GOODYEAR
AEROSPACE

GOODYEAR · AEROSPACE · CORPORATION

CODE IDENT NO. 25500

PRELIMINARY

GOODYEAR AEROSPACE CORPORATION

AKRON, OHIO 44315

FUNCTION ALGORITHMS
FOR MPP
SCIENTIFIC SUBROUTINES
VOL. I

GER-17221

FEBRUARY, 1984

Prepared by:
J. G. Gouch

E-ID-26(7-71)
REF: EOI 380

MPP SCIENTIFIC SUBROUTINES
Table of Contents

Chapter Page

Appendix

1.0 GENERAL SUMMARY

This final report represents design documentation and user documentation for Function Algorithms for the Massively Parallel Processor (MPP) developed by Goodyear Aerospace Corporation (GAC) under NASA contract NAS5-27610.

The contract specifies development of MPP assembler instructions to perform the following functions:

Natural Logarithm
Exponential (e to the x power)
Square Root
Sine
Cosine
Arctangent

To fulfill the requirements of the contract, parallel array and scalar inplementations for these functions have been developed on the PDP11/34 Program Development and Management Unit (PDMU) that is resident at the MPP testbed installation located at the NASA Goddard facility.

1.1 REQUIREMENTS SUMMARY

Each function was specified to perform on parallel array data located in the MPP Array Unit, and serial data located in the MPP Main Control Unit. Function Arguments and results were specified as real VAX-standard 32-bit floating-point format.

Arguments to the sine and cosine functions were required to be in radians. Results of the arctangent functions were to range between -pi/2 and +pi/2.

Specifications for error conditions that were required of each function are:

natural logarithm - overflow on arguments outside domain of implementation.
exponential - overflow on arguments outside domain of implementation.
sine - none.
cosine - none.
arctangent - none.
square root - error on negative input arguments.

Errors for the array functions were to be indicated by a logical one placed in an error bitplane. Errors for the MCU functions were to be indicated by setting error flag bits.

Errors for the array functions were to be indicated by a logical one placed in an error bitplane. Errors for the MCU functions were to be indicated by setting error flag bits.

Accuracy for each function was specified as:

natural logarithm - results accurate to full range of floating point format.

exponential - results accurate to full range of floating point format.

sine - 6 digit precision to the right of the decimal point.

cosine - 6 digit precision to the right of the decimal point.

arctangent - 6 digit precision to the right of the decimal point, and one integer digit to the left of the decimal point.

square root - results accurate to full range of floating point format.

The contract deliverable items are:

- a Final Project Report containing:

  algorithm descriptions for each function,
  error conditions and their effects,
  how each function may be accessed via a Higher Order Language (HOL
  measured execution times,
  MPP implementation descriptions.

- a VAX computer compatible tape containing all source code required to generate the object code for each function to be executed on the MPP.

1.2 RESULTS SUMMARY

Each function has been developed for parallel array data, and serial Main Control Unit (MCU) data of the standard VAX 32-bit floating-point format. The array functions were computed iteratively for the LNA, EXPA, SINA, COSA, and SINCOS subroutines. Therefore, these functions require an MCU subroutine portion as well as one or more PECU portions. The array algorithms are discussed in Chapter 2.

All of the serial MCU functions were performed using Discrete Orthonormal LeGendre Polynomial (DOL) expansion. The MCU algorithms are discussed in Chapter 3.

Chapter 4 describes the filename conventions, library names and user information to incorporate these functions into MPP applications .

1.2.1 TIMING SUMMARY

Although the general theory of each algorithm would apply to other formats, the implementation of the algorithms has been optimized for the VAX format. In addition, optimization of PECU code was performed to execute each array function in the fewest possible array cycles. The execution times for each function are summarized in Table 1.0 .

Note the addition of SINCOS which computes both the Sine and Cosine functions in the array. This function was made available with minimal effort using the required Sine and Cosine functions, and provides both results at a rate near the execution times of the individual subroutines for Sine and Cosine.

Table 1.0 MPP Scientific Function Execution Times

ARRAY FUNCTION	EXECUTION TIME (SEC x 10**-6)
LNA	557.9
EXPA	416.3
SQRTA	74.0
SINA	333.7
COSA	333.7
SINCOS	347.0
ATANA	391.7

MCU FUNCTION	EXECUTION TIME (SEC x 10**-6)
LNM	62.7
EXPM	65.5
SQRTM	55.7
SINM	42.7
COSM	47.8
ATANM	54.5

1.2.2 SUMMARRY OF ERROR CONDITIONS

Errors in array functions are indicated as bits 'set' within an error bitplane. The error bitplane is designated by a function argument. Errors in the MCU subroutines are indicated as bits 'set' within a specific MCU register.

The error conditions are decribed in detail in the algorithm description of each subroutine (See Table 3.0) and are summarized as follows:

Natural Logarithm - Errors occur for negative input values.

Exponential - Errors occur for input values with exponents greater than 2**7 and:

Positive Input - Overflow indicated (output set to maximum VAX number).

Negative Input - Underflow indicated (output set to VAX '0').

(For array inputs less than 2**-31, the output is clipped to VAX '0' and a status bit is set).

Square Root - Errors occur for negative input values.

Sine,Cosine - Errors occur for serial inputs greater than 2**24. (See Section 3.2.2).

Arctangent - None

1.2.3 ACCURACY SUMMARY

The array algorithms described in this report use iterative technique for computing each function and are accurate over the full range of VAX floating-point data.

Detailed theoretical error analysis for the MCU serial algorithms is included in Appendix A.

2.0 ARRAY ALGORITHMS

Section 2.0 describes the array scientific subroutines. See Section 3.0 for the description of the MCU subroutines.

2.1 GENERAL DESCRIPTION OF ALGORITHMS

2.1.1 NATURAL LOGARITHM SUBROUTINE

The MPP array field function, ALOGA[X,Y,O,T], uses the input array field designated by the dummy variable X to create the output array field designated by the dummy variable, Y. At the row i, column j (i=0,...,127; j=0,...,127) location of the X field, the function develops the natural log corresponding to the value of the element of X, namely, x(i,j), and places the result, y(i,j), in the same row and column location of the Y field. In a FORTRAN sense, the field function creates

```
y(i,j)=ALOG(x(i,j)   where

i=0,...,127  and
j=0,...,127.
```

On exit from the routine, the 1 bit slice field, O, provides error status information. The bit slice field, O, is set wherever the input, X, was negative.

Where an X element is zero, Y will be loaded with the most negative VAX number possible, namely, Ymin where Ymin=-(1-(2**(-24)))*(2**(+127)).

Where O is clear the output is in range; all non-zero, non-negative, X element values produce in-range Y element values.

The 56 bit temporary field, T=(t0, t1, t2,...., t55) specifies the array memory to be used for scratch purposes during function execution.

FUNCTION MASKING: Unmasked

FIELD OVERLAP: The X, Y, O, and T fields are not permitted to be overlapped.

INPUT VARIABLE FIELD X:

- o Number of bit slices: 32
- o X element number type: REAL

  Each X element is a VAX11/780 single precision floating point number
  Its characteristics follow:

  normalized
  signed magnitude
  exponent biased up by 128
  base is 2
  bit layout (from left to right):

  1 sign bit,
  8 biased exponent bits, and
  24 mantissa bits (including the suppressed most significant mantissa bit).
- o X units: Dimensionless
- o Bit slice notation for X:
  - Arbitrary bit slice designator:

    x0 is the bit slice that holds the leftmost bits of the elements of X.

    x31 is the bit slice that holds the rightmost bits of the elements of X.

    X=(x0, x1, x2, x3,..., x31), i.e., X comprises the concatenation of the 32 individual bit slices.
  - Notation for sign bit slice: SX=(sx0)=(x0)
  - Biased exponent bit slice designator:

    EX=(ex0, ex1,..., ex7)=(x1, x2,..., x8) .

    The exponents of the elements of X are given by EX-128; the base for all elements is 2.

    When X=0, EX=0.
  - Mantissa bit slice designator:

    MX=(mx0, mx1,..., mx7)=(u, x9, x10,..., x31) where u is implicit. At one element of X,

    u=1 when at the element location, at least one bit of X is non-zero and

    u=0 when at the element location, all bits of X are zero.

    When the most significant bit (MSB) slice of the mantissa field, namely, mx0, is stored into MPP array memory (i.e., when the implicit MSB slice, u, is stored), the x8 bit slice is used for storage. Prior to using the ex7 bit slice to store the implicit u slice, the contents of the ex7 bit slice are stored into the t0 bit slice.

OUTPUT VARIABLE FIELD Y:

- o Number of bit slices: 32
- o Y element number type: REAL

  Each Y element is a VAX11/780 single precision floating point number

  Its characteristics follow:

  - normalized
  - signed magnitude
  - exponent biased up by 128
  - base is 2
  - bit layout (from left to right):
    - 1 sign bit,
    - 8 biased exponent bits, and
    - 24 mantissa bits (including the suppressed most significant mantissa bit).
- o Y units: Dimensionless
- o Bit slice notation for Y:
  - Arbitrary bit slice designator:

    y0 is the bit slice that holds the leftmost bits of the elements of Y.

    y31 is the bit slice that holds the rightmost bits of the elements of Y.

    Y=(y0, y1, y2, y3,..., y31), i.e., Y comprises the concatenation of the 32 individual bit slices.
  - Notation for sign bit slice: SY=(sy0)=(y0)
  - Biased exponent bit slice designator:

    EY=(ey0, ey1,..., ey7)=(y1, y2,..., y8) .

    The exponents of the elements of Y are given by EY-128; the base for all elements is 2.

    When Y=0, EY=0.
  - Mantissa bit slice designator:

    MY=(my0, my1,..., my7)=(u, y9, y10,..., y31) where u is implicit. At one element of Y,

    u=1 when at the element location, at least one bit of Y is non-zero and

    u=0 when at the element location, all bits of Y are zero.

    When the most significant bit (MSB) slice of the mantissa field, namely, my0, is stored into MPP array memory (i.e., when the implicit MSB slice, u, is stored), the y8 bit slice is used for storage. Prior to using the ey7 bit slice to store the implicit u slice, the contents of the ey7 bit slice are stored into the t0 bit slice.

OUT-OF-RANGE Y ELEMENTS:

The range of y for the ALOG function is minus infinity to plus infinity. (As will be seen, y values in only a very small fraction of the y range are generated.) The domain of x corresponding to the y range extends from x=0 through x=+infinity.
Negative X element values imply a complex Y element output. O is loaded with (1,1) in such cases. Where an X element value is zero, a Y element value of minus infinity is implied; in such case, the O field is loaded with (0,1) and the Y element is loaded with

Ymin=-(1-(2**(-24)))*(2**(+127)) .

The next X element larger than zero that can be expressed using the specified number form is

Xsmall=(1+(2**(-23)))*(2**(-129)) ;

it causes the smallest possible signed Y element value, Ysmall, that is derived from an X element value. It is given by

Ysmall=LN(Xsmall)=-89.4159861839 .

The very largest X element value, Xlarge, causes the largest possible signed Y element. Specifically,

Ylarge=LN(Xlarge)=+88.0296918823 where
Xlarge=(1-(2**(-24)))*(2**(+127)) .

For all X element values in the domain extending from Xsmall through Xlarge, O will be loaded with 0.

ALGORITHM DEVELOPMENT:

For each element of X (denoted x), y=ALOG(x) must be computed for all in-range x values. All positive x values are considered in-range; however, the case of x=0 must be treated specially.
The issue of computing y will be addressed first. An in-range x will be assumed. Then the issue of determining whether or not x is in-range and the special case of x=0 will be addressed.

The starting expression is

1) y=LN(x) where LN(x) is the natural log of x.

The inverse form of 1) is

2) x=e**y where e=2.718281828 .

Now e can be expressed as

3) e=2**a where a=1/LN(2)=1.4426950407

Using 3) in 2),

4) x=2**(a*y) .

Now, taking the logarithm of x, base 2 (i.e., LOG2(x)), of 4),

5) y=(LN(2))*LOG2(x)

The independent input variable x is a floating point number and so is expressed as

6) x=S*(f*(2**N)) where

f is a fraction having a value less than 1 but greater than or equal to 0.5 that has the number form (0.0.24),
N is an integer having a value less than 8 but greater than or equal to -128 that has the number form (1.7.0), and
S is +1 if x is positive and is -1 if x is negative.

But x is positive only (S=+1), and so

7) x=(f*(2**N)) .

Using 7) in 5),

8) y=(LN(2))*LOG2(f*(2**N)) which reduces to

9) y=(LN(2))*(N-1+LOG2(2*f))

Let

10) g=2*f where

gMIN=1 and
gMAX=2*(1-(2**(-24))) .

Using 10) in 9),

11) y=(LN(2))*(N-1+LOG2(g)) or,

alternatively, as

12) y=((LN(2))*N)+zz where

zz=((LN(2))*LOG2(g))-LN(2).

The primary task needed to be performed to compute y is that of generating LOG2(g). Let

13) z=LOG2(g) where

zMIN=0 and
zMAX=1+LOG2(1-(2**(-24))) (if no guard bits are used).

But g can be expressed as the product

14) g=a0*a1*a2*a3*....*aM*....*a24*a25*a26*a27*.... where,

in BINARY,

```
a0=1,10.
a1=1, 1.1
a2=1, 1.01
a3=1, 1.001
  .
  .
  .
aM=1, 1+(2**(-M))                         ( for all M )
  .
  .
  .
a24=1, 1.00000000000000000000000001
a25=1, 1.000000000000000000000000001
a26=1, 1.0000000000000000000000000001
a27=1, 1.00000000000000000000000000001
  .
  .
  .
  .
```

Each "a" value can assume the value of 1 or the non-one value (to the right of the comma in the list above). Either "a" value can be written as 2 to some power. In particular,

15) aM=2**uM .

As a result, 14) can be written as

16) g=(2**u0)*(2**u1)*(2**u2)*...*(2**uM)*...*(2**u24)*(2**u25)*(2**u26)*...

or as

17) g=2**(u0+u1+u2+u3+....+uM+....+u24+u25+u26+u27+....)

Substituting 17) into 13) shows that

18) z=u0+u1+u2+u3+....+uM+....+u24+u25+u26+u27+....

When aM=1, uM=0. The list of non-zero "u" values (corresponding to "a" values not equal to 1) is provided in Table 2.1.1 .

An iterative approach is used to find z. Assume that iteration (M-1) has just taken place; an (M-1) iteration "g" value, g[M-1], as well as an (M-1) iteration "z" value, z[M-1], have just been developed. To accomplish the next iteration M values, g[M] and z[M], the following expressions are used:

```
19)    g[M]=g[M-1]*aM   where

            g[M]=g[M-1]+SHFT(g[M-1],-M) and TEST=1  when  aM*g[M-1]<g  or
                                                    when  aM*g[M-1]=g  ; else,
            g[M]=g[M-1]                 and TEST=0  when  aM*g[M-1]>g  .
```

Also,

```
20)    z[M]=z[M-1]+uM   where

           uM=LN(aM)/LN(2)            when  TEST=1     ; else,
           uM=0                       when  TEST=0     .
```

The iterations begin at M=1. For the 1st iteration,

g[0]=1 and z[0]=0 .

Using the expressions 19) and 20), z can be determined to any level of precision. Using the z determined in 13) and then in 11), y=ALOG(x) is determined.

A slightly more efficient way to compute y results by multiplying the uM values with LN(2) prior to performing the iteration operations. Substituting z from 13) into the auxiliary expression of 12) yields

21) zz=LN(2)*z-LN(2) .

Using 18) in 21) yields,

22) zz=(LN(2))*(u0+u1+u2+u3+....+uM+....+u24+u25+u26+u27+....)-LN(2) .

Let
23) vM=(LN(2))*(uM) for all M. Using 23) in 22),

24) $zz=(v0+v1+v2+v3+....+vM+....+v24+v25+v26+v27+....)-LN(2)$.

(See Table 2.1.1 for vM values.)

As for z, an iterative approach can be used to find zz. Assume that iteration (M-1) has just taken place; an (M-1) iteration "g" value, g[M-1], as well as an (M-1) iteration "zz" value, zz[M-1], have just been developed. To accomplish the next iteration M values, g[M] and zz[M], the following expressions are used:

25) $g[M]=g[M-1]*aM$ where

$g[M]=g[M-1]+SHFT(g[M-1],-M)$ and $TEST=1$ when $aM*g[M-1]<g$ or
when $aM*g[M-1]=g$; else,
$g[M]=g[M-1]$ and $TEST=0$ when $aM*g[M-1]>g$.

Also,

26) $zz[M]=zz[M-1]+vM$ where

$vM=LN(aM)$ when $TEST=1$
else; $vM=0$ when $TEST=0$.

The iterations begin at M=1. For the 1st iteration,

$g[0]=1$ and
$zz[0]=-LN(2)$.

Using the zz[M] of 26) for zz in 12) permits y to be determined using an economical common times a narrow 8 bit field multiply.

Table 2.1.1 - Values Of uM And vM Corresponding To Different aM Values

M	aM=1+(.5**M)	uM=LOG2(aM)	vM=LN(2)*uM
0	2	.99999999999	.693147180644
1	1.5	.584962500792	.405465108211
2	1.25	.321928094818	.22314355 1296
3	1.125	.169925001262	.117783035547
4	1.0625	.087462841164	.060624621765
5	1.03125	.044394119491	.030771658763
6	1.015625	.022367812829	.0155041864
7	1.0078125	.011227255583	.00778214055402
8	1.00390625	.0056245491 2855	.00389864037089
9	1.001953125	.00281501548714	.00195122 00484
10	1.0009765625	.00140819422001	.00097608585341
11	1.0004882812	.00070426868758	.00048816185522
12	1.0002441406	.000352177 19666	.00024411063096
13	1.0001220703	.00017609939459	.00012206279888
14	1.0000610351	.0000880523548428	.000061033241509
15	1.0000305175	.0000440268 41807	.0000305170812715
16	1.0000152587	.0000220134209034	.0000152585406357
17	1.0000076293	.0000110068765481	.0000076293854471
18	1.0000038146	.0000055034382739	.0000038146927235
19	1.0000019073	.0000027515530405	.0000019072312325
20	1.0000009536	.0000013756104239	9.53500487011E-7
21	1.0000004768	6.87639115553E-7	4.76635114251E-7
22	1.0000002384	3.44151750584E-7	2.38547815634E-7
23	1.0000001192	1.72075875292E-7	1.19273907817E-7
24	1.0000000596	8.6037937645E-8	5.9636953908E-8
25	1.0000000298	4.2852872417E-8	2.9703347699E-8
26	1.0000000149	2.1592532613E-8	1.4966803104E-8
27	1.000000074	1.0630169902E-8	7.3682722976E-9
28	1.0000000037	4.98289214169E-9	3.4538776395E-9
29	1.0000000018	2.32534966612E-9	1.6118095651E-9
30	1.0000000009	1.32877123778E-9	9.210340372E-10
31	1.0000000004	6.6438561889E-10	4.605170186E-10
32	1.0000000002	0	0

2.1.2 EXPONENTIAL ARRAY SUBROUTINE

DESCRIPTION: The MPP array field function, EXPA(X,Y,O,T), uses the input array field designated by the dummy variable X to create the output array field designated by the dummy variable, Y. At the row i, column j (i=0,...,127; j=0,...,127) location of the X field, the function exponentiates the value of the element of X, namely, x(i,j), and places the result, y(i,j), in the same row and column location of the Y field. In a FORTRAN sense, the field function creates

```
y(i,j)=EXP(x(i,j))   where

           i=0,...,127  and
           j=0,...,127.
```

On exit from the routine, the field, O=(o0, o1 o2), provides out-of-range status information as follows:

o0 - set where the output Y was declared equal to VAX '1' because the input X was less than 2**-31, or nearly zero.

o1 - set where the output was too small to be represented in VAX format because the input X was less than -2**7; the output Y will be cleared to all 0's for this case.

o2 - set where overflow has occurred because the input X was greater than 2**7; the maximun VAX number will be inserted in Y.

The 128 bit temporary field, T=(t0, t1, t2,...., t127), specifies the array memory to be used for scratch purposes during function execution.

FUNCTION MASKING: Unmasked

FIELD OVERLAP: The X, Y, O, and T fields are not permitted to be overlapped.

INPUT VARIABLE FIELD X:
- o Number of bit slices: 32
- o X element number type: REAL
  Each X element is a VAX11/780 single precision floating point number
  Its characteristics follow:
  - normalized
  - signed magnitude

exponent biased up by 128
base is 2
bit layout (from left to right):
1 sign bit,
8 biased exponent bits, and
24 mantissa bits (including the suppressed most significant mantissa bit).

- o X units: Dimensionless
- o Bit slice notation for X:
  - Arbitrary bit slice designator:
    - x0 is the bit slice that holds the leftmost bits of the elements of X.
    - x31 is the bit slice that holds the rightmost bits of the elements of X.
    - X=(x0, x1, x2, x3,..., x31), i.e., X comprises the concatenation of the 32 individual bit slices.
  - Notation for sign bit slice: SX=(sx0)=(x0)
  - Biased exponent bit slice designator:
    - EX=(ex0, ex1,..., ex7)=(x1, x2,..., x8) .
      The exponents of the elements of X are given by EX-128; the base for all elements is 2.
      When X=0, EX=0.
  - Mantissa bit slice designator:
    - MX=(mx0, mx1,..., mx7)=(u, x9, x10,..., x31) where u is implicit. At one element of X,

      u=1 when at the element location, at least one bit of X is non-zero and
      u=0 when at the element location, all bits of X are zero.

When the most significant bit (MSB) slice of the mantissa field, namely, mx0, is stored into MPP array memory (i.e., when the implicit MSB slice, u, is stored), the x8 bit slice is used for storage. Prior to using the ex7 bit slice to store the implicit u slice, the contents of the ex7 bit slice are stored into the t0 bit slice.

OUTPUT VARIABLE FIELD Y:

- o Number of bit slices: 32
- o Y element number type: REAL
  Each Y element is a VAX11/780 single precision floating point number
  Its characteristics follow:
  normalized
  signed magnitude
  exponent biased up by 128
  base is 2
  bit layout (from left to right):

1 sign bit,
8 biased exponent bits, and
24 mantissa bits (including the suppressed most significant mantissa bit).

- o Y units: Dimensionless
- o Bit slice notation for Y:
  - Arbitrary bit slice designator:
    y0 is the bit slice that holds the leftmost bits of the elements of Y.
    y31 is the bit slice that holds the rightmost bits of the elements of Y.
    Y=(y0, y1, y2, y3,..., y31), i.e., Y comprises the concatenation of the 32 individual bit slices.
  - Notation for sign bit slice: SY=(sy0)=(y0)
  - Biased exponent bit slice designator:
    EY=(ey0, ey1,..., ey7)=(y1, y2,..., y8) .
    The exponents of the elements of Y are given by EY-128; the base for all elements is 2.
    When Y=0, EY=0.
  - Mantissa bit slice designator:
    MY=(my0, my1,..., my7)=(u, y9, y10,..., y31) where
    u is implicit. At one element of Y,

    u=1 when at the element location, at least one bit of Y is non-zero and
    u=0 when at the element location, all bits of Y are zero.

    When the most significant bit (MSB) slice of the mantissa field, namely, my0, is stored into MPP array memory (i.e., when the implicit MSB slice, u, is stored), the y8 bit slice is used for storage. Prior to using the ey7 bit slice to store the implicit u slice, the contents of the ey7 bit slice are stored into the t0 bit slice.

OUT-OF-RANGE Y ELEMENTS:

Assuming real values for the X elements the complete range of y for the EXPA function is 0 through plus infinity. Only a countable number of X and Y element can be represented by the specified floating point numbers. In particular, the smallest magnitude non-zero Y element value that can be represented using the VAX floating point form will be designated Ysmall; the largest Y element value that can be represented using the VAX floating point form will be designated Ylarge. Ysmall and Ylarge are given by

Ysmall=(1+(2**(-23)))*(2**(-129)) and
Ylarge=(1-(2**(-24)))*(2**(+127)) ,respectively.

Y element values that lie between 0 and Ysmall must be described as

0 or Ysmall. It is reasonable to assert that a Y element with a value that lies in the upper half of the 0 to Ysmall interval, i.e., between Ymin=(1+(2**(-24)))*(2**(-129)) and Ysmall will be assigned the value Ysmall. A Y element value smaller than Ymin (in the lower half of the 0 to Ysmall interval) will be set to 0 and the o1 bit slice will be set (i.e., O=(0,1)). The X element value corresponding to Ymin is

Xmin=LN(Ymin)=-89.4159862435.

Y element values that lie above Ylarge but are smaller than the value Ymax=(1-(2**(-25)))*(2**(+127)) will be assigned the value Ylarge. Where Y element values lie above Ymax, the overflow out-of-range bit will be set in the o0 bit slice (i.e., O=(1,0)). The X element value corresponding to Ymax is

Xmax=LN(Ymax)=+88.029691912

For Y element values that lie from Ymin through Ymax, O=(0,0).

ALGORITHM DEVELOPMENT:

For each element of X (denoted x), y=EXP(x) must be computed for all in-range x values. In-range x values are larger than or equal to Xmin and are smaller than or equal to Xmax .
The issue of computing y will be addressed first. An in-range x will be assumed. Then the issue of determining whether or not x is in-range will be addressed.

The starting expression is

1) y=e**x where e=2.718281828

Now e can be expressed as

2) e=2**a where a=1/LN(2)=1.4426950407

Using 2) in 1),

3) y=2**(a*x) .

The independent input variable x is a floating point number and so is expressed as

4) x=S*(f*(2**N)) where

f is a fraction having a value less than 1 but greater than or equal to 0.5 that has the number form (0.0.24),
N is an integer having a value less than 8 but greater than or equal to -128 that has the number form (1.7.0), and

S is +1 if x is positive and is -1 if x is negative.

Using 4) in 3),

5) y=2**(S*(a*f)*(2**N))

Let

6) b=a*f where

bMAX= 1.4426950407
bMIN= .72134752035

Using 6) in 5),

7) y=2**(S*b*(2**N))

or since (2**N) simply causes a shift of the radix point of b,

8) y=2**(S*SHFT(b,N)) where

SHFT(b,N) implies a shift of the radix point of b equal to the magnitude of N, to the right if the sign of N is + and to the left if the sign of N is -.

****

As an aside, at the low end of the range of X (Xmin=-89.4159862435),

9) b=LN(Ymin)/(128*LN(2))=+1.0078124994,
N=7, and
S=-1.

At the top end of the range of X (Xmax=+88.029691912),

10) b=LN(Ymax)/(128*LN(2))=+.992187499663,
N=7, and
S=+1.

****

Once SHFT(b,N) has been developed, it can be written as the sum of an integer part, I, (+ or -) and an always + fractional part, g, i.e., as

11) I+g=SHFT(b,N)

Then, using 11) in 8),

12) y=2**(I+g)=(2**I)*(2**g)=(2**(I+1))*((2**g)/2) .

The integer variable (I+1) is the unbiased exponent of the output y value. The mantissa of the output is given by ((2**g)/2). Thus, the primary task to be performed is that of generating 2**g. Let

13) z=2**g where

gMIN=0 and
gMAX=1-(2**(-24)) (if no guard bits are used).

But z can be expressed as the product

14) z=a0*a1*a2*a3*....*aM*....*a24*a25*a26*a27*.... where,

in BINARY,

```
a0=1,10.
a1=1, 1.1
a2=1, 1.01
a3=1, 1.001
  .
  .
  .
aM=1, 1+(2**(-M))                          ( for all M )
  .
  .
  .
a24=1, 1.000000000000000000000001
a25=1, 1.0000000000000000000000001
a26=1, 1.00000000000000000000000001
a27=1, 1.000000000000000000000000001
  .
  .
  .
  .
```

Each "a" value can assume the value of 1 or the non-one value (to the right of the comma in the list above). Either "a" value can be written as 2 to some power. In particular,

15) aM=2**uM .

As a result, 14) can be written as

16) z=(2**u0)*(2**u1)*(2**u2)*...*(2**uM)*...*(2**u24)*(2**u25)*(2**u26)*...

or as

```
17)    z=2**(u0+u1+u2+u3+....+uM+....+u24+u25+u26+u27+....)
```

Comparing 17) to 13) shows that

```
18)    g=u0+u1+u2+u3+....+uM+....+u24+u25+u26+u27+....
```

When aM=1, uM=0. The list of non-zero "u" values (corresponding to "a" values not equal to 1) is provided in Table 2.1.2 .

An iterative approach is used to find z. Assume that iteration (M-1) has just taken place; an (M-1) iteration "g" value, g[M-1], as well as an (M-1) iteration "z" value, z[M-1], have just been developed.
To accomplish the next iteration M values, g[M] and z[M], the following expressions are used:

```
19)    g[M]=g[M-1]-uM   where

            uM=0                when  uM>g[M-1]  ; else,
            uM=LN(aM)/LN(2)     when  uM<g[M-1]  or
                                when  uM=g[M-1]

       and

20)    z[M]=z[M-1]*aM  or

           =z[M-1]                when  uM=0       ; else,
           =z[M-1]+SHFT(z[M-1],-M)  when  uM=g[M-1]    .
```

The iterations begin at M=1. For the 1st iteration,

```
       g[0]=g    (the g of expression 13))  and
       z[0]=1 .
```

Using the expressions 19) and 20), z can be determined to any level of precision. By dividing z by 2 (by shifting the radix point of z left by 1 bit position), the mantissa of y=EXP(x) is determined.

Table 2.1.2 - Values Of uM Corresponding To Different aM Values

M	aM=1+(.5**M)	uM=LOG2(aM)
0	2	1.00000000000
1	1.5	.584962500792
2	1.25	.321928094818
3	1.125	.169925001262
4	1.0625	.087462841164
5	1.03125	.044394119491
6	1.015625	.022367812829
7	1.0078125	.011227255583
8	1.00390625	.0056245491285 5
9	1.001953125	.0028150154871 4
10	1.0009765625	.0014081942200 1
11	1.000488281 2	.00070426868758
12	1.0002441406	.00035217719666
13	1.0001220703	.00017609939459
14	1.0000610351	.000088052354842 8
15	1.0000305175	.000044026841807
16	1.0000152587	.0000220134209034
17	1.0000076293	.0000110068765481
18	1.0000038146	.0000055034382739
19	1.0000019073	.0000027515530405
20	1.0000009536	.0000013756104239
21	1.0000004768	6.87639115553E-7
22	1.0000002384	3.44151750584E-7
23	1.0000001192	1.72075875292E-7
24	1.0000000596	8.60379376 45E-8
25	1.0000000298	4.2852872417E-8
26	1.0000000149	2.1592532613E-8
27	1.0000000074	1.0630169902E-8
28	1.0000000037	4.98289214169E-9
29	1.0000000018	2.32534966612E-9
30	1.0000000009	1.32877123778E-9
31	1.0000000004	6.6438561889E-10
32	1.0000000002	0

2.1.3 SQUARE ROOT ARRAY SUBROUTINE

DESCRIPTION :

SQRTV is a PECU routine that computes the square root of an array (X) and places the result in another array (Q). The entry point is SQRTV$. Arrays X and Q each contain 32-bit floating-point numbers in VAX-F format. The routine requires a 22-plane array for temporary storage (T). No error occurs as long as X is non-negative. The sign of X is stored in an error bit plane (E).

If an element of X is positive then its value is:

```
(0.5 + x2/4 + x3/8 + ... + x24/(2**24)) *
2**(e0 + 2*e1 + 4*e2 + ... + 128*e7 - 128)
```

where x2, x3, ..., x24 are the fraction bits and e0, e1, ..., e7 are the characteristic bits of the element. Similarly, its square root in Q has the value:

```
(0.5 + q2/4 + q3/8 + ... + q24/(2**24)) *
2**(y0 + 2*y1 + 4*y2 + ... + 128*y7 - 128)
```

where q2, q3, ..., q24 are the fraction bits and y0, y1, ..., y7 are the characteristic bits.

If e0 = 1, then the X-fraction should be shifted right once and unity added to the X-characteristic. This puts all X-fractions in the range of 0.25 to 1 and makes all X-exponents even. Then the Q-exponent is half the X-exponent and the Q-fraction is the square root of the X-fraction. When we take into account the characteristic bias of 128 we obtain the following binary addition for the Y-characteristic:

```
       0   e7  e6  e5  e4  e3  e2  e1
   +   0   1   0   0   0   0   0   e0
      --------------------------------
      y7  y6  y5  y4  y3  y2  y1  y0
```

This arithmetic is performed near the end of the routine.

