ELEMENTARY FUNCTIONS ON AN ARRAY PROCESSOR

Diane F. Davis
Floating Point Systems
Beaverton, Oregon

Abstract

The algorithms used for elementary functions on the FPS-164 array processor are described. In each case, the choice of the algorithm depends on the parallel hardware, the capability of the instruction word, and the precision desired. For some, the choice depends on the version -- either scalar or vector. Algorithms for the divide, square root, cosine/sine, exponential, and logarithm are discussed. Those for arctangent, tangent, arc-cosine/sine, cosh, sinh, and tanh are summarized.

Introduction

The FPS-164 array processor is designed to solve the many scientific problems that demand high throughput, high precision, and large data memories. In order to facilitate the task, there is a complete library of scalar elementary or intrinsic functions as well as an extensive library of basic vector routines. The algorithms used for the intrinsic functions are discussed as to how they fill the above needs.

Parallel Processing

The schematic of the FPS-164 architecture is shown in figure 1. In addition to the I/O processors and the program cache, there are eight separate synchronous units that can operate in parallel; they are main data memory, table memory, X and Y data pad registers, address or integer unit, floating adder, floating multiplier, and program execution control (i,e. branch). The clock cycle time is 167 nsec.

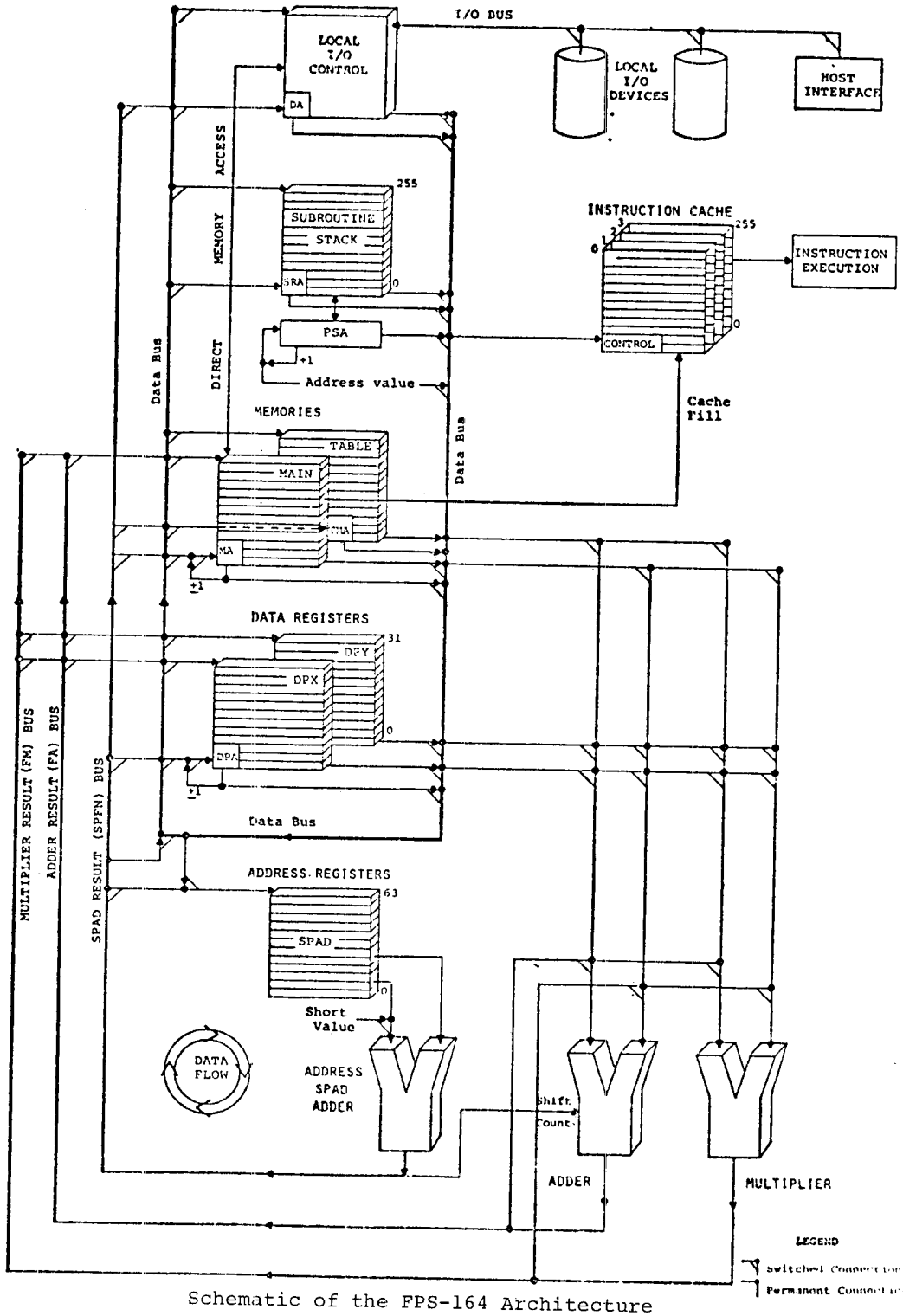
High throughput on the FPS-164 is achieved by parallel hardware complemented by parallel software. A 64 bit instruction word, illustrated in figure 2, allows almost complete parallel control over the hardware units. This can be understood by comparing figures 1 and 2. Both reads and writes can be made simultaneously to the X and Y data pad registers. With a clock cycle time of 167 nsec and all the possibilities allowed, one instruction can contain up to 10 operations giving 60 million operations per second. Several of the operations take more than one cycle but these are pipelined. The memory unit has 3 stages, the floating adder has 2 stages and floating multiplier has 3 stages; in each of these units, a new operation may be initiated every cycle. When maximum use is made of the floating adder and floating multiplier, there are 12 million floating point operation per second (12 megaflops).

