Micro Digital

GoFast® Floating-Point Library User's Guide

November 11, 2009

by Harry Ohlson

Now maintained and extended by
Micro Digital Associates Inc.
2900 Bristol Street, #G204
Costa Mesa, CA 92626
(714) 437-7333
support@smxrtos.com
www.smxrtos.com

1 What Is GoFast?

1.1 A Floating-Point Library

GoFast is a floating-point library for ANSI C. It is a **soft** library, for processors that do not offer floating-point support in hardware. It is **complete**: no other floating-point routines are needed. It complies with the appropriate **IEEE 754** and **ANSI C** standards. However, the exception handling has been simplified a little (see section 5.1), mostly to make the product simple to use in embedded systems. GoFast goes to great pains to provide good **precision**, and to give mathematically **correct** answers even when the standards are silent. GoFast is **portable**: it is easily adopted for new processors and new compilers. GoFast is **maintainable**: it is computer-generated. It will run in **flash** memory: data and code are separate, there's no modification of code or constants, and there's no run-time initialization. It is **re-entrant**, storing nothing into static variables and using very little stack space. It is **well-tested** (there's an automated test suite) and **stable** (the algorithms haven't changed in years).

There's even a simplified GoFast library for some old 8-bit processors: 8051, 8096, Z80. These are not covered in this manual because they differ in several ways from the "real" (computer-generated) GoFast. The old 8-bit compilers are not ANSI C, maybe not even close. (Some 8-bit GoFast users actually code in assembly language.) The available memory would not take a full library. In most cases, the routines are not naturally re-entrant.

1.2 A Fast One

Most importantly, GoFast is **fast**. Replacing the native floating-point library with GoFast might cut timings by 20% for simple functions such as add or multiply, by 75% in transcendentals such as the tangent. You could even see an occasional 90%, but there would be something wrong with the original routine then. (For instance, some C libraries have no single-precision functions.)

How can we make claims like these? Surely the people who supply the compiler do their best to produce a good floating-point library? Yes they do – their practical best. The floating-point routines are typically written in C and operate on floating-point variables. These algorithms are relatively simple, easily found on the Web or in books, and efficient in a floating-point unit. They get heavy when all floating-point is simulated. Not all compilers do it this way of course: the old Borland library, hand-written in assembly code, is about as fast as GoFast. But today's compilers, such as the GNU C, typically support so many different CPU varieties that hand-written assembly code is out of the question.

GoFast performs all calculations using **integers**. The first thing done is the separation of the exponent and the mantissa; the last is their recombination. This method might do an in-line polynomial evaluation (double-precision) in 25 machine instructions per term – no subroutine calls, no simulation, 25 cycles or so in a RISC. Because the mantissa has 64 bits, good precision comes as a bonus. The algorithms can get intricate – and you don't find them on the Web – but they have been thoroughly tested over the years.

GoFast also takes advantage of some **machine instructions** that normally aren't available in C. It will use full divide (64/32 ➔ 32) and full multiply (32*32 ➔ 64) if these are available. It will happily employ such strange (and useful) functions as the PowerPC "rotate and mask". It understands all kinds of status flag strategies, and will optimize out unneeded compares.

1.3 How About Emulation?

The GoFast library will work if the compiler generates library calls for floating-point operations. For a simple addition "**f1 += f2;**" the produced (hypothetical) assembly code might be something like

```
move d0,r4
move d1,r5
call dpadd
move r4,d0
```

Some compilers will however assume floating-point support in hardware, and produce something (again hypothetical) like this:

```
fload     f0,r4
fload     f1,r5
fadd      f0,f1
fstore    r4,f0
```

If there's no floating-point unit present, these instructions will cause a CPU exception, which then has to be handled by a floating-point emulator. The emulator is typically installed as part of the operating system. Different processors will most likely require totally different emulators. An emulator library is also needed, for those ANSI C functions that are not in the floating-point unit.

Emulation makes sense in personal computers and workstations. As a general rule, it is misplaced in an embedded system. Three of our GoFast library targets have a floating-point emulator (several in some cases): the Intel x86, the PowerPC, and the SPARC. Emulation in the last two is optional and up to the user; do it when there's a clear reason. The Intel x86 compilers will almost always force you into emulation. There is actually a full GoFast library for the Microsoft and the Borland compilers, but to use it, you have to do all floating-point operations with subroutine calls.

The x86 emulation is covered in a different manual because the subject is quite complex and has little in common with the ANSI C math library, the subject of this document. Nevertheless, the emulators do share the name GoFast.

1.4 Definitions

Floating point is a method of representing numeric values (integers and non-integers) in a computer. It uses three fields for this:

- The **sign** tells whether the number is positive or negative.
- The **exponent** tells where the decimal point goes.
- The **mantissa** (also called the significand) gives the digits.