After shifting where e0 = 1, the X-fraction is in the range of 0.25 to 1 so the Q-fraction is in the range of 0.5 to 1 and is automatically normalized. The fraction bits of Q are computed in the order q2, q3, q4, ..., q24.

Bit q2 is 1 if and only if the Q-fraction value is 0.75 or more; that is, if and only if the X-fraction value is 0.5625 or more (0.5625 = 0.75 * 0.75). The initial part of the routine computes q2 by setting the shift register lengths to 26, loading the X-fraction into the shift register, adding 0.5 for the hidden bit, shifting right one place where e0 = 1, and subtracting 0.0625. The result will be 0.5 or more where and only where q2 = 1 so the result bit with weight 0.5 is stored as q2. Where e0 = 0, the binary arithmetic looks like:

```
     0   x2   x3   x4   x5   ...  x23  x24    0
+    0    1    1    1    0   ...    0    0    0
    ------------------------------------------
    r1   r2   r3   r4   r5   ...  r23  r24  r25
```

and where e0 = 1, the binary arithmetic looks like:

```
     0    0   x2   x3   x4   ...  x22  x23  x24
+    0    0    1    1    0   ...    0    0    0
    ------------------------------------------
    r1   r2   r3   r4   r5   ...  r23  r24  r25
```

Bit r1 of the result is stored as q2 and the other result bits are used in the computation of the other Q fraction bits.

To compute bits q3 through q24 the routine enters the main loop (starting at MAIN2). For j = 2, 3, ... assume that bits q2, q3, ..., q(j-1) have been determined and bit qj is now being computed. Let Q(j) equal the value of the Q-fraction if qj is replaced by 1 and qk is replaced by 0 for all k > j. Thus:

$$\begin{aligned} Q(2) &= 1/2 + 1/4 \\ Q(3) &= 1/2 + q2/4 + 1/8 \\ Q(4) &= 1/2 + q2/4 + q3/8 + 1/16 \\ Q(24) &= 1/2 + q2/4 + q3/8 + \dots + q23/(2**23) + 1/(2**24) \end{aligned}$$

Bit qj = 1 if and only the value of the X-fraction is Q(j)*Q(j) or more. For j = 2, 3, ... we define:

$$R(j) = 1/2 + (2**(j-2)) * (X - Q(j)*Q(j))$$

With this definition, bit qj = 1 if and only if R(j) is 1/2 or more. Note that when the routine enters the main loop at MAIN2, the value in the shift register equals R(2). Iteration j-1 of the main loop first stores a bit of R(j) into qj (it stores the bit with weight 1/2). Then it calculates R(j+1) from R(j). From the definition of R(j) we have:

$$\begin{aligned} R(j+1) &= 1/2 + (2**(j-1)) * (X - Q(j+1)*Q(j+1)) \\ 2 * R(j) &= 1 + (2**(j-1)) * (X - Q(j)*Q(j)) \end{aligned}$$

So:

R(j+1) = 2*R(j) - 1/2 + (2**(j-1))*(Q(j)*Q(j) - Q(j+1)*Q(j+1))
R(j+1) = 2*R(j) - 1/2 + (2**(j-1))*(Q(j) - Q(j+1))*(Q(j) + Q(j+1))

But Q(j) - Q(j+1) = (1 - 2*qj)/(2**(j+1)), Q(j) + Q(j+1) = 2*Q(j) + (2*qj - 1)/(2**(j+1)), and (2*qj - 1)*(1 - 2*qj) = -1 so:

R(j+1) = 2*R(j) -1/2 + (1-2*qj)*Q(j)/2 - 1/(2**(j+3))

Where qj = 0, we obtain the following binary addition for R(j+1):

$$\begin{array}{lcccccccccc}
2 * R(j): & r2 & r3 & r4 & r5 & \ldots & rj & r(j+1) & r(j+2) & r(j+3) & r(j+4) \\
+ & 1 & 1 & q2 & q3 & \ldots & q(j-2) & q(j-1) & 0 & 1 & 1 \\
\hline
R(j+1): & r1 & r2 & r3 & r4 & \ldots & r(j-1) & rj & r(j+1) & r(j+2) & r(j+3)
\end{array}$$

and where qj = 1, we obtain the following binary addition for R(j+1):

$$\begin{array}{lcccccccccc}
2 * R(j): & r2 & r3 & r4 & r5 & \ldots & rj & r(j+1) & r(j+2) & r(j+3) & r(j+4) \\
+ & 0 & 0 & \overline{q2} & \overline{q3} & \ldots & \overline{q(j-2)} & \overline{q(j-1)} & 0 & 1 & 1 \\
\hline
R(j+1): & r1 & r2 & r3 & r4 & \ldots & r(j-1) & rj & r(j+1) & r(j+2) & r(j+3)
\end{array}$$

The main loop will recirculate 2*R(j) through the A-register while constructing the addend in the P-register, and performing the addition to get R(j+1) in the B-register and the shift register. To ease the construction of the addend in the P-register, bits t2 through t23 are built up in the 22-plane temporary storage array, where t2 = 1 ⊕ q2 and ti = q(i-1) ⊕ qi for i in the range of 3 to 23. When r1 of the previous addition is moved from the B-register to qj, the P-register contains the complement of q(j-1), so a simple logic operation will form tj in the P-register. Bit tj is stored in the following cycle and P is set to 1 (R(j) is also shifted one place and 0 added with SHIFT A and HALFADD operations). Then R(j) is re-circulated 23-j times with 0's added to bring bit r(j+4) into the A-register. Two cycles of SHIFT A and FULLADD add 1's to bits r(j+4) and r(j+3). The next cycle does a SHIFT A and HALFADD (to add 0 to bit r(j+2)) and loads tj = q(j-1) ⊕ qj into the P-register (this is the correct addend to bit r(j+1)). Then j-2 addition cycles are performed while P is exclusive-or'ed with successive t bits (this puts the correct addend in P whether qj = 0 or qj = 1).

When the main loop is finished (END1), bit q24 is stored and another step performed to see if the round-bit (q25) is 0 or 1. The shift register length is set to 30 and the round-bit is added to the q bits to obtain the final fraction in the shift register. Then the characteristic of the result is computed (described above) and stored in the shift register.

If the X-characteristic equals 0, then X = 0 and the result equals 0. The G-register is cleared wherever X = 0 Then the shift register is stored in the result array where G = 1, and 0's are stored where G = 0.

2.1.4 SINE and COSINE ARRAY SUBROUTINE

DESCRIPTION :

The MCL mnemonics using these functions are SINA, COSA, and SINCOSA. The MCL statement:

```
SINA      ang,sin,[temp]
```

generates an array of sines (sin) from an array of angles (ang). The MCL statement:

```
COSA      ang,cos,[temp]
```

generates an array of cosines (cos) from an array of angles (ang). The MCL statement:

```
SINCOSA   ang,sin,cos,[temp]
```

generates an array of sines (sin) and an array of cosines (cos) from an array of angles (ang). Arrays ang, sin, and cos are in the 32-bit VAX-F floating-point format. Angles are in radians. The optional parameter, temp, is used to specify the location of a 90-plane array for temporary storage. If temp is not specified then planes 884 through 973 are used.

Each mnemonic first calls a PECU routine (VFSC1$) to convert the angle array to a fixed-point format. VFSC1$ adjusts the angles to lie in the first quadrant (0 to 90 degrees), leaves them in the planar shift registers, and initializes three planes, Z, COSSGN, and SINSGN, in temporary array storage.

Then each mnemonic calls a PECU routine, VFSC2$, twelve times to adjust the angles and leave them close to 45 degrees.

Then VFSC3$ is called to generate the sine and cosine of the angles close to 45 degrees and leave them in SNA and CS, respectively.

Then VFSC4$ is called twelve times to adjust the sine and cosine arrays until they are fixed-point versions of the desired results.

Finally, each mnemonic calls VFSC5$ to float the results and store them in the result array(s), sin and/or cos. SINCOSA makes two calls to float both the sine and cosine arrays. SINA and COSA each make only one call to float the desired result.

TEMPORARY ARRAY STORAGE

These functions use 90 planes of temporary storage (planes 884 through 973 as a default). Its layout is shown below:

Z	COS SGN	CS	SIN SGN	SNA	SNB
1	1	29	1	29	29

The COSSGN plane contains the sign of the cosine while the SINSGN plane contains the sign of the sine. The CS array contains the fixed-point version of the cosine magnitude; the MSB of CS has a weight of unity and the LSB of CS has a weight of 2**(-28). The SNA array contains the fixed-point version of the sine magnitude with the same scaling as CS. The SNB array is an alternate copy of SNA. The Z-plane shows where the angle sense is reversed - the value of Z depends on the quadrant containing the angle as shown below:

Quadrant	Degrees	Z
First	0 to 90	0
Second	90 to 180	1
Third	180 to 270	0
Fourth	270 to 360	1

VFSC1$ ROUTINE

The VFSC1$ PECU routine converts a 32-bit VAX-F array of angles to fixed-point and leaves the result in the planar shift registers. The routine also initializes Z, COSSGN, and SINSGN.

The input angle array is in radians. If the angle magnitude is larger than a revolution (2*pi radians), the routine should subtract off any integral multiple of revolutions to leave an angle with a magnitude less than a revolution. The easiest way to accomplish this is to divide the angle by 2*pi and only treat the fractional part of the quotient. Another advantage of this scaling is that the left-most pair of bits of the fractional part show which quadrant contains the angle.

The division by 2*pi is accomplished by multiplying the angle by 1/(2*pi). Let the angle be:

$$\pm\ 2^{E-128} * F$$

where E is the 8-bit characteristic of the angle and F is the 23-bit fraction of the angle (plus 0.5 for the hidden bit). Since 1/(2*pi) = 0.25 * 2/pi the product is:

$$\pm 2^{E-130} * (F * 2/pi)$$

We multiply the fraction, F, by 2/pi. Since F < 1, the product is less than 2/pi = 0.6366 and the MSB of the product has a weight of 0.5 revolutions. Where E < 130 the product is shifted right 130-E places with zero bits inserted at the left end. Where E > 130 the product is shifted left E-130 places while discarding all bits shifted off the left end. Where E = 130 the product is not shifted.

The constant, 2/pi, equals 0.1010 0010 1111 1001 1000 0011 0111 in binary with a relative error of 1/1613825000. If we allow negative binary digits (1̶ meaning -1), the constant can be written as 0.1010 0101̶ 0000 1̶010 1̶000 0100 1̶001̶. Thus, F * 2/pi can be obtained with 10 additions and subtractions of F shifted appropriately:

$$F * 2/pi = F/2 + F/8 + F/64 - F/256 - F/8192 + F/32768 - F/131072$$
$$+ F/(2^{**}22) - F/(2^{**}25) - F/(2^{**}28)$$

The product, F * 2/pi, is computed with a precision of 31 bits; the MSB has a weight of 0.5 revolution and the LSB has a weight of 2**(-31) revolutions. The product is left in register A, register B, and the 30-bit-long planar shift register.

If E < 99 then the product should be right-shifted 32 or more places and the result will be zero. This will occur for any angle magnitude less than 2**(-30) radians.

If E > 161 then the product should be left-shifted 32 or more places. All fraction bits of the product will be shifted off the left end to leave a result of zero. This will occur for any angle magnitude greater than 2**33 radians. Such large angles should never occur in any reasonable application. If such a large angle does occur, the weight of its LSB is at least 1024 radians so all significance is lost and an angle of zero is just as good as any other angle.

The angle characteristic, E, is used to compute a six-bit shift constant, S. Where S = 32 the product is not shifted. Where S > 32 the product is shifted left S-32 places. Where S < 32 the product is shifted right 32-S places. Let Ei be the bit of E with weight 2**i and let Si be the bit of S with weight 2**i. The shift constant, S, is computed with the following binary addition:

```
E7  E6  E5  E4  E3  E2  E1  E0
 0  E7   0   1   1   1   1   0
------------------------------
T7  T6  T5  T4  T3  T2  T1  T0
```

For i = 0, 1, 2, 3, 4, and 5, bit Si of S equals the logical-and of T7 and Ti.

Note that where E = 98 the addition will produce a sum of 128, bits T5 through T0 will all be zero, and S will be zero. Where E < 98 the addition will produce a sum less than 128, bit T7 will be zero and S will be zero. Where E > 161 the sum will overflow its limit of 255, bit T7 will be zero and S will be zero. In all these cases S = 0 so the product, F * 2/pi, is right-shifted 32 places to produce a zero angle.

Where 98 < E < 128, the addition will produce a sum of E+30, T7 will be 1 and S will equal E-98 so the product, F * 2/pi, is right shifted 32-S = 130-E places. Where E = 128 or 129, the addition will produce a sum of E+94, T7 will be 1, S will equal E-98, and the product, F * 2/pi, will be right-shifted 32-S = 130-E places. Where 129 < E < 162, the addition will produce a sum of E+94, T7 will be 1, S will equal E-98, and the product, F * 2/pi, will be left-shifted S-32 = E-130 places.

First, VFSC1$ computes the shift constant S and stores it in six bits of the temporary arrays. Then the product, F * 2/pi, is computed. Then the product is right or left shifted depending on S.

When the routine computes the product, F * 2/pi, it leaves the MSB of the product in the B-plane, the LSB at the end of the planar shift register, and clears the A-plane to clear the bit to the right of the LSB. The routine then shifts the products left and right depending on the shift constant, S, to leave the bit with weight-pi (180 degrees) at the end of the planar shift register, the bit with 90-degree weight in the A-plane, and the bit with 45-degree weight in the B-plane. This operation has three phases: pre-rotation, clearing, and post-rotation. Pre-rotation aligns the products so the subsequent clearing phase clears the correct bits. Where S > 32, the clearing phase clears the leftmost S-32 bits of the product and where S < 32, the clearing phase clears the rightmost 32-S bits of the product. Post-rotation performs the final alignment of the product.

Where S < 32 (S5 = 0), the routine pre-rotates the products right 31 places (equivalent to a left rotation of one place), clears 32-S bits by shifting the products right while clearing the A-plane, and then post-rotates the products 31 places (equivalent to another left rotation of one place). Where S5 = 1 (S > 31), the routine pre-rotates the products right 63-S places (equivalent to a left rotation of S-31 places), clears S-32 bits by shifting the products right while clearing the A-plane, and then post-rotates the products right 63-S places.

The pre-rotation phase and the post-rotation phase are identical. Each rotation phase has five parts. In each part, the G-plane is loaded with a certain mask and then a number of SHIFTM A & HALFADDM instructions are performed to shift the products. The table below shows the mask and number of instructions for each part:

Mask	Number of SHIFTM A & HALFADDM instructions
$\overline{S5} \vee \overline{S4}$	16
$\overline{S5} \vee \overline{S3}$	8
$\overline{S5} \vee \overline{S2}$	4
$\overline{S5} \vee \overline{S1}$	2
$\overline{S5} \vee \overline{S0}$	1

The clearing phase has six parts. In each part a number of SHIFTM, HALFADDM, & CLEARAM instructions are performed with the G-plane loaded with a certain mask as shown in the table below:

Mask	Number of SHIFTM, HALFADDM, & CLEARAM instructions
$\overline{S5}$	1
$\overline{S5} \bullet S4$	16
$\overline{S5} \bullet S3$	8
$\overline{S5} \bullet S2$	4
$\overline{S5} \bullet S1$	2
$\overline{S5} \bullet S0$	1

After the three phases of the alignment, VFSC1$ uses the sign of the angle and the two leftmost bits of the product (the bits with weights 180 degrees and 90 degrees) to initialize Z, COSSGN, and SINSGN. The rest of the product is an angle in the first quadrant and is left in the planar shift registers.

The Z-plane equals the product bit with weight 90 degrees. The COSSGN-plane is the logical exclusive-or of the product bits with weights 180 degrees and 90 degrees. The SINSGN-plane is the logical exclusive-or of the product bit with weight 180 degrees and the sign bit of the angle.

The VFSC1$ routine takes 469 machine cycles to execute.

VFSC2$ ROUTINE

The VFSC2$ routine is called twelve times by the MCU. The call index, N, ranges from 1 to 12. Let A(N) = arctan(2**(-N)); A(1) = arctan(0.5) = 26.565 degrees; A(2) = arctan(0.25) = 14.036 degrees; etc. Each call to VFSC2$ checks the values of the angles in the planar shift registers. Where the value is less than 45 degrees call-N adds A(N) to the angle and where the value is 45 degrees or more call-N subtracts A(N) from the angle.

Initially the angles in the planar shift registers range from 0 degrees to 90 degrees. After call-1 the angles will range from 45 - 26.565 = 18.435 degrees to 45 + 26.565 = 71.565 degrees. After call-N the angles will range from 45 degrees - A(N) to 45 degrees + A(N). After call-12 the angles will range from 44.986 degrees to 45.014 degrees.

Twelve bit-planes in SNB are used to store the sign of the angle adjustment of each call. Bit-plane Fn = 1 where A(N) was subtracted from the angle and Fn = 0 where A(N) was added to the angle.

When the MCU calls VFSC2$ it initializes bits 0 through 31 of the PECU common register depending on the value of A(n). The value in the common register is D(n) defined as follows. For i = 0, 1, ..., 29, let a(n,i) be the bit of A(n) with weight 45 * (2**(-n)) degrees and let d(n,i) be the bit of D(n) put into bit i of the common register. Then d(n,29) = a(n.29) and d(n,i) = a(n,i) @ a(n,i+1) for i = 0, 1, ..., 28. Bits 30 and 31 of the common register are always set to 0. The following table shows how the left half of the common register should be initialized for n = 1, 2, ..., 12:

n	Bits 0-31 of Common Register (hex)
1	DCB0 3C88
2	6835 23B4
3	3CCC C9F0
4	1E74 5F14
5	0F39 E08C
6	079C 6888
7	03CE 13FC
8	01E7 0BD4
9	00F3 85C4
10	0079 C2A0
11	003C E150
12	001E 70A8

Each call to VFSC2$ takes 33 machine cycles. Twelve calls require 396 cycles.

VFSC3$ ROUTINE

VFSC3$ computes the sine and cosine of the angles left in the planar shift registers. These angles range from 44.986 degrees to 45.014 degrees. Let such an angle be X + 45 degrees where -0.014 degrees < X < 0.014 degrees. Then:

sin(X + 45 deg) = sin(X) * cos(45 deg) + cos(X) * sin(45 deg)

cos(X + 45 deg) = cos(X) * cos(45 deg) - sin(X) * sin(45 deg)

But sin(45 deg) = cos(45 deg) = sqrt(0.5) so:

sin(X + 45 deg) = sqrt(0.5) * (cos(X) + sin(X))

cos(X + 45 deg) = sqrt(0.5) * (cos(X) - sin(X))

Since X is so close to zero we can approximate sin(X) with X (in radians) and cos(X) with unity to obtain:

sin(X + 45 deg) = sqrt(0.5) * (1 + X)

cos(X + 45 deg) = sqrt(0.5) * (1 - X)

The error magnitude in these approximations is less than 2.11 * 10**(-8).

When VFSC3$ is called the right-half of the common register should be loaded with 09B7 4EDC in hex. VFSC3$ takes 321 machine cycles to execute.

VFSC4$ ROUTINE

The VFSC4$ routine is called twelve times. Call N causes a rotation of ± A(N). After twelve calls SNA and CS contain fixed-point versions of desired results. The well-known trigonometric identities:

$$\sin(Y + Z) = \sin(Y) * \cos(Z) + \cos(Y) * \sin(Z)$$

$$\cos(Y + Z) = \cos(Y) * \cos(Z) - \sin(Y) * \sin(Z)$$

can be rewritten to obtain:

$$\sin(Y + Z) = \cos(Z) * (\sin(Y) + \cos(Y) * \tan(Z))$$

$$\cos(Y + Z) = \cos(Z) * (\cos(Y) - \sin(Y) * \tan(Z))$$

Let $Z = \pm A(N)$ so $\tan(Z) = \pm 2**(-N)$. Then:

$$\sin(Y + A(N)) = \cos(A(N)) * (\sin(Y) + \cos(Y)/(2**N))$$

$$\sin(Y - A(N)) = \cos(A(N)) * (\sin(Y) - \cos(Y)/(2**N))$$

$$\cos(Y + A(N)) = \cos(A(N)) * (\cos(Y) - \sin(Y)/(2**N))$$

$$\cos(Y - A(N)) = \cos(A(N)) * (\cos(Y) + \sin(Y)/(2**N))$$

If the cos(A(N)) factor in these equations is ignored for the moment then simple shifts and adds or subtracts generate the sine and cosine of Y ± A(N) from the sine and cosine of Y. Twelve steps with N = 1, 2, ..., 12, respectively will generate the sine and cosine of the initial first quadrant angle from sin(X + 45 deg) and cos(X + 45 deg). Let K = cos(A(1)) * cos(A(2)) * ... * cos(A(12)). Rather than multiply by cos(A(N)) in each of the twelve steps VFSC3$ multiplies sin(X + 45 degrees) and cos(X + 45 degrees) by K so after the twelfth call to VFSC4$ the results are correct.

Odd-numbered calls to VFSC4$ (N = 1, 3, 5, ..., 11) use SNA as a source of the sine and put the new sine value in SNB. Even-numbered calls (N = 2, 4, 6, ..., 12) use SNB as a source of the sine and put the new sine value in SNA. All calls to VFSC4$ use CS as a source for the cosine and put the new cosine value back into CS.

Call N to VFSC4$ takes 181 - 2*N machine cycles to execute. The twelve calls to VFSC4$ require a total of 2016 cycles.

VFSC5$ ROUTINE

VFSC5$ floats a fixed-point value in SNA or CS and puts the result in the user's sine or cosine array. The SINCOSA mnemonic calls VFSC5$ twice to float both results. SINA and COSA call VFSC5$ once to float only the desired result. Each call to VFSC5$ takes 133 machine cycles.

TIMING

The SINA and COSA mnemonics each require 3335 machine cycles to execute. Thus sines or cosines can be computed at a rate higher than 49 MOPS.

The SINCOSA mnemonic requires 3468 machine cycles to execute. Thus if one wants both the sines and the cosines the rate is higher than 94 MOPS.

2.1.5 ARCTANGENT ARRAY SUBROUTINE

DESCRIPTION :

The MCL mnemonic ARCTNA computes an array of arctangents from an array of slopes. The form of the mnemonic is:

```
ARCTNA     arctan,slope[,temp]
```

where arctan and slope are arrays of 32-bit VAX-F format floating-point numbers. No restriction is placed on the values in the slope array. The values computed in arctan will be in radians and range from -pi/2 to pi/2. The optional parameter, temp, is an array of 82 bit planes used for temporary storage - if temp is omitted then the routine uses bit planes 892 through 973.

METHOD

Since the sign of y = arctan(x) is the same as the sign of x we ignore the sign until the very end. Thus, we assume that both x and y are non-negative.

Let A(i) = arctan(2**(-i)) for i = 0, 1, 2, ... and let z be any angle. Then:

$$\tan(z - A(i)) = \frac{\tan(z) - \tan(A(i))}{1 + (\tan(z)*\tan(A(i)))}$$

Let tan(z) = N(i)/D(i) for some numbers N(i) and D(i) and let tan(z-A(i)) = N(i+1)/D(i+1). Since tan(A(i)) = 2**(-i) we have:

$$\frac{N(i+1)}{D(i+1)} = \frac{(N(i)/D(i)) - (2**(-i))}{1 + (N(i)/(D(i)*(2**i)))}$$

When we multiply the numerator and the denominator of the right side by D(i) we obtain:

$$\frac{N(i+1)}{D(i+1)} = \frac{N(i) - D(i)/(2**i)}{D(i) + N(i)/(2**i)}$$

We equate the two numerators and equate the two denominators to obtain:

$$N(i+1) = N(i) - D(i)/(2**i)$$

$$D(i+1) = D(i) + N(i)/(2**i)$$

One method to compute $y = \arctan(x)$ is to start with two numbers N(0) and D(0) such that $N(0)/D(0) = x$ and initialize a third number, Y(0), to zero. Then we perform the following iteration step for i = 0, 1, 2, ... ,n:

```
WHERE ((2**i) * N(i) ≥ D(i))
    D(i+1) = D(i) + N(i)/(2**i)
    N(i+1) = N(i) - D(i)/(2**i)
    Y(i+1) = Y(i) + A(i)
ELSEWHERE
    D(i+1) = D(i)
    N(i+1) = N(i)
    Y(i+1) = Y(i)
ENDWHERE
```

For i = 0, 1, 2, ... define the angle Z(i) such that tan(Z(i)) = N(i)/D(i). Then Z(0) = y and Z(0) + Y(0) = y. The first iteration, with i = 0, selects those places where the slope, N(0)/D(0), is unity or higher (that is, where Z(0) is A(0) or higher) and subtracts A(0) from Z(0) to obtain Z(1). In the same places it adds A(0) to Y(0) to obtain Y(1). In all places Y(1) + Z(1) = y and Z(1) is less than A(0). Similarly, iteration i+1 selects those places where the slope, N(i)/D(i), is (2**(-i)) or higher (that is, where Z(i) is A(i) or higher) and subtracts A(i) from Z(i) to obtain Z(i+1). In the same places the iteration adds A(i) to Y(i) to obtain Y(i+1). In all places Y(i+1) + Z(i+1) = y and Z(i+1) is less than A(i). Thus, the final iteration, with i = n, leaves Z(n+1) less than A(n) so Y(n+1) is within A(n) of the desired result, y. Since A(n) is less than (2**(-n)) for any n we have computed y to an accuracy of (2**(-n)).

Let W(i) = (N(i)*N(i)) + (D(i)*D(i)) for i = 0, 1, 2, ..., n+1. Then in the places selected by iteration i+1, W(i+1) = W(i) * (1 + (4**(-i))). In the remaining places W(i+1) = W(i) so in all places, W(i+1) $\le$ W(i) * (1 + (4**(-i))). Thus:

$$W(i+1) \le W(0) * 2 * 1.25 * 1.0625 * \ldots * (1 + (4**(-i)))$$

$$W(i+1) \le W(0) * 2.71182$$

Since D(i+1)*D(i+1) $\le$ W(i+1) we obtain:

$$D(i+1) \le 1.6468 * \mathrm{sqrt}(W(0))$$

Thus, D(i+1) can be bounded if W(0) can be bounded. Also we know that

N(i+1)/D(i+1) < (2**(-i)) so N(i+1) can also be bounded.

The test for the where condition in the above method is not convenient so we modify the method as follows. For i = 0, 1, 2, ... let T(i) = ((2**i)*N(i)) - D(i) so N(i) = (T(i)+D(i))/(2**i). Then T(0) = N(0) - D(0) and the iteration step of the above method can be replaced by the following step which involves T instead of N:

```
WHERE (T(i) ≥ 0)
     D(i+1) = D(i) + (T(i) + D(i))/(4**i)
     T(i+1) = 2*T(i) - D(i+1)
     Y(i+1) = Y(i) + A(i)
ELSEWHERE
     D(i+1) = D(i)
     T(i+1) = 2*T(i) + D(i+1)
     Y(i+1) = Y(i)
ENDWHERE
```

Note that as i gets large the adjustment, D(i+1) - D(i), approaches zero and this iteration step approaches the iteration step for the non-restoring division algorithm. Since N(i+1)/D(i+1) < (2**(-i)) we have -D(i+1) ≤ T(i+1) < D(i+1) so T(i+1) is well-bounded.

The input operand, x, is a floating-point number in VAX-F format. Where x is non-zero we have:

$$x = (F + 0.5) * 2^{E-128}$$

where E is the characteristic in the range of 1 to 255 and F is the 23-bit fraction in the range of 0 to 0.5-(2**(-24)). Where x = 0 we have E = F = 0.

Where E ≥ 129 we have x ≥ 1 so y = arctan(x) ≥ A(0). In these places we let N(0) = F + 0.5 and initialize D(0) to 2**(128-E) so N(0)/D(0) = x. T(0) = N(0) - D(0) is initialized to F + 0.5 - 2**(128-E), a non-negative number, so these places are selected by the first iteration step where i = 0. Where E ≥ 153, x is (2**24) or higher so y is pi/2 - (2**(-24)) or more. When y is close to pi/2 the LSB of the y fraction has a weight of (2**(-23)) so the best value for y is simply pi/2. Thus, where E ≥ 153 we initialize D(0) to 0 and T(0) to F + 0.5 so N(0)/D(0) is infinite and we generate y = pi/2.

Where E ≤ 128, x < 1 so the i = 0 iteration will not select this place. In fact, the first iteration that selects this place is when i = 129-E. Rather than waste the i = 0, 1, 2, ..., 128-E iterations we will start this place at i = 129-E. This means that there is an array of iteration numbers in temporary storage so each place can hold its own iteration number. The iteration number, I, is initialized to 129-E where E ≤ 128 and initialized to 0 where E ≥ 129. Where 1 ≤ E ≤ 128 we initialize D(129-E) to 0.5 and T(129-E) to F so N(129-E) = (T(129-E) + D(129-E)) * (2**(129-E)) = x/2 and N(129-E)/D(129-E) = x.

Where E = 0 we have x = 0. The method assumes the hidden bit equals 1 making x = 2**(-129). It computes y = arctan(x) = 2**(-129) and when the hidden bit is dropped from y it stores zeroes so y = arctan(x) = 0.

Thus, the method is initialized with the following rules:

$$\begin{array}{ll} I = 129-E & \text{where } E \le 128 \\ I = 0 & \text{where } E \ge 129 \end{array}$$

$$\begin{array}{ll} D(I) = 0.5 & \text{where } E \le 128 \\ D(I) = 2**(128-E) & \text{where } 129 \le E \le 152 \\ D(I) = 0 & \text{where } E \ge 153 \end{array}$$

$$T(I) = F + 0.5 - D(I)$$

Where $E \ge 129$ we have N(0) = F + 0.5 < 1 and D(0) = 2**(128-E) $\le$ 0.5 so W(0) < 1.25. Where $E \le 128$ we have N(I) = x/2 < 0.5 and D(I) = 0.5 so W(I) < 0.5. Thus, D can be bounded by 1.6468 * sqrt(W(0)) < 1.8412. D is always non-negative so it can be held in an array whose leftmost bit has a weight of unity. Since $-D \le T < D$, T is a signed quantity whose sign bit has a weight of -2.

We compute y with an accuracy equal to the weight of the LSB of its fraction. We will perform 26 iterations with i = I, I+1, I+2, ..., I+25. This will leave $0 \le Z(I+26) < A(I+25)$. If we add A(I+26) to Y(I+26) the maximum error due to stopping the iteration at i = I+25 will be no more than A(I+26).

The LSB of T and D will have a weight of 2**(-26) so D has 27 places and T has 28 places. This means that the D(i+1) = D(i) + (T(i)+D(i))/(4**i) computation in each iteration step may have an error between -(2**(-27)) and 2**(-27) when I > 0. The minimum value for D(i+1) is 0.5 so the magnitude of the relative error is no more than 2**(-26). The relative error in tan(Z(i+1)) is no more than 2**(-26) so the maximum error in Z(i+1) is less than (2**(-26))*Z(i+1) < (2**(-26))*A(i). This error only occurs where $T(i) \ge 0$, that is, only where A(i) is added to the final result, y. Thus, the contribution of this error to the final error is bounded by (2**(-26))*y.

Another error source is in the summation of the various A(n) terms to form y. This error is minimized by delaying the calculation of y until all iterations have been performed - the Y(n+1) calculation in iteration n is replaced by:

$$\begin{array}{l} F(n-I) = 1 \text{ where } T(n) \ge 0 \\ F(n-I) = 0 \text{ where } T(n) < 0 \end{array}$$

The flag bit F(0) of the first iteration is always 1 and need not be stored.

