DATA GENERAL
CORPORATION
Southboro,
Massachusetts 01772
(617) 485-9100

PROGRAM

Floating Point Interpreter

TAPES

Basic Binary: 091-000012 (5600 - 7577)

Extended Binary: 091-000013 (4100 - 7577)

June, 1969

093-000019-01

Table of Contents

1. Introduction

2. Structure of the Interpreter

3. Floating Point Number Representation

4. Interpreter Use and Command Structure

    4.1. General
    4.2. The Writable Area
    4.3. Initialization
    4.4. Entering and Exiting the Interpretive Mode
    4.5. Floating Point Instruction Set
        4.5.1. Memory Reference Instructions
        4.5.2. Arithmetic Instructions
            4.5.2.1. Options
            4.5.2.2. ALC Instructions with Post-normalize Option
            4.5.2.3. ALC Instructions that Always Post-normalize
            4.5.2.4. Floating ALC Instruction Examples
        4.5.3. Input/Output Instructions
        4.5.4. Special Instructions
        4.5.5. Illegal Instructions
    4.6. Requirements for Reentrance

5. Extended Floating Point Features

    5.1. Mathematical Functions
    5.2. "F" Format Conversion

6. Examples

Appendicies

    A Floating Point Instruction Summary
    B Floating Point Instruction Encoding
    C Mathematical Function Methods

1. Introduction

Most small, general purpose computers do not have hardware for the manipulation of floating point numbers. They have, therefore, implemented software packages to provide floating point functions. Two approaches can be taken to solve this software problem.

The first approach is to provide a number of subroutines, each of which performs a specific function, e.g. floating addition. This approach requires the user to pass inputs in a standard manner to the subroutine. Since most small computers have only two accumulators, only one operand (which requires two words to represent) can be passed in the accumulators. The address of the second operand must be provided along with the subroutine call. A typical calling sequence requires two load accumulator instructions, a subroutine call, and an address word. Furthermore, one operand is always destroyed and replaced by the result (usually the accumulator operand). This means that an intermediate result cannot be tested without destroying an operand--a function often useful in computation. A lengthy program requiring these subroutines cannot make efficient use of storage. Further, the manipulation of operands becomes a tedious, cumbersome job.

The second approach is to provide an interpreter in which all the floating point functions are imbedded and which can simulate floating point registers in storage. The interpreter is given control by a subroutine call. Once in control, it accesses succeeding memory words and "interprets" a sixteen bit word in a manner similar to the hardware. These instructions are not executed in the hardware sense but are instead interpreted to provide extended machine features.

This second approach was adopted for the NOVA floating point package and is the subject of this manual. The Interpreter provides four floating point accumulators which can be addressed and manipulated in a manner similar to the actual hardware accumulators. Floating point instructions are syntactically similar to machine instructions and are assembled in a similar way. For example, *the instruction

```
ADD#    Ø,2,SNR
```

which adds ACØ to AC2, skips the next instruction if the result is non-zero, and does not change ACØ or AC2, has a similar floating point version. This instruction

```
FADD#   Ø,2,FSNR
```

---

* This manual requires the reader to have a good understanding of both, "How to Use The NOVA" and the Assembler Manual.

adds floating accumulator Ø (designated as FACØ) to FAC2, skips the next instruction if the result in non-zero, and does not destroy FACØ or FAC2.

The Interpreter has additional desirable properties. It can be implemented in Read Only Storage (ROS), and it is completely reentrant. These properties, in addition to the floating point instruction set, are described in the body of this manual.

2. Structure of the Interpreter

The Interpreter is non-destructive, i.e. no instruction is modified in any manner during execution. Furthermore, it does not store temporary information within itself. Instead, it uses a writable area which must be provided by the user. These properties enable the Interpreter to be wired in ROS as a customer option.

Within the Interpreter, subroutine linkage is via a push down list. This list is maintained in the writable area provided by the user. This property, coupled with the properties already mentioned, make the Interpreter reentrant. This simply means that if a second user routine requires the Interpreter, it may interrupt the present routine, perform its function using the Interpreter and return to the first routine without affecting the state of the Interpreter. One implication of this is that the Interpreter may "call itself." This property has actually been used, enabling an extended version of the package to compute elementary transcendental functions using floating point instructions.

The Interpreter requires eight page Ø locations which the user must not destroy. These locations and their uses are:

ØØ4: will contain the starting address of the Interpreter
ØØ5: will contain the starting address of the initialization routine for the Interpreter
ØØ6: one temporary word used by the Interpreter and saved by any routine reentering the package.

ØØ7: a word set up by the user to contain the address of a writable area for the Interpreter

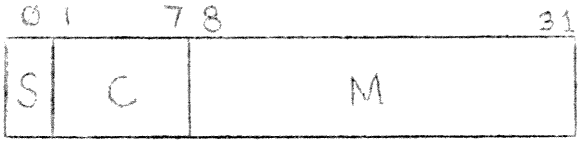
Ø4Ø: a word containing the address of a user written "get character" routine

Ø41: a word containing the address of a user written "put character" routine

Ø42-Ø43: words containing linkage addresses to determine the location of extended code (if present)

3. Floating Point Number Representation*

Floating points numbers are internally stored in two consecutive 16-bit words. The form is:



"S" is the sign of the mantissa, "M", in bits 8-31. The mantissa is considered to be a normalized six digit hexadecimal fraction, and the range of the magnitude of the mantissa is therefore,

$$16^{-1} \leq M \leq (1-16^{-6}).$$

The characteristic, "C", is the integer exponent of 16 in excess 64_{10} code. The total range of magnitudes is this:

$$16^{-1}*16^{-63} \leq F \leq (1-16^{-6})*16^{63}$$

or approximately

$$8.7*10^{-78} \leq F \leq 7.2*10^{75}$$

Any operand having a zero mantissa is represented in true zero from, i.e., bits Ø-31 are Ø. Negative numbers are identical to their positive counterparts except "S"=1 instead of Ø.

The maximum error of a normalized mantissa is less than 16^{-6}.

---

*See also "How to Use The NOVA," Appendix C.

4. Interpreter Use and Command Structure

4.1. General

The Interpreter provides four floating point accumulators. They are numbered Ø,1,2, and 3 similar to the hardware accumulators. The designation FACn will be used for floating accumulator n. Arithmetic is performed accumulator to accumulator as with fixed point instructions. Operands can be accessed and stored using memory reference instructions. Instructions which reference floating point operands in memory should provide an address which points to the first word of the two word operand. If indexing is specified, the hardware index register is used. For example,

```
FLDA  1,4,2
```

loads FAC1 from two consecutive words in memory whose first word address is 4+ contents (AC2). Certain instructions manipulate hardware AC2 and AC3 and will be described in Section 4.5. No facility is provided for manipulating ACØ and AC1 with the floating point instruction set.

