Computer System Support for Scientific and Engineering Computation

Lecture 13 - June 14, 1988 (notes revised July 6, 1988)

This paper discusses three topics. First how the Intel 80387 computes transcendental functions, then some observations about the Intel 80960 floating point architecture, and finally some comments about the IEEE 754 standard.

1 Transcendental Approximations Using Cordic

This section addresses two questions:

- How does the Intel 80387 use the Cordic algorithms in its approximation of transcendental functions?
- Why does the Intel 80387 use the Cordic rather than some other approach?

1.1 The Intel Transcendental Milieu

Intel put support for transcendental functions into the instruction set of the 8087. This is the embarkation point for understanding why and how Intel's current floating-point processors support logarithmic, exponential, and trigonometric instructions.

1.1.1 Lineage

The 8087 has five transcendental functions: FPTAN, FPATAN, FYL2X, FYL2XP1, and F2XM1. Each instruction uses the Cordic technique and is highly accurate [1]. The mathematics and algorithms were developed by W. Kahan. For implementation reasons, the instructions restrict the operands to reduced domains, and this limitation necessitates a software interface layer for nearly every application.

Intel's Common Elementary (CEL) function library [2] provides a relatively complete set of the common real and complex valued elementary transcendental functions. Many math packages are disappointing in the results they deliver. CEL is more complete and produces better results than any other math package that runs on Intel numeric processors. For a variety of reasons, though, few people know of its existence and even fewer use it. The appropriateness of supplying hardware support for just kernel approximation functions has been questioned because of CEL's lack of commercial success.

The 80387 was Intel's first major re-implementation of the 8087. It incorporates myriad improvements over the original 8087 implementation, mostly by bringing the arithmetic up

to date with the IEEE standard. In the area of support for transcendental functions, more complete argument reduction was provided for the 5 instructions, and 3 new instructions were added: FSIN, FCOS, and FSINCOS (which delivers sine and cosine simultaneously). These additions are a compromise response to requests for single-instruction approximation functions suitable for use as inline math functions.

The 80960 has a significantly different floating-point architecture than the 80387, but shows the common 8087 transcendental function heritage. It supports the same functions (less FSINCOS) as the 80387, but provides them in three precisions and implements full argument range reduction for the trigonometric instructions. The two architectures correspond more closely than it might appear on the surface: the 80387 was implemented by wrapping its architecture around the 80960's floating-point unit. One consequence of this is that tradeoffs made in one design are reflected in the other.

1.1.2 Guidelines for Putting Approximations in Silicon

Committing an approximation function to hardware is unlikely to be an unqualified success. Delivering any result other than the mathematically correct result rounded to the destination format is open to criticism. Delivering a result more slowly than a software implementation can raise questions of why the function is in hardware. Dedicating significant amounts of chip area to support transcendental functions is usually better spent improving the speed of vector multiplication. In short, the silicon implementation should be fast, accurate, and cost nothing.

The 80387 and 80960 approach is to provide a small number of kernel functions from which all the usual functions can be readily calculated. The exceptions to this guideline, sine and cosine, are included because they are perhaps the most common approximation functions, and can be effectively calculated from tangent using the few extra bits of precision available in the internal data paths.

The approximations must be highly accurate. The algorithms implemented in the 387/960 turn out to be accurate to over 62 bits, with the error usually under two ulps of 64 significant bits. This accuracy is economically achievable with the careful choice of approximation functions and by taking advantage of the three or four extra mantissa bits available in the hardware to support rounding.

In the absence of correctly rounded results, the most important property to maintain is monotonicity. Preserving the monotonic properties of the mathematical functions tends to preserve the characteristics of the mathematical computations carried out in floating point arithmetic, better than highly accurate but non-monotonic approximations do. Choosing monotonic algorithms turns out to be difficult. One advantage of the Cordic algorithms used in the 387/960 is that they have been proved to be monotonic.

The approximations must be fast. They must compare favorably in execution speed with responsibly implemented table-driven approximations on the same hardware. One way to look at this requirement is that the user should not feel tempted to reimplement the functions. Most approximation algorithms can be improved with minimal expense up to a particular limit for the hardware, after which the improvements become much more expensive and difficult. Another way to interpret the "fast" requirement is the approximation algorithm should not go beyond the point well implemented by the hardware. In the 387/960 environment, this means maintaining all intermediate computations in no more than 68 significant bits, and generally eschewing divisions (and to a lesser extent multiplications).

Last, the approximation instructions should implement full operand range reduction. One important reason to support the full range is that the slightly wider internal data paths are usually sufficient to avoid roundoff errors in the operand reduction. Another reason is that despite all the obvious limitations, programmers will try to use these instructions as substitutes for the corresponding library functions.

1.2 Approximations, such as Tangent

The following sections show in greater detail how the 80387 and 80960 go through the various steps and compute an approximation of the tangent function.

The approximation of a function across the full operand range is usually an involved process. The algorithms are characteristically case analyses followed by straight-line code. Broadly speaking, the case analyses fall into three categories: special operand handling, algebraic reductions, and the function approximation proper. Only the last category has much room for creative algorithms.

Special operand handling weeds out non-numerical operands, operands where the function isn't defined, and operands where the function is exactly defined and no approximation is needed. Algebraic reductions exploit symmetries inherent in the target mathematical function to reduce the operand range to a smaller, more tractable range over which the approximation is carried out. The proper choice of algebraic reductions is crucial to the preservation of mathematical identities in the approximated function. Special operand handling and algebraic reductions steps are found in the 387/960 but not the 8087.

After algebraic reductions, the 387/960 approximation technique is to divide the three ranges of operands. Tiny operands are often trivially approximated using the source operand with perhaps a multiplication. Intermediate range operands are approximated using a simple rational function based on one of the Padé approximations* of the function. The larger operands are transformed into the intermediate or tiny range using the Cordic argument reduction technique, where one of the two previous approximations can be used.

1.2.1 Special Operand Handling

The special operand handling for the tangent function is straightforward. The usual NaN propagation rules are followed for tan(NaN). A subnormal operand is handled as if it were normalized into a wider internal format and then considered as any other normalized operand. The invalid operation exception is signaled for $\tan(\pm\infty)$.

The most interesting special case is $\tan(\pm 0)$, which returns its source operand ± 0 as an exact result. All other finite operands signal the inexact exception†.

1.2.2 Algebraic Reductions

Each numeric operand u is algebraically reduced to u' such that $0 < u' < \pi/4$ in preparation for the approximation step.

The identity

$$\tan(u) = \tan(u \text{ rem } \pi)$$

*A Padé approximation is a quotient of two polynomials. The coefficients of the polynomials are chosen so that as many as possible Taylor coefficents of the quotient match the Taylor coefficents of the function being approximated.