Precision

In order to accomodate the high precision required, the data word consists of 64 bits, 53 in the mantissa and 11 in the exponent. See figure 3. This provides 15-16 decimal digits of precision in the mantissa and a dynamic range of approximately $10^{\pm 308}$. The mantissa is normalized to a fraction F with the radix point immediately following the sign bit and to the left of the representation of F with the sign bit and the most significant bit opposite such that $\frac{1}{2} \leq F < 1$ for positive values or $-1 \leq F < -\frac{1}{2}$ for negative values. The mantissa is expressed in 2's complement notation and the exponent uses excess (bias) notation. The 64 bit data word is maintained in the registers, the floating point arithmetic units, data bus and data memories. The low end of the mantissa is extended to three guard bits which are used for even rounding.

Algorithm Choice

The algorithms used for the intrinsic functions are designed to optimize the parallelism and precision of the processor making efficient tradeoffs when necessary. Taylor's method was chosen over Newton's iterative method for both the scalar divide and square root. The iterative method must wait for one intermediate result before it can calculate the next and so is not appropriate for scalar routines in a parallel pipeline processor. On the other hand, the Taylor's method allows several operations to occur



Schematic of the FPS-164 Architecture

Figure 1

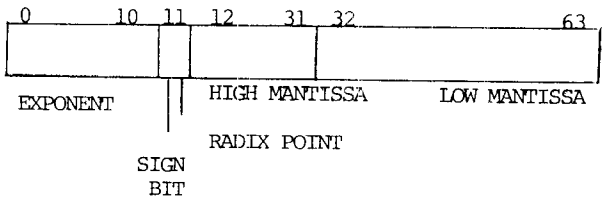
0 – 13	14 – 22	23 – 31	32 – 50	51 – 55	56 – 63
INTEGER OR ADDRESS	FLOATING ADD	BRANCH	DATA PADS DPX AND DPY	FLOATING MULTIPLY	MEMORY DATA TABLE

Figure 2

FPS-164 Instruction Field Layout

Figure 3

FPS-164 Data Word



simultaneously. The vector routines do benefit from the fewer operations in Newton's method.

Divide Algorithm

The full precision scalar floating point divide, called RDIV, is done in software in 23 cycles. RDIV of Y/X begins with a reciprocal lookup from a 256 value table (residing in table memory, TMROM) of the most significant 8 bits of the fraction F where

$$X = F*2^{E}.$$

Let

$$F = Z+R,$$

where Z contains the first 8 bits and R contains the remainder. The table lookup gives the inverse,

$$TI = \frac{1}{Z}$$

Then

$$Y/X = Y/F*2^{E} = \frac{Y*2^{-E}}{Z+R} = \frac{(Y/Z)*2^{-E}}{1+R/Z} = \frac{TI*Y*2^{-E}}{1+TI*R} =$$

$$Y/X = \frac{TI*Y*2^{-E}}{1+A}$$

where $A = TI*R$.