Flag bits F(1), F(2), ..., F(25) are stored in the temporary array until the end of the routine. As discussed above we always add A(I+26) to y so we assume F(26) = 1 as well. Thus,

$$y = F(0)*A(I) + F(1)*A(I+1) + \ldots + F(26)*A(I+26)$$

The Taylor series expansion for A(N) is:

$$A(N) = 2^{-N} - \frac{2^{-3N}}{3} + \frac{2^{-5N}}{5} - \frac{2^{-7N}}{7} + \frac{2^{-9N}}{9} - \frac{2^{-11N}}{11} + \ldots$$

Let B(N,J) = ((-1)**J) * (2**(-N*(2J+1))) / (2J+1) so A(N) = B(N,0) + B(N,1) + B(N,2) + B(N,3) + ... Then:

$$\begin{aligned} y = {} & F(0)*B(I,0) + F(0)*B(I,1) + F(0)*B(I,2) + \ldots \\ & + F(1)*B(I+1,0) + F(1)*B(I+1,1) + F(1)*B(I+1,2) + \ldots \\ & \ldots \\ & + F(26)*B(I+26,0) + F(26)*B(I+26,1) + F(26)*B(I+26,2) + \ldots \end{aligned}$$

We can change the order of summation to obtain:

$$y = C(I,0) + C(I,1) + C(I,2) + C(I,3) + C(I,4) + C(I,5) + \ldots$$

$$\text{where } C(I,J) = F(0)*B(I,J) + F(1)*B(I+1,J) + \ldots + F(26)*B(I+26,J)$$

We want to compute y as a floating-point number. Where I = 0 we have $pi/4 \le y \le pi/2$ and where $I > 0$ we have $A(I) \le y < A(I-1)$. So $pi/4 \le (2**I)*y < 2$ everywhere. Thus, we first compute (2**I)*y which needs only one normalization step to put it into the range of floating-point fractions (0.5 to 1) and then compute the exponent of y.

Let D(I) = C(I,5) + C(I,6) + C(I,7) + ... so (2**I)*y = (2**I)*(C(I,0) + C(I,1) + C(I,2) + C(I,3) + C(I,4) + D(I)). We compute (2**I)*y as follows:

(1) - Compute (2**(9*I)) * C(I,4) * (4095/4096) and shift it right 2I places.

(2) - Add (2**(7*I)) * C(I,3) * (4095/4096) to the result of (1) and shift the sum right 2I places.

(3) - Add (2**(5*I)) * C(I,2) * (4095/4096) to the result of (2) and shift the sum right 2I places.

(4) - Add (2**(3*I) * C(I,1) * (4095/4096) to the result of (3) and shift the sum right 2I places.

(5) - Multiply the result of (4) by 4096/4095.

(6) - Add (2**I) * D(I) to the result of (5).

(7) - Add (2**I) * C(I,0) to the result of (6) to obtain (2**I) * y.

Let F(0) = 1, F(1) = a, F(2) = b, and F(3) = c. Then (2**(9*I)) * C(I,4) * (4095/4096) equals the following binary fraction:

```
0.000 111 000 111 aaa 000 aaa bbb 000 bbb ...
```

which can be formed in the P-register (LSB first) and entered into the shift register in step (1).

Similarly, -(2**(7*I)) * C(I,3) * (4095/4096) equals the following binary fraction:

```
0.0010010 01a01a0 0ab0ab0 0bc0bc0 ...
```

which can be formed in the P-register and subtracted from the shift register in step (2).

Similarly, (2**(5*I)) * C(I,2) * (4095/4096) is the sum of the following two binary fractions:

```
0.00110 01100 11bb0 0bb00 bbdd0 0dd00 ...
0.00000 00aa0 0aa00 aacc0 0cc00 ccee0 ...
```

where F(4) = d and F(5) = e. Each of these fractions is formed in the P-register (LSB first) and added to the shift register in step (3).

Similarly, -(2**(3*I)) * C(I,1) * (4095/4096) is the sum of the following two binary fractions:

```
0.010 1a1 aba bcb cdc ded efe fgf ghg hih ...
0.000 000 010 1a1 aba bcb cdc ded efe fgf ...
```

where F(6) = f, F(7) = g, F(8) = h, and F(9) = i. Each of these fractions is formed in the P-register (LSB first) and subtracted from the shift register in step (4).

In step (5) we multiply the shift register contents by 4096/4095. But:

$$\frac{4096}{4095} = \frac{4097}{4096} * \frac{(2**24)+1}{2**24} * \frac{2**48}{(2**48)-1}$$

With a relative error of (2**(-48)) we perform the multiplication by adding the shift register to itself shifted right 24 places and then adding the shift register to itself shifted right 12 places.

The magnitude of D(I) is less than (2**(-11*I))/11 so for I $\geq$ 3 we have (2**I)*D(I) < (2**(-30))/11 which can be neglected. Thus, D(I) only has significance for I = 0, 1, and 2. Letting a = F(1) and b = F(2) we can find the following binary fractions:

$$-D(0) = 0.0000\ 1100\ 1010\ 11a1\ 1aa0\ a00\overline{a}\ ba\overline{ba}\ \ldots$$

$$-2D(1) = 0.0000\ 0000\ 0000\ 0100\ 1100\ 111a\ 0\overline{a}0\overline{a}\ \ldots$$

$$-4D(2) = 0.0000\ 0000\ 0000\ 0000\ 0000\ 0001\ 0110\ \ldots$$

In step (6) we select the places where I = 2 and add 4D(2) to the result of (5), then we select the places where I = 1 and add 2D(1) to the result of (5), and then we select the places where I = 0 and add D(0) to the result of (5).

Since (2**I) * C(I,0) is simply 1.abcd efgh ..., step (7) is the addition of the appropriate F(n) values to the corresponding bits of the result of (6) to form (2**I) * y.

The magnitude of the error in this computation of (2**I)*y is less than 13.37 * (2**(-30)). If (2**I)*y is less than unity then the A(I+26) error is at most 16 * (2**(-30)) and the D(i+1) error contribution is at most 16 * (2**(-30)) so the maximum error is 45.37 * (2**(-30)). The round-off error in rounding the answer to the VAX-F format is at most 32 * (2**(-30)) giving a worst case error of 1.21 times the weight of the LSB of the final fraction. If (2**I)*y is unity or greater the weight of the LSB is doubled so the effect of the worst error is smaller.

TEMPORARY STORAGE

The 82-plane temporary storage region has the following layout:

```
                  ------------27---------------14---------------27----------
                   !                    !S            S!                   !
                   !          T         !1            1!         D         !
                   !                    !4             !                   !
    ------14--------------------------------------------------------------
    !F           F  F      F!
    !1           1  1      2!
    !            4  5      5!
    -----------25------------
```

The first 25 planes store F(1) through F(25), respectively. Planes F(15) through F(25) are overlayed by the first 11 bits of the T array - array T is not used to compute F(15) through F(25). Array T is only 27 bits long since there is no need to store its MSB - the complement of its MSB equals F(1), F(2), etc. on successive iterations.

The next fourteen planes store S14, S13, S12, ..., S1, respectively, where Sn = 1 only where $I \geq n$. Thus, S1 flags where I > 0 and S14 flags where I > 13.

The last 27 planes store the D array.

VFATN$ ROUTINE

The VFATN$ routine computes the arctangents of elements of the x array and places the results in corresponding elements of the y array. Both x and y are arrays of 32-bit floating-point numbers in VAX-F format. The routine is called with the following setup:

```
Common Register = FE6A C3DD 6CCD 994E (in hex)
             R3 = LSB of y array
             R4 = MSB of temporary storage
             R5 = R4 + 40 = R6 - 41
             R6 = LSB of temporary storage
             R7 = LSB of x array
```

If the default temporary array is being used then R4 = 892, R5 = 932, and R6 = 973.

The basic parts of the VFATN$ routine are: construct S1 through S14; perform iteration I; perform iterations I+1 through I+14; perform iterations I+15 through I+25; and construct y. These are described in the following sections.

Construct S1 through S14 - The following diagram shows the values in planes S1 through S14 as a function of the initial iteration number I:

I	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
2	1	1	0	0	0	0	0	0	0	0	0	0	0	0
3	1	1	1	0	0	0	0	0	0	0	0	0	0	0
4	1	1	1	1	0	0	0	0	0	0	0	0	0	0
5	1	1	1	1	1	0	0	0	0	0	0	0	0	0
6	1	1	1	1	1	1	0	0	0	0	0	0	0	0
7	1	1	1	1	1	1	1	0	0	0	0	0	0	0
8	1	1	1	1	1	1	1	1	0	0	0	0	0	0
9	1	1	1	1	1	1	1	1	1	0	0	0	0	0
10	1	1	1	1	1	1	1	1	1	1	0	0	0	0
11	1	1	1	1	1	1	1	1	1	1	1	0	0	0
12	1	1	1	1	1	1	1	1	1	1	1	1	0	0
13	1	1	1	1	1	1	1	1	1	1	1	1	1	0
≥14	1	1	1	1	1	1	1	1	1	1	1	1	1	1

First the PE shift registers are set to a length of 14. Thirteen zero planes are shifted into the shift registers as the routine computes S14 in P. Let E be the characteristic of x and let E7, E6, ..., E0 be the bits of E where E7 is the MSB and E0 is the LSB. S14 = 1 where I ≥ 14. Since I = 129 - E, then S14 = 1 where E ≤ 115. The logic equation for S14 is:

$$S14 = \overline{E7}\ (\overline{E6} \vee \overline{E5} \vee \overline{E4} \vee \overline{E3}\ \overline{E2})$$

Then the complement of E0 is shifted into the shift registers. Then eight ones are shifted into the shift registers where S14 v (E3 ⊕ (E2 v E1)) = 1 (where I ≥ 14 or where E mod 16 = 2, 3, ..., 8, 9). Then four ones are shifted into the shift registers where S14 v (E2 ⊕ E1) = 1 (where I ≥ 14 or where E mod 8 = 2, 3, 4, or 5). Then two ones are shifted into the shift registers where S14 v E1 = 1 (where I ≥ 14 or where E mod 4 = 2 or 3). Where E ≤ 129 we shift the shift register into S13 through S1, respectively. Where E > 129 we shift zeros into S13 through S1. This completes the construction of S1 through S14. The complement of S1 is left in the P register for use in the next part. Construction of S1 through S14 requires 49 machine cycles.

<u>Perform Iteration I</u> - The shift register lengths are set to 26. First D(I) is constructed by clearing the shift registers to zeros and setting register B to one. Where S1 = 0 (where E > 128) the shift register is shifted right E - 129 places. The first shift clears the B register. the shift registers.

2.2 HOL Interface Requirements for Array Subroutines

All MPP Array functions require loading the MCU Call Queue registers with the arguments of the function with an MCU program, and then calling a specific PECU Subroutine. This is usually performed by an MCL MACRO, or by some Higher-Order-Language (HOL). This Section describes the minimum requirements for a HOL or MACRO to use the Scientific Functions.

Note that the subroutines ARCTNA, and SQRTA call the PECU directly, and that LNA, EXPA, SINA, COSA AND SINCOS call MCU subroutines that perform iterative PECU calls.

Tables 2.2.1 - 2.2.6 describe HOL interfaces for each Array subroutine.

Table 2.2.1 LNA - Natural Logarithm Subroutine HOL Interface Specifications

Name : LNA (X,Y,E,T)

Description : Compute Y = LN(X)

Global MCU Names* : LN$V

Required Arguments : X - Input Array in 32-bit VAX format
Y - Destination Array in 32-bit VAX format
E - Error Output Status bitplane:
Set if source 'X' was Negative
Clear Otherwise
T - Temporary Storage Array of 56 bitplanes

Required Main Control Queue Registers :

Scalar Queue Registers :

R32 - 'PEMODE' Code: 68*4
R33 - Cleared to '0'
R34 - High Half of Ln(2) = X'B172'
R35 - Low Half of Ln(2) = X'16B9'

PECU Queue Registers :

R36 - N/U
R37 - LSB of 'T' Temporary Array
R38 - N/U
R39 - LSB of 'T' Temporary Array
R40 - LSB of 'E' Error Bitplane
R41 - LSB of 'X' Source Array
R42 - LSB of 'T' Temporary Array
R43 - LSB of 'Y' Destination Array

* Call via : CALL R15,LN$V
After Loading Queue Registers

Table 2.2.2 EXPA - Exponential Subroutine HOL Interface Specifications

Name : EXPA (X,Y,E,T)

Description : Compute Y = e**X

Global MCU Names* : EXP$V

Required Arguments :

X - Input Array in 32-bit VAX format
Y - Destination Array in 32-bit VAX format
E - Error Output Status of 3 bitplanes:

E(0) Set if input < 2**-31; output set to X'40800000' (VAX '1').
E(1) Set if overflow; output set to X'7FFFFFFF' (max VAX value).
E(2) Set if underflow; output set to X'0' (VAX 0).

T - Temporary Storage Array of 43 bitplanes

Required Main Control Queue Registers :

Scalar Queue Registers :

R32 - High Half of 1/ln(2) : X'453F
R33 - Low Half of 1/ln(2) : X'D630'
R34 - N/U
R35 - N/U

PECU Queue Registers :

R36 - N/U
R37 - N/U
R38 - N/U
R39 - LSB of 'T' Temporary Array
R40 - LSB of 'E' Error Array
R41 - LSB of 'X' Source Array
R42 - N/U
R43 - LSB of 'Y' Destination Array

* Call via : CALL R15,EXP$V
After Loading Queue Registers

Table 2.2.3 SQRTA - Square Root Subroutine
HOL Interface Specifications

Name :	SQRTA (X,Y,E,T)
Description :	Compute Y = SQRT(X)
Global PECU Names* :	SQRTV$
Required Arguments :	X - Input Array in 32-bit VAX format Y - Destination Array in 32-bit VAX E - Error bitplane, Set if X was Negative T - Temporary Storage Array of 22 bitplanes

Main Control Queue Registers :

Scalar Queue Registers :

R32 - N/U
R33 - N/U
R34 - N/U
R35 - N/U

PECU Queue Registers :

R36 - N/U
R37 - N/U
R38 - N/U
R39 - N/U
R40 - Error bitplane
R41 - MSB of 'T' Temporary Array
R42 - LSB of 'X' Source Array
R43 - LSB of 'Y' Destination Array

* Call via : LR R44,SQRTV$

Table 2.2.4 SIN, COS - Sine, Cosine Subroutine
HOL Interface Specifications

Name : SINA, COSA (X,Y,T)

Description : Compute Y = SIN(X), COS(X)

Global MCU Names * : SNCS$V

Required Arguments : X - Input Array in 32-bit VAX format
Y - Destination Array: Function in 32-bit VAX
T - Temporary Storage Array of 90 bitplanes

Main Control Queue Registers :

Scalar Queue Registers :

R32 - N/U (Set within SNCS$V)
R33 - N/U
R34 - N/U
R35 - N/U

PECU Queue Registers :

R36 - N/U
R37 - N/U
R38 - LSB of 'Y' Destination Array
R39 - N/U
R40 - '1' for SINE; '2' for COSINE
R41 - LSB of 'X' Source Array
R42 - LSB of Source Exponent (R41-23)
R43 - LSB of Temporary Storage Array 'T'

* Call via : CALL R15,SNCS$V

Table 2.2.5 SINCOS - SineCosine Subroutine HOL Interface Specifications

Name : SINCOS (X,Y,Z,T)

Description : Compute Y = SIN(X), and Z = COS(X)

Global MCU Names* : SNCS$V

Required Arguments : X - Input Array in 32-bit VAX format
Y - Destination Array 'SIN' in 32-bit VAX
Z - Destination Array 'COS' in 32-bit VAX
T - Temporary Storage Array of 90 bitplanes

Main Control Queue Registers :

Scalar Queue Registers :

R32 - N/U (Set within SNCS$V)
R33 - N/U
R34 - N/U
R35 - N/U

PECU Queue Registers :

R36 - N/U
R37 - N/U
R38 - LSB of 'Y' Destination Array
R39 - LSB of 'Z' Destination Array
R40 - '4' denotes Sine and Cosine
R41 - LSB of 'X' Source Array
R42 - LSB of Source Exponent (R41-23)
R43 - LSB of Temporary Storage Array 'T'

* Call via : CALL R15,SNCS$V

Table 2.2.6 ATANA - Arctangent Array Subroutine
HOL Interface Specifications

Name : ARCTNA (X,Y,T)

Description : Compute Y = ARCTAN(X)

Global PECU Names* : ATANV$

Required Arguments : X - Input Array in 32-bit VAX format
Y - Destination Array in 32-bit VAX
T - Temporary Storage Array of 82 bitplanes

Main Control Queue Registers :

Scalar Queue Registers :

R32 - X'FE6A'
R33 - X'C3DD'
R34 - X'6CCD'
R35 - X'994E'

PECU Queue Registers :

R36 - N/U
R37 - N/U
R38 - N/U
R39 - LSB of Destination Array 'Y'
R40 - MSB of Temporary Array 'T'
R41 - LSB-41 of Temporary Array
R42 - LSB of Temporary Array
R43 - LSB of Input Array 'X'

* Call via : LR R44,ATANV$

3.0 SEQUENTIAL MCU ALGORITHMS

The MCU sequential (or serial) algorithms are described in this section. The routines are 'sequential' in that a single 32-bit VAX input generates a single VAX output. Each subroutine requires that input data be loaded into specific MCU registers. Upon completion of each subroutine, the output function and error status will also be contained in specific MCU registers. The specific form of the 32-bit VAX real is described in Section 3.3.

3.1 GENERAL DESCRIPTION OF THE POLYNOMIAL METHOD

The iterative algorithms employed to implement the array function modules are most efficient for processors that have no hardware multiplier resources. Because the MCU has an embedded 16 bit hardware multiplier as well as a hardware adder resource, an algorithm that makes effective use of these resources is used in place of the iterative algorithms. The algorithm used is the familiar Discrete Orthonormal Legendre (DOL) polynomial fitting algorithm. For each function, the valid domain of the independent variable is segmented into connected intervals. Within each interval, a polynomial that approximates the function to a specified level of precision (or accuracy) is computed.

The form of the polynomial used to match the scalar function f(u) over an u-interval is:

1) p(u)=A0+u*(A1+u*(A2+u*(A3+u*(A4+u*(A5+u*(A6+...+u*(AN)...))))))
   where
   - the A[j] values (j=0,1,2,...,N) are coefficients to be found while trying to force p(u) to approximate the true function, f(u), over the interval u[k] <= u < u[k+1], k=0,1,...,K ,
   - u represents the independent input variable,
   - k identifies the "k"th interval of the domain of u,
   - K+1 specifies the total number of intervals that comprise the u domain, and
   - p(u) is the polynomial function.

Generally, the degree, "N", of the polynomial (and the number of operations required to evaluate the polynomial) needed to approximate the function f(u) to a given level of accuracy is reduced as K (the number of domain intervals) is increased. It follows that K should be large in order to decrease the number of operations (and execution time) required to compute "p". Within certain "reasonable" K intervals, the statement above is true. However, if K is too large, too many branching operations are required to identify the particular interval in which an input "u" lies. Also, as K increases, the amount of storage required for the "A" coefficients of all intervals tends to increase even though the storage requirements for any one given interval tends to decrease.

To implement the scalar function modules, K values as small as 2 and as large as 14 have been used. In general, the degree of approximating polynomials has been low when K is high. In no case has the degree of the polynomial been allowed to exceed 7. Execution times for modules utilizing small K values can be speeded up by factors of 2:1 simply by increasing K. (In such case, more memory would have to be expended for the modules.)

The input and output variable values for each of the scalar functions are 32 bit VAX real (floating point) values. Usually the "u" value used in the polynomial of Eq* 1* is a biased or scaled and biased version of the mantissa of the input real variable. The implemented version of the Eq* 1* polynomial (POLY32) uses an input "u" that is a signed magnitude fraction that has the magnitude form (0.0.32). (Note: The number form (s.i.f) describes the number of sign bits, s, the number of integer bits, i, and the number of fractional bits, f.) Two MCU registers are used to store the "u" magnitude; an additional 16 bit register is used to store the sign of "u". (When the sign register contains hex 8000, "u" is positive. If the register is loaded with a "0", "u" is negative. No other values are valid "u" sign indicators.) The permitted "u" magnitude can range from "0" to less than 1.

The A[j] coefficients of the polynomial function are 2's complement numbers of the form (1.32.0)*(2**(-32)). Thus, they may take on any value that lies in the interval (-.5) <= A[j] < (+.5) .

The "p" value generated by the POLY32 routine is a 2's complement number of the same form as A[j], namely, (1.32.0)*(2**(-32)). It may also take on any value that lies in the interval (-.5) <= p < (+.5) .

To evaluate functions using POLY32, Eq* 1* is evaluated from right to left. First, AN multiplies u. Then the result is added to A[N-1]. This result again multiplies u, etc., until the A0 addition is completed. To accomplish the multiply using the 32 bit inputs specified for the POLY32 routine required the development of a small 32 bit multiply routine, MULT32, that would accept one operand of the form of A[j], another operand with the form of "u", and would produce a result with the form of "p" (i.e., A[j]). To make best use of the 16 bit MCU hardware multiplier to perform a 32 bit times 32 bit multiply, cardinal multiplies are performed.

Thus, MULT32 first converts the 2's complement input, B, to the routine (e.g., A[j]) to a signed magnitude form like that of "u". Then the high 16 bits of "u" and the high 16 bits of the magnitude of B are cardinal multiplied. Also, the low 16 bits of "u" cardinal multiply the high 16 bits of the magnitude of B; and, in like manner, the low 16 bits of the magnitude of B cardinal multiply the high 16 bits of "u". The three products (with appropriate offsets) are added to form a 32 bit cardinal fraction with a precision of no worse than + or - 2**(-31).

Using the sign of "u" and B, the result is converted back to the same 2's complement form as B and presented as MULT32 output. It should be noted that the MULT32 output permits the immediate additions required by Eq* 1. Also, the output of the addition is of the form of B and so can be used immediately as input to the MULT32 routine, again, as required by Eq* 1.

The MULT32 routine is implemented so as to provide at least 6 guardbits during each POLY32 multiply operation. Since no more than 7 adds are employed in POLY32 and the multiplies have virtually no impact on the variance of each sum, the variance of POLY32 is only about twice that of an "A" coefficient; POLY32 has at least 5 guard bits.

The software modules developed to implement the scalar functions, POLY32, and MULT32 are subroutines and so are re-entrant in character. Prior to a call to any of the modules discussed, values and addresses of values needed to execute the routines are pre-loaded into MCU registers allocated to the routines. Results of executions are returned in MCU registers.

The call of any scalar funccion module will result in the automatic load of both the POLY32 and MULT32 routines. Independent of the number of same or different scalar function module calls made by a user's program, these modules will be loaded once only. If no scalar function modules are called, these modules will not be loaded.

The scalar routines use all MCU registers. Users must save register values they want to preserve prior to calling any scalar module.

Section 3.3 describes the interface register requirements for each function.

The following section, Section 3.2, describes each function algorithm for the MCU subroutines. The descriptions are given in Program Design Language (PDL) form. In addition, the PDL steps have been numbered for reference and include fractional partitioning using odd or even numbers to assist in identifying logical paths.

Appendix A describes the generation of function interval polynomial coefficients for each of the MCU functions.

3.2 DESCRIPTION OF MCU ALGORITHMS

3.2.1 MCU SQUARE ROOT SUBROUTINE : SQRTM

This Subroutine develops the value, "Y", the square root of the input variable, "X". "X", the input, and "Y", the output, are 32 bit VAX floating point numbers. Along with "Y", a 16 bit status value, S, is generated for output; S=3 indicates a negative X. The "Y" value for such X has no meaning; only positive "X" values are permitted as input arguments.

****

When the exponent of "X" is even, the routine demands the calculation of y1=(SQRT(w)) where .5 <= w < 1 ; thus, the range of y1 is (.707...) <= y1 < 1.

The value of "y1" is established using the polynomial, "p1", given by

```
p1=A10*(U**0)+A11*(U**1)+A12*(U**2)+A13*(U**3)+......+A1N*(U**N)
   where  U=2*(w-.75) ; thus, the range of p1 is (.707...-.75) <= p1 < (1-.75)
.
```

The polynomial "p1" is computed from right to left using

```
p1=A10+U*(A11+U*(A12+U*(A13+U*(A14+U*(A15+U*(A16+.....+U*(A1N) .
```

The POLY32 routine used to compute p1 assumes that -1/2 <= U < 1/2 and "U" has the signed magnitude format [S, (0.31.0)]*2**(-32) and that p1 lies in the range -1/4 <= p1 < 1/4 (it does) and has the 2's complement format (1.31.0)*2**(-32) .

The starting location of the memory space that stores the "A1" coefficients needed to compute p1 and then y1 is COEF1. The coefficient data are assumed stored in the sequence:

Address	Item
COEF1+ 0	A10(hi)
COEF1+ 2	A10(lo)
COEF1+ 4	A11(hi)
COEF1+ 6	A11(lo)
.	.
.	.
.	.
COEF1+4*N1	A1N(hi)
COEF1+4*N1+2	A1N(lo) ;

N1, the degree of the "p1" polynomial, is defined within this subroutine.

Y=y1*(2**((EX-128)/2))
  where EX is the VAX biased exponent of X.

****

When the exponent of "X" is odd, the routine demands the calculation of

y2=SQRT(w/2)
   where .5 <= w < 1 ; thus, the range of y2 is .5 <= y2 < (.707...) .

The value of "y2" is established using the polynomial, "p2", given by

p2=A20*(U**0)+A21*(U**1)+A22*(U**2)+A23*(U**3)+.....+A2N*(U**N)
 where U=2*(w-.75) ; thus, the range of p2 is (.707...-.75) <= p2 < (1-.75) .

The polynomial "p2" is computed from right to left using

p2=A20+U*(A21+U*(A22+U*(A23+U*(A24+U*(A25+U*(A26+.....+U*(A2N) .

The POLY32 routine used to compute p2 assumes that -1/2 <= U < 1/2 and "U" has the signed magnitude format [S, (0.31.0)]*2**(-32) and that p2 lies in the range -1/4 <= p2 < 1/4 (it does) and has the 2's complement format (1.31.0)*2**(-32) .

The starting location of the memory space that stores the "A2" coefficients needed to compute p2 and then y2 is COEF2. The coefficient data are assumed stored in the sequence:

Address	Item
COEF2+ 0	A20(hi)
COEF2+ 2	A20(lo)
COEF2+ 4	A21(hi)
COEF2+ 6	A21(lo)
COEF2+ 8	A22(hi)
COEF2+ 10	A22(lo)
.	.
.	.
.	.
COEF2+4*N2	A2N(hi)
COEF2+4*N2+2	A2N(lo) ;

N2, the degree of the "p2" polynomial, is defined within this subroutine.

Once y2 is computed, the output floating point Y value is given by

Y=y2*(2**((EX-128+1)/2))

where EX is the VAX biased exponent of X.

For both the "p1" and "p2" polynomials above, coefficients of the polynomial are assumed to have the same format as is used for the polynomial value. The "A1" and "A2" coefficient blocks for both polynomials are stored as part of this subroutine.

The degree of the polynomials is at least 1.

****

The entry branch and link register for this subroutine is RF. The subroutine calls the POLY32 subroutine by way of register RF. The POLY32 subroutine , in turn, calls the subroutine, MULT32.MS, as an internal subroutine (i.e., no BAL register is used) .

Registers directly required by this subroutine are marked with a "*". Registers indirectly required by the POLY32 routine are marked with a "#". Registers indirectly required by the MULT32.MS routine are marked with a "$".

Register:	RE	RD	RC	RB	RA	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0
SQRTM Use:	*	*	*	*	*	*	*	*				*	*	*	*
POLY32 Use:		#			#	#	#	#					#		#
MULT32.MS Use:		$			$	$					$	$	$	$	$

ON ENTRY:
R9=Xlo
RB=Xhi

ON EXIT:
R0=Ylo
R2=Yhi
.bp

1. SQRTM entry .

Register:	RE	RD	RC	RB	RA	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0
Use:				Xhi		Xlo									

```
2. RE=0  (Set status to 0.)
    Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
             ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
         Use:¦  C¦   ¦   ¦Xhi¦   ¦Xlo¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦

3. IF RB=0  (Check for X=0; return with Y=0 in such case. Also, check for
             X=negative; abort with status value of 3 in such case.)
    Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0
             ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
         Use:¦  0¦   ¦   ¦  0¦   ¦Xlo¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦

         IF R9=0
               R0=0   . (Ylo=0.)
               R2=0   . (Yhi=0.)
               RETURN  (by way of RF).
    Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
             ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦(0)¦   ¦(0)¦
         Use:¦  0¦   ¦   ¦  0¦   ¦  0¦   ¦   ¦   ¦   ¦   ¦   ¦Yhi¦   ¦Ylo¦

            Else
               Continue
    Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
             ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
         Use:¦  0¦   ¦   ¦  0¦   ¦Xlo¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦

         End If.

      Else
         IF RB=negative
               RE=X'0003'
               RETURN  (by way of RF).
    Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
             ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
         Use:¦  3¦   ¦   ¦Xhi¦   ¦Xlo¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦

            Else
               Continue
    Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
             ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
         Use:¦  0¦   ¦   ¦Xhi¦   ¦Xlo¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦

         End If.

   End If.
```

4. (RB, RC)=RB*X'0200', cardinal multiply.
(Put X exponent (and "0" sign bit), right justified, into RB; put true mantissa-.75 into (R9, RA) (radix point for X true mantissa is 1 bit position left of left edge of R9). EX is biased exponent of X.)

Register:	RE	RD	RC	RB	RA	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0
Use:	0		mh1	EX		Xlo									

(R9, RA)=R9*X'0200', cardinal multiply.

Register:	RE	RD	RC	RB	RA	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0
Use:	0		mh1	EX	mlo	mh2									

R9=R9 .OR. RC (Merge mantissa chunks.)

Register:	RE	RD	RC	RB	RA	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0
Use:	0		-	EX	mlo	mhi									

5. (Create true mantissa-.75 . Then conceptually multiply result by 2 to create U. Radix point will then be at left edge of R9.)

```
RD=.NOT. R9  . (Lead bit of RD, RD(0), now contains sign bit of U.)
RD=RD .AND. X'8000'  . (Clears all but U sign bit.)
If RD=0
     (Sign bit of U is 0, i.e., +.)
     R9=R9 .EXCLUSIVE OR. X'8000'  . (Clear lead bit of R9 when
                                      true mantissa-.75 is +; creates
                                      Uhi. Ulo already exists. Uhi,Ulo
                                      is the magnitude of U.)
  Else
     R9=R9 .EXCLUSIVE OR. X'7FFF'  . (Clear lead bit of R9 when
                                      true mantissa-.75 is -; complement
                                      remaining bits of R9 to create
                                      Uhi. Now proceed to complement
                                      RA which becomes Ulo. Uhi,Ulo
                                      is the magnitude of U.)
     RA=RA .EXCLUSIVE OR. X'FFFF'  . (Complement complete.)
```

Register:	RE	RD	RC	RB	RA	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0
Use:	0	SU	-	EX	Ulo	Uhi									

6. RC=X'0001' (Detect odd/even character of X exponent.)

Register:	RE	RD	RC	RB	RA	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0
		0													
Use:	0	SU	1	EX	Ulo	Uhi									

```
    RC=RC .AND. RB  (RC=1 if EX is odd; RC=0 if EX is even.)
     Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
              ¦   ¦  0¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
          Use:¦  0¦ SU¦0/E¦ EX¦Ulo¦Uhi¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦


 7. If RC=0
     Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
              ¦   ¦  0¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
          Use:¦  0¦ SU¦  0¦ EX¦Ulo¦Uhi¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦

          R7=N1   (Set degree for calculating "p1"  (even EX).)
          R8=COEF1+4*N1+2  (Set to last 16 bit address of COEF1 block.)
     Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
              ¦   ¦  0¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
          Use:¦  0¦ SU¦  0¦ EX¦Ulo¦Uhi¦loc¦cnt¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦

          BAL,RF     POLY32.  (Compute "p1" polynomial.)
     Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
              ¦   ¦  0¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦ p1¦   ¦ p1¦
          Use:¦  0¦ SU¦  0¦ EX¦Ulo¦Uhi¦ - ¦ - ¦   ¦   ¦ - ¦ - ¦ hi¦ - ¦ lo¦

      (Warning: Polynomial coefficients must be chosen so that the maximum
      value of p1 is X'FFFFFFFF' for the maximum input X=X'FFFFFFFF'. Also,
      polynomial coefficients must be chosen so that the minimum
      value of p1 is X'C0000000' for the minimum input X=.5  .)
      R2=R2+X'4000'  (Unbias p1 to create p1+.25=Y true mantissa-.5  .
                          Result is positive; lead sign bit is 0.  Radix
                          point is at left edge of R2.)
      RB=RB-128+0+256  (Preliminary development for biased "Y" exponent.)
      (RB, RC)=RB*X'0040'  , 2's complement multiply.
                                      (Develop "Y" exponent and sign bit.)
      (R2, R3)=R2*X'0100'  , cardinal multiply. (Line up "Y" mantissa hi.)
      (R0, R1)=R0*X'0100'  , cardinal multiply. (Line up "Y" mantissa lo.)
      R2=R2 .OR. RC  (Merge "Y" sign, exponent, and mantissa hi.)
      R0=R0 .OR. R3  (Merge "Y" mantissa lo1 with mantissa lo2.)
      RETURN  (by way of RF).
     Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
              ¦   ¦  0¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
          Use:¦  0¦ SU¦  0¦ EX¦Ulo¦Uhi¦ - ¦ - ¦   ¦   ¦ - ¦ - ¦Yhi¦ - ¦Ylo¦

       Else
     Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
              ¦   ¦  0¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
          Use:¦  0¦ SU¦  0¦ EX¦Ulo¦Uhi¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦

          R7=N2   (Set degree for calculating "p2"  (odd EX).)
          R8=COEF2+4*N2+2  (Set to last 16 bit address of COEF2 block.)
     Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
              ¦   ¦  0¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
          Use:¦  0¦ SU¦  0¦ EX¦Ulo¦Uhi¦loc¦cnt¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦

          BAL,RF     POLY32.  (Compute "p2" polynomial.)
     Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
              ¦   ¦  0¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦ p2¦   ¦ p2¦
          Use:¦  0¦ SU¦  0¦ EX¦Ulo¦Uhi¦ - ¦ - ¦   ¦   ¦ - ¦ - ¦ hi¦ - ¦ lo¦

       (Warning: Polynomial coefficients must be chosen so that the minimum
          value of p2 is X'C0000000' for the minimum input X=.25  .)
          R2=R2+X'4000'  (Unbias p2 to create p2+.25=Y true mantissa-.5  .
                          Result is positive; lead sign bit is 0. Radix
                          point is at left edge of R2.)
          RB=RB-128+1+256  (Preliminary development for biased "Y" exponent.)
          (RB, RC)=RB*X'0040'  , 2's complement multiply.
                                               (                                .)
          (R2, R3)=R2*X'0100'  , cardinal multiply. (Line up "Y" mantissa hi.)
          (R0, R1)=R0*X'0100'  , cardinal multiply. (Line up "Y" mantissa lo.)
          R2=R2 .OR. RC  (Merge "Y" sign, exponent, and mantissa hi.)
          R0=R0 .OR. R3  (Merge "Y" mantissa lo1 with mantissa lo2.)
          RETURN  (by way of RF).
     Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
              ¦   ¦  0¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
          Use:¦  0¦ SU¦  0¦ EX¦Ulo¦Uhi¦ - ¦ - ¦   ¦   ¦ - ¦ - ¦Yhi¦ - ¦Ylo¦

    End If.

END
```

3.2.2 MCU SINE SUBROUTINE DESCRIPTION : SINM

Subroutine develops the value, "Y", the sine of the input variable, "X". "X", the input, and "Y", the output, are 32 bit VAX floating point numbers. Along with "Y", a 16 bit status value, S, is generated for output; it is 0 when ¦X¦ is less than 2**24. When ¦X¦ is equal to or greater than 2**24, the angular uncertainty is on the order of 2 radians and so the Y answer becomes meaningless; the status is set to 3 in such case (no processing is performed).