†This is mathematically correct, because evaluating a trigonometric function at a nonzero rational value always results in an irrational number. Look up the Gelfand-Schneider theorem in a number theory book.

is guaranteed by computing

$$u' = u \text{ ieee\_rem } \pi$$

and working with that modified operand, which takes on values in the range $-\pi/2 < u' < \pi/2$. Since $\tan(-u) = -\tan(u)$, and $\tan(\pi/2 - u) = \cot(u)$ for $0 < u < \pi/2$. the correct value is returned by computing:

$$\tan(u) = \begin{cases} T(u') & \text{if } 0 < u' < \pi/4 \\ \frac{1}{T(\pi/2-u')} & \text{if } \pi/4 < u' < \pi/2 \\ -T(-u') & \text{if } -\pi/4 < u' < 0 \\ \frac{-1}{T(\pi/2+u')} & \text{if } -\pi/2 < u' < -\pi/4 \end{cases}$$

where $T(x)$ represents the approximation of the tangent of x.

The preceding paragraph's references to π are a prevarication. Actually, the 387/960 use a machine representation of π with 66 significant bits. Using the slightly wider representation avoids the problematic cases (sign selection at the zeros and singularities) in the sine, cosine and tangent where, with only 64 bits of significance, the computed u' is 0 modulo the machine-π. Note that full range reduction requires that the trigonometric instructions be interruptable and resumable because the full range reduction can take a long time. (The worst case for the 80960 is nearly 5 milliseconds.)

1.2.3 The 8087 Tangent Approximation

The 387/960 use a modified version of the Cordic and rational approximations used by the 8087; the 8087 approach is explained first, and the 387/960 technique is described in terms of its differences from the 8087.

The first step of the 8087 tangent approximation is the Cordic "pseudo-divide", where an operand u is decomposed into a sequence of quotient bits $q_i \in \{0,1\}$ and a reduced operand u_n, using:

$$u = \sum_{i=0}^{n-1} q_i \cdot \tau_i + u_n, \quad u_n < \tau_{n-1} < 2^{-(n-1)}$$

where $\tau_i = \arctan(2^{-i})$. Since $\arctan(x) < x$ for postive x, this decomposition guarantees that $\tau_{n-1} < 2^{-(n-1)}$. The decomposition can always be performed because each time τ_i is subtracted, the relation $u_i < \tau_{i-1}$ continues to hold. Initially, $u = q_0\tau_0 + u_1$, with $q_0 = 0$ and $u_1 = u < \frac{\pi}{4} = \tau_0$. Using induction, assume $u_i < \tau_{i-1}$. If $u_i < \tau_i$, $q_i = 0$ so $u_{i+1} = u_i < \tau_i$; otherwise, $q_i = 1$ and $u_i = \tau_i + u_{i+1}$, so again

$$u_{i+1} = u_i - \tau_i < \tau_{i-1} - \tau_i < \tau_i.$$

The last inequality relies on $2\tau_i > \tau_{i-1}$. This result can be derived from the identity

$$\tan(x \pm y) = \frac{\tan(x) + \tan(y)}{1 \mp \tan(x)\tan(y)}$$

by taking the arctangent of both sides of

$$\tan(\arctan(\frac{x}{2}) + \arctan(\frac{x}{2})) = \frac{x}{1 - x^2/4} > x.$$

The rational approximation computes two values, x_n and y_n, whose quotient approximates the tangent function:

$$\frac{y_n}{x_n} = \frac{3 \cdot u_n}{3 - u_n^2}.$$

The relative error E_r of this approximation is about than $\frac{1}{45} u_n^4$[1].

The Cordic "pseudo-multiply" preserves the relationship of x and y being proportional to the $\cos(u)$ and $\sin(u)$:

$$\begin{aligned} y_i &= y_{i+1} + q_i \cdot x_{i+1} \cdot 2^{-i} \\ x_i &= x_{i+1} - q_i \cdot y_{i+1} \cdot 2^{-i} \end{aligned}$$

That is, if $\frac{y_{i+1}}{x_{i+1}} = \tan(u_{i+1})$, and if y_i and x_i are computed as shown, then from the tangent sum-of-angles identity given above, it is easy to check that $\frac{y_i}{x_i} = \tan(u_{i+1} + q_i \tau_i) = \tan(u_i)$. The accuracy of the pseudo-multiply depends on the accuracy of the original τ_i and the cumulative error when computing the x and y values.

The monotonicity of this approximation is demonstrated by proving that the rational approximation is monotonic, that the Cordic transformations are monotonic and that the composition is monotonic.

The 8087 chooses $n = 16$. This guarantees that $E_r < 2^{-65}$ for the rational approximation. Choosing $n = 33$ would allow the simple approximation $\tan(u_n) = u_n$, at the cost of usually 17 more Cordic pseudo-divide and pseudo-multiply steps. It turns out that the rational approximation is faster, since it only requires one multiply, and two shifted adds.

1.2.4 The 387/960 Tangent Approximation

The 387/960 tangent approximation incorporates several improvements to the 8087 approach. Algorithmically, two of the improvements are interesting.

First, since the 387/960 do argument reduction and don't require the argument $u \leq \pi/4$. the x and y results can be divided as y/x or x/y, depending on whether the tangent or cotangent branch of the approximation is desired. This division happens with the unrounded. 67-bit x and y values, which delivers a more accurate result than the 8087 approach of rounding x and y and delivering them as results.

The other significant improvement is, rather than computing x_i as a number proportional to $\cos(u)$, computing x_i as proportional to $1 - \cos(u)$. The changes are to split the rational approximation step into two sequences. The first computes x_n and y_n as:

$$\frac{y_n}{x_n} = \frac{3 \cdot u_n}{u_n^2}.$$

The pseudo-multiply then computes:

$$\begin{aligned} y_i &= y_{i+1} + q_i \cdot (3 - x_{i+1}) \cdot 2^{-i} \\ x_i &= x_{i+1} + q_i \cdot y_{i+1} \cdot 2^{-i} \end{aligned}$$

---

[1] The Taylor series for $\tan(u)$ is $u + \frac{u^3}{3} + \frac{2u^5}{15} + \cdots$ while $\frac{3u}{3-u^2} = u(1 - \frac{u^2}{3})^{-1} = u(1 + \frac{u^2}{3} + (\frac{u^2}{3})^2 + \cdots)$. The difference of these two series is $\frac{u^5}{45} + O(u^7)$

After this, the actual tangent approximation $T(u)$ is:

$$T(u) = \frac{y_0}{3 - x_0}.$$

The new approach allows x_i to accumulate error only in the last few bits, whereas the relative error in the original approach tends to swamp the x value.

At the cost of another 32 mantissa constants and a somewhat slower tangent Cordic approximation, the x and y could be computed to the actual $\sin(u)$ and $1 - \cos(u)$ rather than just proportional to the sine and cosine.