4.2. The Writable Area

The basic Floating Point Interpreter requires 64 (decimal) words of contiguous writable memory. If the Extended Interpreter is used, 11Ø (decimal) words must be provided. The Extended Interpreter and its added features are described in detail in

Section 5. Other than the above differences in the length of the writable area, this description applies to both versions.

The address of the first word of the writable area provided by the user must be stored in location ØØ7 of page Ø before any Interpreter commands are executed. If a second routine (or third, etc.) may reenter the Interpreter, this routine must provide an address pointing to a different memory area before reentrance is made.

A number of flags are stored in this writable area which may be examined by the user. To access these flags, the user must exit from the interpretive mode (as described in Section 4.4). If an index register is loaded from location ØØ7, the user may access these flag words by an instruction of the form:

LDA Ø,n,i

where i is the index register (2 or 3), and n is a constant displacement as described below.

n = Ø: The first word contains the overflow/underflow flags. If the result of an operation is less than 16**-64, bit 15 of this word will be set to indicate underflow. If the result of an operation is greater than (1-16**-6)*16**+63, bit 14 of the word will be set to indicate overflow. Other conditions may set overflow/underflow. These will be discussed where appropriate.

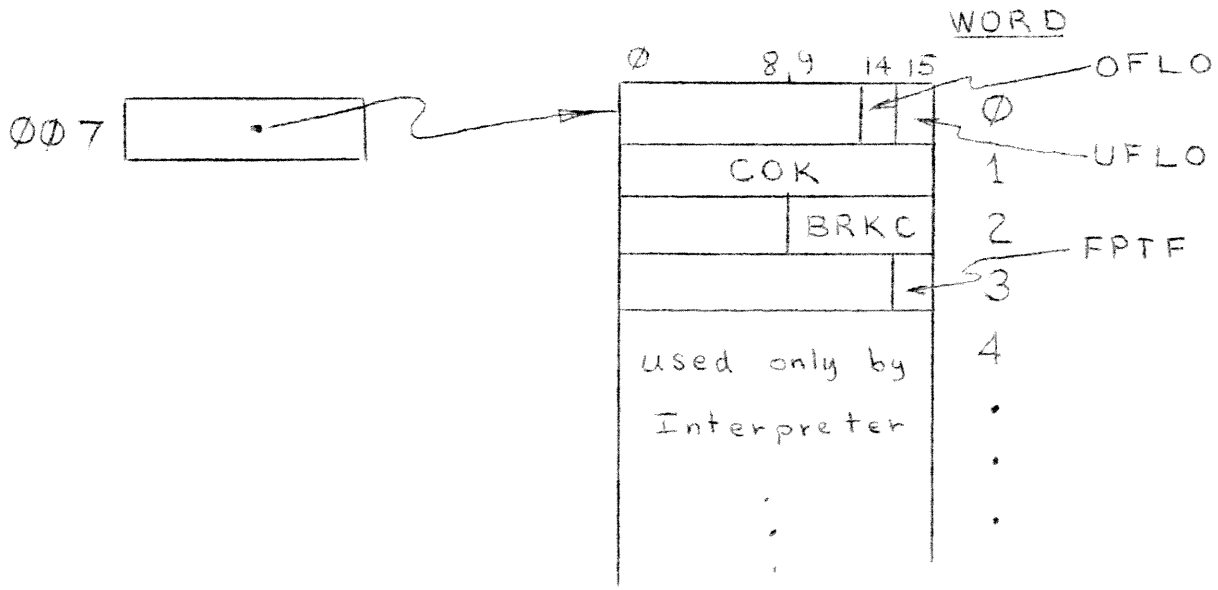
This word is initially cleared. Once a bit is set, it will remain set. It is the user's responsibility to reset these bits.

n = 1: After an input conversion (see Section 4.5.3.) this word will be zero if no conversion was performed (because of an input error). Otherwise the word will be non-zero.

n = 2: After an input conversion, this word will contain the 7-bit ASCII character, right justified, that served as the break character in the input stream.

n = 3: After an input conversion, this word will be Ø if no decimal point (".") was encountered in the input stream. If a point was seen, the word will be 1.

The diagram below summarizes these flags.



A convenient means of accessing these flags is to define four symbolic equivalences such as:

```
FLGS = Ø                    ;overflow/underflow flags
COK  = 1                    ;conversion OK
BRKC = 2                    ;break character
FPTF = 3                    ;floating point flag
```

Now if AC2 contains the address of the writable area, any flag may be accessed by statements like the following:

```
LDA         Ø,FLGS,2
LDA         1,COK,2
LDA         1,BRKC,2
LDA         1,FPTF,2
```

4.3. Initialization

The Interpreter, or more correctly the writable area, must be initialized before floating point instructions are executed. This initialization should be given once for every writable area to be used. The command is simply

```
FINI
```

which generates the instruction

```
JSR  @5
```

(note that the initialization routine address is in location ØØ5). Location ØØ7 must point to the writable area. This command destroys AC3, but preserves all other accumulators and the carry. The initialization routine clears the overflow/underflow flags and sets up linkage for the push down list.

Entering-Exiting the Interpreter Mode

To use the Floating Point Interpreter, it is necessary to distinguish between the processing of normal instructions (described in "How to Use The NOVA") and the processing of floating point instructions. Since the latter instructions are "interpreted" (not executed per se), the Interpreter must be given control before floating point code will be executed properly. Whenever the Interpreter is in control, this will be called the "interpretive mode." Otherwise, the machine will be referred to as in the "normal mode." To enter the Interpretive mode, the command is

FETR

which generates the instruction

JSR @4

As noted earlier, location ØØ4 will contain the starting address of the Interpreter. Once in the interpretive mode, only the floating point instruction set can be used (normal instructions will cause unpredictable results). To return to normal processing, the command

FEXT

must be given.

FETR destroys AC3, but the contents of all other accumulators and Carry will be saved. AC2 and AC3 can be used for indexing in the interpretive mode. Certain instructions also enable modification of the contents of AC2 and AC3. FEXT will restore AC$\emptyset$ and Carry to their state before entering the Interpreter. AC2 and AC3 will reflect any changes caused by floating point instructions that modified their contents.