To get the actual value of the number, you raise 2 to the power of the exponent and multiply this with the mantissa. (For details such as bias and scaling, see the IEEE 754 document.)

In the IEEE 754 standard, **single-precision** numbers take up 32 bits, **double-precision** numbers twice that. The useful **range** for singles is approximately 10^{-38} to 10^{38}, for doubles 10^{-308} to 10^{308}. The relative **precision** (typical rounding error in one arithmetic operation) is of the order of 10^{-7} for singles, 10^{-16} for doubles.

1.5 Notations

In the following text, these symbols denote different kinds of variables:

d1, **d2**, **d3**	double-precision number
f1, **f2**, **f3**	single-precision number
si	standard integer (16 or 32 bits usually)
li	long integer (32 bits)
ul	unsigned long integer (32 bits)
ll	64-bit integer
ull	unsigned 64-bit integer
NaN	not-a-number, an invalid floating-point value
INF	infinity, an overflowed floating-point value

2 Included Functions

2.1 Intrinsic Functions

These are functions called by the C compiler to handle simple floating-point operations such as add or compare. The names of the intrinsics depend on the compiler in question, and there's even some (small) variation in what exact functions are needed. All these functions also have a private GoFast name, for testing and documentation purposes. The following tables give the GoFast name.

function type	operation	generated call	notes
double arithmetic	`d3 = d1 + d2`	`d3 = dpadd(d1,d2)`	
	`d3 = d1 + 1`	`d3 = dpinc(d1)`	`rare`
	`d3 = d1 – d2`	`d3 = dpsub(d1,d2)`	
	`d3 = d1 – 1`	`d3 = dpdec(d1)`	`rare`
	`d3 = -d1`	`d3 = negdf2(d1)`	
	`d3 = d1 * d2`	`d3 = dpmul(d1,d2)`	
	`d3 = d1 / d2`	`d3 = dpdiv(d1,d2)`	
single arithmetic	`f3 = f1 + f2`	`f3 = fpadd(f1,f2)`	
	`f3 = f1 + 1`	`f3 = fpinc(f1)`	`rare`
	`f3 = f1 – f2`	`f3 = fpsub(f1,f2)`	
	`f3 = f1 – 1`	`f3 = fpdec(f1)`	`rare`
	`f3 = -f1`	`f3 = negsf2(f1)`	
	`f3 = f1 * f2`	`f3 = fpmul(f1,f2)`	
	`f3 = f1 / f2`	`f3 = fpdiv(f1,f2)`	
conversion	`d1 = f1`	`d1 = fptodp(f1);`	`float to double`
	`f1 = d1`	`f1 = dptofp(d1);`	`double to float`
	`d1 = li`	`d1 = litodp(li);`	`long to double`
	`li = d1`	`li = dptoli(d1);`	`double to long`
	`d1 = ul`	`d1 = ultodp(ul);`	`uns. long to double`
	`ul = d1`	`ul = dptoul(d1);`	`double to uns. long`
	`f1 = li`	`f1 = litofp(li);`	`long to float`
	`li = f1`	`li = fptoli(f1);`	`float to long`
	`f1 = ul`	`f1 = ultofp(ul);`	`uns. long to float`
	`ul = f1`	`ul = fptoul(f1);`	`float to uns. long`

Some old compilers do the unsigned integer conversions either improperly or not at all. There's quite a bit of variation in how **NaN** and overflow show up in an integer answer. GoFast will return 0x7FFFFFFF (assuming 32 bits) for positive overflow, 0x80000000 for negative overflow and for **NaN**.

function type	operation	generated call	return value
64-bit conversion	`ull = f1`	`ull = fptoull(f1)`	
	`ll = f1`	`ll = fptoll(f1)`	
	`ull = d1`	`ull = dptoull(d1)`	
	`ll = d1`	`ll = dptoll(d1)`	
	`fp = ll`	`fp = lltofp(ll)`	
	`dp = ll`	`dp = lltodp(ll)`	
	`fp = ull`	`fp = ulltofp(ull)`	
	`dp = ull`	`dp = ulltodp(ull)`	
comparison	`d1 :: d2`	`si = dpcmp(d1,d2)`	`-2 -1 0 1`
	`d1 == d2`	`si = _d_feq(d1,d2)`	`true or false`
	`d1 != d2`	`si = _d_fne(d1,d2)`	`true or false`
	`d1 > d2`	`si = _d_fgt(d1,d2)`	`true or false`
	`d1 >= d2`	`si = _d_fge(d1,d2)`	`true or false`
	`d1 <= d2`	`si = _d_fle(d1,d2)`	`true or false`
	`d1 < d2`	`si = _d_flt(d1,d2)`	`true or false`
	`f1 :: f2`	`si = fpcmp(f1,f2)`	`-2 -1 0 1`
	`f1 == f2`	`si = _f_feq(f1,f2)`	`true or false`
	`f1 != f2`	`si = _f_fne(f1,f2)`	`true or false`
	`f1 > f2`	`si = _f_fgt(f1,f2)`	`true or false`
	`f1 >= f2`	`si = _f_fge(f1,f2)`	`true or false`
	`f1 <= f2`	`si = _f_fle(f1,f2)`	`true or false`
	`f1 < f2`	`si = _f_flt(f1,f2)`	`true or false`