1.2.5 Cordic Equations and Implied Exponents

The mathematical formulation of the Cordic transformations tends to obscure the magnitudes of the numbers being computed. The Cordic transformations have the important property that the magnitudes of the intermediate and final results are tightly bounded. The 387/960 exploits these bounds to maximize the precision of the approximations.

For the Cordic (and other) operations, the 387/960 hardware treats each floating-point number as a fixed-point mantissa with an implied exponent. Both the mantissa and exponent are manipulated in scratch registers of the appropriate width.

The inner loop for the tangent pseudo-divide is roughly this (for clarity in the pseudo-code presentation of the algorithms below, the record elements ".m" and ".e" represent the mantissa and (unbiased) exponent portions of the number):

```
/* u.e is both the exponent of u and the loop counter */
loop while u.e >= -15;
    q := q shl 1;       /* the bits q[i] are packed into q */
    if u.m >= arctan_2[u.e].m then
        u.m := u.m - arctan_2[u.e].m;
        q := q + 1;     /* q[i] = 1 */
    end if;
    u.m := u.m shl 1;
    u.e := u.e - 1;
end loop;
```

On the 387/960, each iteration of the pseudo-divide loop requires one clock cycle.
The inner loop for the tangent pseudo-multiply is roughly this:

```
y.e = -15;
x.e = 2*y.e;
loop while q <> 0;
    if (q and 1) <> 0 then
        x.m := (x.m + y.m) shr 2;
        y.m := (3 - (y.m shr (-x.e)) + x.m) shr 1;
    else
        x.m := x.m shr 2;
        y.m := y.m shr 1;
    end if;
    x.e := x.e + 2;
    y.e := y.e + 1;
```

```
        q := q shr 1;
    end loop;
```

On the 387/960, the pseudo-multiply loop requires one clock cycle if the quotient bit is 0, or five cycles if the quotient bit is 1.

Both the pseudo-multiply and pseudo-divide loops may iterate less than sixteen times. The pseudo-divide jumps into the first iteration where the quotient bit might be non-zero, and the pseudo-multiply stops as soon as the final x and y results are known. For single precision computations, no rational approximation is necessary after the Cordic reduction of the operand.

1.3 Implementation Requirements and Costs

The other six transcendental approximations are similar in complexity and implementation to the tangent. One potentially non-obvious attribute of the Cordic algorithm for arctangent is that it uses the same set of mantissa constants used by the tangent approximation. (The arctangent pseudo-divide does simple shifts and adds, and the pseudo-multiply uses the arctangent constants.) A similar inverse relationship holds between the logarithm and exponential instructions: only fifteen constants of the form $\log_2(1 + 2^{-i})$ are required.

It turns out that only four Cordic approximations are required: $\tan(u)$, $\arctan(y/x)$, $-\log_2(1-u)$, and $\exp_2(u) - 1$. The two logarithm functions use different argument reductions to get to the same core approximation function. Because the 387/960 does full range reduction on the logarithm approximation, the result is guaranteed to be monotonic (unlike the 8087/CEL implementation). The sine and cosine functions are calculated using the algebraic half-angle identities with tangent:

$$\begin{aligned} \sin x &= \frac{2\tan(x/2)}{1+\tan^2(x/2)} \\ \cos x &= \frac{1-\tan^2(x/2)}{1+\tan^2(x/2)} \\ &= 1-\tan(x/2)\sin(x) \end{aligned}$$

In the 80387, FSINCOS requires one additional multiplication beyond the algebraic computations performed for FSIN alone.

The argument for supporting transcendental instructions has always been stronger for the 80387 than it was for the 80960. Besides compatibility issues, the 80387 is a dedicated floating-point processor, whereas the 80960 is a general-purpose processor. This translates into the 960 having more alternatives for other ways to best use the chip area. The end result was the 960 FPU design is area limited, and generally trades off speed for space. However, the 387 dependency on the 960 design set the base guidelines for functionality.

When evaluating the cost of implementing transcendental instructions, it is convenient to divide the costs into two categories: hardware and microcode. For the 387/960, the hardware costs were relatively small, but the microcode cost was more significant.

Both the 80387 and 80960 are microcoded machines, so the additional costs of decoding extra opcodes for transcendental and dispatching the microcode is neglible. The major hardware costs are illustrated by the dotted area in Figure 1.

The Cordic operations require 33 mantissa constants (67 bits wide) and a 16-bit shift register for the quotient bits. The rational approximations need two additional mantissa constants for the logarithm and exponent. The only other significant hardware impact is

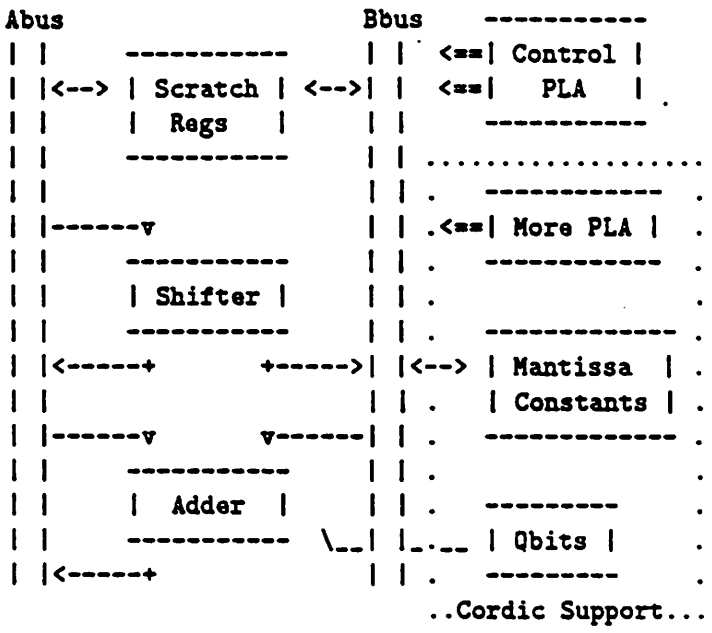


Figure 1: A Generic Floating-Point ALU with Cordic Support

Item	Sq Mils	% FPU area
16-bit shift register	180	0.9
40 PLA minterms	625	3.1
33 mantissa constants	500	2.5
Total	1305	6.5

Table 1: FPU Area for Transcendental Support

the control PLA, which has 40 of 115 minterms dedicated to implementing the 8 Cordic operations and the primitive arithmetic operations needed to compute the rational approximations.

The chip area required for this hardware support is summarized in Tables 1 and 2.

1.4 Measures of Effectiveness

Although the size of the implementation is a valid measure of the quality of a hardware-supported transcendental function implementation, the more interesting and more important measures are execution speed and result quality.

Table 3 lists the execution times, in microseconds, of the transcendental instructions.