The normal sequence of code using the Interpreter would be of the form:

```
FINI        ;INITIALIZE INTERPRETER
LDA         ∅,CNST
STA         ∅,TEMP  ; NORMAL INSTRUCTIONS
FETR                ;ENTER INTERPRETER
 .
 .                  ;FLOATING POINT INSTRUCTIONS
 .
FEXT        ; EXIT INTERPRETER
LDA         1,C5
```

4.5. Floating Point Instruction Set

This section contains a description of the instruction set for the basic Floating Point Interpreter. These instructions obey the syntactic rules described in the Assembler Manual. For example, all floating ALC instructions require a source accumulator designation and a destination accumulator designation. All floating point instructions begin with an "F" to distinguish them from the normal instruction set. Appendix A summarizes the instructions and Appendix B gives their octal encoding.

4.5.1. Memory Reference Instructions

The instruction

FLDA n,adr

causes FACn to be loaded with the two word operand at adr,adr+1.

The instruction

FSTA n,adr

causes FACn to be stored in memory at adr,adr+1.

The instruction

FJMP adr

causes control to be transfered to the floating point instruction at adr.

The instruction

FJSR adr

causes control to be transfered to the instruction at adr and AC3 to be set to the value of the current location counter +1. For example, a floating point subroutine can be executed by the following:

```
                    ;MAIN PROGRAM

          FLDA      Ø,LOC     ;LOAD FACØ
          FJSR      SUBR      ;JUMP TO SUBROUTINE
          FSTA      Ø,RSLT    ;STORE RESULT
            .
            .
            .
SUBR:
            .
            .       ;SUBROUTINE
            .
          FJMP      Ø,3 ;RETURN
```

The instruction

FFIX adr

causes the floating point number at adr, adr+1 to be converted to a fixed point, double precision integer (truncated) at adr, adr+1. If the conversion results in an integer whose absolute value is greater than 2**24-1, the overflow flag will be set and 2**24-1 will be returned as the magnitude. Note that while floating point numbers are represented in signed-magnitude format, fixed point values will always be represented in two's complement notation.

The instruction

FFLO adr

causes the fixed point, double precision integer at adr,adr+1 to be converted to a floating point number at adr,adr+1. Negative integers must be represented in two's complement format.

The instruction

FISZ adr

causes the contents of adr to be incremented by one and the next floating point instruction in sequence to be skipped if the result is zero.

The instruction

FDSZ adr

causes the contents of adr to be decremented by one and the next floating point instruction in sequence to be skipped if the result in zero. FISZ and FDSZ should be used with fixed point, single precision integers--not with floating point numbers.

The instruction

FST3 adr

causes AC3 to be stored at adr.

The instruction

FLD3 adr

causes AC3 to be loaded from the contents of adr. FST3 and FLD3 operate on real accumulator 3. FST3 is useful for saving the return address inside a floating point subroutine. Note that the return address should always be saved if a floating point subroutine exits and then enters the interpretive mode, since FETR destroys AC3. These two instructions also provide a means for initializing loop counts without leaving the interpretive mode.

For example,

```
                    FEXT
                    LDA    Ø,CNT
                    STA    Ø,TEMP
          LP1:      FETR
                     .
                     .
                     .
                    FEXT
                    DSZ    TEMP
                    JMP    LP1
                    FETR
```

can be replaced by

```
                    FLD3   CNT
                    FST3   TEMP
          LP1:       .
                     .
                     .
                    FDSZ   TEMP
                    FJMP   LP1
```

4.5.2. Arithmetic Instructions (ALC)

4.5.2.1. Options

The floating point ALC instructions are similar to normal ALC instructions. Two floating accumulators must be specified. The first is the source accumulator, the second the destination accumulator.

Seven skip conditions are defined (in addition to the default "no skip"). These conditions are listed in Table 4-1. The conditions FSZR and FSNR should be used with caution. Since

floating point arithmetic is inherently approximate, the probability of obtaining true zero is very low. The normal procedure is to test a result (or difference) against a small quantity, ε. For example, if we wish to test for the convergence of an iterative procedure we might use the following:

```
EPSLN:  3544Ø          ;EPSILON IS 2*16**-4
        Ø
        .
        .
        .
        FLDA    Ø,ORSLT     ;GET OLE RESULT
        FLDA    1,NRSLT     ;GET NEW RESULT
        FSUB    1,Ø         ;OLD-NEW
        FLDA    1,EPSLN     ;EPSILON
        FPOS    Ø,Ø         ;ABS (OLD-NEW)
        FSUB    1,Ø,FSLE    ;SKIP IF  <  EPSILON
```

Table 4-1

Skip Mnemonic	Effect
FSLT	skip if result < Ø.
FSLE	skip if result ≤ Ø.
FSGT	skip if result > Ø.
FSGE	skip if result ≥ Ø.
FSNR	skip if result ≠ Ø.
FSZR	skip if result = Ø.
FSKP	unconditional skip

All ALC instructions permit the load/no load option. As with normal instructions, if a floating point instruction mnemonic is suffixed with "#", the results of the operation will not replace the contents of the destination register.

A further option is available with one class of ALC instructions. This option will prevent post-normalization of the result. The instructions described in Section 4.5.2.2. permit this option. It is called for by suffixing "U" (for unnormalized) to the instruction mnemonic. For example, FMOV Ø,1 moves FACØ to FAC1 and normalizes the result, while FMOVU Ø,1 moves FACØ to FAC1 without normalization.

4.5.2.2. ALC Instructions with Post-Normalize Option

The instruction

FMOV n,m

moves FACn to FACm.

The instruction

FPOS n,m

moves the absolute value of FACn to FACm.

The instruction

FMNS n,m

moves the negative of the absolute value of FACn to FACm.

The instruction

FNEG n,m

moves the negative value of FACn to FACm.

The instruction

FRND n,m

rounds the value of FACn and moves it to FACm. By "round" we mean the following.

$$F = 16^{**}-6^{*} \lfloor 16^{**}6^{*}F+1/2 \rfloor \dagger$$

The instruction

FADD n,m

adds FACn to FACm and moves the result to FACm. If underflow occurs, the underflow flag is set and true ∅ returned as the result. If overflow occurs, the overflow flag is set and a magnitude of (1-16**-6)*16**+63 is returned as the result. The operands are assumed to be pre-normalized.

The instruction

FSUB n,m

subtracts FACn from FACm and moves the result to FACm. The overflow conditions are handled as with FADD. Prenormalized operands are assumed.