The "Y" value range is   -1 <= Y <= +1   .

****

The subroutine demands the calculation of

```
Y=SIN(X)=((-1)**SX)*SIN(¦X¦)  where
          SX is the value of the sign bit of X, and
          ¦X¦ symbolizes the absolute value of X.
```

¦Y¦ must be computed using a number of different approximations. For true X exponents (TEX) less than 0, the ¦X¦ interval is dissected into 3 different sub-intervals, namely,

```
     1)              TEX < (-10)      or               ¦X¦ < 2**(-11)  ,
     2)      (-10) <= TEX < ( -4)     or     2**(-11) <= ¦X¦ < 2**( -5)  ,
and  3)      ( -4) <= TEX < (  0)     or     2**( -5) <= ¦X¦ < 2**( -1)  .
```

For true X exponents greater than or equal to 0, ¦Y¦ is computed only after converting X to units of 1/4 rotations and then converting this resultant value, R, to an integer form. To find ¦Y¦ in such case, the fractional part of R, Rf , is first dissected into 8 equal sub-intervals. For the appropriate sub-interval "j", the associated polynomial, sj, is used to approximate ((SIN(Rf*(PI/2)))/2)-.25 . Then, SIN(ARG) is approximated by 2*(sj+.5). For odd quadrants,

```
ARG=(1+Rf)*PI/2  .
```

For even quadrants,

```
ARG=(Rf)*PI/2  .
¦Y¦=2*(sj+.5)  .
```

The final Y value is determined by the ¦X¦ quadrant index and the sign of X.

Specifically,

```
|Y|=2*(sj+.5)  and the sign of Y, SY, is
SY=SX .EXCLUSIVE OR. Qhi .EXCLUSIVE OR. Qlo   where
   SX is the sign of X,
   Qhi is the top bit of the |X| quadrant index, and
   Qlo is the bottom bit of the |X| quadrant index.
```

The specific approximations for the 3 |X| sub-intervals corresponding to negative X exponents are listed below. Then, the 8 approximations used to approximate `((SIN(Rf*(PI/2)))/2)-.25` are listed.

APPROXIMATIONS

Sub-interval 1)

When the true exponent of X is less than -10,

```
|Y|=|X|.
```

Sub-interval 2)

When the true exponent of X is less than -4 but greater than or equal to -10, |Y| is given by

```
|Y|=|X|-(|X|**3)/6  and so by

|Y|=|X|*(1-(2/3)*X2)   where
  X2=(|X|/2)**2.
```

POLY32, the polynomial expansion routine, is not used to compute this approximation.

Sub-interval 3)

When the true exponent of X is less than 0 but greater than or equal to -4, |Y| is found using

```
|Y|=|X|*(1-G(|X|/2))  where
  G(|X|/2)=1-(SIN(|X|))/|X|
```

The value of G(|X|/2) is established using the polynomial, "p1(U)", given by

```
p1(U)=A10*(U**0)+A11*(U**1)+A12*(U**2)+A13*(U**3)+.....+A1N1*(U**N1)  where
```

U=|X|/2 ,
the |X| range is 1/32 <= |X| < .5 ,
the U range is 1/64 <= U < .25 , and
p1 approximates (G(|X|/2))/2 .

The polynomial "p1" is computed from right to left us'ng

p1(U)=A10+U*(A11+U*(A12+U*(A13+U*(A14+U*(A15+U*(A16+.....+U*(A1N) .

The POLY32 routine used to compute p1 assumes that 1/64 <= U < 1/4 and "U" has the signed magnitude format [S, (0.31.0)]*2**(-32) and that p1 lies within the range -1/4 <= p1 <= 1/4 (it does) and has the 2's complement format (1.31.0)*2**(-32) .

The starting location of the memory space that stores the "A1" coefficients needed to compute p1 is COEF1. The coefficient data are assumed stored in the sequence:

Address	Item
COEF1+ 0	A10(hi)
COEF1+ 2	A10(lo)
COEF1+ 4	A11(hi)
COEF1+ 6	A11(lo)
COEF1+ 8	A12(hi)
COEF1+ 10	A12(lo)
COEF1+ 12	A13(hi)
COEF1+ 14	A13(lo)
.	.
.	.
.	.
COEF1+4*N1	A1N(hi)
COEF1+4*N1+2	A1N(lo) .

N1, the degree of the "p1" polynomial, and the COEF1 coefficient data are defined within this subroutine.

Once p1 is computed, the output |Y| value is given by

|Y|=|X|*(1-2*p1) .

****

X => .5 approximations

When the true exponent of X is 0 or greater, |Y| is found by first converting the angle |X| from a radian measure to a 1/4 rotations measure. The converted angle is called "|R|" and is described in terms of units of radians. |R|, is defined by

```
|R|=|X|/(PI/2)=|X|*(2/PI/)   (Units=1/4 rotations).
```

The fractional part of |R|, Rf, is dissected into 8 equal sized sub-intervals that are indexed from 0 through 7. For each sub-interval, the SIN((PI/2)*Rf) is approximated using the particular polynomial associated with the sub-interval. The "j"th (j=0, 1,..., 7) sub-interval approximation of SIN((PI/2)*Rf) is described in terms of the polynomial sj(U) where U is related to Rf using

```
Rf=j/8+(U/8)   where
   0 <= U < 1  .
```

The polynomial, "sj(U)", is defined by

```
sj(U)=Bj0*(U**0)+Bj1*(U**1)+Bj2*(U**2)+Bj3*(U**3)+.....+BjMj*(U**Mj)  where
     the Rf range is       j/8 <= Rf < (j+1)/8  , j=1,...,7  ,
     the U range is       0 <= U < 1  , and
     sj(U) approximates     (SIN((Rf*PI/2)))/2  .
```

The polynomial "sj" is computed from right to left using

```
sj=Bj0+U*(Bj1+U*(Bj2+U*(Bj3+U*(Bj4+U*(Bj5+U*(Bj6+.....+U*(BjMj)  .
```

The POLY32 routine used to compute sj assumes that   0 <= U < 1 and "U" has the signed magnitude format [S, (0.31.0)]*2**(-32) and that sj lies in the range -.5 <= sj <= .5 (it does) and has the 2's complement format (1.31.0)*2**(-32) .

The starting location of the memory space that stores the "Bj" coefficients needed to compute sj is KOEFj. The coefficient data are assumed stored in the sequence:

Address	Item
KOEFj+ 0	Bj0(hi)
KOEFj+ 2	Bj0(lo)
KOEFj+ 4	Bj1(hi)
KOEFj+ 6	Bj1(lo)
KOEFj+ 8	Bj2(hi)
KOEFj+ 10	Bj2(lo)
KOEFj+ 12	Bj3(hi)
KOEFj+ 14	Bj3(lo)
.	.
.	.
.	.
KOEFj+4*N3	BjN(hi)

KOEFj+4*N3+2 BjN(lo) .

Mj, the degree of the "sj" polynomial, and the KOEFj coefficient data are defined within this subroutine.

Once sj is computed, the output SIN(Rf*(PI/2)) value is given by

SIN(Rf*(PI/2))=2*sj .

For the "p1" and "sj" polynomials above, coefficients of the polynomial have the 2's complement format (1.31.0)*2**(-32) , the same format as is used for the polynomial value.

The degree of the polynomials is at least 1.
****

The entry branch and link register for this subroutine is RF. The subroutine calls the POLY32 subroutine by way of register RF. The POLY32 subroutine , in turn, calls the subroutine, MULT32.MS, as an internal subroutine (i.e., no BAL register is used) .

Registers directly required by this subroutine are marked with a "*".
Registers indirectly required by the POLY32 routine are marked with a "#".
Registers indirectly required by the MULT32.MS routine are marked with a "$".

Register:	RE	RD	RC	RB	RA	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0
	*	*	*	*	*	*	*	*	*	*		*	*	*	*
POLY32 Use:		#			#	#	#	#	#				#		#
MULT32 Use:		$			$	$					$	$	$	$	$

ON ENTRY:
R9=Xlo
RB=Xhi

ON EXIT:
R0=Ylo
R2=Yhi

```
*******************************************************************************

                                  SINM entry

*******************************************************************************

 1. SINM entry  .
Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
         ¦   ¦   ¦           ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
     Use:¦   ¦   ¦   ¦Xhi¦   ¦Xlo¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦


 2. R0=X'4000'  .  (Capture complement of true exponent sign bit in R0.)
Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
         ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
     Use:¦   ¦   ¦   ¦Xhi¦   ¦Xlo¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦Use¦

 3. R0=R0 .AND. RB  (Complement of sign of X true exponent in R0 after step.)
Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
         ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦___¦
     Use:¦   ¦   ¦   ¦Xhi¦   ¦Xlo¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦SEX¦


 4. IF R0=0  (Split processing based on true exponent (-) or (+,0).)
Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
         ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
     Use:¦   ¦   ¦   ¦Xhi¦   ¦Xlo¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦ - ¦

          (The true X exponent must lie in the range, -128 <= EXP-128 <= -1  ,
          to branch in this direction.
          Develop X sign bit in R2(0) and "0" in other 15 bits.)
 4.0      R2=X'8000'  .
          R2=R2 .AND. RB  . (Sign bit now in R2(0).)
Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
         ¦   ¦   ¦   ¦   ¦   ¦   ¦                               ¦
     Use:¦   ¦   ¦   ¦Xhi¦   ¦Xlo¦   ¦   ¦   ¦   ¦   ¦   ¦ SX¦   ¦ - ¦

 4.2      RB=RB .EXCLUSIVE OR. R2  . (Zero's out X sign bit in RB;
                                     X becomes ¦X¦.)
Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ P5¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
         ¦   ¦   ¦   ¦¦X¦¦   ¦¦X¦¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
     Use:¦   ¦   ¦   ¦ hi¦   ¦ lo¦   ¦   ¦   ¦   ¦   ¦   ¦ SX¦   ¦ - ¦

          (Determine if X true exponent is less than -10.)
 4.4      R0=RB  . (Replicate ¦X¦ high.)
          R0=R0-((128-10)*128)  . (Remove exponent bias; add 10 to result.)
          R0=R0 .AND. X'FF80'  . (Clear mantissa bits out of R0; R0 value is
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
                                (true X exponent + 10)*128  ; final result
                                is T1.)
          Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
                   |            ||   ||X||   |   |   |   |   |   |   |   |   |
              Use:|   |   |   | hi|   | lo|   |   |   |   |   |   | SX|   | T1|

 4.6      If R0=negative
                (X true exponent is less than -10,  T1 < 0; Y=X.)
 4.6.0          R2=R2 .OR. RB . (Re-insert sign bit into |X| high in R2.
                                 Yhi=Xhi.)
                R0=R9  . (Ylo=Xlo.)
                RE=0   . (Set status.)
                RETURN  (by way of RF).
Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
        |   |   |   ||X||   ||X||   |   |   |   |   |   |   |   |   |
    Use:|  0|   |   | hi|   | lo|   |   |   |   |   |   |Yhi|   |Ylo|
    ****************************************************************

             Else
                (X true exponent is greater than or = to -10 but less than 0.
                (0 <= T1 < 10).)
Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
        |   |   |   ||X||   ||X||   |   |   |   |   |   |   |   |   |
    Use:|   |   |   | hi|   | lo|   |   |   |   |   |   | SX|   | T1|

 4.6.1          (R0, R1)=R0*X'0400' (card). (Let E2A=2*(X true exponent + 10);
                                      put E2A into R0.)
Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
        |   |   |   ||X||   ||X||   |   |   |   |   |   |   |   |   |
    Use:|   |   |   | hi|   | lo|   |   |   |   |   |   | SX| - |E2A|

                (Put X true mantissa magnitude into RC, RA.)
 4.6.3          (RB, RC)=RB*X'0100' (card) . (Put X true mantissa (high) into
                                              RC; radix point is on left edge
                                              of RC.)
                (R9, RA)=R9*X'0100' (card) . (Put X true mantissa (low) into
                                              R9, RA.)
                RC=RC .OR. R9  . (Merge high mantissa bits.)
                RC=RC .OR. X'8000'  . (Insert lead "1" bit into X mantissa;
                                       result is X true mantissa, mX, in
                                       (RC, RA) .)
Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
        |   |   | mX|   | mX|   |   |   |   |   |   |   |   |   |   |
    Use:|   |   | hi| - | lo| - |   |   |   |   |   |   | SX| - |E2A|

                (Replicate true mantissa (magnitude); put (RC,RA) into (R5,R3).)
 4.6.5          R5=RC  .
                R3=RA  .
```

Register:	RE	RD	RC	RB	RA	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0
			mX		mX					mX		mX			
Use:			hi	-	lo	-				hi		lo	SX	-	E2A

(Make ¦X¦/2 an integer. Its radix point is to be at the left edge of R6. If R6, R5, and R3 are considered contiguous, and the radix point now sits at the left edge of R6, then the X true mantissa magnitude has been multiplied by 2**(-16). To make the integer ¦X¦/2, the mX value should have been multiplied by 2**(tru exp-1) and not by 2**(-16). To correct the (R6, R5, R3) value, it must be multiplied by 2**(tru exp + 15) or, since the R0 value divided by 2 is (tru exp + 10), by 2**((R0value/2)+5).
The values of R0 range from 2*0 to 2*9. For the various values of R0, the values of 2**((R0value/2)+5) will be pulled from the MCU memory table, SHF . The table follows:

SHF Table

Location	R0	2**(R0/2)
SHF + 0	2*0	X'0001'
SHF + 2	2*1	X'0002'
SHF + 4	2*2	X'0004'
SHF + 6	2*3	X'0008'
SHF + 8	2*4	X'0010'
SHF +10	2*5	X'0020'
SHF +12	2*6	X'0040'
SHF +14	2*7	X'0080'
SHF +16	2*8	X'0100'
SHF +18	2*9	X'0200'
SHF +20	2*10	X'0400'
SHF +22	2*11	X'0800'
SHF +24	2*12	X'1000'
SHF +26	2*13	X'2000'
SHF +28	2*14	X'4000'
SHF +30	2*15	X'8000'

4.6.7 R0=R0+SHF5 . (SHF5=SHF+2*5 . Point to correct location in shift table. PNT is the pointer value in R0. E+S=2*(true X exponent+10+SHF5)=PNT.)

R1=0(R0) . (Put the scaling value, SCL, in memory location referenced by R0 into R1.)

Register:	RE	RD	RC	RB	RA	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0
			mX		mX					mX		mX			PNT
Use:			hi	-	lo	-				hi		lo	SX	SCL	E+S

4.6.9 (R5, R6)=R5*R1 (card) . (Shift X true mantissa high according to SCL; create higher part of ¦X¦/2 .)

(R3, R4)=R3*R1 (card) . (Shift X true mantissa low according to SCL; create lower part of ¦X¦/2 .)
R3=R3 .OR. R6 . (Merge low parts of integer ¦X¦/2; integer ¦X¦/2 is in (R5, R3) after step. Radix point is at left edge of R5.)

Register:	RE	RD	RC	RB	RA	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0
			mX		mX					int		int			PNT
Use:			hi	-	lo	-			-	¦X¦	-	¦X¦	SX	SCL	E+S
										/2		/2			
										hi		lo			

(Split processing where X true exponent is less than -4.)
4.6.B RO=RO-(SHF5+2*(10-4)) . (R0/2=tru exp + 4; E24=2*(tru exp + 4).)

Register:	RE	RD	RC	RB	RA	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0
					mX					int		int			
Use:			hi	-	lo	-			-	¦X¦	-	¦X¦	SX	-	E24
										/2		/2			
										hi		lo			

4.6.D If RO=negative
(X true exponent is less than -4 but greater than or equal to -10. Use ¦Y¦=¦X¦*(1-(2/3)*(¦X¦/2)**2) for computing ¦Y¦.
Square integer ¦X¦/2 1st.)
4.6.D.0 (R3, R4)=R3*R5 (card) . (Multiply lo times hi (int X).)
(R5, R6)=R5*R5 (card) . (Multiply hi times hi (int X).)
R6=R6+R3 ,save carry out . (Partial square lo.)
R5=R5+carry in . (Partial square hi.)
R3=R3+R6 ,save carry out . (Full square lo.)
R5=R5+carry in . (Full square hi.)

Register:	RE	RD	RC	RB	RA	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0
			mX		mX					int		int			
Use:			hi	-	lo	-			-	¦X¦	-	¦X¦	SX	-	E24
										/2		/2			
										sqr		sqr			
										hi		lo			

(Square of integer ¦X¦/2 is in (R5, R3); radix point is at left edge of R5. Multiply square times (2/3)=.6666... and call the result "D" . Then subtract "D" ,namely , (.6666...*(¦X¦/2)**2) from 1 and call the result C.)
4.6.D.2 (R3, R4)=R3*X'AAAA' (card) . (Multiply lo (int ¦X¦/2 sqr) times both (2/3) hi and (2/3) lo. The hi and lo part of (2/3)=X'AAAA'; radix point of (2/3) hi is at left edge of

```
          at X'AAAA'.)
          (R5, R6)=R5*X'AAAA' (card) . (Multiply hi (int ¦X¦/2 sqr)
                                     times both (2/3) hi and
                                     (2/3) lo.)
           R6=R6+R5 ,save carry out . (Add for partial multiply, lo.)
           R5=R5+carry in  . (Add for partial multiply hi.)
           R3=R3+R6 ,save carry out . (Add for full multiply, Dlo.)
           R5=R5+carry in  . (Add for full multiply, Dhi.)
Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
         ¦   ¦   ¦ mX¦   ¦ mX¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
     Use:¦   ¦   ¦ hi¦ - ¦ lo¦ - ¦   ¦   ¦ - ¦Dhi¦ - ¦Dlo¦ SX¦ - ¦E24¦

     ("C" is now in (R5, R3). Develop C=1-D=1-(X**2)/6  .)
 4.6.D.4    R5=.NOT. R5  . (1-(X**2)/6 hi.)
            R3=.NOT. R3  . (1-(X**2)/6 lo. "C" is now in (R5, R3).)
Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
         ¦   ¦   ¦ mX¦   ¦ mX¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
     Use:¦   ¦   ¦ hi¦ - ¦ lo¦ - ¦   ¦   ¦ - ¦Chi¦ - ¦Clo¦ SX¦ - ¦E24¦

                     (Multiply the ¦X¦ true mantissa times "C" to get the
                     value, mX*(1-(X**2)/6). Call the result mV. Put into
                     (R5, R3).)
 4.6.D.6        (RA, RB)=RA*R5 (card)  . (Multiply mXlo times Chi.)
                (R5, R6)=R5*RC (card)  . (Multiply Chi times mXhi.)
                RA=RA+R6 ,save carry out . (Add for partial multiply, lo.)
                R5=R5+carry in  . (Add for partial multiply hi.)
                (R3, R4)=R3*RC (card)  . (Multiply Clo times mXhi.)
                R3=R3+RA ,save carry out  . (Add for full multiply; mVlo.)
                R5=R5+carry in  . (Add for full multiply; mVhi.)
Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
         ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦ mV¦   ¦ mV¦   ¦   ¦   ¦
     Use:¦   ¦   ¦ - ¦ - ¦ - ¦ - ¦   ¦   ¦ - ¦ hi¦ - ¦ lo¦ SX¦ - ¦E24¦

                     (If R5 lead bit is 1 (i.e., if it looks negative, the
                     value is in the mantissa range of the output Y. Else,
                     mV is a hair below .5 and needs to be multiplied by 2.
                     Fix and produce output.)
 4.6.D.8             If R5=negative     (mV=.5 test . )
                          (Pseudonegative low likelihood branch direction;
                          the mV magnitude is => .5  .)
 4.6.D.8.0                R0=R0+2*(128-4-1)  (((Biased exponent of Y)-1)*2;
                                              to align exponent, need to
                                              multiply R0 by 64.)
                          (R5, R6)=R5*256  (Y true mantissa hi 8 bits
                                            properly positioned in R5.)
                          (R3, R4)=R3*256  (Y true mantissa lowest 16 bits
                                            properly positioned in R3.)
```

```
                        Else
4.6.D.8.1                  RO=RO+2*(128-5-1)  (((Biased exponent of Y)-1)*2;
                                               t  align exponent, need to
                                               multiply RO by 64.)
                           (R5, R6)=R5*512  (Y true mantissa hi 8 bits
                                              properly positioned in R5.)
                           (R3, R4)=R3*512  (Y true mantissa lowest 16 bits
                                            properly positioned in R3.)

                      End (4.6.D.8) If.

                      (Fix exponent.)
4.6.D.A               (RO, R1)=RO*X'0040'  . (Properly positioned biased
                                              Y exponent is now in R1.)
                      (Fix mantissa.)
                      R3=R3 .OR. R6  . (Merge lo bits of Y mantissa; Ylo.)
                      R5=R5 .AND. X'FF7F'  (Clear lead mantissa bit.)
                      R2=R2 .OR. R5  (Sign and biased mantissa merge.)
                      R2=R2+R1  (Add aligned biased Y exponent to aligned
                                ¦Yhi¦ mantissa.)
                      RO=R3          (Move Ylo into R1.)
                      RE=0  . (Status.)
                      RETURN  (by way of RF).
Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ RO¦
         ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
     Use:¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦Yhi¦ - ¦Ylo¦
   *****************************************************************

                    Else
                      (¦X¦ true exponent is less than 0 but greater than or
                      equal to -4. Use ¦Y¦=¦X¦*(1-G(¦X¦/2)) for computing Y.)
Register:¦ RE¦ RD, RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ RO¦
         ¦   ¦   ¦ mX¦   ¦ mX¦   ¦   ¦   ¦   ¦int¦   ¦int¦   ¦   ¦   ¦
     Use:¦   ¦   ¦ hi¦ - ¦ lo¦ - ¦   ¦   ¦ - ¦¦X¦¦ - ¦¦X¦¦ SX¦ - ¦E24¦
         ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦ /2¦   ¦ /2¦   ¦   ¦   ¦
         ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦ hi¦   ¦ lo¦   ¦   ¦   ¦

4.6.D.1               R6=RA  . (Save ¦X¦ true mantissa low in R6.)
                      RB=RO  . (Save 2*(¦X¦ true exponent + 4) in RB.)
                      RE=R2  . (Save X sign in RE.)
                      R9=R5  . (Move int ¦X¦/2 hi (Uhi) into R9.)
                      RA=R3  . (Move int ¦X¦/2 lo (Ulo) into RA.)
                      RD=0   . (Load sign bit of U, a magnitude, into RD.)
                      R7=N1   (Set degree for calculating "p1" where p1
                           approximates (G(¦X¦/2))/2=(1-SIN(¦X¦)/¦X¦)/2  .)
                      R8=COEF1+4*N1+2  (Set to last 16 bit address of COEF1
                                         block.)
```

```
Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
         |   |  0| mX|   |int|int|   |   | mX|   |   |   |   |   |   |
     Use:| SX| SU| hi|E24||X|||X||loc|cnt| lo| - | - | - | - | - | - |
         |   |   |   |   | /2| /2|   |   |   |   |   |   |   |   |   |
         |               'o| hi|   |   |   |   |   |   |   |   |   |
         |                         |   |   |   |   |   |   |   |   |   |
         |   |   |   |   |Ulo|Uhi|   |   |   |   |   |   |   |   |   |
```

4.6.D.3   BAL,RF   POLY32. (Compute "p1" polynomial (G(|X|/2)) .)

```
Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
         |   |  0| mX|   |int|int|   |   | mX|   |   |   | p1|   | p1|
     Use:| SX| SU| hi|E24||X|||X|| - | - | lo| - | - | - | hi| - | lo|
         |   |   |   |   | /2| /2|   |   |   |   |   |   |   |   |   |
         |   |   |   |   | lo| hi|   |   |   |   |   |   |   |   |   |
         |   |   |   |   | = | = |   |   |   |   |   |   |   |   |   |
         |   |   |   |   |Ulo|Uhi|   |   |   |   |   |   |   |   |   |
```

(Generate C=(1-G(|X|/2)) approximation. Note: G(|X|/2) radix point is 1 bit location right of the left edge of R2.)

4.6.D.5   R2=R2 .EXCLUSIVE OR. X'7FFF' . (Use p1 to approximate G(|X|/2). Radix point of p1 (but not G) is at left edge of R2. Operation on (R2, R0) yields the hi part of (1-G(|X|/2)), i.e., Chi, an unsigned number.)

R0=R0 .EXCLUSIVE OR. X'FFFF' . (Low part of (1-G(|X|/2)); Clo.)

```
Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
         |       | mX|   |   |   |   |   | mX|   |   |   |   |   |   |
     Use:| SX| SU| hi|E24| - | - | - | - | lo| - | - | - |Chi| - |Clo|
```

(Multiply the X true mantissa times "C" to get the value, mX*(1-G(|X|/2)). Call the result mV. Put into (R2, R0).)

4.6.D.7   (R6, R7)=R6*R2 (card) . (mXlo*Chi.)

(R0, R1)=R0*RC (card) . (Clo*mXhi.)

(R2, R3)=R2*RC (card) . (Chi*mXhi.)

R6=R6+R3, save carry out. (Combine lower bits of partial product, mV=mX*(1-G(|X|/2)). The radix point of mV is 1 bit right of the left edge of R2.)

R2=R2+ carry in . (Combine upper bits of partial product, mV=mX*(1-G(|X|/2)).)

R0=R0+R6, save carry out. (Combine lower bits of complete product, mV=mX*(1-G(|X|/2)).)