Table 4 summarizes the results of running the Whetstone benchmark. This benchmark is quoted because of its unnaturally high sensitivity to transcendental function performance, especially cosine. The numbers represent thousands of Whetstones per second.

Table 5 summarizes the results of running Alex Liu's elementary function accuracy and monotonicity tests. The minimum and maximum observed result is reported for the tested

Processor	*Microcode*		*Microcode Area*		*Hardware+Microcode*	
	Words	% ROM	Sq Mils	% chip	Sq Mils	% chip
80387	850	33.2	1200	1.4	2505	3.0
80960	315	10.3	1070	0.7	2375	1.5

Table 2: Chip Area for Transcendental Support

Instruction	16 MHz	20 MHz	25 MHz
$z = 2^x - 1$	20.9	16.7	13.4
$z = y \log_2(x)$	27.4	21.9	17.5
$z = y \log_2(1 + x)$	36.3	21.0	16.8
$z = \text{atan2}(y, x)$	21.9	17.5	14.0
$z = \tan(x)$	20.2	16.2	12.9
$z = \sin(x)$	27.6	22.1	17.6
$z = \cos(x)$	27.6	22.1	17.6

Table 3: 80960 Transcendental Instruction Times

System	MWhetstones	
	Single	Double
Sun386i	1.3	1.0
Sun-3/280+68882	1.4	1.4
80960KB, 16MHz	3.4	3.3
Sun-3/280+FPA	4.1	2.9
80960KB, 20MHz	4.2	4.0
Sun-4/280	5.8	3.9

Table 4: Various Whetstone Benchmark Times

Function	Interval	Min	Max	NME
Single Precision				
SIN(X)	[+0.000e+00, +1.570e+00)	-5.3100e-01	+5.3100e-01	0
COS(X)	[+0.000e+00, +1.570e+00)	-5.3100e-01	+3.0000e+00	0
EXP(X)	[-1.037e+00, +1.008e+00)	-5.0200e-01	+5.0200e-01	0
EXPM1(X)	[-1.037e+00, +1.008e+00)	-5.0800e-01	+5.0800e-01	0
LOG1P(X)	[-2.928e-01, +4.142e-01)	-5.3100e-01	+5.3100e-01	0
ATAN(X)	[-6.553e+04, +6.553e+04)	-5.1600e-01	+5.1600e-01	0
LOG(X)	[+7.071e-01, +1.414e+00)	-5.3100e-01	+5.3100e-01	0
Double Precision				
SIN(X)	[+0.000e+00, +1.570e+00)	-5.4100e-01	+5.3600e-01	0
COS(X)	[+0.000e+00, +1.570e+00)	-1.8686e+01	+5.3300e-01	0
EXP(X)	[-1.037e+00, +1.008e+00)	-5.0200e-01	+5.0300e-01	0
EXPM1(X)	[-1.037e+00, +1.008e+00)	-5.0900e-01	+5.0900e-01	0
LOG1P(X)	[-2.928e-01, +4.142e-01)	-5.3200e-01	+5.3500e-01	0
ATAN(X)	[-6.553e+04, +6.553e+04)	-5.1700e-01	+5.1600e-01	0
LOG(X)	[+7.071e-01, +1.414e+00)	-5.3500e-01	+5.3200e-01	0
Extended Precision				
SIN(X)	[+0.000e+00, +1.570e+00)	-1.1970e+00	+1.6350e+00	0
COS(X)	[+0.000e+00, +1.570e+00)	-4.0000e+04	+1.6580e+00	0
EXP(X)	[-1.037e+00, +1.008e+00)	-1.5040e+00	+1.3090e+00	0
EXPM1(X)	[-1.037e+00, +1.008e+00)	-1.9670e+00	+2.5880e+00	0
LOG1P(X)	[-2.928e-01, +4.142e-01)	-1.0670e+00	+1.0040e+00	0
ATAN(X)	[-6.553e+04, +6.553e+04)	-1.4480e+00	+1.7720e+00	0
LOG(X)	[+7.071e-01, +1.414e+00)	-1.0730e+00	+1.1450e+00	0

Table 5: Alex Liu's Elementary Function Test Results for 80960

interval for each function, along with the number of monotonicity errors (NME) detected.

The 80960 numbers were obtained selecting 64 partitions with 2500 random points per partition. This caused the test of all three precisions to run for most of a three-day weekend on a dedicated 80960 system. The large errors seen in cosine are due to the test program not correcting for the machine representation of π used by the 80960[§].

No results are currently available for the 80386/80387.

More interesting is a comparison of hardware approximation speeds to today's best software implementations. Two high quality software packages are those of K-C Ng and Peter Tang. Ng's is a double precision package distributed with the Berkeley Unix 4.3bsd libm source. Tang's is a single precision package that uses large tables and double precision arithmetic to simplify approximations. Neither package is directly comparable with the 387/960 approximations, although both have important, comparable qualities. Both packages share the property of being highly accurate (error bounded by about two ulps), and no monotonicity violations.

The double precision package computes results roughly eleven bits less accurate than the 387/960. For the higher precision more terms must be added to the polynomial ap-

[§]When $x \approx \frac{\pi}{2}$, $\cos(x) = \sin(\frac{\pi}{2} - x) \approx \frac{\pi}{2} - x$, so the value of cosine is very sensitive to the stored value of π.

Library	Time Units	Operations	Constants
Double	49-52	13–16 adds, 16 mul, 1 div	13 (12 shared)
Single	19	3 adds, 6 mul, 1 div	5 (all unique)
80960	14	2 adds, 1 mul, 1 div, 1 cordic	16 (shared)

Table 6: Comparison of Polynomial Tangent Approximation Execution Speed

proximation. The double precision package takes the approach of minimizing the number of constants required, which is attractive for a microcode implementation.

The single precision package would also need several more terms to compute an approximation as accurate as the 387/960. This package's approach of using the wider precision for intermediate results matches the advantage exploited by microcode for better precision, although microcode doesn't have the substantial padding that a single precision result computed using double precision affords. The large table approach taken by the single precision package is unacceptable for a microcode implementation, but fair to consider because any hardware/microcode solution will be measured against software solutions.

Table 6 summarizes how the tangent approximation compares between the two software packages and the 80960 hardware implementation. The argument is assumed to be in the $[0, \pi/4)$ interval since all three implementations apply the same special operand checks and algebraic reductions to reduce the operand to that range. The inapplicabilities of the packages, as discussed above, are ignored. The register setup and branching overhead, memory latency for constant fetches, and so on are assumed to be comparable for all implementations, and therefore not counted; only the conceptual floating-point operations are counted. These assumptions are all generous to the software implementations.

The "time units" column represents the total operation time for the approximation, normalized to a generic 80960 add time unit. Multiplies are optimistically considered twice an add time, divides twice a multiply time, and a Cordic 3 times slower than a divide. The "constants" column reports the number of unique floating-point constants required by the approximation.