Table 4-2 summarizes the floating ALC instructions with post-normalize option.

---

† $\lfloor X \rfloor$ gives the maximum integer K such that $K \leq X$

Table 4-2

ALC Instructions with Post-Normalize Option

Instruction	Effect
FMOV n,m	FACn → FACm
FPOS n,m	|FACn| → FACm
FMNS n,m	− |FACn| → FACm
FNEG n,m	− FACn → FACm
FRND n,m	rounded FACn → FACm
FADD n,m	FACn+FACm → FACm
FSUB n,m	FACm−FACn → FACm

4.5.2.3. ALC Instructions that Always Post-Normalize

This class of ALC instructions always post-normalizes the result. They assume pre-normalized operands. Overflow is checked and indicated by setting the overflow flag and returning a magnitude of (1-16**-6)*16**63 as the magnitude of the result. Underflow is checked and indicated by setting the underflow flag and returning true ∅ as the result.

The instruction

FMPY n,m

multiplies FACn by FACm and moves the result to FACm.

The instruction

FDIV n,m

divides FACm by FACn and moves the result to FACm.

The instruction

FHLV n,m

halves FACn and moves the result to FACm.

Table 4-3 summarizes the ALC instructions that always post-normalize the result.

Table 4-3

ALC Instructions that Always Post-Normalize

Instruction	Effect
FMPY n,m	FACm*FACn → FACm
FDIV n,m	FACm/FACn → FACm
FHAV n,m	FACn/2. → FACm

4.5.2.4. Floating ALC Instruction Examples

Some examples of legal floating point ALC instructions are:

```
FMPY    1,Ø
FADD    Ø,1,FSGE
FSUB#   1,Ø,FSLT
FMOVU   3,Ø
FMOV    Ø,Ø,FLST
FNEG    Ø,Ø,FSKP
```

4.5.3. Input/Output Instructions

The use of I/O instructions requires the user to provide two special routines. The first is an input routine which, when called, must return an ASCII input character, right justified in ACØ with bit 8 = Ø. The address of this routine must be stored by the user in location Ø4Ø of page Ø.

The second routine is an output routine which, when called, must accept an ASCII output character, right justified in ACØ with bit 8 = Ø. The address of this routine must be stored in location Ø41 of page Ø. All output messages to this routine will be terminated by a null (all zero) character.

These I/O routines must be reentrant for the Interpreter to be reentrant. (If the routines which interrupt and use the Interpreter do not use I/O instructions, the user I/O routines need not be reentrant).

The instruction

```
FDFC n  ;FLOATING POINT DECIMAL TO FLOATING
        ; CONVERT
```

will cause an ASCII character string in engineering notation to be converted to internal floating point form and loaded in FACn. The input characters must be provided by the user routine whose address is stored in location 40_8 of page 0. Numbers in the following form will be converted.

$$\left\{ [\pm]\, n[\cdots n[.]\, n \cdots n[E\, [\pm]\, m[m]]] \begin{matrix} b \\ r \\ e \\ a \\ k \end{matrix} \right\}$$

where n is the decimal mantissa (the first seven non-zero digits will be converted and the remaining digits ignored). The signs of the mantissa and characteristic are optional with the default assumed +. The break character is any character other than

1. a decimal digit

or 2. an "E"

If the break character is a rubout (177), the entire string will be ignored and a new one must be given, i.e. the conversion starts over.

If the conversion results in a number less than 16**-1*16**-63, the underflow flag will be set and true zero will replace FACn. If the conversion results in a number whose magnitude is greater than (1-16**-6)*16**63, this latter magnitude will replace FACn and overflow will be set. As described in Section 4.2, input conversion returns three additional words of information: conversion OK flag, the break character, and decimal point seen flag. These may be examined and used as necessary.

Examples of legal character strings are:

```
 1*
 1.*
-1*
+1*
 1E3*
 3.1415926*
 1.E+7Ø*
 1.E-7Ø*
```

where * will be returned as the break character and conversion OK will be non-zero.

Some illegal character strings are:

```
 A     (break character will be A)
+*     (break character will be *)
+.!    (break character will be !)
```

Conversion OK will be zero in all these illegal cases.

The instruction

```
FDFCI    n      ;FDFC WITH INDICATION
```

will provide the user with an indication before the conversion begins. The ASCII character "F" followed by a null character will be passed to the output routine whose address is given in location Ø41. For example, if the user has provided for I/O from the teletype, the use of FDFCI will print "F" on the page copy every time an input is required. In all other respects it is identical to FDFC.

The instruction

```
FFDC n  ; FLOATING POINT FLOATING TO DECIMAL CONVERT
```

will convert the number in FACn to an ASCII character string in engineering notation. The output characters will be passed one at a time, right justified in ACØ, to a user routine whose address is stored in location 41_8 of page Ø. The output string will be of the form:

$$\{\pm\}.nnnnnnnE\{\pm\}mm$$

where the n's represent the decimal digits of the mantissa, and the m's represent the decimal characteristic. The string will be terminated by a null character.

4.5.4. Special Instructions

Two special instructions are defined which modify the index registers.

The instruction

FIC2

causes AC2 to be incremented by two

The instruction

FIC3

causes AC3 to be incremented by two. These instructions are useful for indexing through a table of floating point numbers (see Section 6, example #2).

A third special instruction provides a HALT feature within the interpretive mode. The instruction

FHLT

will cause the Interpreter to HALT. Hardware ACØ will contain the address of the FHLT instruction. The address lights will have no apparent relationship to the HALT, since the address is within the Interpreter. The user may press CONTINUE to resume after this HALT.

4.5.5. Illegal Instructions

The proper encoding for all floating point instructions is given in Appendix B. The Interpreter will HALT if an illegal instruction is encountered. Hardware ACØ will contain the

address where the illegal instruction was found. This HALT will occur if extended instructions are used and only the Basic Interpreter is loaded, or on any bit configuration that cannot be decoded into a floating point instruction. The user cannot press CONTINUE to resume after this HALT.

4.6. Requirements for Reentrance

A number of points regarding reentrance of the Interpreter have been mentioned. This Section explicitly defines the rules which must be followed by any routine which interrupts a base level routine and reenters the Interpreter.

A. All hardware accumulators, Carry, and page Ø locations ØØ6 and ØØ7 must be saved.

B. A new writable area address must be provided in location ØØ7.

C. An FINI must be issued after location ØØ7 has been set up (only necessary the first time).

D. If I/O instructions are to be used, the user I/O routines must be reentrant. (Alternatively, locations Ø4Ø and Ø41 must be saved and addresses provided to different I/O routines).