R2=R2+ carry in . (Combine upper bits of complete

```
                                    product, mV=mX*(1-G(¦X¦/2)).)
Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
         ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦ mV¦   ¦ mV¦
     Use:¦ SX¦ SU¦ - ¦E24¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦ hi¦ - ¦ lo¦

                  (If R2(1) bit is 1, the value is in the mantissa range
                  of the output Y. Else, mV is a hair below .5 and needs
                  to be multiplied by 2. Note: The radix point of mV is
                  1 bit right of the left edge of R2.
                  Fix and produce output.)
 4.6.D.9          If R2(1)=0  (Less than .5 test.)
                       (The mV magnitude is  < .5   branch direction.)
 4.6.D.9.0             RB=RB+2*(128-5-1)  (((Biased exponent of Y)-1)*2;
                                           to align exponent, need to
                                           multiply R0 by 64.)
                       (R2, R3)=R2*1024  (Y true mantissa hi 8 bits
                                          properly positioned in R2.)
                       (R0, R1)=R0*1024  (Y true mantissa lowest 16 bits
                                          properly positioned in R0.)

                    Else
                       (The mV magnitude is  => .5   branch direction.)
 4.6.D.9.1             RE=RB+2*(128-4-1)  (((Biased exponent of Y)-1)*2;
                                           to align exponent, need to
                                           multiply R0 by 64.)
                       (R2, R3)=R2*512  (Y true mantissa hi 8 bits
                                         properly positioned in R2.)
                       (R0, R1)=R0*512  (Y true mantissa lowest 16 bits
                                         properly positioned in R0.)

                  End (4.6.D.9) If.

                  (Fix exponent.)
 4.6.D.B          (RB, RC)=RB*X'0040'  . (Properly positioned biased
                                          Y exponent is now in RB.)
                  (Fix mantissa.)
                  R0=R0 .OR. R3  . (Merge lo bits of Y mantissa; Ylo.)
                  R2=R2 .AND. X'FF7F'  (Clear lead mantissa bit.)
                  R2=R2+RC  (Add aligned biased Y exponent to aligned
                            ¦Yhi¦ mantissa.)
                  R2=R2 .OR. RE  (Sign and ¦Yhi¦ merge.)
                  RE=0  . (Status.)
                  RETURN  (by way of RF).
Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
         ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
     Use:¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦Yhi¦ - ¦Ylo¦
     ****************************************************************
```

```
                End (4.6.D) If.

             Else
     ********************************************************************
          (X true exponent is 0 or + starting here.)
Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
         |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
     Use:|   |   |   |Xhi|   |Xlo|   |   |   |   |   |   |   |   | - |
****************************************************************************
 4.1  RE=X'8000'.(Prepare to capture X sign in RE(0); clear other bits.)
          RE=RE .AND. RB  . (Sign of X in RE; other RE bits cleared.)
          RB=RB .EXCLUSIVE OR. RE  . (X changed to a magnitude, |X|.)
Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
         |   |   |   ||X||   ||X||   |   |   |   |   |   |   |   |   |
     Use:| SX|   |   | hi|   | lo|   |   |   |   |   |   |   |   | - |


 4.3  If RB => (128+EXmax)*128 (EXmax=25. |X| greater than or equal to
                                 2**24 test.)
          (|X| => 2**24; too much angular uncertainty results. Return
          with status value of 3.)
 4.3.0    RE=3  . (Status=S=3. Worthless results exist in Yhi and Ylo
                            registers.)
          RETURN via RF  .
Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
         | 3 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
     Use:| S |   |   | - |   | - |   |   |   |   |   |   |   |   | - |
          *********************************************************

             Else
                (|X| < 2**24 along this path.)
 4.3.1          Continue.

          End (4.3) If

          (True exponent of |X|, TEX, lies in the range, 0 <= TEX <= 24 ,
          along this path.)
          (Isolate |X| biased exponent (BEX) and biased mantissa (mx).)
 4.5      (RB, RC)=RB*X'0200'  . (|X| biased mantissa hi left justified in RC;
                                  |X| biased exponent, BEX, is in RB, right
                                  justified.)
          (R9, RA)=R9*X'0200'  . (|X| biased mantissa lo left justified in
                                  (R9, RA).)
          R9=R9 .OR. RC  . (|X| biased mantissa hi merged into R9.)
Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
         |   |   |   |   | mX| mX|   |   |   |   |   |   |   |   |   |
     Use:| SX|   | - |BEX| lo| hi|   |   |   |   |   |   |   |   | - |

          (Multiply BEX by 2.)
          (RB, RC)=RB*2  . (Two times BEX, 2BE, is now in RC.)
Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
         |   |   |   |   | mX| mX|   |   |   |   |   |   |   |   |   |
     Use:| SX|   |2BE| - | lo| hi|   |   |   |   |   |   |   |   | - |

          (The input angle value, |X|, is described in terms of units of
          radians. For computational convenience, the angle must be converted
```

to units of 1/4 rotations. This unit transformation is accomplished by multiplying |X| by (2/PI); the resultant value, |R|, expresses the input angle in terms of the 1/4 rotation units. The actual transformation is accomplished by developing the true mantissa of | times (2/PI); this result is called "m2p". |R| is just "m2p" times 2**TEX where TEX is the true exponent of |X|. After the development of |R|, it is converted to an integer form (0.2.30). The fractional part of this integer form, Rf (a single quadrant an le), in conjunction with the 2 integer bits of integer |R| (the quadrant index bits), will be used to find SIN(|X|.)

```
        (Develop the X true mantissa times 2/PI. Note that in this section of
        the PDL, mX is the biased mantissa; thus, to develop m2p,
        mX*(2/PI)+K is computed where K=.5*(2/PI)=1/PI . (The mX bias is
        -.5  .)
        The radix point of mX is 1 bit position left of the left edge of R9.
        The value 2/PI is X'145F306E'*(2**(-29)).)
4.7     R5=R9  . (Replicate mX hi in R5.)
        (RA, RB)=RA*X'145F' (card) . (Product of mX lo times 2/PI hi.)
        (R5, R6)=R5*X'306E' (card) . (Product of mX hi times 2/PI lo.)
        R5=R5+RA, save carry . (Add for partial product lo.)
        (R9, RA)=R9*X'145F' (card) . (Product of mX hi times 2/PI hi.)
        R9=R9+carry in . (Add for partial product hi.)
        R5=R5+RA, save carry . (Add for full product lo.)
        R9=R9+carry in . (Add for full product hi. Product of mX*(2/PI)
                          is now in (R9, R5). The radix point of the
                          product is 2 bit positions to the right of
                          the left edge of R9. Now add (2/PI)/2 to
                          account for the fact that mX is the true
                          X mantissa biased down by .5  .)
        R5=R5+X'306E', save carry . (Add 1/PI lo to lo part of product.)
        R9=R9+X'145F'+carry in . (Add 1/PI hi to hi part of product.
                                  Product of X true mantissa times
                                  (2/PI) now is in (R9, R5). Call this
                                  result "m2P".)
```

Register:	RE	RD	RC	RB	RA	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0
Use:	SX		2BE	-	-	m2P hi			-	m2P lo					-

```
        (The product of |X|*(2/PI) is called |R|. Now develop integer
        |R| from m2P and the exponent info of 2BE.)
4.9     If RC(10)=0   (|X| true exponent less than 16 test.)
               (True X exponent is less than 16.)
4.9.0          Continue.
            Else
               (True X exponent is at least 16 (but less than EXmax).)
4.9.1          R9=R5
```

```
                R5=0
         End (4.9) If

 4.B     RC=RC-(2*128-SHF)  . ((True X exponent)*2 +SHF is now in RC.
                                 RC=entry point to SHF table. SEE ATANF2
                                 FOR SHF TABLE DETAILS.)
         R8=0(RC)  . (Content of SHF table now in R8.)
         (R5, R6)=R5*R8  (card). (Shift lo.)
         (R8, R9)=R8*R9  (card). (Shift hi.)
         R9=R9 .OR. R5  . (Merge hi part of lo product into hi result.)
         RA=R6  . (Shifted product is in (R9, RA).)
Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
         |   |   |   |   |int|int|   |   |   |   |   |   |   |   |   |
     Use:| SX|   | - | - ||R|||R|| - |   | - | - |   |   |   |   | - |
         |                     hi|   |   |   |   |   |   |   |   |   |

     (The lead 2 bits of integer |R| (R9(0) & R9(1)) are integer
     bits and define the quadrant in which the angle lies. The most
     significant 3 fractional bits of |R| (R9(2), R9(3), & R9(4)) comprise
     the argument range index, "j",epolynomial to be used in approximating
     the output Y. The remaining bits of (R9, RA), appropriately aligned,
     are used as input into the polynomial routine.)

     (Fix output Y sign to account for quadrant; quadrant 2, 3 indicator
     is in R9(0).)
 4.D     RE=RE .EXCLUSIVE OR. R9  . (Sign of Y, SY, in RE(0); garbage
                                      elsewhere in RE.)
         RE=RE .AND. X'8000'  . (Sign of Y in RE(0); "0" elsewhere in
         RE.)
Register:| RE| RD| RC| RE| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
         |   |   |   |   |int|int|   |   |   |   |   |   |   |   |   |
     Use:| SY|   | - | - ||R|||R|| - |   | - | - |   |   |   |   | - |
         |   |   |   |   | lo| hi|   |   |   |   |   |   |   |   |   |

 4.F     If R9(1)=1      (Odd quadrant test.)
               (Odd quadrant 1 or 3 involved. Since the angular range
               to be approximated is 0 to less than 1 quarter rotations,
               the angle value that is the fractional part of int |R|
               must be subtracted from 1 (PI/2) for these quadrants.)
 4.F.0         RA=.NOT. RA  . (Subtract angle lo from 1 (unit=1/4
                               rotations).)
               R9=.NOT. R9  . (Subtract angle hi from 1 (unit=1/4
                               rotations). Radix point 2 bits right of
                               left edge of R9.))
            Else
               (Even quadrants 0 or 2 involved.)
 4.F.1         Continue.
         End (4.F) If
```

```
        (The approximation subinterval index, "j", is defined by the bits
        R9(2), R9(3), & R9(4) . The index "j" establishes the appropriate
        polynomial, "sj", to be used in finding |Y|.)
        (Isolate "j" and slide the lower bits (R9(5) thru RA(15)) to the
        left 5 bit positions so that they butt up against the left edge
        of R9. The result in (R9, RA) will be "U".)
 4.11   R4=R9    . (Replicate int |R| hi in R4.)
        (R4, R5)=R4*X'0020'. (Multiply int |R| hi in R4 by 32 in order
                             isolate quadrant/"j" data in R4; the lower
                             order bits remain, left justified, in
                             R5. They are all but the 3 top fractional
                             bits of int |R| hi.)
        (RA, RB)=RA*X'0020'. (Multiply int |R| lo in RA by 32 in order
                             to slide int |R| lo bits 5 bit positions
                             to the left; lo bits reside in RB.)
        R5=R5 .OR. RA   . (Merge Uhi bits in R5.)
        RA=RB   . (Ulo moved to RA.)
        R9=R5   . (Uhi moved to R9.)
        RD=0   . (Clear sign bit of U. U is positive only; radix point is
                 at left edge of R9.)
        (Create 4*j to be able to access polynomial info GET table.)
        R4=SHIFT(R4, +2)   . (Shift value in R4 left 2 bit positions.)
        R4=R4 .AND. X'001C'.(Mask out all but 4*j bits of R4; clear all
                                    but bits 11,...,13  .)
Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
         |   | 0|   |   |   |   |   |   |   |   |   |   |   |   |   |
     Use:| SY| SU| - | - | U | U | - |   | - |4*j|   |   |   |   | - |
         |                    hi|   |   |   |   |   |   |   |   |   |

     (R5 is used to get access to the appropriate polynomial, sj(U), for
        approximating ((SIN(Rfm*(PI/2)))/2)   .
        Note that Rfm=(j/8)+(U/8) and that  0 <= U < 1  .)
        (The use of 4*j to retrieve polynomial parameters follows.)
 4.13   R5=R5+GET   . (R5 points to polynomial degree number
                      in "GET" table.)
```

"GET" Table

j	k	4*j+k	Address	Value
0	0	4*0+0	GET+ 0	M(0)
0	2	4*0+2	GET+ 2	KOEF(0)+4*M(0)+2
1	0	4*1+0	GET+ 4	M(1)
1	2	4*1+2	GET+ 6	KOEF(1)+4*M(1)+2
2	0	4*2+0	GET+ 8	M(2)
2	2	4*2+2	GET+10	KOEF(2)+4*M(2)+2
3	0	4*3+0	GET+12	M(3)

```
                       3  2  4*3+2    GET+14   KOEF(3)+4*M(3)+2
                       4  0  4*4+0    GET+16         M(4)
                       4  2  4*4+2    GET+18   KOEF(4)+4*M(4)+2
                       5  0  4*5+0    GET+20         M(5)
                       5  2  4*5+2    GET+22   KOEF(5)+4*M(5)+2
                       6  0  4*6+0    GET+24         M(6)
                       6  2  4*6+2    GET+26   KOEF(6)+4*M(6)+2
                       7  0  4*7+0    GET+28         M(7)
                       7  2  4*7+2    GET+30   KOEF(7)+4*M(7)+2

          R7=0(R5); R5=R5+2  . (Load R7 with degree of selected
                               polynomial, M(j), held at address
                               p'inted to by value in R5. Then
                               bump "GET" table pointer, R5.)
          R8=0(R5)  . (Load R8 with KOEF(j)+4*M(j)+2  ; the last
                      16 pit address of the KOEF(j) block, the
                      coefficients of the polynomial, r(j), held
                      at address pointed to by value in R5.)
Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
         |   |  0|   |   |   |   |   |   |   |   |   |   |   |   |   |
     Use:| SY| SU| - | - |Ulo|Uhi|loc|cnt| - | - |   |   |   |   | - |

 4.15     BAL,RF    POLY32.  (Compute "r(j)" polynomial.)
Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
         |   |   |   |   |   |   |   |   |   |   |   |   | sj|   | sj|
     Use:| SY| - | - | - | - | - | - | - | - | - | - | - | hi| - | lo|

          (The value of sj(U) polynomial approximates

               (SIN(ARG))/2   where
                  ARG=(j/8+U/8)*(PI/2)  ,  j=0,...,7,  &
                                           0 <= U < 1     and
                  -.5 <= sj <= .5  .)

          (If R2(0) bit is 1 , the fi..' |Y| true exponent must be 1. If
          R2(0)is 0, exponents will be less than 1.
 4.17     If R2(0)=1    (|Y|=1 test.)
                (|Y|=1 branch direction. (SIN(Rfm*(PI/2)))/2  is equal to
                .5  ; argument of SIN function is 90 degrees.)
 4.17.0         R2=X'4080'  . (|Y| hi  in R2.)
                R0=X'0000'  . (|Y| lo  in R0.)
                R2=R2 .OR. RE  . (Merge sign bit into |Y|; Y results.)
                RE=0  . (Status bit.)
                RETURN via RF  .
Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
         | 0 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
     Use:| S | - | - | - | - | - | - | - | - | - | - | - |Yhi| - |Ylo|
          ***********************************************************
```

```
             Else
                (|Y| < 1 branch direction. (SIN(Rfm*(PI/2)))/2  is less
                than .5  ; argument of SIN function is less than 90
                degrees.)
4.17.1          Continue  .

          End (4.17) If

4.19      If R2(1)=0
                (|Y| < .5 branch direction. (SIN(Rfm*(PI/2)))/2  is less
                than .25  ; argument of SIN function is less than 30
                degrees.)
4.19.0          Continue  .

             Else
                (.5 <= |Y| < 1  branch direction. (SIN(Rfm*(PI/2)))/2
                is less than .5 but greater than or equal to .25;
                argument of SIN function is less than 90 degrees but
                greater than or equal to 30 degrees.)
4.19.1          (R2, R3)=R2*X'0200'  . (Shift |Y| mantissa kernal hi
                                        left 9 bit places.)
                (R0, R1)=R0*X'0200'  . (Shift |Y| mantissa kernal lo
                                        left 9 bit places.)
                R0=R0 .OR. R3  . (Merge true mantissa lo parts.)
                R2=R2+((128-1)*128)  . (Establish |Y| biased exponent.)
                R2=R2 .OR. RE  . (Merge sign bit into |Y|; Y results.)
                RE=0  . (Status bit=S.)
                RETURN via RF  .
Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
         | 0 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
     Use:| S | - | - | - | - | - | - | - | - | - | - | - |Yhi| - |Ylo|
           **********************************************************

          End (4.19) If

4.21      If R2(2)=0
                (|Y| < .25 branch direction. (SIN(Rfm*(PI/2)))/2  is less
                than .125  ; argument of SIN function is less than
                14.47751219  degrees.)
4.21.0          Continue  .

             Else
                (.25 <= |Y| < .5  branch direction. (SIN(Rfm*(PI/2)))/2
                is less than .25 but greater than or equal to .125;
                argument of SIN function is less than 30 degrees but
                greater than or equal to 14.47751219  degrees.)
4.21.1          (R2, R3)=R2*X'0400'  . (Shift |Y| mantissa kernal hi
                                        left 10 bit places.)
                (R0, R1)=R0*X'0400'   . (Shift |Y| mantissa kernal lo
                                        left 10 bit places.)
                R0=R0 .OR. R3   . (Merge true mantissa lo parts.)
                R2=R2+((128-2)*128)   . (Establish |Y| biased exponent.)
                R2=R2 .OR. RE   . (Merge sign bit into |Y|; Y results.)
                RE=0   . (Status bit=S.)
                RETURN via RF   .
Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
         | 0 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
     Use:| S | - | - | - | - | - | - | - | - | - | - | - |Yhi| - |Ylo|
         **********************************************************

         End (4.21) If

 4.23    If R2(3)=0
                (|Y| < .125 branch direction. (SIN(Rfm*(PI/2)))/2  is less
                than .0625  ; argument of SIN function is less than
                7.180755781  degrees.)
 4.23.0         Continue  .

            Else
                (.125 <= |Y| < .25 branch direction. (SIN(Rfm*(PI/2)))/2
                is less than .125 but greater than or equal to .0625;
                argument of SIN function is less than 14.47751219 degrees
                but greater than or equal to 7.180755781 degrees.)
 4.23.1         (R2, R3)=R2*X'0800'   . (Shift |Y| mantissa kernal hi
                                        left 11 bit places.)
                       )=R0*X'0800'   . (Shift |Y| mantissa kernal lo
                                        left 11 bit places.)
                R0=R0 .OR. R3   . (Merge true mantissa lo parts.)
                R2=R2+((128-3)*128)   . (Establish |Y| biased exponent.)
                R2=R2 .OR. RE   . (Merge sign bit into |Y|; Y results.)
                RE=0   . (Status bit=S.)
                RETURN via RF   .
Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
         | 0 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
     Use:| S | - | - | - | - | - | - | - | - | - | - | - |Yhi| - |Ylo|
         **********************************************************

         End (4.23) If


         (The 4 lead bits of "sj", R2(0), R2(1), R2(2), & R2(3)i,
         are "0"; the most likely result cases have now been examined.
         Now take care of the cases in which "sj" is very small;
         i.e.,  (SIN(Rfm*(PI/2)))/2  is less than .0625 or argument
```

```
        of SIN function is less than 7.180755781  degrees.)
        (First, define a new constant that eliminates the lead 4
        zero bits of "sj".)
 4.25   (R2, R3)=R2*X'0010'  . (Shift |Y| mantissa kernal hi left 4
                                bit places.)
        (R0, R1)=R0*X'0010'  . (Shift |Y| mantissa kernal lo left 4
                                bit places.)
        R3=R3 .OR. R0  . (Merge 4 bit left-shifted (SIN(Rfm*(PI/2)))/2
                         hi parts; radix point for the above is 3 bit
                         intervals left of the left edge of R3. Note
                         that R3(0)=0. The lo part of this value lies
                         in R1. Call this value "lsr" and consider its
                         radix point to lie at the left edge of R3.
                         Then, "lsr"=8*(SIN(Rfm*(PI/2)))  .)
Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
         ¦   ¦   ¦   ¦   ¦   ¦   ¦                  lsr¦   ¦lsr¦   ¦
     Use:¦ SY¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦ hi¦   ¦ lo¦ - ¦

 4.27   If R3=0
             (The new variable "lsr" is smaller than 2**(-16)  .)
             Shift "lsr" value left by 16 bit positions.)
 4.27.0      R3=R1  . (Shift "lsr" lo into "lsr" hi.)
             R1=0  . (Shift "0" into "lsr" lo.)
             RA=(128-1-3-16)*2  . (2*(|Y| biased exponent kernal) in RA.)
           Else
 4.27.1      RA=(128-1-3)*2  . ((2*(|Y| biased exponent kernal) in RA.)
        End (4.27) If

 4.29   If R5 < X'FF00'  . (Test mask for R3 lead bits, 8 bits wide.)
             (The variable in (R3, R1) is smaller than 2**(-8)  .)
             Shift value left by 8 bit positions.)
 4.29.0      (R3, R4)=R3*X'0100'  . (Shift hi left 8 places.)
             R3=R4  . (Put shifted value back into R3.)
             RC=8*2  . (Shift index is 8.)
           Else
 4.29.1      RC=0*2  . (Shift index is 0.)
        End (4.29) If

 4.2B   If R5 < X'F000'  . (Test mask for R3 lead bits, 4 bits wide.)
             (The variable in (R3, R1) is smaller than 2**(-4)  .)
             Shift value left by 4 bit positions.)
 4.2B.0      (R3, R4)=R3*X'0010'  . (Shift hi left 4 places.)
             R3=R4  . (Put shifted value back into R3.)
             RC=RC+4*2  . (Shift index delta is 4.)
           Else
 4.2B.1      Continue  .
        End (4.2B) If
```

```
4.2D      If R5 < X'C000'  . (Test mask for R3 lead bits, 2 bits wide.)
              (The variable in (R3, R1) is smaller than 2**(-2)  .)
              Shift value left by 2 bit positions.)
4.2D.0        (R3, R4)=R3*X'0004'  . (Shift hi left 2 places.)
              R3=R4  . (Put shifted value back into R3.)
              RC=RC+2*2  . (Shift index delta is 2.)
            Else
4.2D.1        Continue  .
          End (4.2D) If

4.2F      If R5 < X'8000'  . (Test mask for R3 lead bits, 1 bits wide.)
              (The variable in (R3, R1) is smaller than 2**(-1)  .)
              Shift value left by 1 bit position.)
4.2F.0        (R3, R4)=R3*X'0002'  . (Shift hi left 1 places.)
              R3=R4  . (Put shifted value back into R3.)
              RC=RC+1*2  . (Shift index delta is 1.)
            Else
4.2F.1        Continue  .
          End (4.2F) If

4.31      If R5=0
              (The variable in (R3, R1) is 0.)
4.31.0        R2=0  . (|Y|=0; R2=Yhi.)
              R0=0  . (|Y|=0; R2=Ylo.)
              RE=0  . (Status=S.)
              RETURN via RF  .
Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
         | 0 |   |   |   |   |   |   |   |   |   |   |   | 0 |   | 0 |
     Use:| S | - | - | - | - | - | - | - | - | - | - | - |Yhi| - |Ylo|
          **********************************************************

            Else
              (The variable in (R3, R1) is at least X'80000000'.)
4.31.1        RA=RA-RC  . (Unaligned biased |Y| exponent-1  is in RA.)
              (RA, RB)=RA*64  . ((Biased |Y| exponent-1)*128 in RB;
                                  aligned data.)
              RC=RC .OR. X'001F' . (Modulo 32 mask shift index times 2.)
              RC=RC+SHF  . (Entry to shift table value.)
              R6=0(RC)  . (Put shift value into R6.)
              (R1, R2)=R1*R6 . (Shifted & aligned lo part of (R3, R1).)
              (R3, R4)=R3*X'0100'  . (Aligned hi part of (R3, R1) is in
                                      R3; |Y| hi biased, aligned
                                      mantissa.)
              R1=R1 .OR. R4  . (Merge |Y| mantissa lo pieces; Ylo
                                results.)
              R3=R3+RB  . (Add aligned |Y| exponent data (short by 1)
                           to aligned mantissa data; lead mantissa bit
                           fixes exponent data.)
```

```
                R3=R3 .OR. RE  . (¦Y¦ hi merged with sign of Y yields
                                   Y hi.)
                R2=R3  . (Put Yhi into R2.)
                R0=R1  . (Put Ylo into R0.)
                RE=0  . (Status=S.)
                RETURN via RF  .
Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
         ¦ 0 ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
     Use:¦ S ¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦Yhi¦ - ¦Ylo¦
           *********************************************************

          End (4.31) If

    End (4) If

    END
```

3.2.3 MCU SUBROUTINE DESCRIPTION : COSM

This Subroutine develops the value, "Y", the cosine of the input variable, "X". "X", the input, and "Y", the output, are 32 bit VAX floating point numbers. Along with "Y", a 16 bit status value, S, is generated for output; it is 0 when |X| is less than 2**24. When |X| is equal to or greater than 2**24, the angular uncertainty is on the order of 2 radians and so the Y answer becomes meaningless; the status is set to 3 in such case (no processing is performed).

The "Y" value range is -1 <= Y <= +1 .

****

The subroutine demands the calculation of

```
Y=COS(X)=((-1)**SX)*COS(|X|)  where
          SX is the value of the sign bit of X, and
          |X| symbolizes the absolute value of X.
```

|Y| is computed using the SINF subroutine after PI/2 has been added to |X| and SX has been negated.

For all |X| values less than 2**24, |Y| is computed only after converting X to units of 1/4 rotations and then converting this resultant value, R, to an integer form. After negating SX and executing R=1+R, the new R takes on the role of R of the SINF routine. The SINF code is then used to complete the processing.

****

The entry branch and link register for this subroutine is RF. The subroutine calls the POLY32 subroutine by way of register RF. The POLY32 subroutine , in turn, calls the subroutine, MULT32.MS, as an internal subroutine (i.e., no BAL register is used) .

Registers directly required by this subroutine are marked with a "*".
Registers indirectly required by the POLY32 routine are marked with a "#".
Registers indirectly required by the MULT32.MS routine are marked with a "$" .

Register:	RE	RD	RC	RB	RA	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0
COSF Use:	*	*	*	*	*	*	*	*	*	*		*	*	*	*
POLY32 Use:		#			#	#	#	#	#				#		#
MULT32.MS Use:		$			$	$					$	$	$	$	$

ON ENTRY:
R9=Xlo
RB=Xhi

ON EXIT:
R0=Ylo
R2=Yhi

```
*************************************************************************

                                COSM entry

*************************************************************************


 1. COSM entry  .
Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
         ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
     Use:¦   ¦   ¦   ¦Xhi¦   ¦X1o¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦

 2        RE=X'8000'  . (Prepare to capture X sign in RE(0); clear other bits.)
          RE=RE .AND. RB  . (Sign of X in RE; other RE bits cleared.)
          RB=RB .EXCLUSIVE OR. RE  . (X changed to a magnitude, ¦X¦.)
Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
         ¦              ¦¦   ¦¦X¦¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
     Use:¦ SX¦   ¦   ¦ hi¦   ¦ lo¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦

 3        If RB => (128+EXmax)*128    (EXmax=25. ¦X¦ greater than or equal to
                                       2**24 test.)
                (¦X¦ => 2**24; too much angular uncertainty results. Return
                with status value of 3.)
 3.0            RE=3  . (Status=S=3. Worthless results exist in Yhi and Ylo
                         registers.)
                RETURN via RF  .
Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ P2¦ R1¦ R0¦
         ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
     Use:¦ S ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
           *********************************************************


             Else
                (¦X¦ < 2**24 along this path.)
 3.1            Continue.

          End (3) If

 4   If RB => (128-12)*128    (¦X¦ greater than or equal to 2**(-13) test.)
                (Y not equal to 1.)
 4.0            Continue.
             Else
                (Y = 1.)
 4.1            R2=X'4080'  . (Y hi.)
                R0=X'0000'  . (Y lo.)
                RE=0  . (Status=S.)
                RETURN via RF  .
Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
         | 0 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
     Use:| S |   |   |   |   |   |   |   |   |   |   |   |Yhi|   |Ylo|
     ****************************************************************

          End (4) If

          (|X| must be converted to 1/4 rotation angular units; then
          the result will be converted to integer form. The fractional
          part of this integer form will be used to find SIN(|X|.)
          (Isolate |X| biased exponent (BEX) and biased mantissa (mx).)
 5        (RB, RC)=RB*X'0200'  . (|X| biased mantissa hi left justified in RC;
                                  |X| biased exponent, BEX, is in RB, right
                                  justified.)
          (R9, RA)=R9*X'0200'  . (|X| biased mantissa lo left justified in
                                  (R9, RA).)
          R9=R9 .OR. RC  . (|X| biased mantissa hi merged into R9.)
Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
         |   |   |   |   | mX| mX|   |   |   |   |   |   |   |   |   |
     Use:| SX|   |   |BEX| lo| hi|   |   |   |   |   |   |   |   |   |

          (Multiply BEX by 2.)
 6        (RB, RC)=RB*2  . (Two times BEX, 2BE, is now in RC.)
Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
         |   |   |   |   | mX| mX|   |   |   |   |   |   |   |   |   |
     Use:| SX|   |2BE| - | lo| hi|   |   |   |   |   |   |   |   |   |

          (Multiply the X true mantissa times 2/PI. Note that mX is
          the biased mantissa and that the radix point of mX is
          1 bit position left of the left edge of R9. The value 2/PI
          is X'145F306E'*(2**(-13)).)
 7        R5=R9  . (Replicate mX hi in R5.)
          (RA, RB)=RB*X'145F' (card) . (Product of mX lo times 2/PI hi.)
          (R5, R6)=R5*X'306E' (card) . (Product of mX hi times 2/PI lo.)
          R5=R5+RA. save carry . (Add for partial product lo.)
          (R9, RA)=R9*X'145F' (card) . (Product of mX hi times 2/PI hi.)
          R9=R9+carry in . (Add for partial product hi.)
          R5=R5+RA, save carry . (Add for full product lo.)
          R9=R9+carry in . (Add for full product hi. Product of mX*(2/PI)
                            is now in (R9, R5). The radix point of the
                            product is 2 bit positions to the right of
                            the left edge of R9. Now add (2/PI)/2 to
                            account for the fact that mX is the true
                            X mantissa biased down by .5  .)
          R5=R5+X'306E', save carry . (Add 1/PI lo to lo part of product.)
          R9=R9+X'145F'+carry in . (Add 1/PI hi to hi part of product.
                                    Product of X true mantissa times
                                      /PI) now is in (R9, R5). Call this
                                    result "m2P". Radix point of "m2P" is
                                    between R9(1) and R9(2).)
Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0
         |           |   |   |m2P|   |   |   |m2P|   |   |   |   |   |
     Use:| SX|   |2BE| - | - | hi|   |   | - | lo|   |   |   |   |   |

          (The product of |X|*(2/PI) is called |R|. Now develop integer
          |R| from m2P and the exponent info of 2BE.)
 8        If RC < 128*2       (Negative |X| exponent test.)
            (|X| true exponent is negative.)
            (Consider that m2P has been multiplied by 2**(-16) and exists
            in (RA, R9, R5) where the radix point is between RA(1) and
            RA(2). To make m2P an integer, the number in the three register
            set must be shifted left by an amount  j=|X| true exponent+16.
            The value SHF+2*j points to the correct shift value to be
            pulled from the SHF table.)
            (Proceed to develop m2P as an integer.)
 8.0        RC=RC-((128-16)*2-SHF)  . (RC points to shift value.)
            R8=0(RC)  . (Shift constant into R8.)
            (R9, RA)=R9*R8 (card)  . (Output hi.)
            (R5, R6)=R5*R8 (card)  . (Output lo.)
            RA=RA .OR. R5  . (Merge lo parts.)

            (Note that (R9, RA, R6) took on the role of the starting
            3 registers, (RA, R9, R5), in the 3 preceding steps.)
            (Add 1 (1/4 rotation) to the value in (R9, RA, R6). The
            radix point of this value lies between R9(1) and R9(2).)
 8.2        R9=R9+X'4000'  . (Quarter rotation has been added. Call result
                                  int R.)
Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
         |   |   |   |   |int|int|   |   |   |   |   |   |   |   |   |
     Use:| SR|   | - | - | lo| hi| - |   | - | - |   |   |   |   |   |

 8.4            BAL,RF   SINF3., 4.D  .
                RETURN via RF  .
Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
         | O |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
     Use:| S | - | - | - | - | - | - | - | - | - | - | - |Yhi| - |Ylo|
      *****************************************************************

             Else
```

```
                (|X| true exponent is 0 or positive.)
Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
         |   |   |   |   |   |m2P|   |   |   |m2P|   |   |   |   |   |
     Use:| SX|   |2BE| - | - | hi|   |   | - | lo|   |   |   |   |   |

 8.1            If RC(10)=0    (Exponent less than 16 test.)
                    (True X exponent is less than 16.)
 8.1.0              Continue.
                  Else
                    (True X exponent is at least 16 (but less than EXmax).)
 8.1.1              R9=R5
                    R5=0
                End (8.1) If

 8.3            RC=RC-(2*128-SHF). ((True X exponent)*2 +SHF is now in RC.
                                    RC=entry point to SHF table. SEE ATANF2
                                    FOR SHF TABLE DETAILS.)
                R8=0(RC)  . (Content of SHF table now in R8.)
                (R5, R6)=R5*R8  (card). (Shift lo.)
                (R8, R9)=R8*R9  (card). (Shift hi.)
                R9=R9 .OR. R5. (Merge hi part of lo product into hi result.)
                RA=R6  . (Shifted product is in (R9, RA).)

 8.5            R9=R9+X'4000'. (Quarter rotation has been added. Call result
                                int R.)
Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
         |   |   |   |   |int|int|   |   |   |   |   |   |   |   |   |
     Use:| SX|   | - | - ||R|||R|| - |   |   | - | - |   |   |   | - |
         |   |   |   |   | lo| hi|   |   |   |   |   |   |   |   |   |

 8.7            BAL,RF   SINF3., 4.D  .
                RETURN via RF  .
Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
         | 0 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
     Use:| S | - | - | - | - | - | - | - | - | - | - | - |Yhi| - |Ylo|

           End (8) If

      ****************************************************************

 END
```

3.2.4 MCU SUBROUTINE DESCRIPTION : ATANM

This Subroutine develops the value, "Y", the arctangent of the input variable, "X". "X", the input, and "Y", the output, are 32 bit VAX floating point numbers. Along with "Y", a 16 bit status value, S, is generated for output; it should always be 0. The "Y" value range is (- PI/2) <= Y <= (PI/2) .

****

The subroutine demands the calculation of

```
Y=ATAN(X)=((-1)**SX)*ATAN(|X|)  where
          SX is the value of the sign bit of X, and
          |X| symbolizes the absolute value of X.
```

|Y| must be computed using a number of different approximations. For true X exponents (TEX) less than 2, the |X| interval is dissected into 5 different sub-intervals, namely,

```
      1)              TEX < (-11)      or                |X| < 2**(-12)  ,
      2)      (-11) <= TEX < ( -5)     or     2**(-12) <= |X| < 2**( -6)  ,
      3)      ( -5) <= TEX < (  0)     or     2**( -6) <= |X| < 2**( -1)  ,
      4)              TEX = (  0)      or     2**( -1) <= |X| < 2**(  0)  ,
and   5)              TEX = (  1)      or     2**(  0) <= |X| < 2**(  1)  .
```

For true X exponents greater than or equal to 2, |Y| is computed using |Y|=(PI/2)-ATAN(1/|X|) . To find |Y| in such case, 1/|X| is first computed. Then the angle approximation associated with one of the 5 |X| sub-intervals shown above is used to compute ATAN(1/|X|). Finally, this result is subtracted from PI/2 .
The specific approximations for the 5 |X| sub-intervals shown above are listed below. The 8 approximations used to approximate the reciprocal are then listed.

APPROXIMATIONS

Sub-interval 1)

When the true exponent of X is less than -11,

|Y|=|X|.

Sub-interval 2)

When the true exponent of X is less than -5 but greater than or equal to -11, |Y| is given by

|Y|=|X|-(|X|**3)/3 and so by

|Y|=|X|*(1-(1/3)*X2) where
X2=|X|**2.

POLY32, the polynomial expansion routine, is not used to compute this approximation.