So, for the 80960 hardware, the Cordic approach approximates the tangent function more accurately and faster than some very good software libraries. On the other hand, tangent is probably one of the most favorable comparisons. The most demanding comparisons would be $\sin(x)$, $\cos(x)$, $\ln(x)$, and $\exp(x)$.

1.5 Why Cordic?

It is commonly thought that in today's world of multiplier arrays and cheap memory, transcendental approximations are best implemented with polynomial algorithms and large table interpolation. But the above data show that for the 387/690 the Cordic approach is faster and more accurate.

But speed is not the only consideration in the 387/960 context:

- The 80387 requires a high degree of compatibility with the 8087. Using the same approximation technique ensures this compatibility.
- Fast and accurate polynomial or rational approximations may not, even today, be known.

- Space limitations in the 80960 significantly affect FPU implementation. There is insufficient die area for an array multiplier or for storing large tables. This precludes any algorithm that requires a large number of multiplications or table interpolation.
- The Cordic algorithms are well understood by Intel. The microcode requirements, constant table sizes, and special purpose hardware costs are all easy to estimate from prior experience.
- Polynomial approximations can be difficult to prove monotonic. The 387/960's Cordic and rational approximations have been proved to be monotonic.

If the 387/960 were to be reimplemented with a fast hardware multiplier so that a software polynomial approach was faster than the present Cordic hardware approach, other considerations could still mitigate in favor of the Cordic approach.

2 A Case for 80960-Style Extended Precision

The 80960 microprocessor architecture presents an IEEE floating-point programming model in which floating-point operands, operations, and exceptions are well integrated with the rest of the architecture. This section is a collection of insights obtained during the definition implementation, and subsequent programming of the 80960 processor. For an overview of the 80960 architecture, and its support for floating-point operations and data types, see [4]. One novel attribute of the architecture is its support for mixed-precision arithmetic. This support leads to a natural and desirable style of evaluating floating-point expressions.

2.1 Computing Q = X*Y/Z

Miriam Blatt's "One Line" C Program to compute `Q = X*Y/Z`, found in Lecture 7a, looks like the following when coded for the 80960 C compiler:

```
#define REAL    float
#define LEN     1

asm REAL logb(REAL x)
{
%reglit(LEN) return; reglit(LEN) x;
        logbnr  x,return
}

asm REAL scalb(REAL x, int e)
{
%reglit(LEN) return; reglit(LEN) x; reglit e;
        scalor  e,x,return
}

/*
 * Compute so as to avoid all spurious overflow and underflow (sic).
 *      q = x*y/z;
 */
```

```
REAL
mblatt(REAL x, REAL y, REAL z)
{
        int ex, ey, ez;
        REAL sx, sy, sz;
        REAL sq;

        ex = (int) logb(x);
        sx = scalb(x, -ex);
        ey = (int) logb(y);
        sy = scalb(y, -ey);
        ez = (int) logb(z);
        sz = scalb(z, -ez);

        sq = sx*sy;
        sq = sq/sz;
        return scalb(sq, ex+ey-ez);
}
```

This compiles into:

```
        .align  4
        .globl  _mblatt
_mblatt:
        logbnr  g0,g13          # ex = (int) logb(x);
        cvtzri  g13,r3
        subi    r3,0,r11        # sx = scalb(x, -ex);
        scaler  r11,g0,r7
        logbnr  g1,g13          # ey = (int) logb(y);
        cvtzri  g13,r15
        subi    r15,0,r11       # sy = scalb(y, -ey);
        scaler  r11,g1,r6
        logbnr  g2,g13          # ez = (int) logb(z);
        cvtzri  g13,r14
        subi    r14,0,r11       # sz = scalb(z, -ez);
        scaler  r11,g2,r5
        mulr    r7,r6,r4        # sq = sx * sy;
        divr    r5,r4,r4        # sq = sq / sz;
        addi    r3,r15,r11      # return scalb(sq, ex+ey-ez);
        subi    r14,r11,r9
        scaler  r9,r4,g0
        ret
```

While better than sorting the numbers, this is not a particularly fast way to compute Q. The approach also has other deficiencies: for example, it does not correctly handle cases where the operands are infinities or NaNs.

For the 80960, the one-line C program to compute the expression is the simple writing of the expression, which compiles to the following code:

```
# q = x*y / z;
        mulr    g0,g1,fp0
        divr    g2,fp0,g0
```

2.2 Extended is not Quad

It is important to distinguish "doubling" a data type from "extending" a data type. You go to Double from Single, or Quad from Double, because you want a different data type. You go from Double to Double-Extended because you want to simplify the evaluation of double expressions.

There is an implicit assumption, not there when a precision is doubled, that extended precision is almost as fast as the base precision being extended. A reasonable rule of thumb is to expect the performance impact of using extended precision to be no more than 10 percent over using the base precision. A larger impact seems to trap users into avoiding extended precision for routine expression evaluation.

The wider exponent range of extended precision is at least as important as the extended precision because it eliminates most concerns about spurious overflow and underflow. Extra exponent range is cheap and does not affect performance.

2.3 Extended Precision Reduces Cycle Count

Consider how to accurately compute $\sqrt{x^2+y^2}$ in double precision without the benefit of extended precision. Ignoring some setup conditions, here is how the cabs is done in the Berkeley 4.3bsd math library:

$$\mathrm{cabs}(x,y) = \begin{cases} x + \dfrac{y}{\sqrt{1+(x/y)^2}+x/y} & \text{if } x/y > 2 \\ x + \dfrac{y}{(\sqrt{2}+1)+\frac{x-y}{y}+\frac{(x/y)^2-2}{\sqrt{1+(x/y)^2}+\sqrt{2}}} & \text{if } x/y \le 2 \end{cases}$$

Clearly an extended precision evaluation of $\sqrt{x^2+y^2}$ will be faster here. The difference in speed attainable by not supporting extended precision will never account for the additional time to compute the additional adds and divides.

Admittedly, not all functions are this difficult to compute in double precision. However, nearly every non-trivial expression is easier to accurately compute with an extended precision. In the limit as operations get faster and have more uniform execution times, extended precision allows faster evaluation of expressions by sheer reduction in operation counts.

2.4 Mixing Extended and Other Data Types Reduces Cycle Count

It is probably not sufficient to support operations on extended operands so that explicit conversions to and from extended precision are required before operating on the data.