The Taylor expansion $(1+A)^{-1}$ is carried out through the A^{7} term. The error due to cut off of the series is bounded by A^{8}. The reciprocal of a fraction between $\frac{1}{2}$ and 1 gives values

$$1 < TI \leq 2.$$

The largest value of the remainder is 2^{-8}. Thus the largest value of A is 2^{-7} giving

$$A^{8} \leq 2^{-56},$$

an insignificant number in a 53 bit fraction. The Taylor expansion gives

$$(1+A)^{-1} = 1 - A + A^{2} - A^{3} + A^{4} - A^{5} + A^{6} - A^{7}.$$

In this form, the polynomial requires 6 multiplies and 7 adds. This is also the situation when written as

$$(1+A)^{-1} = 1-A(1-A(1-A(1-A(1-A(1-A(1-A))))))$$

In fact, this representation is the worst for a parallel processor because each operation waits on the results of the previous one. Happily, a factorization as

$$(1+A)^{-1} = (1 - A)(1 + A^{2})(1 + A^{4})$$

may be implemented in 4 floating multiplies and 3 floating adds and, in addition, operations may be performed in parallel. The entire algorithm contains 8 floating multiplies and 6 floating adds. By producing software that makes use of the hardware's parallel and pipeline structure, the routine is coded for optimum speed.

The parallelism of the computer can be more fully realized in a vector routine such as the vector divide which produces a result every 7 cycles. The vector divide, called VDIV, for Y/X uses Newton's interactive method as follows:

R_0 = FRECIP X (floating point approximation to 1/X)

$R_1 = R0*(2.0-R0*X)$

$R_2 = R1*(2.0-R1*X)$

$R_3 = R2*(2.0-R2*X)$

$Y/X = Y*R_3$

The opcode FRECIP (located in the floating add group) gives an approximation of 8 bits precision to the reciprocal. It differs from the approximation used in RDIV in that the table is software transparent, it includes both the exponent and mantissa, and it is an approximation where the bits after the most significant eight are all zeros. In addition, if the operand of FRECIP is zero, an interrupt bit is set in the status register; in fact, it is used in the scalar divide for this reason only and not for the approximation given. Because the convergence of the method is quadratic, the number of significant bits doubles with each approximation. The initial value R_0 has 8 bits of precision giving the final value 64 bits of precision. The three iterations show that this method is not suitable for a scalar routine because one iteration must wait for the result from the previous one and thus the algorithm allows no parallel or pipeline instructions. However, this algorithm contains only 7 floating multiplies and 4 floating adds and is more suitable for vectorization than the Taylor's method. In a vector routine, floating adds and multiplies of one vector help fill the pipeline of those for another vector. When the algorithm is applied to a vector

$$Yi/Xi \quad i = 1,2,...N$$

different phases of the calculation are going on simultaneously for several input vectors.

Figure 4

Vector Divide Loop

The Vector Divide Y/X produces a result every 7 cycles. Simultaneously, 6 vector elements are being processed. The memory fetch pipline is at least 3 cycles, the MULT pipeline is 3 cycles and the ADD pipeline is 2 cycles. Only one ADD or MULT can be initiated in each cycle; the ADD and MULT "pushes" are supplied by the corresponding initiation on another element. In this example, several intermediate results are saved in the ADDER because of data pad and/or data bus conflicts. If the latter are included in the tally, an ADD is initiated in all by one cycle and a MULT is initiated in every cycle giving 11.2 megaflops.

Vector Divide Loop

	1	2	3	4	5	6
1		ADD FRECIP X (1) $\frac{}{X}$ = R_0	ADD push	MULT push	MULT R_2*X	MULT push
2		ADD push save X in data pad	MULT R_0*(2-R_0*X) = R1	MULT push	MULT push	ADD save R_3 in adder R_3+0
3		MULT R_0*X	MULT push	ADD 2-R1*X	MULT push Fetch next Y	ADD push
4		MULT push	MULT push	ADD push	ADD 2-R_2*X	MULT Y*R_3 = Y/X
5	Fetch next X	MULT push	ADD save R1 in adder R1+0	MULT R1*(2-R1*X) = R_2	ADD push	MULT push
6		ADD 2-R_0*X	ADD push	MULT push	MULT R_2*(2-R_2*X) = R_3 y in data	MULT push pad
7			MULT R_1*X	MULT push	MULT push	Store Y/X

The actual routine illustrated in figure 4 consists of a 7 cycle loop with 6 columns indicating that 6 input vectors are being processed simultaneously.