E. Upon exit to the base level routine, the hardware accumulators, Carry and locations ØØ6 and ØØ7 must be restored.

5. Extended Floating Point Features

The instructions described up to this point are available with a 1K Interpreter. An extended version of the Interpreter is available which occupies 2K of storage.* The extended version provides the instructions already described, in addition to a number of mathematical functions and "F" format output. If the extended Interpreter is used, 11Ø (decimal) words of writable storage must be provided by the user.

5.1. Mathematical Functions

The math functions are implemented using ALC instructions which always post-normalize (see Section 4.5.2.3.). They permit the no load option as well as the floating skip options. Appendix C provides a detailed description of the methods used to implement these functions as well as a discussion of their accuracy. The following is a general description of each instruction.

The instruction

```
FALG  n,m
```

computes the natural logarithm of the contents of FACn and moves the result to FACm. If the argument is less than Ø, the overflow flag is set and -(1-16**-6)*16**63 is returned as the result.

---

*With a 4K configuration, the Interpreter requires locations 56ØØ-6577 (octal) and the Extended Interpreter requires locations 41ØØ-5577.

The instruction

FATN n,m

computes the arctangent of the contents of FACn and moves the result to FACm. The result is an angle expressed in radians in the range $-pi/2 \leqslant \arctan(x) \leqslant pi/2$.

The instruction

FCOS n,m

computes the cosine of the contents of FACn and moves the result to FACm. The argument is assumed to be an angle expressed in radians. If the argument is greater than 2**24, the overflow flag is set and the result will be incorrect.

The instruction

FSIN n,m

computes the sine of the contents of FACn and moves the result to FACm. The argument is assumed to be an angle expressed in radians. If the argument is greater than 2**24, the overflow flag is set and the result will be incorrect.

The instruction

FTAN n,m

computes the tangent of the contents of FACn and moves the result to FACm. The argument is assumed to be an angle expressed in radians. If the argument is greater than 2**24, the overflow flag will be set and the result will be incorrect.

The instruction

FEXP n,m

computes e raised to the power contained in FACn and moves the result to FACm. If the argument is less than -177.5, true Ø is returned and the underflow flag is set. If the argument is greater than 174.673, +(1-16**-6)*16**63 is returned and the overflow flag is set.

The instruction

FSQR n,m

computes the square root of the argument in FACn and moves the result to FACm. If the argument is less than Ø, the underflow flag is set and true Ø is returned as the result.

Table 5-1 summarizes the math functions.

5.2. "F" Format Conversion

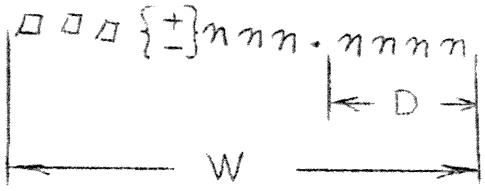
The standard Interpreter provides floating point to decimal conversion with "E" format output. The extended version provides "F" format output as well.

The instruction

FFDCF n ;FFDC WITH "F" FORMAT OUTPUT

will convert the floating point number in FACn to decimal and output a character string in "F" format via the user routine

whose address is stored in location Ø41. The output will be of the form:



The width of the field (including sign and decimal point), W, and the number of places to be given after the decimal point, D, must be set up in the writable area before FFDCF is given. The displacements of these words are

W = 136
D = 137.

They can be accessed in a manner similar to the flags described in Section 4.2. Two conditions will cause the overflow flag to be set and no conversion to be performed.

Table 5-1

MATH FUNCTIONS

Instruction	Effect
FALG n,m	ln(FACn) → FACm
FATN n,m	arctan (FACn) → FACm
FCOS n,m	cosine (FACn) → FACm
FSIN n,m	sine (FACn) → FACm
FTAN n,m	tangent (FACn) → FACm
FEXP n,m	e** (FACn) → FACm
FSQR n,m	(FACn)**½ → FACm

1.) W > 32 (entire width of field limited to 32 characters)

2.) W < D+2 (W must be 2 greater than D to provide for sign and decimal point)

If W is not large enough to accommodate the number, significant digits will be lost.

Assume W has been set to 12 (decimal) and D to 6. Examples of "E" format versus "F" format outputs are:

"E" Format,	"F" Format
+.13746ØØE+Ø2	+13.746ØØØ
-.7968433E-Ø3	-.ØØØ796
+.1ØØØØØØE+Ø4	+1ØØØ.ØØØØØØ
-.1ØØØØØØE+Ø5	1ØØØØ.ØØØØØØ (note sign lost)
+.1ØØØØØØE+Ø6	ØØØØØ.ØØØØØØ (all lost)
-.35ØØØØØE-Ø6	-.ØØØØØØ (significance
+.4713279E-Ø1	+.Ø47132 lost)