Consider the evaluation of

$$z = \sqrt{x^2+y^2}$$

on a machine that has separate arithmetic instructions for each precision and a full complement of conversion instructions:

```
fcvtd.e x,r0
fmul.e  r0,r0,r0
fcvtd.e y,r1
```

```
fmul.e  r1,r1,r1
fadd.e  r0,r1,r0
fsqrt.e r0,r0
fcvte.d r0,z
```

Conversions between floating-point data formats is relatively cheap to do at the execution interface. As floating-point operations get faster, the conversions will start to account for a significant amount of the execution time. The solution is not to avoid extended precision, but to support extended precision operands mixed with other data widths in operands. The computation of z looks like this on the 80960:

```
mulrl   x,x,fp0
mulrl   y,y,fp1
addrl   fp0,fp1,fp0
sqrtrl  fp0,z
```

2.5 Problematic Extended Implementations

In extended-accumulation register models, like the 68881 and the 8087, the results always are stored in extended precision. The means that a double rounding will be incurred when the computed result is stored back in single or double format.

The *precision control* abomination exists to mitigate the double rounding problem. In the 68881, for example, one can set the precision control to a narrower format and obtain a correctly rounded single or double precision result. However, by doing this one loses the benefits of extended precision. The 8087 family doesn't lose all the benefits, in that the significand is rounded but the exponent range remains extended. Unfortunately this means that the 8087 family cannot produce correctly rounded double precision results in the event of overflow or underflow[1].

In both cases, changing the precision control is rarely done. This is because, done with any regularity, it significantly slows down the code.

2.6 How iC960 Compiler uses Extended

Here's what DAXPY looks like in 80960 assembly language:

```
#       /*
#        * level-1 BLAS simplified
#        */
#       void
#       daxpy(int n, REAL da, REAL *dx, int incx, REAL *dy, int incy)
#       {
#               int i;
        .align  4
        .globl  _daxpy
_daxpy:
#               for (i = 0; i < n; i += 1)
        lda     (g4)[g0*8],r3
```

[1] However, rounding a number first to double precision and then to single precison is the same as rounding immediately to single precision, as was proven earlier in these lectures.

```
        cmpobge g4,r3,.I80
        subo    8,g6,g6
.I81:
#                       dy[i] = dy[i] + da*dx[i];
        ldl     (g4),r4
        addo    8,g4,g4
        cmpo    r3, g4
        addo    8,g6,g6
        ldl     (g6),r6
        mulrl   r4,g2,fp1
        addrl   r6,fp1,r6
        stl     r6,(g6)
        bge     .I81
.I80:
        ret
#       }
```

An example how key variables can use extended precision is seen in the 80960 implementation of DDOT:

```
#       /*
#        * forms the dot product of two vectors.
#        * uses unrolled loops for increments equal to one.
#        * jack dongarra, linpack, 3/11/78.
#        */
#       REAL
#       ddot(int n, REAL *dx, int incx, REAL *dy, int incy)
#       {
#               int i;
#               long double dtemp;
#
        .align  4
        .globl  _ddot
_ddot:

#               dtemp = (REAL) 0.0;
        movre   0f0.0,fp0

#               for (i = 0; i < n; i += 1)
        lda     (g1)[g0*8],r3
        cmpobge g1,r3,.I99
.I100:
#                       dtemp = dtemp + dx[i]*dy[i];
        ldl     (g1),r12
        ldl     (g3),r8
        addo    8,g1,g1
        cmpo    r3,g1
        addo    8,g3,g3
```

```
        mulrl   r12,r8,fp3
        addrl   fp0,fp3,fp0
        bge     .I100
.I99:
#               return dtemp;
        movl    fp0,g0
        ret
# }
```

3 Some Problems with the IEEE 754 Standard

1. IEEE 754 has not been enough to get libraries "in the spirit of 754". Libraries need to be characterized in terms of quality, just like the arithmetic.

2. The trapping exception handling model wrong. It is the worst part of the 80960 floating-point architecture. Exponent wrap around is useless. Instead, hardware should deliver original operands and default result for traps. This allows consistent software/hardware exception handling. Also need to spec what operations can change results (e.g. comparisons? integer conversions?).

3. Extended formats should not need to support denormalized numbers. Should be explicit strong recommendations on how to use and think of extended.

4. The required double rounding of wider→narrower should be relaxed.

5. Integer overflow in association with fp operations should be better specified? Draw dividing line between int/fp exceptions. Spec integer overflow as something different than invop.

6. Round to integral value operation should always produce exact result.

7. The rounding mode should be compilation mode. Allow option of encoding in instruction (not dynamically changeable). Library routines effectively required to save/restore. Mode prevents compile-time constant evaluation. Mode not useful for interval arithmetic. Use as indicator for numerical stability is questionable: recompile ok[‖]. And why not 3 modes: rn, rz, ro (for bindec conversions).

8. Expression evaluation needs good guidelines.

9. Binary/decimal conversions too loosely specified. Too much hand waving. Too few good routines out there. Need good test suite. S/W implementation issue, not hardware. Difficult issues address formatting in fixed amount of space.

10. Truncated integer conversion needs better ack in spec.

11. NaNs need better spec, as is aren't and can't be portably used.

[‖] Unless you are testing the stability of commercial code, for which the source is not usually available

References

[1] Rafi Nave. "Implementation of Transcendental Functions on a Numerics Processor". *Microprocessing and Microprogramming* 11 (1983) 221–225. North-Holland.

[2] *80287 Support Library Reference Manual.* Intel Corporation. 1985.

[3] Eugene H Spafford, John C. Flaspohler. *A Report on the Accuracy of Some Floating Point Math Functions on Selected Computers.* Software Engineering Research Center, Georgia Institute of Technology. GIT-SERC-86/02.

[4] *80960KB Programmer's Reference Manual.* Intel Corporation, 1988.

Transcendental Approximations Using Cordic

- The Intel Transcendental Milieu
- Approximations, such as Tangent
- Implementation Requirements and Costs
- Measures of Effectiveness
- Why Cordic?

Valerio
14th June

Lineage

- 8087
- CEL library
- 80387
- 80960

Guidelines for Putting Approximations in Silicon

- kernel functions
- accurate
- monotonic
- fast
- full operand reduction

Approximations, such as Tangent

- Special Operand Handling
- Algebraic Reductions
- The 8087 Tangent Approximation
- The 387/960 Tangent Approximation
- Cordic Equations and Implied Exponents

Special Operand Handling

- usual rules for tan(NaN)
- $\tan(\pm\infty)$ signals invalid operation
- $\tan(\pm 0) = \pm 0$

Algebraic Reductions

- $\tan(u) = \tan(u \text{ rem } \pi)$

- Tangent/Cotangent

$$\tan(u) = \begin{cases} T(u') & \text{if } 0 < u' < \pi/4 \\ \dfrac{1}{T(\pi/2 - u')} & \text{if } \pi/4 < u' < \pi/2 \\ -T(-u') & \text{if } -\pi/4 < u' < 0 \\ \dfrac{-1}{T(\pi/2 + u')} & \text{if } -\pi/2 < u' < -\pi/4 \end{cases}$$