Sub-interval 3)

When the true exponent of X is less than 0 but greater than or equal to -5, |Y| is found using

|Y|=|X|*(1-G(|X|)) where
G(|X|)=1-(ATAN(|X|))/|X|

The value of G(|X|) is established using the polynomial, "p1(U)", given by

p1(U)=A10*(U**0)+A11*(U**1)+A12*(U**2)+A13*(U**3)+.....+A1N1*(U**N1) where
U=|X| ,
the |X| range is 1/64 <= |X| < .5 ,
the U range is 1/64 <= U < .5 , and
p1 approximates G(|X|) .

The polynomial "p1" is computed from right to left using

p1(U)=A10+U*(A11+U*(A12+U*(A13+U*(A14+U*(A15+U*(A16+.....+U*(A1N) .

The POLY32 routine used to compute p1 assumes that 1/64 <= U < 1/2 and "U" has the signed magnitude format [S, (0.31.0)]*2**(-32) and that p1 lies within the range -1/4 <= p1 < 1/4 (it does) and has the 2's complem format (1.31.0)*2**(-32) .

The starting location of the memory space that stores the "A1" coefficients needed to compute p1 is COEF1. The coefficient data are assumed stored in the sequence:

Address	Item
COEF1+ 0	A10(hi)
COEF1+ 2	A10(lo)

COEF1+ 4	A11(hi)
COEF1+ 6	A11(lo)
COEF1+ 8	A12(hi)
COEF1+ 10	A12(lo)
COEF1+ 12	A13(hi)
COEF1+ 14	A13(lo)
.	.
.	.
.	.
COEF1+4*N1	A1N(hi)
COEF1+4*N1+2	A1N(lo) .

N1, the degree of the "p1" polynomial, and the COEF1 coefficient data are defined within this subroutine.

Once p1 is computed, the output ¦Y¦ value is given by

¦Y¦=¦X¦*(1-p1) .

Sub-interval 4)

When the true exponent of X is 0, ¦Y¦ is found using

¦Y¦=p2(U)+.5 where
U=2*¦X¦-1.5 ,

The polynomial, "p2(U)", is defined by

p2=A20*(U**0)+A21*(U**1)+A22*(U**2)+A23*(U**3)+......+A2N1*(U**N1) where
the ¦X¦ range is 1/2 <= ¦X¦ < 1.0 ,
the U range is -.5 <= U < .5 , and
p2(U) approximates .5*(ATAN((U+1.5)/2))-.5 .

The polynomial "p2" is computed from right to left using

p2(U)=A20+U*(A21+U*(A22+U*(A23+U*(A24+U*(A25+U*(A26+......+U*(A2N) .

The POLY32 routine used to compute p2 assumes that -1/2 <= U < 1/2 and "U" has the signed magnitude format [S, (0.31.0)]*2**(-32) and that p2 lies in the range -1/4 <= p2 < 1/4 (it does) and has the 2's complement format (1.31.0)*2**(-32) .

The starting location of the memory space that stores the "A2" coefficients needed to ccmpute p2 is COEF2. The coefficient data are assumed stored in the sequence:

Address Item

COEF2+ 0	A20(hi)
COEF2+ 2	A20(lo)
COEF2+ 4	A21(hi)
COEF2+ 6	A21(lo)
COEF2+ 8	A22(hi)
COEF2+ 10	A22(lo)
COEF2+ 12	A23(hi)
COEF2+ 14	A23(lo)
.	.
.	.
.	.
COEF2+4*N2	A2N(hi)
COEF2+4*N2+2	A2N(lo) .

N2, the degree of the "p2" polynomial, and the COEF2 coefficient data are defined within this subroutine.

Once p2 is computed, the output |Y| value is given by

```
   |Y|=2*(p2+.5)  .
```

Sub-interval 5)

When the true exponent of X is 1, |Y| is found using

```
   |Y|=2*(p3(U)+.5)  where
       U=|X|-1.5  ,
```

The polynomial, "p3(U)", is defined by

```
p3(U)=A30*(U**0)+A31*(U**1)+A32*(U**2)+A33*(U**3)+.....+A3N1*(U**N1)  where
   the |X| range is      1.0 <= |X| < 2.0  ,
   the U range is     -.5 <= U < .5  , and
   p3(U) approximates      .5*(ATAN(U+1.5)-.5  .
```

The polynomial "p3" is computed from right to left using

```
p3=A30+U*(A31+U*(A32+U*(A33+U*(A34+U*(A35+U*(A36+.....+U*(A3N)  .
```

The POLY32 routine used to compute p3 assumes that -1/2 <= U < 1/2 and "U" has the signed magnitude format [S, (0.31.0)]*2**(-32) and that p3 lies in the range -1/4 <= p3 < 1/4 (it does) and has the 2's complement format (1.31.0)*2**(-32) .

The starting location of the memory space that stores the "A3" coefficients needed to compute p3 is COEF3. The coefficient data are assumed stored in the sequence:

Address	Item
COEF3+ 0	A30(hi)
COEF3+ 2	A30(lo)
COEF3+ 4	A31(hi)
COEF3+ 6	A31(lo)
COEF3+ 8	A32(hi)
COEF3+ 10	A32(lo)
COEF3+ 12	A33(hi)
COEF3+ 14	A33(lo)
.	.
.	.
.	.
COEF3+4*N3	A3N(hi)
COEF3+4*N3+2	A3N(lo) .

N3, the degree of the "p3" polynomial, and the COEF3 coefficient data are defined within this subroutine.

Once p3 is computed, the output |Y| value is given by

```
   |Y|=2*(p3+.5)  .
```

****

Reciprocal

When the true exponent of X is 2 or greater, |Y| is found by first computing 1/|X|. The reciprocal of |X|, |R|, is defined by

```
   |R|=(1/(4*mX))*(2**(-TEX+2)) where
       mX here is the true mantissa of X, and
       TEX is the true exponent of X.
```

The term 1/(4*mX) is approximated using 8 different polynomials. Each polynomial corresponds to 1 of 8 equal sub-intervals into which the mX range .5 <= mX < 1.0 , is divided. The "j"th (j=0, 1,..., 7) sub-interval approximation of (1/(4*mX)) is described by the polynomial rj(U) where U is related to mX using

```
   mX=.5+j/16+(U*2)   where
      0 <= U < 1/32  .
```

The polynomial, "rj(U)", is defined by

```
rj(U)=Bj0*(U**0)+Bj1*(U**1)+Bj2*(U**2)+Bj3*(U**3)+.....+BjMj*(U**Mj)  where
   the mX range is      .5 <= |X| < 1.0  ,
```

the U range is 0 <= U < 1/32 , and
rj(U) approximates (1/(4*mX)) .

The polynomial "rj" is computed from right to left using

rj=Bj0+U*(Bj1+U*(Bj2+U*(Bj3+U*(Bj4+U*(Bj5+U*(Bj6+......+U*(BjMj) .

The POLY32 routine used to compute rj assumes that 0 <= U < 1/32 and "U" has the signed magnitude format [S, (0.31.0)]*2**(-32) and that rj lies in the range 0 <= rj <= 1/2 (it does) and has the 2's complement format (1.31.0)*2**(-32) .

The starting location of the memory space that stores the "Bj" coefficients needed to compute rj is KOEFj. The coefficient data are assumed stored in the sequence:

Address	Item
KOEFj+ 0	Bj0(hi)
KOEFj+ 2	Bj0(lo)
KOEFj+ 4	Bj1(hi)
KOEFj+ 6	Bj1(lo)
KOEFj+ 8	Bj2(hi)
KOEFj+ 10	Bj2(lo)
KOEFj+ 12	Bj3(hi)
KOEFj+ 14	Bj3(lo)
.	.
.	.
.	.
KOEFj+4*N3	BjN(hi)
KOEFj+4*N3+2	BjN(lo) .

Mj, the degree of the "rj" polynomial, and the KOEFj coefficient data are defined within this subroutine.

Once rj is computed, the output 1/¦X¦ value is given by

1/¦X¦=rj*(2**(-TEX+2)) .

****
For the "p1", "p2", "p3" and "rj" polynomials above, coefficients of the polynomial have the 2's complement format (1.31.0)*2**(-32) , the same format as is used for the polynomial value.

The degree of the polynomials is at least 1.

The entry branch and link register for this subroutine is RF. The subroutine calls the POLY32 subroutine by way of register RF. The POLY32 subroutine , in turn, calls the subroutine, MULT32.MS, as an internal subroutine (i.e., no BAL register is used) .

Registers directly required by this subroutine are marked with a "*".
Registers indirectly required by the POLY32 routine are marked with a "#".
Registers indirectly required by the MULT32.MS routine are marked with a "$".

Register:	RE	RD	RC	RB	RA	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0
ATANM Use:	*	*	*	*	*	*	*	*	*	*		*	*	*	*
POLY32 Use:		#			#	#	#	#	#				#		#
MULT32 Use:		$			$	$					$	$	$	$	$

ON ENTRY:
R9=Xlo
RB=Xhi

ON EXIT:
R0=Ylo
R2=Yhi

****

ATANF entry

****

```
1. ATANF entry   .
    Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
             ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
         Use:¦   ¦   ¦   ¦Xhi¦   ¦Xlo¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦


2. R0=X'4000'   .  (Capture complement of true exponent sign bit in R0.)
    Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
             ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
         Use:¦   ¦   ¦   ¦Xhi¦   ¦Xlo¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦Use¦

3. R0=R0 .AND. RB  (Complement of sign of X true exponent in R0 after step.)
    Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
             ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦___¦
         Use:¦   ¦   ¦   ¦Xhi¦   ¦Xlo¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦SEX¦


4. IF R0=0  (Split processing based on true exponent (-) or (+,0).)
    Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
             ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
         Use:¦   ¦   ¦   ¦Xhi¦   ¦Xlo¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦ - ¦

         (The true X exponent must lie in the range, -128 <= EXP-128 <= -1   ,
         to branch in this direction.
         Develop sign bit in R2(0) and "0" in other 15 bits.)
4.0      R2=X'8000'   .
         R2=R2 .AND. RB   . (Sign bit now in R2(0).)
    Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R6¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
             ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
         Use:¦   ¦   ¦   ¦Xhi¦   ¦Xlo¦   ¦   ¦   ¦   ¦   ¦   ¦ SX¦   ¦ - ¦

4.2      RB=RB .EXCLUSIVE OR. R2   . (Zero's out X sign bit in RB;
                                       X becomes ¦X¦.)
    Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
             ¦   ¦   ¦   ¦¦X¦¦   ¦¦X¦¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
         Use:¦   ¦   ¦   ¦ hi¦   ¦ lo¦   ¦   ¦   ¦   ¦   ¦   ¦ SX¦   ¦ - ¦

         (Determine if X true exponent is less than -11.)
4.4      R0=RB   . (Replicate ¦X¦ high.)
```

```
          R0=R0-((128-11)*128)  . (Remove exponent bias; add 11 to result.)
          R0=R0 .AND. X'FF80'  . (Clear mantissa bits out of R0; R0 value is
                                  (true X exponent + 11)*128  ; final result
                                  is T1.)
     Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
              ¦   ¦   ¦   ¦¦X¦¦   ¦¦X¦¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
          Use:¦   ¦   ¦   ¦ hi¦   ¦ lo¦   ¦   ¦   ¦   ¦   ¦   ¦ SX¦   ¦ T1¦

4.6       If R0=negative
                (X true exponent is less than -11,  T1 < 0; Y=X.)
4.6.0           R2=R2 .OR. RB . (Re-insert sign bit into X high in R2. Yhi=Xhi.
                R0=R9  . (Ylo=Xlo.)
                RE=0  . (Set status.)
                RETURN  (by way of RF).
     Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
              ¦   ¦   ¦   ¦¦X¦¦   ¦¦X¦¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
          Use:¦  0¦   ¦   ¦ hi¦   ¦ lo¦   ¦   ¦   ¦   ¦   ¦   ¦Yhi¦   ¦Ylo¦
           ****************************************************************

             Else
                (X true exponent is greater than or = to -11 but less than 0.
                (0 <= T1 < 11).)
     Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
              ¦   ¦   ¦   ¦¦X¦¦   ¦¦X¦¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
          Use:¦   ¦   ¦   ¦ hi¦   ¦ lo¦   ¦   ¦   ¦   ¦   ¦   ¦ SX¦   ¦ T1¦

4.6.1           (R0, R1)=R0*X'0400' (card). (Let E2B=2*(X true exponent + 11);
                                             put E2B into R0.)
     Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
              ¦   ¦   ¦   ¦¦X¦¦   ¦¦X¦¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
          Use:¦   ¦   ¦   ¦ hi¦   ¦ lo¦   ¦   ¦   ¦   ¦   ¦   ¦ SX¦ - ¦E2B¦

                (Put X true mantissa magnitude into RC, RA.)
4.6.3           (RB, RC)=RB*X'0100' (card) . (Put X true mantissa (high) into
                                              RC; radix point is on left edge
                                              of RC.)
                (R9, RA)=R9*X'0100' (card) . (Put X true mantissa (low) into
                                              R9, RA.)
                RC=RC .OR. R9  . (Concatenate high mantissa bits.)
                RC=RC .OR. X'8000'  . (Insert lead "1" bit into X mantissa.)
     Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
              ¦   ¦   ¦ mX¦   ¦ mX¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
          Use:¦   ¦   ¦ hi¦ - ¦ lo¦ - ¦   ¦   ¦   ¦   ¦   ¦   ¦ SX¦ - ¦E2B¦

             (Replicate true mantissa (magnitude); put (RC,RA) into (R5,R3).)
```

```
4.6.5          R5=RC  .
               R3=RA  .
    Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
             |           |   | mX|   |   |   |   | mX|   | mX|   |   |   |
         Use:|   |   | hi| - | lo| - |   |   |   | hi|   | lo| SX| - |E2B|
```

(Make integer X magnitude. Its radix point is to be at the left edge of R6. If R6, R5, and R3 are considered contiguous, then the true X mantissa magnitude has been multiplied by 2**(-16). Instead, to make |X| an integer, it should have been multiplied by 2**(tru exp). Thus, the (R6, R5, R3) value should be multiplied by 2**(tru exp + 16) or, since the R0 value divided by 2 is (tru exp + 11), by 2**((R0value/2)+5). The values of R0 range from 2*0 to 2*10. For the various values of R0, the values of 2**((R0value/2)+5) will be pulled from the MCU memory table, SHF . The table follows:

SHF Table

Location	R0	2**(R0/2)
SHF + 0	2*0	X'0001'
SHF + 2	2*1	X'0002'
SHF + 4	2*2	X'0004'
SHF + 6	2*3	X'0008'
SHF + 8	2*4	X'0010'
SHF +10	2*5	X'0020'
SHF +12	2*6	X'0040'
SHF +14	2*7	X'0080'
SHF +16	2*8	X'0100'
SHF +18	2*9	X'0200'
SHF +20	2*10	X'0400'
SHF +22	2*11	X'0800'
SHF +24	2*12	X'1000'
SHF +26	2*13	X'2000'
SHF +28	2*14	X'4000'
SHF +30	2*15	X'8000'

```
4.6.7          R0=R0+SHF5   . (SHF5=SHF+2*5  . Point to correct location in
                              shift table. PNT is the pointer value.
                              E+S=2*(true |X| exponent+11)+SHF5.)
               R1=0(R0)   . (Put value in memory location referenced by R0 into
                             R1.)
    Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
             |   |   | mX|   | mX|   |   |   |   | mX|   | mX|   |   |   |
         Use:|   |   | hi| - | lo| - |   |   |   | hi|   | lo| SX|PNT|E+S|

4.6.9          (R5, R6)=R5*R1 (card)  . (Shift X true mantissa high.)
               (R3, R4)=R3*R1 (card)  . (Shift X true mantissa low.)
```

```
                   R3=R3 .OR. R6   . (Concatenate low parts of integer |X|; integer
                                      |X| is in (R5, R3) after step.)
    Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
             |   |   | mX|   | mX|   |   |   |   |int|   |int|   |   |   |
         Use:|   |   | hi| - | lo| - |   |   | - ||X|| - ||X|| SX|PNT|E+S|
             |   |   |   |   |   |   |   |   |   | hi|   | lo|   |   |   |

                   (Split processing where X true exponent is less than -5.)
4.6.B              R0=R0-(SHF5+2*(11-5)) . (R0/2=tru exp + 5; E25=2*(tru exp + 5).)
    Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
             |   |   | mX|   | mX|   |   |   |   |int|   |int|   |   |   |
         Use:|   |   | hi| - | lo| - |   |   | - ||X|| - ||X|| SX| - |E25|
             |   |   |   |   |   |   |   |   |   | hi|   | lo|   |   |   |

4.6.D              If R0=negative
                      (X true exponent is less than -5 but greater than or
                      equal to -11. Use Y=X*(1-(1/3)*X**2) for computing Y.
                      Square integer |X| 1st.)
4.6.D.0               (R3, R4)=R3*R5 (card)  . (Multiply lo times hi (int X).)
                      (R5, R6)=R5*R5 (card)  . (Multiply hi times hi (int X).)
                      R6=R6+R3 ,save carry out  . (Partial square lo.)
                      R5=R5+carry in  . (Partial square hi.)
                      R3=R3+R6 ,save carry out  . (Full square lo.)
                      R5=R5+carry in  . (Full square hi.)
    Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
             |   |   | mX|   | mX|   |   |   |   |int|   |int|   |   |   |
         Use:|   |   | hi| - | lo| - |   |   | - ||X|| - ||X|| SX| - |E25|
             |   |   |   |   |   |   |   |   |   |sqr|   |sqr|   |   |   |
             |   |   |   |   |   |   |   |   |   | hi|   | lo|   |   |   |

                      (Square of integer |X| is in (R5, R3); radix point is at
                      left edge of R5. Multiply square times (1/3)=.3333... and
                      call the result "D" . Then subtract "D" (.3333...*X**2)
                      from 1 and call the result C.)
4.6.D.2               (R3, R4)=R3*X'5555' (card)  . (Multiply lo (int X sqr)
                                                    times both (1/3) hi and
                                                    (1/3) lo. The hi and lo
                                                    part of (1/3)=X'5555';
                                                    radix point of (1/3) hi
                                                    is at left edge of
                                                    at X'5555'.)
                      (R5, R6)=R5*X'5555' (card)  . (Multiply hi (int X sqr)
                                                    times both (1/3) hi and
                                                    (1/3) lo.)
                      R6=R6+R5 ,save carry out  . (Partial multiply, lo.)
                      R5=R5+carry in  . (Partial multiply hi.)
                      R3=R3+R6 ,save carry out  . (Full multiply, Dlo.)
                      R5=R5+carry in  . (Full multiply, Dhi.)
    Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
             |   |   | mX|   | mX|   |   |   |   |   |   |   |   |   |   |
        Use:|   |   | hi| - | lo| - |   |   | - |Dhi| - |Dlo| SX| - |E25|

                    ("C" is now in (R5, R3). Develop C=1-D=1-(X**2)/3  .)
4.6.D.4             R5=.NOT. R5  . (1-(X**2)/3 hi.)
                    R3=.NOT. R3  . (1-(X**2)/3 lo. "C" is now in (R5, R3).)
    Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
             |   |   | mX|   | mX|   |   |   |   |   |   |   |   |   |   |
        Use:|   |   | hi| - | lo| - |   |   | - |Chi| - |Clo| SX| - |E25|

                    (Multiply the X true mantissa times "C" to get the value,
                    mX*(1-(X**2)/3). Call the result mV. Put into (R5, R3).)
4.6.D.6             (RA, RB)=RA*R5 (card)  . (Multiply mXlo times Chi.)
                    (R5, R6)=R5*RC (card)  . (Multiply Chi times mXhi.)
                    RA=RA+R6 ,save carry out  . (Partial multiply, lo.)
                    R5=R5+carry in  . (Partial multiply hi.)
                    (R3, R4)=R3*RC (card)  . (Multiply Clo times mXhi.)
                    R3=R3+RA ,save carry out  . (Full multiply; mVlo.)
                    R5=R5+carry in  . (Full multiply; mVhi.)
    Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
             |   |   |   |   |   |   |   |   |   | mV|   | mV|   |   |   |
        Use:|   |   | - | - | - | - |   |   | - | hi| - | lo| SX| - |E25|

                    (If R5 lead bit is 1 (i.e., if it looks negative, the
                    value is in the mantissa range of the output Y. Else,
                    mV is a hair below .5 and needs to be multiplied by 2.
                    Fix and produce output.)
4.6.D.8             If R5=negative
                        (Pseudonegative branch direction; really, the mV
                        magnitude is = to or > .5  .)
4.6.D.8.0               R0=R0+2*(128-5)  (Biased exponent of Y times 2;
                                          need to multiply it by 64.)
                        (R5, R6)=R5*256  (Y true mantissa hi 8 bits
                                          properly positioned in R5.)
                        (R3, R4)=R3*256  (Y true mantissa lowest 16 bits
                                          properly positioned in R3.)

                      Else
4.6.D.8.1               R0=R0+2*(128-6)  (Biased exponent of Y times 2;
                                          need to multiply it by 64.)
                        (R5, R6)=R5*512  (Y true mantissa hi 8 bits
                                          properly positioned in R5.)
                        (R3, R4)=R3*512  (Y true mantissa lowest 16 bits
                                        properly positioned in R3.)

                    End (4.6.D.8) If.
```

```
                        (Fix exponent.)
4.6.D.A                 (R0, R1)=R0*X'0040'  . (Properly positioned biased
                                                Y exponent is now in R1.)
                        (Fix mantissa.)
                        R3=R3 .OR. R6  . (Merge lo bits of Y mantissa; Ylo.)
                        R5=R5 .AND. X'FF7F'  (Clear lead mantissa bit.)
                        R2=R2 .OR. R5  (Sign and biased mantissa merge.)
                        R2=R2 .OR. R1  (Merge biased Y exponent into Yhi.)
                        R0=R3          (Move Ylo into R1.)
                        RE=0  . (Status.)
                        RETURN  (by way of RF).
    Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
             |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
         Use:| - | - | - | - | - | - | - | - | - | - | - | - |Yhi| - |Ylo|
         ****************************************************************


                      Else
                        (X true exponent is less than 0 but greater than or
                        equal to -5. Use Y=X*(1-G(X)) for computing Y.)
    Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
             |           |   | mX|   |   |   |   |int|   |int|   |   |   |
         Use:|   |   | hi| - | lo| - |   |   | - ||X|| - ||X|| SX| - |E25|
             |   |   |   |   |   |   |   |   |   | hi|   | lo|   |   |   |

4.6.D.1                 R6=RA   . (Save X true mantissa low in R6.)
                        RB=R0   . (Save 2*(X true exponent + 5) in RB.)
                        RE=R2   . (Save X sign in RE.)
                        R9=R5   . (Move int X hi (Uhi) into R9.)
                        RA=R3   . (Move int X lo (Ulo) into RA.)
                        RD=0    . (Load sign bit of U, a magnitude, into RD.)
                        R7=N1   (Set degree for calculating "p1" where
                                 p1=G(X)=(1-ATAN(U)/U)  .)
                        R8=COEF1+4*N1+2  (Set to last 16 bit address of COEF1
                                          block.)
    Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
             |       | mX|   |int|int|   |   | mX|   |   |   |   |   |   |
         Use:| SX| SU| hi|E25||X|||X||loc|cnt| lo| - | - | - | - | - | - |
             |   |   |   |   | lo| hi|   |   |   |   |   |   |   |   |   |
             |   |   |   |   | = | = |   |   |   |   |   |   |   |   |   |
             |   |   |   |   |Ulo|Uhi|   |   |   |   |   |   |   |   |   |

4.6.D.3                 BAL,RF    POLY32.  (Compute "p1" polynomial.)
    Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
             |   |  0| mX|   |int|int|   |   | mX|   |   |   | p1|   | p1|
         Use:| SX| SU| hi|E25||X|||X|| - | - | lo| - | - | - | hi| - | lo|
             |   |   |   |   | lo| hi|   |   |   |   |   |   |   |   |   |
             |   |   |   |   | = | = |   |   |   |   |   |   |   |   |   |
             |   |   |   |   |Ulo|Uhi|   |   |   |   |   |   |   |   |   |
                       (Generate C=(1-G(X)) .)
4.6.D.5                R2=.NOT. R2  . (G(X)=p1. Radix point of p1 is at left
                                      edge of R2. Complement of (R2, R0)
                                      is (1-G(X)) as an unsigned number; Chi.)
                       R0=.NOT. R0  . (Low part of (1-G(X)); Clo.)
    Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
             |   |  0| mX|   |   |   |   |   | mX|   |   |   |   |   |   |
         Use:| SX| SU| hi|E25| - | - | - | - | lo| - | - | - |Chi| - |Clo|

                       (Multiply the X true mantissa times "C" to get the value,
                       mX*(1-G(X)). Call the result mV. Put into  (R2, R0).)
4.6.D.7                (R6, R7)=R6*R2 (card)   . (mXlo*Chi.)
                       (R0, R1)=R0*RC (card)   . (Clo*mXhi.)
                       (R2, R3)=R2*RC (card)   . (Chi*mXhi.)
                       R6=R6+R3, save carry out. (Combine lower bits of partial
                                                 product, mV=mX*(1-G(X)).)
                       R2=R2+ carry in   . (Combine upper bits of partial
                                           product, mV=mX*(1-G(X)).)
                       R0=R0+R6, save carry out. (Combine lower bits of complete
                                                 product, mV=mX*(1-G(X)).)
                       R2=R2+ carry in   . (Combine upper bits of complete
                                           product, mV=mX*(1-G(X)).)
    Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
             |   |  0|   |   |   |   |   |   |   |   |   |   | mV|   | mV|
         Use:| SX| SU| - |E25| - | - | - | - | - | - | - | - | hi| - | lo|

                       (If R2 lead bit is 1 (i.e., if it looks negative, the
                       value is in the mantissa range of the output Y. Else,
                       mV is a hair below .5 and needs to be multiplied by 2.
                       Fix and produce output.)
4.6.D.9                If R2=negative  (Pseudonegative; really a magnitude.)
                            (Pseudonegative branch direction; really, the mV
                            magnitude is = to or > .5  .)
4.6.D.9.0                   RB=RB+2*(128-5)  (Biased exponent of Y times 2;
                                              need to multiply it by 64.)
                            (R2, R3)=R2*256  (Y true mantissa hi 8 bits
                                              pr perly positioned in R2.)
                            (R0, R1)=R0*256  (Y true mantissa lowest 16 bits
                                              properly positioned in R0.)

                          Else
4.6.D.9.1                   RB=RB+2*(128-6)  (Biased exponent of Y times 2;
                                              need to multiply it by 64.)
                            (R2, R3)=R2*512  (Y true mantissa hi 8 bits
                                              properly positioned in R2.)
                            (R0, R1)=R0*512  (Y true mantissa lowest 16 bits
                                              properly positioned in R0.)
```

```
                    End (4.6.D.9) If.

                    (Fix exponent.)
4.6.D.B             (RB, RC)=RB*X'0040'  . (Properly positioned biased
                                            Y exponent is now in RB.)
                    (Fix mantissa.)
                    R0=R0 .OR. R3  . (Merge lo bits of Y mantissa; Ylo.)
                    R2=R2 .AND. X'FF7F'  (Clear lead mantissa bit.)
                    R2=R2 .OR. RE  (Sign and biased mantissa merge.)
                    R2=R2 .OR. RC  (Merge biased Y exponent into Yhi.)
                    RE=0  . (Status.)
                    RETURN  (by way of RF).
    Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
             |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
         Use:| - | - | - | - | - | - | - | - | - | - | - | - |Yhi| - |Ylo|
         ****************************************************************

                End (4.6.D) If.


             Else
```

```
*****************************************************************************

                              ATANF entry

*****************************************************************************

         (X true exponent is 0 or + starting here.)
Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
         |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
     Use:|   |   |   |Xhi|   |Xlo|   |   |   |   |   |   |   |   | - |

 4.1     RE=X'8000'  . (Prepare to capture X sign in RE(0); clear other bits.)
         RE=RE .AND. RB  . (Sign of X in RE; other RE bits cleared.)
         RB=RB .EXCLUSIVE OR. RE  . (X changed to a magnitude, |X|.)
Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
         |   |   |   ||X||   ||X||   |   |   |   |   |   |   |   |   |
     Use:| SX|   |   | hi|   | lo|   |   |   |   |   |   |   |   | - |

 4.3     (RB, RC)=RB*X'0200'  . (|X| biased mantissa hi left justified in RC;
                                 |X| biased exponent, EXP, is in RB, right
                                 justified.)
         (R9, RA)=R9*X'0200'  . (|X| biased mantissa lo left justified in
                                 (R9, RA).)
         R9=R9 .OR. RC  . (|X| biased mantissa hi merged into R9.)
Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
         |   |   |   |   | mX| mX|   |   |   |   |   |   |   |   |   |
     Use:| SX|   | _ |EXP| lo| hi|   |   |   |   |   |   |   |   | - |

 4.5     RB=RB-128  . (Unbiased exponent=EX.)
 4.7     If RB=0
               (True exponent of |X| is 0 along this path. Polynomial to be
               used is setup for the |X| interval, .5 <= |X| < 1.0  .)
     Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
              |   |   |   |   | mX| mX|   |   |   |   |   |   |   |   |   |
          Use:| SX|   | - | EX| lo| hi|   |   |   |   |   |   |   |   | - |

               (Bias down mantissa by an additional .25; then, describe
               the resultant U as a magnitude with sign in RD.)
               (Create true |X| mantissa-.75  . Then conceptually multiply
               result by 2 to create U. Radix point will then be at left
               edge of R9.)
 4.7.0         RD=.NOT. R9  . (Lead bit of RD, RD(0), now contains sign bit
                               of U.)
               RD=RD .AND. X'8000'  . (Clears all but U sign bit.)
 4.7.2         If RD=0
                     (Sign bit of U is 0, i.e., +.)
 4.7.2.0             R9=R9 .EXCLUSIVE OR. X'8000'  . (Clear lead bit of R9
                                                when true mantissa-.75 is
                                                +; creates Uhi. Ulo
                                                already exists. Uhi,Ulo
                                                is the magnitude of U.)
                    Else
4.7.2.1                R9=R9 .EXCLUSIVE OR. X'7FFF'  . (Clear lead bit of R9
                                                when true mantissa-.75 is
                                                -; complement remaining
                                                bits of R9 to create Uhi.
                                                Now proceed to complement
                                                RA which becomes Ulo. Uhi
                                                Ulo is the magnitude of
                                                U.)
                       RA=RA .EXCLUSIVE OR. X'FFFF'  . (Complement complete.)
     Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
              |                  |   |   |   |   |   |   |   |   |   |   |
          Use:| SX| SU| - | EX|Ulo|Uhi|   |   |   |   |   |   |   |   | - |

                End (4.7.2) If.


4.7.4           R7=N2   (Set degree of polynomial for calculating "p2",
                         p2=ATAN(ARG)-.5, ARG=(U+1.5)/2  .)
                R8=COEF2+4*N2+2  (Set to last 16 bit address of COEF2 block,
                                  the coefficients of p2.)
     Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
              |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
          Use:| SX| SU| - | EX|Ulo|Uhi|loc|cnt|   |   |   |   |   |   | - |

                BAL,RF    POLY32.  (Compute "p2" polynomial.)
     Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
              |   |   |   |   |   |   |   |   |   |   |   |   | p2|   | p2|
          Use:| SX| SU| - | EX|Ulo|Uhi| - | - |   |   | - | - | hi| - | lo|

                (If R2 lead bit is 1 (i.e., if p2 is negative), the value
                produces a Y mantissa that is a hair below .5  . In such case
                the final Y true exponent must be -1 and the bits of p2 must
                be shifted left 1 bit position more than normal. Else, when p2
                is positive, the Y true exponent is 0; a normal alignment
                shift is required.)
                (Fix p2 and produce output.)
4.7.6           If R2=negative
                       (The p2 value is negative branch direction.)
4.7.6.0                (R2, R3)=R2*512  (Y true mantissa hi 8 bits
                                          properly positioned in R2.)
                       (R0, R1)=R0*512  (Y true mantissa lowest 16 bits
                                          properly positioned in R0.)
                       R2=R2 .OR. X'3F80'  . (Merge Y biased exponent of 127
                                      into Yhi.)

                Else
                   (The p2 value  is .5 or somewhat greater branch
                   direction.)
4.7.6.1            (R2, R3)=R2*256  (Y true mantissa hi 8 bits
                                     properly positioned in R2.)
                   (R0, R1)=R0*256  (Y true mantissa lowest 16 bits
                                     properly positioned in R0.)
                   R2=R2 .OR. X'4000'  . (Merge Y biased exponent of 128
                                         into Yhi.)

             End (4.7.6) If.


             (Fix mantissa.)
4.7.8        R0=R0 .OR. R3  . (Merge lo bits of Y mantissa; Ylo.)
             R2=R2 .OR. RE  . (Merge Y sign into Yhi.)
             RE=0  . (Status.)
             RETURN  (by way of RF).
   Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
            ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
        Use:¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦Yhi¦ - ¦Ylo¦
        *****************************************************************

          Else

             (True exponent of ¦X¦ is greater than 0 along this path.
             It will be necessary to differentiate the ¦X¦ true exponent=+1
             case from the cases for which the exponent is greater..)
   Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
            ¦   ¦   ¦   ¦   ¦ mX¦ mX¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
        Use:¦ SX¦   ¦ - ¦ EX¦ lo¦ hi¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦ - ¦

             (Develop EX2=EX-2.)
4.7.1        RB=RB-2  . (, RB contains the ¦X¦ true exponent -2.)
   Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
            ¦   ¦   ¦   ¦   ¦ mX¦ mX¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
        Use:¦ SX¦   ¦ - ¦EX2¦ lo¦ hi¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦ - ¦

4.7.3        If RB=negative
                (The ¦X¦ true exponent is +1 to branch in this direction.
                Polynomial to be used is setup for the ¦X¦ interval,
                1.0 <= ¦X¦ < 2.0  .)
                (Bias down mantissa by an additional .25; then, describe
                the resultant U as a magnitude with sign in RD, i.e.,
                create (true ¦X¦ mantissa-.75)  . Then conceptually
                multiply result by 2 to create U. Radix point will then
```

```
                          be at left edge of R9.)
4.7.3.0                   RD=.NOT. R9  . (Lead bit of RD, RD(0), now contains sign
                                         bit of U.)
                          RD=RD .AND. X'8000'  . (Clears all but U sign bit.)
4.7.3.2                   If RD=0
                               (Sign bit of U is 0, i.e., +.)
4.7.3.2.0                      R9=R9 .EXCLUSIVE OR. X'8000'  . (Clear lead bit of
                                                    R9 when true mantissa-.75
                                                    is +; creates Uhi. Ulo
                                                    already exists. Uhi,Ulo
                                                    is the magnitude of U.)
                            Else
4.7.3.2.1                       R9=R9 .EXCLUSIVE OR. X'7FFF'  . (Clear lead bit of
                                                    R9 when true mantissa-.75
                                                    is -; complement remaining
                                                    bits of R9 to create Uhi.
                                                    Now proceed to complement
                                                    RA which becomes Ulo. Uhi,
                                                    Ulo is the magnitude of
                                                    U.)
                               RA=RA .EXCLUSIVE OR. X'FFFF'  . (Complement done.)
    Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
             ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
         Use:¦ SX¦ SU¦ - ¦EX2¦Ulo¦Uhi¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦ - ¦

                          End (4.7.3.2) If.

4.7.3.4                   R7=N3   (Set degree of polynomial for calculating "p3",
                                  p3=(ATAN(ARG)/2)-.5, ARG=(U+1.5)/2  .)
                          R8=COEF3+4*N2+2  (Set to last 16 bit address of COEF3
                                           block, the coefficients of p3.)
    Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
             ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
         Use:¦ SX¦ SU¦ - ¦EX2¦Ulo¦Uhi¦loc¦cnt¦   ¦   ¦   ¦   ¦   ¦   ¦ - ¦

                          BAL,RF    POLY32.  (Compute "p3" polynomial.)
    Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
             ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦ p3¦   ¦ p3¦
         Use:¦ SX¦ SU¦ - ¦EX2¦Ulo¦Uhi¦ - ¦ - ¦   ¦   ¦ - ¦ - ¦ hi¦ - ¦ lo¦

                          (If R2 lead bit is 1 (i.e., if p3 is negative), the value
                          produces a Y mantissa that is a hair below .5  . In such
                          case the final Y true exponent must be 0 and the bits of
                          p3 must be shifted left 1 bit position more than normal.
                          Else, when p3 is positive, the Y true exponent is 1; a
                          normal alignment shift is required.)
                          (Fix p3 and produce output.)
4.7.3.6                   If R2=negative
```

```
                              (The p3 value is negative branch direction.)
4.7.3.6.0                     (R2, R3)=R2*512  (Y true mantissa hi 8 bits
                                                properly positioned in R2.)
                              (R0, R1)=R0*512  (Y true mantissa lowest 16 bits
                                                properly positioned in R0.)
                              R2=R2+(125*128)  . (Develop Y biased exponent of
                                                  128 in R2.)

                           Else
                              (The p3 value  is .5 or somewhat greater branch
                              direction.)
4.7.3.6.1                     (R2, R3)=R2*256  (Y true mantissa hi 8 bits
                                                properly positioned in R2.)
                              (R0, R1)=R0*256  (Y true mantissa lowest 16 bits
                                                properly positioned in R0.)
                              R2=R2 .OR. X'4080'  . (Merge Y biased exponent of
                                                     129 into Yhi.)

                        End (4.7.3.6) If.


                        (Fix mantissa.)
4.7.3.8                 R0=R0 .OR. R3  . (Merge lo bits of Y mantissa; Ylo.)
                        R2=R2 .OR. RE  . (Merge Y sign into Yhi.)
                        RE=0  . (Status.)
                        RETURN  (by way of RF).
    Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
             ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
         Use:¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦ - ¦Yhi¦ - ¦Ylo¦
         *****************************************************************

                     Else
                        (The ¦X¦ true exponent is greater than +1 to branch in
                        this direction. The ¦X¦ values will be 2.0 or greater.
                        W                                           ¦ in order
                        to get suitable expansions for the ATAN function.
                        Proceed to find the reciprocal of ¦X¦.
    Register:¦ RE¦ RD¦ RC¦ RB¦ RA¦ R9¦ R8¦ R7¦ R6¦ R5¦ R4¦ R3¦ R2¦ R1¦ R0¦
             ¦   ¦   ¦   ¦   ¦ mX¦ mX¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
         Use:¦ SX¦   ¦ - ¦EX2¦ lo¦ hi¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦ - ¦

4.7.3.1                 R5=R9   . (Replicate mXhi in R5.)
                        (R5, R6)=R5*X'0020'  . (Multiply range index, "j",
                                                j=0,...,7  , by 32; R5=4*j
                                                results in bits 11,...,13 .)
                        R5=R5 .AND. X'001C'  . (Clear all but bits 11,...,13  .)
                        R9=R9 .AND. X'1FFF'  . (Clear range interval index bits,
                                               "j", of mXhi. Creates Uhi in R9
```

```
                         and Ulo in RA.)
RD=0  . (Clear sign bit of U. U is positive only.)
R5=R5+GET  . (R5 points to polynomial degree number
              in "GET" table.)
```