Square Root Algorithm

The technique followed for the square root is similar to that for the divide where Taylor's method is used for the scalar version and Newton's method is used for the vector version. The scalar square root, called SQRT, consists of a software table look and a Taylor series carried out to the 6th power of the variable as follows

$$(X)^{\frac{1}{2}} = (F*2^{E})^{\frac{1}{2}} = 2^{E/2}*(Z+R)^{\frac{1}{2}} = 2^{E/2}*Z^{\frac{1}{2}}*(1+R/Z)^{\frac{1}{2}}$$

$$= 2^{E/2}*TS*(1+A)^{\frac{1}{2}} \text{ for even E}$$

$$= 2^{E/2}*(2)^{\frac{1}{2}}*TS*(1+A)^{\frac{1}{2}} \text{ for odd E}$$

where TS is the square root table lookup approximation and A = TI*R as before. The address for the table is given by the same 8 significant bits used for the divide table. Because only positive operands are considered, the table consists of only 128 values. The Taylor expansion with B = A/2 is

$$(1+A)^{\frac{1}{2}} = 1 + B - \frac{B^2}{2} + \frac{B^3}{2} - \frac{5}{8}B^4 + \frac{7}{8}B^5 - \frac{21}{16}B^6$$

with the cut off error bounded by the first term omitted is given by $\frac{33}{16}B^7$. As before $A \leq 2^{-7}$

$$B = A/2 \leq 2^{-8}$$

and so $\frac{33}{16}B^7 \lesssim 2^{-55}$

which is insignificant with a 53 bit mantissa. If the series is written as

$$1 + B(1 - B(\frac{1}{2} - B(\frac{1}{2} - B(\frac{5}{8} - B(\frac{7}{8} - \frac{21}{16} B)))))$$

there are 6 floating multiplies and 6 floating adds but, as before, this representation is not suitable for parallel processors. Although the representation

$$1 + B + B^2 (\frac{-1}{2} + \frac{1}{2}B) + B^4 (\frac{-5}{8} + \frac{7}{8} B - \frac{21}{16} B^2)$$

requires 7 floating multiplies and 6 floating adds, it can be handled more efficiently in a parallel processor. The entire routine has 11 floating multiplies and 12 floating adds and takes 28 cycles.

The vector square root using Newton's method with 3 iterations produces a result every 11 cycles. The algorithm is as follows

A_0 = FRSQRT X (floating point approximation to $1/X^{\frac{1}{2}}$)

$A_1 = \frac{A_0}{2} * (3.0 - A_0^2 * X)$

$A_2 = \frac{A_1}{2} * (3.0 - A_1^2 * X)$

$A_3 = \frac{A_2}{2} * (3.0 - A_2^2 * X)$

$X^{\frac{1}{2}} = X * A_3$

This contains 10 floating multiplies and 7 floating adds which is less than that required in the scalar version. The loop takes 11 cycles and has 5 columns. The opcode FRSQRT (located as is FRECIP in the floating add group) gives an approximation of 8 bits precision with exponent and trailing zeros to the reciprocal of the square root. If the operand is negative, an interrupt bit is set in the status register and for this purpose only, it is used in the scalar routine. For an operand of zero, FRECIP gives a finite result such that $X * A_0^{\frac{1}{2}}$ is zero.

Cosine and Sine Algorithm

The cosine and sine is also evaluated by a table lookup and interpolation. In fact, of the three large table read only memories, the cosine/sine is the largest, consisting of 4096 values between 0 and $\pi/2$. In this case, the scalar and vector routines use the same algorithm. The cosine/sine table forms the foundation of the Fast Fourier Transform routines used for signal and image processing. Here, the angle in radians is scaled by $\frac{4096}{\pi/2}$ to its normalized value, $\frac{X}{\pi/2} * 4096$, from which the 12 lowest bits of the nearest integer is taken as the table address in the quadrant. The next two higher bits denote the quadrant. The remaining higher bits are ignored. In this manner, the table accomodates angles without losing precision in modulo 2π and also negative values (2's complement notation). The table gives both the cosine and sine in the following manner. The table address is multiplied by two by left shifting with zero fill in the low bit. The low bit then designates either the cosine or sine table such that two adjacent values of the table address point consecutively to the cosine and sine of the same angle. A bit called the FFT bit is set in the status register so that the hardware interprets an increment in the cosine table address to give the sine address of the same angle.