- machine-π and true π

The 8087 Tangent Approximation

- Pseudo Divide

$$u = \sum_{i=0}^{n-1} q_i \cdot \tau_i + u_n$$

- Rational Approximation

$$\frac{y}{x} = \frac{3 \cdot u_n}{3 - u_n^2}.$$

- Pseudo Multiply

$$\begin{aligned} y_i &= y_{i+1} + q_i \cdot x_{i+1} \cdot 2^{-i} \\ x_i &= x_{i+1} - q_i \cdot y_{i+1} \cdot 2^{-i} \end{aligned}$$

The 387/960 Tangent Approximation

- Pseudo Divide

$$u = \sum_{i=0}^{n-1} q_i \cdot \tau_i + u_n$$

- Rational Approximation, Part 1

$$\frac{y_n}{x_n} = \frac{3 \cdot u_n}{u_n^2}.$$

- Pseudo Multiply

$$\begin{aligned} y_i &= y_{i+1} + q_i \cdot (3 - x_{i+1}) \cdot 2^{-i} \\ x_i &= x_{i+1} + q_i \cdot y_{i+1} \cdot 2^{-i} \end{aligned}$$

- Rational Approximation, Part 2

$$T(u) = \frac{y_0}{3 - x_0}.$$

Implied Exponents, Pseudo Divide

```
loop while u.e >= -15;
    q := q shl 1;
    if u.m >= arctan_2[u.e].m then
        u.m := u.m - arctan_2[u.e].m;
        q := q + 1;
    end if;
    u.m := u.m shl 1;
    u.e := u.e - 1;
end loop;
```

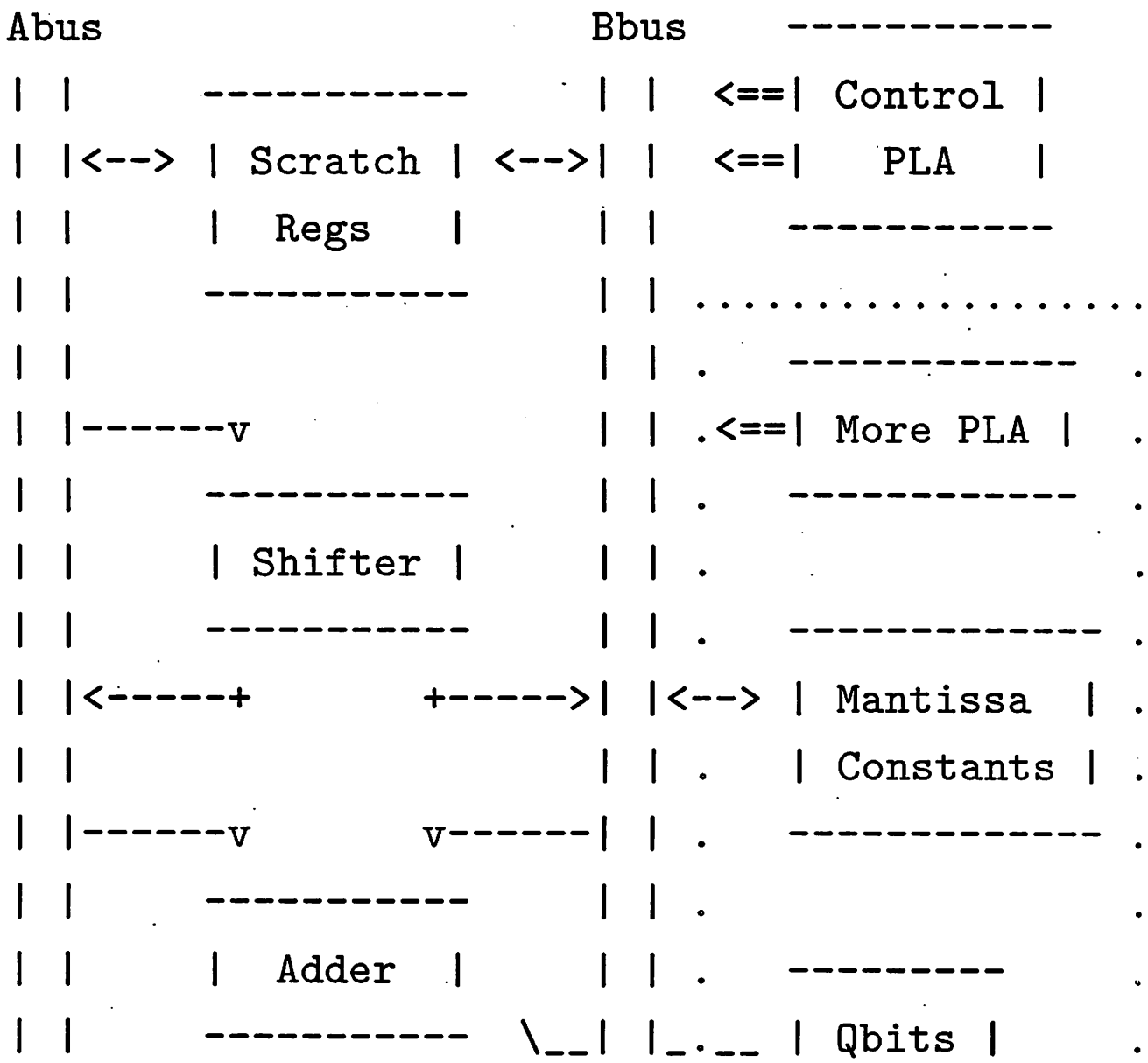
Implied Exponents, Pseudo Multiply

```
y.e = -15;
x.e = 2*y.e;
loop while q <> 0;
    if (q and 1) <> 0 then
        x.m := (x.m + y.m) shr 2;
        y.m := (3 - (y.m shr (-x.e)) + x.m)
    else
        x.m := x.m shr 2;
        y.m := y.m shr 1;
    end if;
    x.e := x.e + 2;
    y.e := y.e + 1;
    q := q shr 1;
end loop;
```

Cordic Flows

- $\arctan(y/x)$ uses $\arctan(2^{-i})$ constants
- $\exp_2(u) - 1$ uses $\log_2(1 + 2^{-i})$ constants
- $-\log_2(1 - u)$ uses $\log_2(1 + 2^{-i})$ constants
- sin/cos from $\tan(u/2)$

A Generic Floating-Point ALU with Cordic Support



FPU Area for Transcendental Support

Item	Sq Mils	% FPU area
16-bit shift register	180	0.9
40 PLA minterms	625	3.1
33 mantissa constants	500	2.5
Total	1305	6.5

Chip Area for Transcendental Support

Processor	*Microcode*		*Microcode Area*	
	Words	% ROM	Sq Mils	% chip
80387	850	33.2	1200	1.4
80960	315	10.3	1070	0.7