"GET" Table

j	k	4*j+k	Address	Value
0	0	4*0+0	GET+ 0	M(0)
0	2	4*0+2	GET+ 2	KOEF(0)+4*M(0)+2
1	0	4*1+0	GET+ 4	M(1)
1	2	4*1+2	GET+ 6	KOEF(1)+4*M(1)+2
2	0	4*2+0	GET+ 8	M(2)
2	2	4*2+2	GET+10	KOEF(2)+4*M(2)+2
3	0	4*3+0	GET+12	M(3)
3	2	4*3+2	GET+14	KOEF(3)+4*M(3)+2
4	0	4*4+0	GET+16	M(4)
4	2	4*4+2	GET+18	KOEF(4)+4*M(4)+2
5	0	4*5+0	GET+20	M(5)
5	2	4*5+2	GET+22	KOEF(5)+4*M(5)+2
6	0	4*6+0	GET+24	M(6)
6	2	4*6+2	GET+26	KOEF(6)+4*M(6)+2
7	0	4*7+0	GET+28	M(7)
7	2	4*7+2	GET+30	KOEF(7)+4*M(7)+2

```
R7=0(R5); R5=R5+2  . (Load R7 with degree of selected
                      polynomial, M(j), held at address
                      pointed to by value in R5. Then
                      bump "GET" table pointer, R5.)
R8=0(R5)  . (Load R8 with KOEF(j)+4*M(j)+2  ; the last
             16 bit address of the KOEF(j) block, the
             coefficients of the polynomial, r(j), held
             at address pointed to by value in R5.)
```

Register:	RE	RD	RC	RB	RA	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0
Use:	SX	SU	-	EX2	Ulo	Uhi	loc	cnt	-	-					-

```
4.7.3.3          BAL,RF     POLY32.  (Compute "r(j)" polynomial.)
```

Register:	RE	RD	RC	RB	RA	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0
													rj		rj
Use:	SX	SU	-	EX2	Ulo	Uhi	-	-			-	-	hi	-	lo

(The value of rj is given by

rj=1/(8*ARG) for ARG=1/2+j/16+U/2 , 0 <= U < 1/8.

If R2(1) bit is 1 , the final |X| reciprocal true

```
                    exponent must be -(EX2+2)+2=-EX2+0=(.NOT. EX2)+1 ; a
                    normal alignment shift is required. If R2 lead bit is 0 ,
                    the final |X| reciprocal true exponent must be
                    -(EX2+2)+1=-EX2-1=(.NOT. be shifted left 1 bit position
                      more than normal.)
                    (Fix r(j) and produce output |X| reciprocal.)
4.7.3.5             RB=.NOT. RB  . (Complement EX2.)
4.7.3.7             If R2(1) is set  (Rare case; ARG=.5 test.)
                          (Normal alignment branch direction.)
4.7.3.7.0                 (R2, R3)=R2*X'200'  . (Y true mantissa hi 8 bits
                                            properly positioned in R2.)
                          (R0, R1)=R0*X'200'  . (Y true mantissa lowest 16
                                                 bits properly positioned in
                                                 R0.)
                          RB=RB+128  . (Biased exponent of reciprocal-1.)

                       Else
                          (Additional bit shift alignment branch direction.)
4.7.3.7.1                 (R2, R3)=R2*X'400'  . (Y true mantissa hi 8 bits
                                                 properly positioned in R2.)
                          (R0, R1)=R0*X'400'  . (Y true mantissa lowest 16
                                                 bits properly positioned in
                                                 R0.)
                          RB=RB+127  . (Biased exponent of reciprocal-1.)
                    End (4.7.3.7) If.

                    (Assemble X reciprocal, XR, in (RB, R9).)
4.7.3.9             (RB, RC)=RB*X'0080'  . (Position exponent in RC.)
                    RB=RC  . (Move aligned, biased exponent into RB.)
                    RB=RB+R2  . (Add aligned (biased exponent-1) & aligned
                                 hi mantissa.)
                    RB=RB .OR. RE  . (Merge sign, biased exponent & hi
                                     mantissa.)
                    R0=R0 .OR. R3  . (Merge lo mantissa bits into R0.)
                    R9=R0  . (Move mantissa lo into RB.)
    Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
             |   |   |   | XR|   | XR|   |   |   |   |   |   |   |   |   |
         Use:| - | - | - | hi| - | lo| - | - |   |   | - | - | - | - | - |


    **********************************************************************
    **********************************************************************
                    (Reciprocal of X is now in the input X slot; find the
                    ATAN of the reciprocal.)
4.7.3.B             BAL,RF   ATANF entry  . (Get ANG. Answer is PI/2-ANG.)
    Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
             |   |   |   |   |   |   |   |   |   |   |   |   |ANG|   |ANG|
         Use:| - | - | - | - | - | - | - | - | - | - | - | - | hi| - | lo|
*****************************************************************************
*****************************************************************************


                         (Compute PI/2-ANG. ANG is floating point.)
4.7.3.D                  RE=X'8000'  . (Prepare for sign bit isolation.)
                         RE=RE .AND. R2  . (ANG sign bit in RE(0); 0 elsewhere.
                                             RE value is SA.)
                         R2=R2 .EXCLUSIVE OR. RE  . (Absolute value of ANG, |A|,
                                                      now in (R2, R0).)
   Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
            |   |   |   |   |   |   |   |   |   |   |   |   ||A||   ||A||
        Use:| SA| - | - | - | - | - | - | - | - | - | - | - | hi| - | lo|

                         (Put 2 times the ANG biased exponent into R2. Also, save
                         |A| hi in R7.)
4.7.3.F                  R7=R2  . (Save |A| in R7.)
                         (R2, R3)=R2*X'0400'  . (Bits 7,...,14 of R2 now holds
                                                 biased |A| exponent.)
                         R2=R2 .AND. X'01FE'  . (R2 now holds
                                                 2*(biased |A| exponent)=2EB.)
   Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
            |   |   |   |   |   |   |   ||A||   |   |   |   |   |   ||A||
        Use:| SA| - | - | - | - | - | - | hi| - | - | - | - |2EB| - | lo|

                         (Align |A| biased mantissa so that radix point is at
                         left edge of R7; then unbias it.)
4.7.3.11                 (R7, R8)=R7*X'0100'  . (Move |A| biased mantissa hi into
                                                 R8.)
                         (R0, R1)=R0*X'0100'  . (Shift hi part of biased mantissa
                                                 lo into R0 and lo part into R1.)
                         R8=R8+R0  . (Merge biased mantissa hi pieces. Results in
                                      biased mantissa hi of |A| in R8; lo part is
                                      in R1.)
                         R8=R8 .OR. X'8000'  . (Unbias |A| biased mantissa hi
                                                pieces. |A| true mantissa hi, mAu
                                                hi, is in R8; |A| true mantissa
                                                hi, mAu lo, is in R1.)
   Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
            |   |   |   |   |   |   |mAu|   |   |   |   |   |   |mAu|   |
        Use:| SA| - | - | - | - | - | hi| - | - | - | - | - |2EB| lo| - |

                (Develop |A| as an integer. Use the data derived from
                ANG (SA, mAu, and 2EB) to do this. The 2EB value will be
                used to access a shift constant from the SHF table; that
                constant will be used to shift the mantissa (mAu) bits
                to the left (relative to the mantissa's radix point); the
                resultant value is the integer form of |A|.
```

```
          Initially, consider that mAu has been divided by 2**30;
          the radix point is then 30 bit positions to the left
          of the left edge of R8. In fact, if the 4 register
          combination, (RC, RA, R8, R1), contains this value, then
          RC and RA are conceptually filled with "0"'s and the
          radix point lies 2 bits in from the left edge of RC, i.e.,
          between RC(1) and RC(2). To get the true integer value of
          |A|, this value must be multiplied by 2**(EXA+30) where
          EXA is the true exponent of |A| and the "30" compensates
          for the earlier division by 2**30. The quantity
          2**(EXA+30) is the value extracted by appropriately
          entering the SHF table.)
          (Develop SHF table entry address.)
4.7.3.13    R2=R2-((128-30)*2)  . (Two times (|A| true exponent+30) is
                                   put into R2 after removal of
                                   exponent bias; result is called
                                   2EA.)
    Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
             |   |   |   |   |   |   |mAu|   |   |   |   |   |   |mAu|   |
         Use:| SA| - | - | - | - | - | hi| - | - | - | - | - |2EA| lo| - |

            (Create output Y for different exponent ranges of |A|. The
            true exponent value of |A| can't be greater than -1.)
4.7.3.15              If R2=negative
                         (|A| true exponent is -31 or less. |Y|=PI/2.)
4.7.3.15.0               R2=X'40C9'  . (PI/2 (hi) for exponent -31 to -128.)
                         R0=X'0FDB'  . (PI/2 (lo).)
                         R2=R2 .OR. RE  . (Merge in sign bit of output.)
                         RE=0  . (Set status.)
                         RETURN via RF  .
   Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0
            |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
        Use:| - | - | - | - | - | - | - | - | - | - | - | - |Yhi| - |Ylo|
            **********************************************************

                      Else
   Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
            |   |   |   |   |   |   |mAu|   |   |   |   |   |   |mAu|   |
        Use:| SA| - | - | - | - | - | hi| - | - | - | - | - |2EA| lo| - |
            (The integer form of |A| must be developed in order
            to execute the subtraction from PI/2; the mAu
            value must be shifted relative to the radix point by
            an amount determined by |A|'s true exponent. The
            |A| exponent range is from -31 up through -1.)
4.7.3.15.1             If R2(10)=1  .
                          (Data of (RC, RA, R8, R1) must be shifted left
                          by 16.)
4.7.3.15.1.0              RA=R8
```

```
                                     R8=R1
                                     R2 .AND. X'001E'  . (Clear lead bit of 2EA.)
                                     Else
4.7.3.15.1.1                         RA=0  . (Initialize RA.)
                                    End (4.7.3.15.1) If

                                    (Do the remaining (less than 16) shift to convert
                                    |A| to an integer.)
4.7.3.15.3                     R2=R2+SHF  . (Develop address into table to get
                                             shift factor.)
                               R5=0(R2)  . (Put shift value into R5.)
                               (RA, RB)=RA*R5  . (Do hi shift. The role of RC of
                                            the 4 register set, (RC,RA,R8,R1),
                                             is now taken on by RA.)
                               (R8, R9)=R8*R5  . (Do lo shift. The role of RA of
                                           the 4 register set, (RC,RA,R8,R1),
                                           is now taken on by R8.)
                               R8=R8 .OR. RB  . (Merge bits of similar level of
                                           significance in R8.)

        Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
                 |   |   |   |   |int|   |int|   |   |   |   |   |   |   |   |
             Use:| SA| - | - | - ||A|| - ||A|| - | - | - | - | - | - | - | - |
                 |   |   |   |   | hi|   | lo|   |   |   |   |   |   |   |   |
                                ((RA, R8) now contains the integer value of |A|; the
                                    radix point lies between RA(1) and RA(2). Now
                                    execute the (PI/2-(int |A|)) process.)
4.7.3.15.5                          R8=.NOT.R8  . (Negate int |A| lo.)
                                    RA=.NOT.RA  . (Negate int |A| hi.)
                                    R8=R8+X'PI/2' ,save carry out. (Add PI/2 lo.)
                                    RA=RA+X'PI/2' ,carry in. (Add PI/2 hi.)

                                    (Convert int |Y| into floating point Y.)
4.7.3.15.7                          (R8, R9)=R8*X'0400'  . (Align |Y| mantissa lo.)
                                    (RA, RB)=RA*X'0400'  . (Align |Y| mantissa hi.)
                                    R8+R8 .or. RB  . (Merge mantissa pieces.)
                                    R0=R8  . (Move Ylo into R0.)
                                    R2=RA  . (Move |Y| hi into R2.)
                                    R2=R2 .OR. X'4080'  . (Merge in biased exponent of
                                    |Y| corresponding to Y true exponent of 1.)
                                    R2=R2 .OR. RE  . (Insert sign bit of Y.)
                                    RE=0  . (Set status.)
                                    RETURN via RF  .
       Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
                |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
            Use:| - | - | - | - | - | - | - | - | - | - | - | - |Yhi| - |Ylo|
                ********************************************************
```

```
                  End (4.7.3.15) If

             End (4.7.3) If

         End (4.7) If

   End (4) If

   END
```

3.2.5 MCU NATURAL LOG SUBROUTINE DESCRIPTION : LNM

This Subroutine develops the value, "Y", the natural logarithm of the input variable, "X". "X", the input, and "Y", the output, are 32 bit VAX floating point numbers. Along with "Y", a 16 bit status value, S, is generated for output; it is 0 when X is positive and non-zero. The theory used for this MCU computation if the natural logarithm, is identical to that for the parallel array algorithm.

****

The subroutine demands the calculation of :

```
          Y=LN(X)        .
```

****

```
0.    (ENTER LOGM.)


1. (Set the status bits to 0.)
     Load R14 with X'0000'.


2. (Check for X=negative.)

   If bit 0 of R11 is set

      then,

         Load register R2 with X'0003'.

         RETURN.     Fatal error status if X was negative.

    End If.

   Else,

      Continue.
```

```
3. (Check for X=0.)
     If R9=0,

          If R11=0,

               Load register R2 with X'FFFF'.

               Load register R0 with X'FFFF'.

               Load register R14 with '2' to indicate underflow.

               RETURN.

          Else,

               Continue.

     Else

          Continue.
```

4. R9 and R11 contain X. Adjust the R11 X bits 9 bit positions to the left so that the shifted R11 X bits lie in R12 and R11 as follows:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Address R11:	m	m	m	m	m	m	m	0	0	0	0	0	0	0	0	0
	1	2	3	4	5	6	7									

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Address R12:	0	0	0	0	0	0	0	S	e	e	e	e	e	e	e	e
									0	1	2	3	4	5	6	7

Perform cardinal multiply :

```
MLU R11,#X'200'  .
```

5. Adjust the R9 X bits 9 bit positions to the left so that the shifted R9 X bits lie in R10 and R9 as follows:

```
              |                              1  1  1  1  1  1|
              | 0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5|
              |                        |                      |
Address R10:  | m| m| m| m| m| m| m| 0| 0| 0| 0| 0| 0| 0| 0| 0|
              | 1| 1| 1| 2| 2| 2| 2|  |  |  |  |  |  |  |  |  |
              | 7| 8| 9| 0| 1| 2| 3|  |  |  |  |  |  |  |  |  |
              |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

              |                                               |
              |                              1  1  1  1  1  1|
              | 0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5|
              |                        |                      |
Address R9:   | 0| 0| 0| 0| 0| 0| 0| m| m| m| m| m| m| m| m| m|
              |  |  |  |  |  |  |  |  |  | 1| 1| 1| 1| 1| 1| 1|
              |  |  |  |  |  |  |  | 8| 9| 0| 1| 2| 3| 4| 5| 6|
              |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
```

Perform cardinal multiply :

```
MLU R9,#X'200'  .
```

6. "OR" R9 with R12 to get:

```
              |                                               |
              |                              1  1  1  1  1  1|
              | 0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5|
              |                        |                      |
Address R9:   | m| m| m| m| m| m| m| m| m| m| m| m| m| m| m| m|
              |  |  |  |  |  |  |  |  |  | 1| 1| 1| 1| 1| 1| 1|
              | 1| 2| 3| 4| 5| 6| 7| 8| 9| 0| 1| 2| 3| 4| 5| 6|
              |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
```

7. If the most significant mantissa bit is set,

then,   use LNCOEFF1 coefficients;

else,   use LNCOEFF0 coefficients;

In general the polynomial coefficients are:

HEX address	Coefficient
COEF+ 0	A(0) (low)
COEF+ 1	A(0) (high)
COEF+ 2	A(1) (low)
COEF+ 3	A(1) (high)
.	.
.	.
.	.
COEF+2*N	A(N) (low)
COEF+2*N+1	A(N) (high) .

Note that POLY computes:

H=A(0)+fb*(A(1)+fb*(A(2)+fb*(A(3)+fb*(A(4)+...+fb*(A(N-1)+fb*(A(N)))...).

The steps to use POLY follow.)

Load R8 with COEF block starting address.

Load R7 with the degree of the polynomial, N.

```
CALL R15,PLY32$  to compute LN(ARG)-ARG.
```

8. Then, because the desired function is actually LN(ARG), add in the input ARG contained in R10 and R9:

```
ADD   R0,R10

ADDC  R2,R9 .
```

We thus have computed the LN(1+U) term of the required:

Y = EXP*LN(2) - (128+1)*LN(2) + LN(1+U) - U .

9. Sutract 128+1 from the input exponent:

```
SUB  R11,#128+1 for use in above equation;
```

```
        If R11 is negative,

           then,   Perform 2's complement of R11

                   and remember sign bit in R14 .

           else,   continue.
```

10. Save the exponent of R11 in R6. Perform the multiplication of the exponent by the constant LN(2):

```
         MLU     R11,#X'B172'  which is the high half of LN(2),

and,     MLU     R6,#X'17F9'   which is the low half.
```

Merge the two multiplication halves:

```
         ADD     R6,R12

         ADDC    R11      .
```

11. If EXP-129 was negative, the product is negative,

    then, complement R6, R7, and R11 which contain (EXP-129)*LN(2).

12. Add the Polynomial results to R6, and R7 :

```
        ADD     R7,R0

        ADDC    R6,R2

        INCRC   R11    .
```

13. If this final result is negative,

    then, remember the final sign bit in R14,

    and complement R11, R6, and R7 to get a positive number.

Now the un-normalized Y value is contained in registers

R11, R6 and R7.

14. Normalize the Y value as follows:

```
  If R11 is '0',
      then,  R6 and R7 contain the fractional bits;
             CLR   R10          no exponent bias required;
             CALL  R15,NORMV$   to normalize Y.

            The output of NORMV$ is VAX 32-bit format in R2 and R0.
            CLR   R14          clear status and,
            RETURN .
      else,
          continue.
```

15. Normalize the 48 possible bits in R11, R6 and R7:

Since the maximum number of exponent bits is 7,

Shift the value in R11 by 9 places (16 bit register -7) in order to speed the normalization:

```
     MLU   R11,#X'200'
     LR    R8,R7        save R7 temporarily;
     MLU   R6,#X'200'
     MLU   R8,#X'200'  .
```

Then merge the results:

```
       OR    R6,R12

       OR    R7,R8  .
```

Save the shift count in R10 to be passed to NORMV$

so that the exponent may be adjusted; include also the 16 bits of R11:

```
      LR    R10,#16-9  .
```

For the normalize routine NORMV$, the input registers are R3 and R1:

```
     LR     R3,R6

     LR     R1,R7

     CALL   R15,NORMV$ .
```

Clear the status register:

```
     CLR    R14   and,

     RETURN  .
```

3.3.6 MCU EXP SUBROUTINE DESCRIPTION : EXPM

For the Exponential subroutine, X is the input variable, Y is the output variable, and S is the output status indicator.

All 16 bits of S are normally 0. When S is 1, overflow has occurred; S set to 2 indicates underflow, a non fatal condition.

Y is set at X'7FFFFFFF' if Y goes out of range (overflows); Y is set at X'00000000' if X underflows.

The algorithm used for this subroutine is identical to the algorithm used for the array Logarithm function (LNA) described Section 2.

****

```
0. (ENTER EXPM.)


1. Set the status bits to 0.
      Load R14 with X'0000'.


2. (Check for X=0.)

      If R9=0,

            If R11=0,

                  Load register R2 with X'0000'.

                  Load register R0 with X'4080' , a VAX '1'.

                  RETURN.

       Else,.

            Continue.
```

3. R11 and R9 contain X. Adjust the X bits 9 bit positions to the left so that the shifted X bits lie in R9 and R10 as follows.

```
     |                                               |
     |                               1  1  1  1  1  1|
     | 0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5|
     |                       |                       |
R11: | m| m| m| m| m| m| m| 0| 0| 0| 0| 0| 0| 0| 0| 0|
     |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
     | 1| 2| 3| 4| 5| 6| 7|  |  |  |  |  |  |  |  |  |
     |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
```

```
     |                                               |
     |                               1  1  1  1  1  1|
     | 0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5|
     |                       |                       |
R12: | 0| 0| 0| 0| 0| 0| 0| S| e| e| e| e| e| e| e| e|
     |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
     |  |  |  |  |  |  |  |  | 0| 1| 2| 3| 4| 5| 6| 7|
     |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
```

Perform cardinal multiply :

```
MLU  R11,#X'200'.
```

4. Adjust the R9 X bits 9 bit positions to the left so that the shifted R9 X bits lie in R10 and R9 as follows:

```
     |                                               |
     |                               1  1  1  1  1  1|
     | 0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5|
     |                       |                       |
R10: | m| m| m| m| m| m| m| 0| 0| 0| 0| 0| 0| 0| 0| 0|
     | 1| 1| 1| 2| 2| 2| 2|  |  |  |  |  |  |  |  |  |
     | 7| 8| 9| 0| 1| 2| 3|  |  |  |  |  |  |  |  |  |
     |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
```

```
     |                                               |
     |                               1  1  1  1  1  1|
     | 0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5|
     |                       |                       |
R9:  | 0| 0| 0| 0| 0| 0| 0| m| m| m| m| m| m| m| m| m|
     |  |  |  |  |  |  |  |  |  | 1| 1| 1| 1| 1| 1| 1|
     |  |  |  |  |  |  |  | 8| 9| 0| 1| 2| 3| 4| 5| 6|
     |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
```

Perform cardinal multiply :

```
MLU R9,#X'200'
```

5. "OR" R9 with R12 to get:

```
     |                                               |
     |                              1  1  1  1  1  1|
     | 0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5|
     |                       |                       |
R9:  | m| m| m| m| m| m| m| m| m| m| m| m| m| m| m| m|
     |  |  |  |  |  |  |  |  |  | 1| 1| 1| 1| 1| 1| 1|
     | 1| 2| 3| 4| 5| 6| 7| 8| 9| 0| 1| 2| 3| 4| 5| 6|
     |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
```

6. Load register R2 with aHI=X'2E2A' where 'a' is 1/LN(2).
   Load register R0 with aLO=X'8EC9'.

   Then call the common multiply subroutine to obtain a*f:

```
CALL  R15,MULT32$.
```

   The results are stored in R0 and R2.

7. Add in the a/2 term:

```
ADD   R0,#X'8ECA'

ADDC  R2,#X'2E2A'.
```

These registers are the Ig bits of the final result.

8. Save the sign bit in R4, and Clear the sign bit of the exponent in R11.

9. If the integer part of Ig is a '0', then the input exponent will determine overflow if greater than 8. If the integer part of Ig is '1' the overflow or underflow exists if the input exponent is greater than 6:

   If bit #1 of R2 is 0,   (Ig integer bit)

```
            then,

                 If R11 (input exponent) >=  128+8   (include bias),

                    then,

                           Perform step 10 error return.

           else,

                 If R11 (input exponent) <   128+7  (include bias),

                   then,

                           Perform step 10 error return.


                   else   proceed to step 11 for normal execution
                          no overflow or underflow exists.
```

10. When the input exponent becomes out of range:

```
         If the sign bit of the exponent was positive,

            then,

               An overflow has occurred so

               Set the output to the maximun VAX number

               And set the status to '1'.

               LR  R0,#X'FFFF

               LR  R2,#X'7FFF'

               LR  R14,#1

               RETURN.

          else,

              since the exponent was large and the sign negative,

              a non-fatal underflow condition exists; the output

              becomes '0' and an underflow status is indicated:
```

```
          CLR R0

          CLR R2

          LR  R14,#2

          RETURN.
```

11. (NORMAL entry)

```
    Compare R11 (the input exponent) with 128-31;

    If R11 less than 2**-31

      then,

            force the output to '1' (since e**0 = 1).

      else,

           continue.
```

12. Move the Ig values to R6 and R7.

13.(The registers, (R6, R7) save the positive sum Ig=a*f=(Ig1+Ig2+Ig3)+(a/2) with the form (0.1.30). The value of Ig must be shifted an amount determined by the pure biased exponent value of X stored in R10. But before proceeding with the shift, multiply Ig by 2**(-34). The scaled positive Ig value is assumed to reside in the concatenation of registers, R10, R9, R8, R7, R6. The bits of registers R10, R9, R8 must be loaded with the sign bit of 0. The radix point of the scaled Ig value lies at the boundary between R10 and R9 (i.e., between R10(15) and R9(0)). Now load the sign into the registers.)

```
    Load R8 with 0.

    Load R9 with 0.

    Load R10 with 0.
```

(R10 will store the integer bits of the shifted Ig value.)

14.(The registers, (R6, R7) save the sum Ig=a*f=(Ig1+Ig2+Ig3)+(a/2) with the form (0.1.30). The bits of R10, R9, R8, R7, R6 must be shifted left an amount that is ultimately determined by the pure biased exponent value of X stored in R11. Before proceeding, create the shift constant by subtracting 128 and adding 34 to the value stored in R11.

Add R11 to the negative of the exponent bias (-128) plus 34; store result into R11. (R11 now contains the left shift value; it can't be bigger than 41 because of the operations of item 12.)