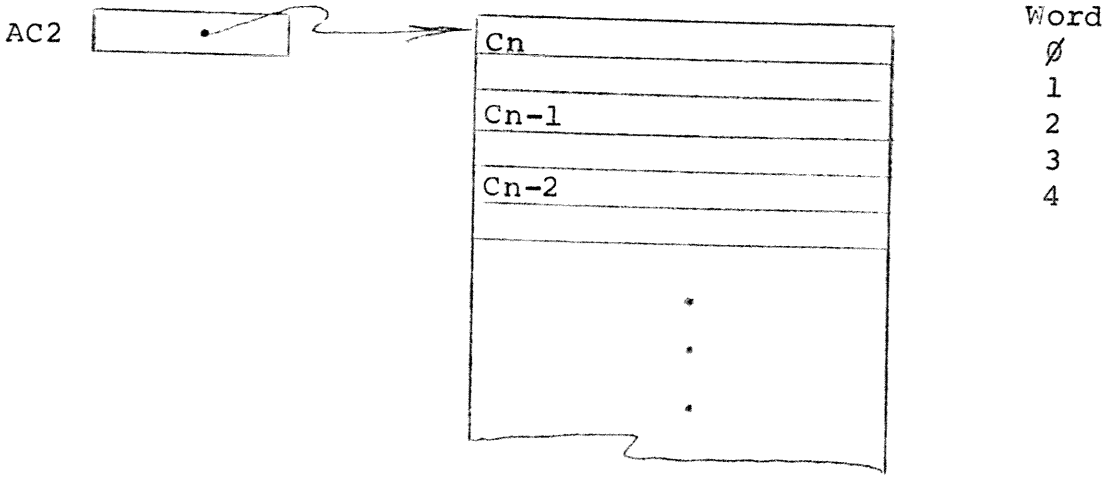
6. Examples

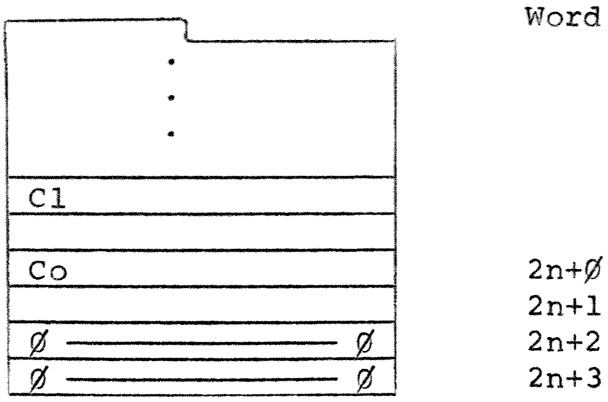
The following routine is an example of a floating point subroutine that performs a square root Newton iteration on the trial guess in FACØ given the argument in FAC1.

```
;
;THIS ROUTINE PERFORMS A SQUARE ROOT
;      NEWTON ITERATION
;FAC1 CONTAINS THE ARGUMENT AND IS
;      NOT DESTROYED
;FACØ CONTAINS THE PREVIOUS GUESSTIMATE
;COMPUTES (FACØ+(FAC1/FACØ))/2.
;CALLING SEQUENCE
;      FJSR  NSR
;      RETURN : RESULT IN FACØ
;


NSR:  FMOV 1,2   ;SAVE ARGUMENT
      FDIV Ø,2   ;FAC1/FACØ
      FADD 2,Ø   ;FACØ+FAC1/FACØ
      FHLV Ø,Ø   ;(FACØ+FAC1/FACØ)/2.
      FJMP Ø,3   ;RETURN
```

The next routine is an example of polynomial evaluation. This routine requires AC2 to point to the first word of a table of floating point coefficients, ordered high order coefficient down and terminated by true Ø. For example:





The routine uses Horner's method for evaluation, i.e.

$$f(x) = (\cdots(((x + C_n)\ x + C_{n-1})\ x + C_{n-2})\ x \cdots + C_1)\ x + C_o$$

```
;
;POLYNOMIAL EVALUATION
;FAC2 CONTAINS ARGUMENT (X)
;AC2  POINTS TO COEFFICIENT LIST TERMINATED BY Ø,
;     AND ORDERED HIGH TO LOW
;RESULT RETURNED IN FACØ
;FACØ,FAC1 DESTROYED
;CALLING SEQUENCE
;     FJSR      FPLY
;     RETURN
;
FZRO: Ø
      Ø
FPLY: FLDA     Ø,FZRO    ;CLEAR RESULT
FPLY1: FLDA    1,Ø,2,    ;GET COEFFICIENT
       FMOV    1,1,FSNR
       FJMP    Ø,3       ;RETURN IF ZERO
       FMPY    2,Ø       ;SUM * ARGUMENT
       FADD    1,Ø       ;SUM * ARG. + COEF.
       FIC2              ;BUMP POINTER TO NEXT COEF.
       FJMP    FPLY1
```

APPENDICES

Appendix A

Floating Point Instruction Summary

Standard

FADD	Floating Add
FDFC	Floating Decimal to Floating Convert
FDFCI	FDFC with Indication
FDIV	Floating Divide
FDSZ	Floating Decrement and Skip if Zero
FETR	Floating Mode Enter
FEXT	Floating Mode Exit
FFDC	Floating Floating to Decimal Convert
FFIX	Floating to Fixed
FFLO	Fixed to Floating
FHLT	Floating Halt
FHLV	Floating Halve
FIC2	Floating Increment AC2
FIC3	Floating Increment AC3
FINI	Floating Initialize
FISZ	Floating Increment and Skip if Zero
FJMP	Floating Jump
FJSR	Floating Jump to Subroutine
FLD3	Floating Load AC3
FLDA	Floating Load Floating Accumulator
FMOV	Floating Move
FMNS	Floating Move Minus
FMPY	Floating Multiply
FNEG	Floating Negate
FPOS	Floating Move Positive
FRND	Floating Round
FST3	Floating Store AC3
FSTA	Floating Store Floating Accumulator
FSUB	Floating Subtract

Extended

FALG	Floating Natural Logarithm
FATN	Floating Arctangent
FCOS	Floating Cosine
FEXP	Floating Exponential
FFDCF	Floating Floating to Decimal Convert with "F" Format
FSIN	Floating Sine
FSQR	Floating Square Root
FTAN	Floating Tangent

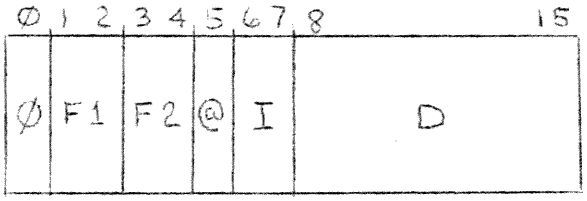
Floating Point Options

#	No Load
FSGE	Floating Skip on Greater Than or Equal
FSGT	Floating Skip on Greater Than
FSKP	Floating Skip
FSLE	Floating Skip on Less Than or Equal
FSLT	Floating Skip on Less Than
FSNR	Floating Skip on Non Zero Result
FSZR	Floating Skip on Zero Result
U	Unnormalize (no post-normalization)