80960 Transcendental Instruction Times

Instruction	16 MHz	20 MHz	25 MHz
$z = 2^x - 1$	20.9	16.7	13.4
$z = y \log_2(x)$	27.4	21.9	17.5
$z = y \log_2(1 + x)$	36.3	21.0	16.8
$z = \text{atan2}(y, x)$	21.9	17.5	14.0
$z = \tan(x)$	20.2	16.2	12.9
$z = \sin(x)$	27.6	22.1	17.6
$z = \cos(x)$	27.6	22.1	17.6

Various Whetstone Benchmark Times

System	MWhetstones	
	Single	Double
Sun386i	1.3	1.0
Sun-3, 68882	1.4	1.4
80960KB, 16MHz	3.4	3.3
Sun-3, FPA	4.1	2.9
80960KB, 20MHz	4.2	4.0
Sun-4	5.8	3.9

Function	Interval	
Single Precision		
SIN(X)	[+0.000e+00, +1.570e+00)	−5.
COS(X)	[+0.000e+00, +1.570e+00)	−5.
EXP(X)	[−1.037e+00, +1.008e+00)	−5.
EXPM1(X)	[−1.037e+00, +1.008e+00)	−5.
LOG1P(X)	[−2.928e−01, +4.142e−01)	−5.
ATAN(X)	[−6.553e+04, +6.553e+04)	−5.
LOG(X)	[+7.071e−01, +1.414e+00)	−5.
Double Precision		
SIN(X)	[+0.000e+00, +1.570e+00)	−5.
COS(X)	[+0.000e+00, +1.570e+00)	−1.
EXP(X)	[−1.037e+00, +1.008e+00)	−5.
EXPM1(X)	[−1.037e+00, +1.008e+00)	−5.
LOG1P(X)	[−2.928e−01, +4.142e−01)	−5.
ATAN(X)	[−6.553e+04, +6.553e+04)	−5.
LOG(X)	[+7.071e−01, +1.414e+00)	−5.
Extended Precision		
SIN(X)	[+0.000e+00, +1.570e+00)	−1.
COS(X)	[+0.000e+00, +1.570e+00)	−4.
EXP(X)	[−1.037e+00, +1.008e+00)	−1.
EXPM1(X)	[−1.037e+00, +1.008e+00)	−1.
LOG1P(X)	[−2.928e−01, +4.142e−01)	−1.
ATAN(X)	[−6.553e+04, +6.553e+04)	−1.
LOG(X)	[+7.071e−01, +1.414e+00)	−1.

Comparison of Polynomial Tangent Approximation Execution Speed

Library	Time Units	Operations
Double	49-52	13–16 adds, 16 mul, 1 div
Single	19	3 adds, 6 mul, 1 div
80960	14	2 adds, 1 mul, 1 div, 1 co

Why Cordic?

- 8087 compatibility
- fast, accurate polynomial approx unknown
- chip area limitations
- prior experience with Cordic
- provable monotonicity

Some 80960-Related Exercises

1. Prove or disprove that signaling the inexact exception for all non-zero finite IEEE format floating-point tangent operands is mathematically correct.

2. Under what conditions can the computed tangent signal overflow or underflow on the 80960?

3. An early version of the 387/960 hardware accumulated a sticky bit in the least significant mantissa bit on right shifts, rather than simply dropping the bits shifted off. Explain why the hardware was changed, and how this affected the representation of the arctan(2^-i) values.

4. Explain the larger error reported by Alex Liu's tests for the extended precision EXPM1 function.

Valerio
13 Jun 88

"Floating-point numbers are like sand piles: every time you move one you lose a little sand and pick up a little dirt."

The Elements of Programming Style

"That's not funny!"

Jim Valerio

Valerio
14 June 86

80960 Floating-Point Architecture Observations

- Overview of 80960 Floating-Point Architecture
- A Case for 80960-Style Extended Precision
- Improving the IEEE Standard

A Case for 80960-Style Extended Precision

- Computing `Q = X*Y/Z`
- Extended is not Quad
- Extended Precision Reduces Cycle Count
- Mixing Extended, Other Formats
- Problematic Extended Implementations

```
#define REAL    float
#define LEN     1

asm REAL logb(REAL x)
{
%reglit(LEN) return; reglit(LEN) x;
        logbnr  x,return
}

asm REAL scalb(REAL x, int e)
{
%reglit(LEN) return; reglit(LEN) x; reglit e
        scaler  e,x,return
}

/*
 * Compute so as to avoid all spurious overf
 *      q = x*y/z;
 */
REAL
```

```
mblatt(REAL x, REAL y, REAL z)
{
        int ex, ey, ez;
        REAL sx, sy, sz;
        REAL sq;

        ex = (int) logb(x);
        sx = scalb(x, -ex);
        ey = (int) logb(y);
        sy = scalb(y, -ey);
        ez = (int) logb(z);
        sz = scalb(z, -ez);

        sq = sx*sy;
        sq = sq/sz;
        return scalb(sq, ex+ey-ez);
}
```

```
        .align  4
        .globl  _mblatt
_mblatt:
        logbnr  g0,g13          # ex = (int)
        cvtzri  g13,r3
        subi    r3,0,r11        # sx = scalb
        scaler  r11,g0,r7
        logbnr  g1,g13          # ey = (int)
        cvtzri  g13,r15
        subi    r15,0,r11       # sy = scalb
        scaler  r11,g1,r6
        logbnr  g2,g13          # ez = (int)
        cvtzri  g13,r14
        subi    r14,0,r11       # sz = scalb
        scaler  r11,g2,r5
        mulr    r7,r6,r4        # sq = sx *
        divr    r5,r4,r4        # sq = sq /
        addi    r3,r15,r11      # return sca
        subi    r14,r11,r9
        scaler  r9,r4,g0
```

```
# q = x*y / z;
        mulr    g0,g1,fp0
        divr    g2,fp0,g0
```

Extended is not Quad

- doubling vs. extending
- simplify expression evaluation
- 10 percent penalty
- wider exponent range

Answers to Exercises

1. You've got me.
   I seem to recall there's a theorem that guarantees this, but I can't produce it.

2. Tangent can never signal overflow because the largest possible result is less than 1/(2^-66), which is representable in single precision. Tangent will signal underflow when the source operand 'u' is less than the smallest normalized number in the destination format. Note that underflow is guaranteed because the 80960 detects loss of accuracy by inexact result, and the tangent is guaranteed to be inexact.

3. The change was made for two reasons, both associated with the error analysis. First, the error analysis was easier when the direction of the rounding caused by right shifts was known. This precipitated rounding up each log2(1 + 2^-i) constant, as a way to counteract the rounding down performed by right shifts. The second reason for the shift change as to make monotonicity proofs feasible.

4. The operand is multiplied by 64-bit rounded value of log10(2) before issuing the exponential instruction; the operand of the exponential instruction has been rounded twice.