The interpolation giving the cosine and sine is given by the trigonometric identities

$$COS(X) = COS(T+D) = COS(T)*COS(D) - SIN(T)*SIN(D)$$
$$SIN(X) = SIN(T+D) = SIN(T)*COS(D) + COS(T)*SIN(D)$$

where T is the integer part of the angle used for the table address and D is the fractional difference in radians. Values of COS(D) and SIN(D) are approximated by the Taylor series

$$COS(D) = 1 - \frac{D^2}{2!}$$

$$SIN(D) = D - \frac{D^3}{3!}$$

where the maximum error is given by the maximum value $D = 2^{-12}$ in the first term omitted as

$$\frac{D^4}{4!} \sim \frac{2^{-48}}{2^4} = 2^{-52}$$

close to the precision of the processor. The scalar cosine is evaluated in 30 cycles and the scalar sine in 32 cycles. The vector routines produce a result every 8 cycles.

Exponential and Logarithmic Algorithms

The exponential and logarithmic functions are outlined as follows. The exponential e^X is given by the following.

Let $2^Y = e^X$

then $Y = X * LOG_2(e)$

If $Y = I + FR$ where I = integer part, FR = fraction part

then $2^Y = 2^{I+FR} = 2^I * 2^{FR}$

where 2^{FR} is calculated by a polynomial of degree 11. The coefficients of the polynomial are given in the University of Chicago notes.[1] The scalar version takes 32 cycles and the vector version produces a result every 18 cycles.

The logarithmic function is given by the following algorithm.

$LOG_e(X) = LOG_e(F*2^E) = LOG_e(F) + E*LOG_e(2)$

$= LOG_e(F) + E*LOG_e(2) + \frac{1}{2}LOG_e(2) - \frac{1}{2}LOG_e(2)$

$= (E - \frac{1}{2})*LOG_e(2) + LOG_e(SQRT(2)*F)$

where E = exponent of X

and F = mantissa of X

Then if $Y = SQRT(2)*F$

and $Z = (Y-1) / (Y+1)$

then $LOG_e(Y) = Z*P(Z^2)$

where P is a polynomial of degree 6 of Z^2. The coefficients are found in Hart and Cheney, #2665.[2]

The scalar version takes 55 cycles and the vector version produces a result every 20 cycles.

Summary of Intrinsic Functions

A summary of the intrinsic functions and their vector versions is shown in the table 1. Times are shown for the scalar and vector versions in the range indicated. Routines below the line were not strictly "vectorized'; in these, the vector routine only calls the scalar routine for each input variable.

Table 1

Summary of Intrinsic Functions

and their Vector Versions

Speed in USEC

NAME	SCALAR	RANGE	VECTOR
Real Divide	3.833	$X \neq 0$	1.167
Reciprocal	3.833	$X \neq 0$	1.000
Integer Divide	4.500	$X \neq 0$	1.167
Square Root	4.667	$0 \leq X$	1.833
Cosine	5.000	all	1.333
Sine	5.333	all	1.333
Exponential base e	5.333	$X < 709$	3.000
Exponential base 10	5.833	$X < 308$	3.167
Logarithm base e	9.167	$0 < X$	3.333
Logarithm base 10	9.667	$0 < X$	3.333
Arctangent	4.667	$0 \leq X \leq \tan(\pi/8)$	5.000
	9.833	$\tan(\pi/8) < X \leq \tan(3\pi/8)$	5.000
	10.500	$\tan(3\pi/8) < X$	5.000
Tangent	9.667	$-\pi \leq X \leq \pi$	10.333
Arcosine	6.833	$\lvert X\rvert \leq \frac{1}{2}$	7.500
	12.500	$\frac{1}{2} < \lvert X\rvert \leq 1$	13.167
Arcsine	6.833	$\lvert X\rvert \leq \frac{1}{2}$	7.500
	12.167	$\frac{1}{2} < \lvert X\rvert \leq 1$	12.833
Cosh	10.833	all	11.500
Sinh	3.167	$0 \leq \lvert X\rvert < 0.3465$	3.833
	12.000	$0.3465 < \lvert X\rvert$	12.567
Tanh	8.000	$0 \leq \lvert X\rvert \leq 0.5493$	8.667
	11.833	$0.5493 < \lvert X\rvert < 19.4$	12.500
	1.333	$19.4 \leq \lvert X\rvert$	2.000