Appendix B

Floating Point Instruction Encoding

Memory Reference



F1 = Ø1
F1 = 1Ø

LDA
STA

F2 is FAC

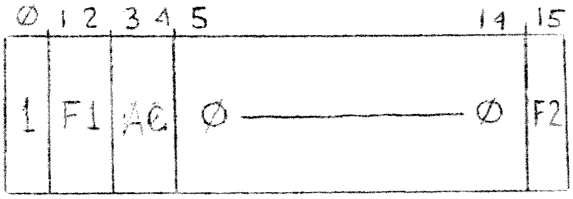
F1 = ØØ

F2 = ØØ FJMP
F2 = Ø1 FJSR
F2 = 1Ø FISZ
F2 = 11 FDSZ

F1 = 11

F2 = ØØ FFLO
F2 = Ø1 FLD3
F2 = 1Ø FST3
F2 = 11 FFIX

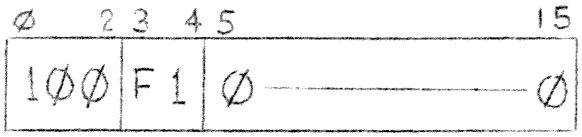
Instructions Requiring an Accumulator



F1 = Ø1
F1 = 1Ø
F1 = 11

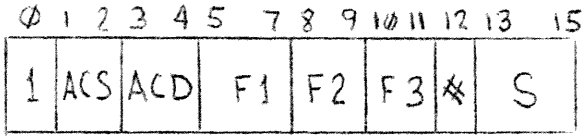
FDFC F2 = 1 "I"
FFDC F2 = 1 "F"
Illegal

Special Instructions



F1 = ØØ	FEXT
F1 = Ø1	FIC2
F1 = 1Ø	FIC3
F1 = 11	FHLT

ALC Instructions



S = Ø	No skip	S = 4	FSZR
S = 1	FSGT	S = 5	FSGE
S = 2	FSLT	S = 6	FSLE
S = 3	FSNR	F = 7	FSKP

F1 ≠ Ø (8-1Ø must be Ø; 11 is "U" option)

F1 = 1	FNEG
F1 = 2	FMOV
F1 = 3	FPOS
F1 = 4	FMNS
F1 = 5	FSUB
F1 = 6	FADD
F1 = 7	FRND

F1 = Ø

F3 = Ø

F2 = 1	FMPY
F2 = 2	FDIV
F2 = 3	FHLV

F3 ≠ Ø (Extended)

F2,F3 = 1	FALG
F2,F3 = 2	FATN
F2,F3 = 3	FCOS
F2,F3 = 4	Illegal (FMPY)
F2,F3 = 5	FSIN
F2,F3 = 6	FTAN
F2,F3 = 7	Illegal
F2,F3 = 1Ø	Illegal (FDIV)
F2,F3 = 11	FEXP
F2,F3 = 12	FSQR
F2,F3 = 13	Illegal
F2,F3 = 14	Illegal (FHLV)
F2,F3 = 15	Illegal
F2,F3 = 16	Illegal
F2,F3 = 17	Illegal

Appendix C

The following pages describe in some detail the function, method, and accuracy of the mathematical routines supplied with the extended Interpreter. The accuracy of the routine is influenced by two factors:

1. The accuracy of the argument and 2. The accuracy of the algorithm. These two factors will be mentioned for each routine. The accuracy of the algorithm itself assumes an argument that is exact, i.e. no argument error.

The relative and absolute errors of a function routine are defined as follows:

Let f(x) = the true value of the function at x

Let g(x) = the result returned by the function routine given x

Now the absolute error of the result is,

ABS(f(x) - g(x))

and the relative error of the result is,

ABS((f(x) -g(x))/f(x)).

Floating Point Arctangent

Function: To calculate the arctangent of x, where x is a floating point number, and return an angle in radians in the range $-pi/2 \leq \arctan(x) \leq pi/2$

Method: The range of x is immediately reduced to

$$\emptyset \leq x \leq 1$$

by means of the identities

$$\arctan(-abs(x)) = -\arctan(abs(x))$$

and if $abs(x) > 1$,

$$\arctan(abs(x)) = pi/2 - \arctan(1/abs(x))$$

For $x > \tan(pi/12)$ the range is reduced to

$$\tan(-pi/12) \leq y \leq \tan(pi/12)$$

by means of the identity

$$\arctan(x) = \arctan(x\emptyset) + \arctan((x-x\emptyset)/(1-x*x\emptyset)).$$

For $x\emptyset = 1/3**\emptyset.5$, we obtain

$$\arctan(x) = pi/6 + \arctan(x*3**\emptyset.5-1/(x+3**\emptyset.5))$$
$$= pi/6 + \arctan(y)$$

where y satisfies the range given above. The arctan is computed using the first four terms of a polynomial approximation of the form:

$$\arctan(x) \cong x * \sum_{i=0}^{n} C_{i}*x**(2*i)$$

Accuracy:

Argument Error

If x is the argument, the absolute error of the result is approximately

$$\varepsilon/(1+x^{**}2)$$

where ε is the absolute error in x. Thus for small values of x, the errors are almost equal, while as x becomes larger, the effect of the argument error decreases.

Maximum Relative Error

For the range

$$-\tan(\pi/12) \leq x \leq \tan(\pi/12),$$

the maximum relative error is approximately 1Ø**-7.6.

Floating Point Exponential

Function: To calculate e to the power x, where x is a floating point number.

Method: If $x < -177.5$, return $\emptyset$ as the result and set underflow flag.

If $x > 174.673$, return the largest positive number as the result and set overflow flag.

Otherwise, we use the equality

$$e^{**}x = 2^{**}(x*\log_2(e))$$

Let $x*\log_2(e) = m*16^{**}c$

Further, let $q = \lfloor m*16^{**}c \rfloor$

and $f = m*16^{**}c - q$

Therefore $\emptyset \leq f < 1.\emptyset$

Now $2^{**}(f+q) = 2^{**}f*2^{**}q$

Let us compute $2^{**}f$.

The range of f can be further reduced if we let

$g = f - \emptyset.5$ where $-\emptyset.5 \leq g < \emptyset.5$

We compute $2^{**}g$ directly if $g \geq \emptyset$, otherwise we compute $2^{**}g$ as

$$1/(2^{**}\text{abs}(g))$$

$2^{**}g$ (where $\emptyset \leq g < \emptyset.5$) is computed using the first five terms of a polynomial approximation of the form:

$$2^{**}g \cong \sum_{i=\emptyset}^{n} c_i*g^{**}i$$

Now f = g+Ø.5

Therefore, 2**f = 2**g*2**Ø.5

But the answer is 2**f*2**q

q is an integer and we let q = 4i + j

Now 2**f*2**q = 2**f*16**i*2**j

The characteristic of f is added to i to obtain 2**f*16**i. This result is shifted j positions left if i>Ø or j position right if i<Ø.

The result is of course e**x.

Accuracy:

Argument Error:

The relative error of the result is approximately equal to the absolute error of the argument. Thus for large values of x, substantial relative errors in the results can occur.

Maximum Relative Error

For the range

$$Ø \leqslant x < Ø.5$$

the maximum relative error of 2**x is approximately 1Ø**-7.Ø.

Floating Point Natural Logarithm

Function: To calculate the natural logarithm of x, where x is a floating point number.