15.(Left shift the bits of R10, R9, R8, R7, R6 by the amount stored in R11. The radix point between R10 and R9 remains fixed during the shift operations. The shift is performed in 3 phases: shift by 32, shift by 16, and shift by less than 16. The steps follow.)

```
(Check MSB of R11 exponent.)

    If bit #10 of R11 = 0,

        Continue. (MSB of R11 exponent is not set.)

      Else

        Load R9 with R7. (MSB of R11 exponent is set.)

        Load R8 with R6. (Shift 32 bits).

        Load R7 with 0.

        Load R6 with 0.

    End If.

(Check next MSB of exponent).

    If bit #11 of R11 exponent is 0,

        Continue. (Next MSB of R3 is not set.)

      Else

        Load R9 with R8. (Next MSB of exponent is set.)

        Load R8 with R7

        Load R7 with R6.

        Load R6 with 0.
      End If.

   "AND" X'000F' with R11; store result in R11. (Lowest 4 bits of shift value.)

    Note: The SHFM data table contains 16 bit storage locations.
        The starting address of this set of 16 values is SHF$. The values are:
```

Address	Hex Value	Address	Hex Value
SHIFT+ 0	0001	SHIFT+ 8	0100
SHIFT+ 1	0002	SHIFT+ 9	0200
SHIFT+ 2	0004	SHIFT+10	0400
SHIFT+ 3	0008	SHIFT+11	0800
SHIFT+ 4	0010	SHIFT+12	1000
SHIFT+ 5	0020	SHIFT+13	2000
SHIFT+ 6	0040	SHIFT+14	4000
SHIFT+ 7	0080	SHIFT+15	8000 .

```
    The shifts will be accomplished with multiplies using the factors from
    the SHIFT block.)

      Load R0 with value in SHIFT(R11*2).

      Cardinal multiply R9 times R11; store result low in R9 and result high in
      R11.

      Cardinal multiply R8 times R11; store result low in R11 and result
      high in R12.

      "OR" R12 with R9; store result in R9.

      Cardinal multiply R7 times R12; store result low in R7 and result high in
      R8.

      "OR" R11 with R8; store result in R8.

16.(The R10, R9, R8 registers hold the shifted positive sum Ig=a*f
    =(Ig1+Ig2+Ig3)+(a/2) value which has the form (0.8.32). The radix point
    remains between R10 and R9. But before proceeding, make the shifted Ig value
    a 2's complement number.)

      If R4=0,

          Continue.

        Else,
        Complement R10. (R10 stores the integer bits of the signed shifted
        Ig value.)

        Complement R9.

        Complement R8.

    End If.
```

16.(The R10, R9, R8 registers hold the signed, shifted positive sum Ig=a*f =(Ig1+Ig2+Ig3)+(a/2) value which has the form (8.8.32). The radix point remains between R10 and R9. To generate the biased exponent of the output Y value, add the bias of 128 plus 1 to the R10 value.)

```
    Add 128+1 to R10.
```

17.(The R9, R8 registers hold the signed, shifted positive fraction of the sum Ig=a*f=(Ig1+Ig2+Ig3)+(a/2) value which has the form (0.0.32). The radix point remains between R10 and R9. R10 contains the biased exponent of Y. Check to see that the exponent is still in bound.)

```
    Load R12 with X'FF00'.

    "AND" R12 with R10; store results into R12.

    If R12=0

        Continue.

      Else

        Load R14 status with X'0001'.

        Load R2 with X'7FFF'.

        Load R0 with X'FFFF'.

        RETURN.

    End If.
```

18.(The R9, R8 registers hold the signed, shifted positive fraction of the sum Ig=a*f=(Ig1+Ig2+Ig3)+(a/2) value which has the form (0.0.32). The radix point remains between R10 and R9. R10 contains an in range biased exponent of Y. The fraction in (R9, R8), called ff, determines the value of the function, H=(2**(ff))/2)-.75 . The range of ff in HEX is X'00000000'*2**(-32) to X'FFFFFFFF'*2**(-32) . Bias ff by .5 to create fb. Then H=(2**(fb+.5)))/2)-.75 and the range of fb in HEX is X'80000000'*2**(-32) to X'7FFFFFFF'*2**(-32) , i.e., -.5 <= fb <+.5 . (The range of H in HEX is -.25 <= H <+.25 ; its form is (1.0.31).). Proceed to bias ff by .5 .)

19.(Call POLY32(R9, R10, R8, R5, R4, R3, R2). The input fraction fb is put into R9, R10. The coefficients of the polynomial that fits H are in the COEF block of the MCU memory. R8 holds the starting address of the COEF block. The degree of the polynomial to approximate, N, is held by R7. Each coefficient has the form (1.0.31). The coefficients are stored in the COEF block in ascending order. In particular,

HEX address	Coefficient
COEF+ 0	A(0) (low)
COEF+ 1	A(0) (high)
COEF+ 2	A(1) (low)
COEF+ 3	A(1) (high)
.	.
.	.
.	.
COEF+2*N	A(N) (low)
COEF+2*N+1	A(N) (high) .

Finally, after execution of POLY, H with form (1.0.31) is returned in R2, R0. Note that POLY computes
H=A(0)+fb*(A(1)+fb*(A(2)+fb*(A(3)+fb*(A(4)+...+fb*(A(N-1)+fb*(A(N)))...).

The steps to use POLY follow.)

Load R8 with COEF block starting address.

Load R7 with the degree of the polynomial, N.

CALL R15,POLY32$

R2 and R0 contain the output of POLY32, R11 contains the exponent. complement the sign bit of R2 (this is the hidden fraction bit).

20. (Pack data.)
Cardinal multiply R11 exponent with X'0080'; store into R11, R12. (Ignore hi part that fills R11. The Y sign bit and biased exponent are now properly positioned in R12.)

Cardinal multiply R2 with X'0100'; store into R2, R3. (The Y mantissa, biased by .5, now is positioned correctly for merging with the sign

```
and exponent data.)
"AND" R2 with X'007F'; store into R2.
"OR" R2 with R12; store into R2.
Cardinal multiply R0 with X'0100'; store into R0.
"OR" R0 with R3; store into R3.
RETURN
```

3.2.7 MCU COMMON SUBROUTINE : POLY32

This subroutine develops the value, "P", of a polynomial of degree "N" using the independent variable "U" as the input. "U", the input, and "P", the output, are 32 bit 2's complement numbers. The value "P" is given by

P=A0*(U**0)+A1*(U**1)+A2*(U**2)+A3*(U**3)+.....+AN*(U**N) .

"P" is computed from right to left using

P=A0+U*(A1+U*(A2+U*(A3+U*(A4+U*(A5+U*(A6+.....+U*(AN) .

The routine assumes that -1/4 <= U < 1/4 and "U" has the signed magnitude format [SU, (0.31.0)]*2**(-32) and that -1/4 <= P < 1/4 and "P" has the 2's complement format (1.31.0)*2**(-32) .

Whatever the starting location of the memory space that stores the "A" coefficients, they are assumed stored in the sequence:

Address	Item
"A"+ 0	A0(hi)
"A"+ 2	A0(lo)
"A"+ 4	A1(hi)
"A"+ 6	A1(lo)
"A"+ 8	A2(hi)
"A"+ 10	A2(lo)
"A"+ 12	A3(hi)
"A"+ 14	A3(lo)
.	.
.	.
.	.
"A"+4*N	AN(lo)
"A"+4*N+2	A2(hi) ;

the "A" coefficients are assumed to have the same form as "P". The "A" block is stored outside of this subroutine.

The routine is given the location of the last 2 byte word of the "A" coefficient block in R8. The degree of the polynomial is given in R7.

The degree of the polynomial is at least 1.

****
The entry branch and link register for this subroutine is RF. This subroutine, in turn, calls the subroutine, MULT32, as an internal subroutine (i.e., no BAL register is used) .

Registers directly required by the POLY32 routine are marked with a "#" below. Registers indirectly required by the MULT32 routine are marked with a "$" below.

Register:	RE	RD	RC	RB	RA	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0
POLY32 Use:		#			#	#	#	#					#		#
MULT32 Use:		$			$	$					$	$	$	$	$

ON ENTRY:
R7=N
R8=COEF[function]+4*N+2
R9=Uhi
RA=Ulo
RD=SU (the sign of U exists in the most significant bit location; all remaining bits are "0".)

ON EXIT:
R0=Plo
R2=Phi

1. POLY32 entry .

Register:	RE	RD	RC	RB	RA	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0
					Uhi	Ulo	loc	cnt							

2. R0=0(R8); R8=R8-2 . (Load R0 with ANlo; decrement the address register.)

Register:	RE	RD	RC	RB	RA	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0
Use:	SAV	SU			Uhi	Ulo	loc -2	cnt							AN lo

3. R2=0(R8); R8=R8-2 . (Load R2 with ANhi; decrement the address register.)

Register:	RE	RD	RC	RB	RA	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0
Use:	SAV	SU			Uhi	Ulo	loc -2	cnt					AN hi		AN lo

4. Bal(RF) to MULT32.MS . (V is the 2's complement multiplier result.)

Register:	RE	RD	RC	RB	RA	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0
Use:	SAV	SU			Uhi	Ulo	loc	cnt	bal				Vhi		Vlo

```
5. R0=R0+0(R8); R8=R8-2  . (No carry-in; save carry-out.)
   Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
            |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
        Use:|SAV| SU|   |   |Uhi|Ulo|loc|cnt| - |   |   |   |Vhi|   |Vlo|
            |   |   |   |   |   |   | -2|   |   |   |   |   |   |   | + |
            |   |   |   |   |   |   |   |   |   |   |   |   |   |   |Alo|
            |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
            |   |   |   |   |   |   |   |   |   |   |   |   |   |   |Alo|


6. R2=R2+0(R8); R8=R8-2  . (Carry-in.)
   Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
            |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
        Use:|SAV| SU|   |   |Uhi|Ulo|loc|cnt| - |   |   |   |Vhi|   |Alo|
            |   |   |   |   |   |   | -2|   |   |   |   |   | + |   |   |
            |   |   |   |   |   |   |   |   |   |   |   |   |Ahi|   |   |
            |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
            |   |   |   |   |   |   |   |   |   |   |   |   |Ahi|   |   |


 7. R7=R7-1  . (Decrement count register.)
    Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
             |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
         Use:|SAV| SU|   |   |Uhi|Ulo|loc|cnt| - |   |   |   |Ahi|   |Alo|
             |   |   |   |   |   |   |   | -2|   |   |   |   |   |   |   |


8. IF R7=0
         RETURN via RF register.
    Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| K1| R0|
             |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
         Use:|SAV| SU|   |   |Uhi|Ulo|loc|cnt| - |   |   |   |Ahi|   |Alo|


      Else
         Go to 4.

   End If.

END
```

3.2.8 MCU COMMON SUBROUTINE : MULT32

This subroutine multiplies a 2's complement 32 bit "A" times a 32 bit signed magnitude number, "U", to produce a 2's complement 32 bit "V". The low 32 bits of the product are dropped; the high 32 bits form "V".

****

Registers directly required by the MULT32 routine are marked with a "$" below.

Register:	RE	RD	RC	RB	RA	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0
MULT32 Use:		$			$	$					$	$	$	$	$

ON ENTRY:
R0=Alo
R2=Ahi
R9=Uhi
RA=Ulo
RD=SU (the sign of U)

ON EXIT:
R0=Vlo
R2=Vhi

1. MULT32 entry .

Register:	RE	RD	RC	RB	RA	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0
Use:		SU			Ulo	Uhi							Ahi		Alo

2. R4=X'8000' . (Generate magnitude of "A" input; sign in R4.)

Register:	RE	RD	RC	RB	RA	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0
Use:		SU			Ulo	Uhi					Use		Ahi		Alo

R4=R4.AND.R2 . (Sign of "A" ends up in R4(0), 0 in remaining bits.)

Register:	RE	~ RC	RB	RA	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0

3. IF R4=0

Continue .

Register:	RE	RD	RC	RB	RA	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0
Use:		SU			Ulo	Uhi					SA		Ahi		Alo

Else

R0=.NOT.R0 .

R2=.NOT.R2 .

Register:	RE	RD	RC	RB	RA	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0
Use:		SU			Ulo	Uhi					SA		Ahi		Alo

End If.

4. R4=R4.EXCLUSIVE OR.RD . (Sign of "V", the product output.)

Register:	RE	RD	RC	RB	RA	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0
Use:		SU			Ulo	Uhi					SV		Ahi mag		Alo mag

5. (R0, R1)=R0*R9 . (Cardinal multiply; Alo*Uhi.)

Register:	RE	RD	RC	RB	RA	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0
Use:		SU			Ulo	Uhi					SV		Ahi mag	Alo mag * Uhi mag lo	Alo mag * Uhi mag hi

6. R1=R2 .

Register:	RE	RD	RC	RB	RA	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0
Use:		SU			Ulo	Uhi					SV		Ahi mag	Ahi mag	Alo mag * Uhi mag hi

```
7. (R3, R2)=R2*R9  .  (Cardinal multiply; Ahi*Uhi.)
   Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
            |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
        Use:|   | SU|   |   |Ulo|Uhi|   |   |   |   | SV|Ahi|Ahi|Ahi|Alo|
            |   |   |   |   |   |   |   |   |   |   |   |mag|mag|mag|mag|
            |   |   |   |   |   |   |   |   |   |   |   | * | * |   | * |
            |   |   |   |   |   |   |   |   |   |   |   |Uhi|Uhi|   |Uhi|
            |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
            |   |   |   |   |   |   |   |   |   |   |   |mag|mag|   |mag|
            |   |   |   |   |   |   |   |   |   |   |   | lo| hi|   | hi|
```

```
8. R3=R3+R0   . (No carry-in; preserve carry-out.) "P"=|A|hi*Uhi+|A|lo*Uhi
   R2=R2+0    . (Carry-in.)
  Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
           |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
       Use:|   | SU|   |   |Ulo|Uhi|   |   |   |   | SV|Plo|Phi|Ahi| - |
           |   |   |   |   |   |   |   |   |   |   |   |   |   |mag|   |
```

```
9. R0=R1   . (Put |A|hi into R0.)
  Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
           |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
       Use:|   | SU|   |   |Ulo|Uhi|   |   |   |   | SV|Plo|Phi| - |Ahi|
           |   |   |   |   |   |   |   |   |   |   |   |   |   |   |mag|
```

```
10. (R0, R1)=R0*RA  .  (Cardinal multiply; Ahi*Ulo.)
  Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
           |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
       Use:|   | SU|   |   |Ulo|Uhi|   |   |   |   | SV|Plo|Phi|Ahi|Ahi|
           |   |   |   |   |   |   |   |   |   |   |   |   |   |mag|mag|
           |   |   |   |   |   |   |   |   |   |   |   |   |   | * | * |
           |   |   |   |   |   |   |   |   |   |   |   |   |   |Ulo|Ulo|
           |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
           |   |   |   |   |   |   |   |   |   |   |   |   |   |mag|mag|
           |   |   |   |   |   |   |   |   |   |   |   |   |   | lo| hi|
11. R0=R0+R3   . (No carry-in; preserve carry-out.) |V|=P+|A|hi*Ulo   .
    R2=R2+0    . (Carry-in.)
   Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
            |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
        Use:|   | SU|   |   |Ulo|Uhi|   |   |   |   | SV| - |Vhi| - |Vlo|
            |   |   |   |   |   |   |   |   |   |   |   |   |mag|   |mag|


12. IF R4=0
         Continue   . (Product sign bit is 0.)
  Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
           |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
       Use:|   | SU|   |   |Ulo|Uhi|   |   |   |   | - | - |Vhi| - |Vlo|


       Else
         R0=.NOT.R0   .
         R2=.NOT.R0   .
     Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
              |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
          Use:|   | SU|   |   |Ulo|Uhi|   |   |   |   | - | - |Vhi| - |Vlo|


    End If.


13. RETURN   .
    Register:| RE| RD| RC| RB| RA| R9| R8| R7| R6| R5| R4| R3| R2| R1| R0|
             |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
         Use:|   | SU|   |   |Ulo|Uhi|   |   |   |   | - | - |Vhi| - |Vlo|


END
```

3.2.9 MCU COMMON SUBROUTINE : NORMV

This subroutine takes 32 bits of fractional data input contained in two 16-bit halves and performs a floating point normalization. A VAX 32-bit format floating point number with sign and biased exponent becomes the resultant output.

Input registers:

```
R3  contains the High half of the input fraction.
R1  contains the Low half of the input fraction.
R10 contains a 2's complement scale factor (known bias)
    or '0' if none.
R14 contains the sign bit of the result; given as
    either X'8000' or '0'.
```

Output registers:

```
R2  High Half of VAX aligned, normalized input (sign and exponent).
R0  Low half of Vax aligned, normalized input.
```

```
0. ENTRY NORMV.

1. If the most significant half of the input is '0'
      then,   the input is less than 2**-16.
              If the least significant half of the input is '0',
              then,   the output must be true VAX '0'.

                   CLR R0
                   CLR R2
                   RETURN.

              else,  swap the input halves and adjust the exponent of R10
                     by 128-16 (bias plus a shift of 16).
```

```
                LR   R3,R1
                CLR  R1
                ADD  R10,#128-16.
```

else, some bits are set in the high half; add bias to exponent.

```
                ADD   R10,#128
```

2. Check the high byte of the input fraction for any bits set. Note R12 counts the number of shifts required to obtain bit(0) set in the input fraction; initialize R12 to 0.

```
                CLR  R12
                LR   R5,#X'FF00' (mask for high byte).
                AND  R5,R3
```

If no bits are set in the high half, a shift of 8 may be performed by a multiplication, and the exponent adjusted by 8:

```
                MLU  R3,#X'100'
                MLU  R1,#X'100'
                OR   R4,R1
                SUB  R10,#8    (exponent adjust).
```

Otherwise, starting with bit 0 and working through bit 6 of R3,
test each bit for a '1', and increment R12 if not set:

```
                BBS   (R3,0),NORMALIZE
                INCR  R12
                    .
                    .
                    .
                BBS   (R3,6)NORMALIZE
                INCR  R12.
```

3. (NORMALIZE).
Adjust the final exponent in R10 by the shift as indicated by R12. From the number of shifts required, obtain a shift multiplication

factor contained in Table SHF$ (R12 + SHF$) and perform the exact number of shifts required to normalize R3 and R1:

```
SUB  R10,R12
ASL  R12
ADD  R12,SHF$
MLU  R3,(R12)
MLU  R1,(R12)
OR   R4,R1.
```

The nomalized fraction is now contaned in registers R4 and R2.

4. Now align the normalized fraction of R4 and R2 into VAX format (mantissa bits only) and clear out the suppressed MSB of the mantissa:

```
MLU  R2,#X'100'
MLU  R4,#X'100'
AND  R4,#X'007F'.
```

R4 now contains the correct high half mantissa bits. R2 and R5 must be "ORed" to obtain the low half mantissa bits:

```
OR   R2,R5
LR   R0,R2.
```

R0 now contains the ouput data for the low half VAX format .

5. The exponent of R10 must be inserted into the high half of the VAX format, and the final sign bit of R14 "ORED" in:

```
MLU  R10,#X'100'  (shift the exponent).
OR   R11,R4
LR   R2,R11
OR   R2,R14

RETURN.
```

3.3 MCU SEQUENTIAL ALGORITHMS HOL INTERFACE

All of the MCU subroutines use identical interface conventions for input and output data formats. Each subroutine requires that input data be loaded (from MCU memory) into MCU registers R9 and R11. Upon completion of each subroutine, the output function will be contained in MCU registers R2 and R0. Error status is returned in R14.

Because the VAX 32-bit format requires two 16-bit MCU storage locations, a "high half" register (which includes the sign and exponent) and "low" half register (least significant mantissa bits) will be defined as shown below.

X input registers:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R11 = High half X	S	e 0	e 1	e 2	e 3	e 4	e 5	e 6	e 7	m 1	m 2	m 3	m 4	m 5	m 6	m 7

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R9 = Low Half X	m 8	m 9	m 10	m 11	m 12	m 13	m 14	m 15	m 16	m 17	m 18	m 19	m 20	m 21	m 22	m 23

Y output registers:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	^								^							
R2 = High half Y	S	e0	e1	e2	e3	e4	e5	e6	e7	m1	m2	m3	m4	m5	m6	m7

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R0 = Low Half Y	m8	m9	m10	m11	m12	m13	m14	m15	m16	m17	m18	m19	m20	m21	m22	m23

(The symbol, ^, indicates the location of the radix point for the value stored in a register.)

R14 = Status storage :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
s0	s1	s2	s3	s4	s5	s6	s7	s8	s9	s10	s11	s12	s13	s14	s15

The loading of the MCU registers upon input, and the eventual storage to MCU memory is the responsibility of either a HOL or MCL MACRO, however the recommended convention for storage is as follows:

ADDRESS	High Half Data
ADDRESS+2	Low Half Data

Note - All MCU registers will be destroyed upon completetion of each function, with the output and status as shown above.

Each MCU algorithm description provides the details of error detection in section 3.2 . For interface convenience, the specific error conditions have been summarized in Table 3.0; normal status is "0" for all functions.

Table 3.0 MCU SUBROUTINE ERROR STATUS

FUNCTION	ERROR STATUS (R15)	Y OUTPUT
Natural Logarithm....	'1' denotes overflow	X'7FFFFFFF'
	'2' denotes underflow	X'0'
	'3' denotes X negative	undetermined (original X)
Exponential..........	'1' denotes overflow	X'7FFFFFFF'
	'2' denotes undeflow	X'0'
Square Root..........	'3' denotes negative X	undetermined (original X)
Sine.................	'3' denotes X > 2**24	undetermined*
Cosine...............	'3' denotes X > 2**24	undetermined*
Arctangent...........	'0' always	Arctangent

* angular uncertainty nea 2 radians

4.0 USAGE OF THE MPP SCIENTIFIC SUBROUTINES

The following sections describe the filenames, library names, and conventions for using the MPP scientific functions decribed in this document. The information contained in this section describes the configuration of NASA MPP disk files in UIC [2,5]. The details of creation of libraries and tasks may differ slightly for use with other configurations.

Table 4.0 contains a list of all PRL and MCL filenames and their global entry points for the parallel array routines. The serial MCU routine filenames and global entry points are listed in Table 4.1.

Section 4.1 contains MPP applications programmer information.

Section 4.2 contains systems programmer information for generating the new VAX subroutines and/or libraries from source.

Section 4.3 describes the MCL macros that exist for the VAX 32-bit format functions, and contain programmer information for using the functions directly from MCL.

4.1 MCU APPLICATIONS

MCU applications programmers assume the existance of 'PELIBR.PST', (a global symbol table that must be used to build MCU tasks) and 'PELIBR.PTK' the parallel subroutine library task (that must be loaded into PECU memory to provide array functions). These files reside in UIC [2,1].

In order to preserve existing MPP applications, the VAX scientific functions described in this document have been incorporated into 'PELIBR.PTK'. (See Section 4.2 for instructions to include the VAX subroutines).

Existing array functions have not been affected, and existing MPP applications should experience no difficulty in using the new version of 'PELIBR.TSK'. The VAX subroutines reside in high PECU memory addresses, therefore entry points to previously existing subroutines have not changed. In the event of incompatibility with the new VAX version of the array functions, a task build of the MCU programs (with reference to the new VAX PELIBR.PST) will be required.

The VAX serial MCU algorithms and the MCU portions of the array functions have been included in the existing library file '[2,1]MCLIBR.MOL' (See Section 4.2 for instructions to include the VAX routines).

All existing MCU tasks must make reference to 'MCLIBR.MOL' ; therefore, if sequential MCU functions or parallel VAX functions are incorporated into existing applications, the task build procedure for the MCU portion will not be affected. New applications that may include VAX scientific functions, should refer to Section 3.5.4 of MPP User's Guide GER-17141 which describes the procedure for building an MCU program.

4.2 SYSTEM GENERATION and MAINTENANCE

This Section describes the procedures for incorporating the MPP VAX Scientific Subroutines into the MPP system. A complete list of filenames for the array subroutines, and MCU subroutines is provided in Table 4.0, and Table 4.1 . The files reside in UIC [2,5] of the NASA PDP-11 system disk.

Generation of the scientific functions presumes the existence of PE Subroutine object library [2,17]PELIBR.POL and MCU object library [2,1]MCLIBR.MOL . Files are obtained from [2,1]PELIBR.POL, and files are inserted into [2,1]MCLIBR.MOL; therefore, these libraries must exist prior to generation of the VAX scientific functions.

Table 4.0 ARRAY SCIENTIFIC SUBROUTINES
Filename References

PE SUBR MACRO NAME	MACRO SOURCE FILE	MCU SUBROUTINE SOURCE FILE	MCU GLOBAL ENTRY NAME	PECU SUBR SOURCE FILE	PECU GLOBAL ENTRY NAME
LNA	LNA.MCL	LNV.MCL	LN$V	LNV.PRL	LNV$
				NRMZV.PRL	NRMZV$
EXPA	EXPA.MCL	EXPV.MCL	EXP$V	EXPV.PRL	EXPV$
				EXPSHIFT.PRL	EXPSH$
				EXPUM.PRL	EXPUM$
SINA	SINA.MCL	SNCSNV.MCL	SNCS$V	VFSC1.PRL	VFSC1$
COSA	COSA.MCL	"	"	VFSC2.PRL	VFSC2$
SINCOS	SINCOS.MCL	"	"	VFSC3.PRL	VFSC3$
				VFSC4.PRL	VFSC4$
				VFSC5.PRL	VFSC5$
SQRTA	SQRTA.MCL	N/A	N/A	SQRTV.PRL	SQRTV$
ARCTNA	ARCTNA.MCL	N/A	N/A	ATANV.PRL	ATANV$

Table 4.1 MCU SCIENTIFIC SUBROUTINES
Filename References

MAIN CONTROL UNIT SUBROUTINES

MCU SUBROUTINE NAME	MCU SOURCE CODE	MCU GLOBAL ENTRY
LNM	LNMV.MCL	LNM$V
EXPM	EXPM.MCL	EXPM$V
SINM	SINM.MCL	SINM$V
COSM	COSMV.MCL	COSM$V
SQRTM	SQRTM.MCL	SQRTM$V
ATANM	ATANMV.MCL	ATNM$V
COMMON SUBROUTINES	MULT32.MCL NORMV.MCL POLY32.MCL SHFM.MCL	MULT32$ NORMV$ POLY32$ SHFM$

Several RSX-11M command files have been written to create the VAX functions from source, create intermediate libraries, and to build a new PE subroutine library task. These command files may be invoked in the logical order necessary to generate all the required files by executing the MCR command:

```
 MCR> @VAXLIBGEN
```

This invokes a command file which prompts the operator to make decisions on system generation steps. All, part, or none of the steps may be performed. This provides the operator with several options of starting, or terminating the system generation of the scientific functions.

The results of the VAXLIBGEN command file are:

- that all PECU object files are concatenated into:

  [2,5]VAXPELIBR.POL which is a new object library;

- all MCU object files are inserted into:

  [1,2]MCLIBR.MOL which is an existing object library.

- a new PECU Subroutine libray task and symbol table are created:

  [2,5]VAXPELIBR.PTK which is the task,
  and [2,5]VAXPELIBR.PST which is the symbol table.

The libraries should be maintained with the current versions of the VAX scientific functions.

From the command file VAXLIBGEN, as described above, the final PECU subroutine library task and symbol table are created in UIC [2,5] (VAXPELIBR.PTK and VAXPELIBR.PST).

These are temporary versions of the PECU subroutines that may be loaded and tried. After verification, these two files must be re-named and transferred to the library UIC [2,1] as PELIBR.TSK and PELIBR.PST to provide the system compatibility described in Section 4.1 .

A seperate command file may be requested to perform this transfer and re-name by executing:

```
MCR> @VAXSYSLIB
```

Note - This Transfer command file also may be selected as part of the previously mentioned VAXLIBGEN command file.

4.3 MCU MACROS

The array functions described in this document must initialize MCU Call Queue registers and request either an MCU or PECU program to perform the desired function. The operations required to execute array functions are listed in Section 2.2 . These operations have also been incorporated into MCL Macros for the array functions only. The interface requirements for the serial MCU functions are described in Section 3.3 ; since the MCU serial functions required only register loads and stores, Macros have not been developed for these functions.

The following Sections describe the Macros that exist for the VAX scientific functions. Table 4.0 lists the Macro name for each function. These Macros have been incorporated into the the MCL Macro library and may be called by any MCL program.

4.3.1 LNA - NATURAL LOGARITHM OF AN ARRAY

This instruction will compute the natural lorarithm of a VAX 32-bit floating point format variable in the array, and store the result in the array also in VAX 32-bit format.

Format	LABEL	COMMAND	ARGUMENTS
	[s]	LNA	X,Y,E,[T]

*Label The label field is optional.

*Command LNA

*Arguments Three arguments are required. T is an optional temporary storage array of at least 56 bits; if T is not specified, the top 56 bits of array memory will be used with the LSB at 973.

*X VAX 32-bit floating point source.

*Y VAX 32-bit floating point destination where Y = LN(X).

*E Error bitplane; Set where X was negative, Clear otherwise. Y not determined.

*T Optional parameter specifying a 56-bit temporary storage area to be used by this subroutine. If not specified, array memory starting at LSB 973 will be used as scratch area.

4.3.2 EXPA - EXPONENTIAL OF AN ARRAY

This instruction will compute the exponential of a VAX 32-bit floating point format variable in the array, and store the result in the array also in VAX 32-bit format.

Format	LABEL	COMMAND	ARGUMENTS
	[s]	EXPA	X,Y,E,[T]

*Label The label field is optional.

*Command EXPA

*Arguments Three arguments are required. T is an optional temporary storage array of at least 43 bits; if T is not specified, the top 43 bits of array memory will be used with the LSB at 973.

*X VAX 32-bit floating point source.

*Y VAX 32-bit floating point destination where Y = EXP(X).

*E 3-bit error array

E(0)	set where output clipped to 0 because X <2**-31.
E(1)	set where Y overflowed.
E(2)	set where Y underflowed

*T Optional parameter specifying a 43-bit temporary storage area to be used by this subroutine. If not specified, array memory starting at LSB 973 will be used as scratch area.

4.3.3 SQRTA - SQUARE ROOT OF AN ARRAY

This instruction will compute the square root of a VAX 32-bit floating point format variable in the array, and store the result in the array also in VAX 32-bit format.

Format	LABEL	COMMAND	ARGUMENTS
	[s]	SQRTA	X,Y,E,[T]

*Label The label field is optional.

*Command SQRTA

*Arguments Three arguments are required. T is an optional temporary storage array of at least 22 bits; if T is not specified, the top 22 bits of array memory will be used with the LSB at 973.

*X VAX 32-bit floating point source.

*Y VAX 32-bit floating point destination where Y = SQRT(X).

*E Error bitplane set where X was negative, clear otherwise.

*T Optional parameter specifying a 22-bit temporary storage area to be used by this subroutine. If not specified, array memory starting at LSB 973 will be used as scratch area.

4.3.4 SINA,COSA - SINE or COSINE OF AN ARRAY

These instructions will compute the sine or cosine of a VAX 32-bit floating point format variable in the array, and store the result in the array also in VAX 32-bit format.

Format	LABEL	COMMAND	ARGUMENTS
	[s]	SINA COSA	X,Y,[T]

*Label The label field is optional.

*Command SINA or COSA

*Arguments Two arguments are required. T is an optional temporary storage array of at least 90 bits; if T is not specified, the top 90 bits of array memory will be used with the LSB at 973.

*X VAX 32-bit floating point source.

*Y VAX 32-bit floating point destination where Y = SIN(X) or Y = COS(X).

*T Optional parameter specifying a 90-bit temporary storage area to be used by this subroutine. If not specified, array memory starting at LSB 973 will be used as scratch area.

4.3.5 SINCOS - SINE AND COSINE OF AN ARRAY

This intructions will compute the sine and cosine of a VAX 32-bit floating point format variable in the array, and store the results in the array also in VAX 32-bit format.

Format	LABEL	COMMAND	ARGUMENTS
	[s]	SINCOS	X,Y,Z,[T]

*Label The label field is optional.

*Command SINCOS

*Arguments Three arguments are required. T is an optional temporary storage array of at least 90 bits; if T is not specified, the top 90 bits of array memory will be used with the LSB at 973.

*X VAX 32-bit floating point source.

*Y VAX 32-bit floating point destination where Y = SIN(X) .

*Z VAX 32-bit floating point destination where Z = COS(X) .

*T Optional parameter specifying a 90-bit temporary storage area to be used by this subroutine. If not specified, array memory starting at LSB 973 will be used as scratch area.

4.3.6 ARCTNA - ARCTANGENT OF AN ARRAY

This instruction will compute the arctangent of a VAX 32-bit floating point format variable in the array, and store the result in the array also in VAX 32-bit format.

Format	LABEL	COMMAND	ARGUMENTS
	[s]	ARCTNA	X,Y,[T]

*Label The label field is optional.

*Command ARCTNA

*Arguments Two arguments are required. T is an optional temporary storage array of at least 82 bits; if T is not specified, the top 82 bits of array memory will be used with the LSB at 973.

*X VAX 32-bit floating point source.

*Y VAX 32-bit floating point destination where Y = ATAN(X) .

*T Optional parameter specifying a 82-bit temporary storage area to be used by this subroutine. If not specified, array memory starting at LSB 973 will be used as scratch area.