Table 2 Comparison of Intrinsic Functions on AP-120 and FPS-164 Speed in USEC

Routine	AP-120				FPS-164			
	SCALAR		VECTOR		SCALAR		VECTOR	
Real Divide	3.7	Table & P3	1.7	Table & P3	3.8	Table & P7	1.2	N3
Square Root	3.8	Table & P3	1.8	Table & P3	4.7	Table & P6	1.8	N3
Cosine	5.0	P9	1.3	Table & P2	5.0	Table & P2	1.3	Table & P2
Sine	4.5	P9	1.3	Table & P3	5.3	Table & P3	1.3	Table & P3
Exponential Base e	4.2	P6	2.3	P6	5.3	P11	3.0	P11
Exponential Base 10			2.3	P6			3.2	P11
Logarithm Base e	4.0	P6	2.7	P6	9.2	P13	3.3	P13
Logarithm Base 10	4.7	P6	2.7	P6	9.7	P13	3.3	P13
Artangent	8.7	P11	9.7	calls scalar	4.7-10.0	P21	5.0	P21

Pn refers to order of polynomial, n
Nn refers to number n of iterations in Newton's method

Table 2 compares the speed and methods used by the AP-120 and the FPS-164. The increased precision of the FPS-164 over the AP-120 (15-16 compared to 7-8 decimal digits) is achieved with little or no sacrifice in speed. In one case where the routines are slower, the LOG_e/LOG_{10}, the old AP-120 algorithm is retrieved as an alternate or QUICK LOG.

Table 3 shows the relative error of the intrinsic functions when compared with double precision CYBER 175 function values (word size 120 bits, mantissa 96 bits). For each function, 1000 double precision data points in the appropriate range were generated on the CYBER 175. With the mantissas rounded to 53 bits, the functions were then evaluated on the CYBER and also rounded to 53 bits. The same input values were used to calculate the functions using the FPS-164 Simulator on the PRIME. Table 3 summarizes the results using these definitions.

Absolute Error = ABS (164 Value - Rounded CYBER 175 Value).

Relative error is according to 164 value, so

Relative Error = Absolute Error / 164 Value.

The median relative error indicates the 15 + decimal digits of precision desired.

Table 3

Relative Error of Functions

	MAXIMUM RELATIVE ERROR	MEDIAN RELATIVE ERROR
ACOS	0.1125D-14	0.0000D 00
ALOG	0.4679D-13	0.3005D-15
ALOG10	0.3726D-11	0.2874D-15
ASIN	-0.2015D-14	0.2757D-15
ATAN	0.8837D-15	0.0000D 00
COS	0.2874D-12	0.3569D-15
COSH	0.8727D-13	0.4441D-15
CSQRT	0.1202D-14	0.2752D-15
EXP	0.8238D-13	0.4645D-15
RDIV	0.1281D-14	0.2589D-15
SIN	0.3274D-12	0.3042D-15
SINH	0.8675D-13	0.7990D-15
SQRT	0.9643D-15	0.2425D-15
TAN	0.9511D-11	0.7688D-15
TANH	0.1113D-14	0.0000D 00

Courtesy of Harry Sedinger,

Floating Point Systems

Conclusion

The algorithms used for elementary functions are designed to optimally use the parallelism and precision of the FPS-164 array processor. In some cases (divide and square root), the choice of the algorithm depends on whether the scalar or vector version is desired. The parallel properties of the processor are most manifest in the vector algorithms and it is here that speed is of greatest importance.

Acknowledgements

For their help and cooperation, I thank R. Norin and the Application Software Group at Floating Point Systems. I acknowledge the work of C. Hsiung (now at Cray Research) while at Floating Point. Special thanks go to H. Sedinger of Floating Points Systems, for extensive assistance and discussions and for supplying the error analysis data. Thanks also to A. Charlesworth of Floating Point Systems, for many helpful comments.

References

1). Hirondo Kuki, "University of Chicago Computation Center Report-Mathematical Functions", Feb. 1966

2). John F. Hart, E. W. Cheney, et al., "Computer Approximations", SIAM series (edited by Johy Wiley and Sons, Inc., New York, 1968)

3). William S. Dorn, Daniel D. McCracken, "Numerical Methods with Fortran IV Case Studies" (edited by John Wiley and Sons, Inc. New York, 1972)