Method: If $x < \emptyset$, overflow flag is set and minus the largest floating point number is returned as the value. Otherwise, let x = m*16**c. By means of a binary normalization, the range of m' is reduced to

$$1/2 \leqslant m' < 1,$$

and x = m' *16**p*2**(-q) where q = number of left shifts required to normalize ($\emptyset \leqslant q \leqslant 3$).

Now for $1/2 \leqslant m' < 1/2**\emptyset.5$

let a = 1/2, b = 1

for $1/2**\emptyset.5 \leqslant m' < 1$

let a = 1, b = $\emptyset$

Define y = (m'-a)/(m'+a)

then m' = a*(y+1)/(-y+1)

Now x = 16**p*2**(-q)*a*(1+y)/(1-y)
= 2**(4p-q-b)*(1+y)/(1-y)

Using ln(x) = ln(2) *log2(x) we obtain

ln(x) = ln(2)* [(4p-q-b)+log2((1+y)/(1-y))]

ln(x) = ln(2)* (4p-q-b)+ln((1+y)/(1-y))

From the above, we can determine that

$2**-\emptyset.5 \leqslant (1+y)/(1-y) \leqslant 2**\emptyset.5$

The ln((1+y)/(1-y)) is determined for the above range using the first three terms of a polynomial approximation of the form:

$$\ln(z) \cong z * \sum_{i=\emptyset}^{n} ci*z**(2*i)$$

where z = (1+y)/(1-y).

Accuracy:

Argument Error

The absolute error in the result is approximately equal to the relative error in the argument. Therefore, an argument close to 1 can give a large error since the function at this value is quite small.

Maximum Absolute Error

For the range

$$1/2**-\emptyset.5 \lessapprox (1+x)/(1-x) \lessapprox 2**\emptyset.5$$

the maximum absolute error of ln(x) is approximately

$$1\emptyset**-7.6.$$

Floating Point Sine and Cosine

Function: To calculate sin(x) or cos(x), where x is the floating point angle in radians.

Method: Compute p = abs(x)*4/pi

Let q = $\lfloor p \rfloor$, f = p-q where $\emptyset \leq f < 1$

Now q represents the half-quadrant in which the abs(x) falls.

Using the following equalities.

$$\sin(x) = -\sin(-x)$$
$$\cos(x) = \cos(-x)$$
$$\cos(x) = \sin(x+\pi/2)$$

we define ql = q if sin is required and $x \geq \emptyset$
ql = q + 2 if cos is required
ql = q + 4 if sin is required and $x < \emptyset$

Then for all values of x, the computation has been reduced to

$$\sin(\pi/4*(q_1+f)) = \sin(t)$$

Since sine (and cosine) are periodic in 2*pi, we take $q_1 = q_1 \bmod 8$

Using the further equality

$$\sin(\pi/4+x) = \text{cosine}(\pi/4-x)$$

we finally can produce the table below

q_1	sin(t)
$\emptyset$	sin(pi/4*f)
1	cos(pi/4*(1-f))
2	cos(pi/4*f)
3	sin(pi/4*(1-f))
4	-sin(pi/4*f)
5	-cos(pi/4*(1-f))
6	-cos(pi/4*f)
7	-sin(pi/4*(1-f))

In all cases, the argument has been reduced to the range $\emptyset \leqslant t < pi/4$. The sin is computed using the first four terms of a polynomial approximation of the form:

$$\sin(x) \cong x^* \sum_{i=\emptyset}^{n} Ci*x**(2*i)$$

The cosine is computed using the first four terms of a polynomial approximation of the form:

$$\cos(x) \cong \sum_{i=\emptyset}^{n} Ci*x**(2*i)$$

Accuracy:

Argument Error

The absolute error of the result is approximately equal to the absolute error in the argument. Thus, the larger the argument, the larger the absolute error of the result.

Maximum Relative Error

For the range

$$\emptyset \leqslant x < pi/4$$

the maximum relative error for sin(x) and cos(x) is approximately 1Ø**-7.4

Floating Point Square Root

Function: To calculate the square root of a floating point number, x.

Method: Let x = m*16**c

If m<Ø, set underflow flag and return Ø. as the result.

If m=Ø, return Ø. as the result.

Otherwise, let c = 2*p + q where p is an integer and q = Ø or 1. Now if q = Ø, we have x = m*16**2p and x**1/2 = m**1/2 * 16**p.

If q = 1, we have x = m*16**(2p+1)
= m*16**(2p+2)/16

and x** 1/2 = (m**1/2)/4 *16**(p+1)

Therefore, the characteristic of the result is p + q, and the problem has been reduced to finding a suitable first guess for m**1/2 if q = Ø or (m**1/2)/4 if q = 1.

An initial guess is taken in the hyperbalic form

yØ = a+b/(c+m)

where for q = Ø, a = 1.8Ø713
b = -1.57727
c = Ø.954182

and for q = 1, a = Ø.428795
b = -Ø.343Ø368
c = Ø.877552

The initial guess is now

```
y1 = yØ*16**(p+q)
```

Two Newton iterations give us the result.

```
          y2 = (y1+x/y1)/2
x**1/2  =  y3 = (y2 + x/y2)/2
```

Accuracy:

Argument Error

The relative error of the result is approximately half the relative error in the argument.

Maximum Relative Error

The maximum relative error for x**Ø.5 is approximately 1Ø**-6.Ø.

Floating Point Tangent

Function: To calculate the tangent of x, where x is the floating point angle in radians.

Method: Compute p = abs(x)*4/pi

Let q = $\lfloor p \rfloor$, f = p-q where $\emptyset \leq f < 1$

Take q_1 = q mod 4 and in a manner similar to sine-cosine we can obtain the table below

q_1	tan(x)
$\emptyset$	tan (pi/4*f)
1	cot (pi/4*(1-f))
2	-cot (pi/4*f)
3	-tan (pi/4*(1-f))

In all cases, the argument has been reduced to the range $\emptyset \leq arg < pi/4$.

The tangent is computed using the first six terms of a polynomial approximation of the form:

$$\tan(x) = x * \sum_{i=\emptyset}^{n} Ci*x**(2*i)$$

Accuracy:

Argument Error

The absolute error of the result is approximately equal to

$$\varepsilon * (1 + \tan(x)**2)$$

where ε is the absolute error of the argument. Thus if x is near an odd multiple of pi/2, an argument error will produce a larger absolute error in the result.

Maximum Relative Error

For the range

$$\emptyset \leq x < pi/4$$

the maximum relative error of tan(x) is approximately 10**-6.6.