The **dpcmp** and **fpcmp** returns are:

- -2 no meaningful comparison (few compilers care)
- -1 argument 1 < argument 2
- 0 arguments equal
- +1 argument 1 > argument 2

Usually a C compiler generates either the **dpcmp/fpcmp** calls (with its own names and return values), or it uses the relational routines, again with different names, maybe even swapping true and false. However, some GNU C compilers adopt a compromise: the relational routines are just synonyms for **dpcmp/fpcmp;** they all return -1, 0 or +1, the same value for the same arguments. Some GNU's use real relational calls and some use fake ones, and this can really confuse the unwary.

2.2 User Functions

This group comprises the ANSI C math functions. The names of the user-level functions are of course fixed (**sin**, **atan2** and so on), but the compiler often adds something to this, perhaps an underscore or two.

function type	user call	function performed
double, simple	d2 = fabs(d1)	d2 = absolute value of d1
	d2 = ceil(d1)	d2 = smallest integer not smaller than d1
	d2 = floor(d1)	d2 = largest integer not larger than d1
	d3 = fmod(d1,d2)	d3 = remainder of d1/d2
	d3 = modf(d1,&d2)	d3 = fraction of d1, d2 = integer of d1
	d2 = frexp(d1,&si)	d2 = mantissa of d1, li = exponent of d1
	d2 = ldexp(d1,si)	d2 = d1 * (2 ^ si)
	d2 = sqrt(d1)	d2 = square-root of d1
single, simple	f2 = fabsf(f1)	f2 = absolute value of f1
	f2 = ceilf(f1)	f2 = smallest integer not smaller than f1
	f2 = floorf(f1)	f2 = largest integer not larger than f1
	f3 = fmodf(f1,f2)	f3 = remainder of f1/f2
	f3 = modff(f1,&f2)	f3 = fraction of f1, f2 = integer of f1
	f2 = frexpf(f1,&si)	f2 = mantissa of f1, si = exponent of f1
	f2 = ldexpf(f1,si)	f2 = f1 * (2 ^ si)
	f2 = sqrtf(f1)	f2 = square-root of f1
double, transc.	d2 = asin(d1)	d2 = arcsine of d1
	d2 = acos(d1)	d2 = arccosine of d1
	d2 = atan(d1)	d2 = arctangent of d1
	d3 = atan2(d1,d2)	d3 = atan(d1/d2) range -**π** to **π**
	d2 = cos(d1)	d2 = cosine of d1
	d2 = cosh(d1)	d2 = hyperbolic cosine of d1
	d2 = exp(d1)	d2 = **e** to the power d1
	d2 = log(d1)	d2 = natural logarithm of d1
	d2 = log10(d1)	d2 = base 10 logarithm of d1
	d3 = pow(d1,d2)	d3 = d1 to power d2
	d2 = sin(d1)	d2 = sine of d1
	d2 = sinh(d1)	d2 = hyperbolic sine of d1
	d2 = tan(d1)	d2 = tangent of d1
	d2 = tanh(d1)	d2 = hyperbolic tangent of d1
single, transc.	f2 = asinf(f1)	f2 = arcsine of f1
	f2 = acosf(f1)	f2 = arccosine of f1
	f2 = atanf(f1)	f2 = arctangent of f1
	f3 = atan2f(f1,f2)	f3 = atanf(f1/f2) range –**π** to **π**
	f2 = cosf(f1)	f2 = cosine of f1
	f2 = coshf(f1)	f2 = hyperbolic cosine of f1

	f2 = expf(f1)	f2 = **e** to the power f1
	f2 = logf(f1)	f2 = natural logarithm of f1
	f2 = log10f(f1)	f2 = base 10 logarithm of f1
	f3 = powf(f1,f2)	f3 = f1 to power f2
	f2 = sinf(f1)	f2 = sine of f1
	f2 = sinhf(f1)	f2 = hyperbolic sine of f1
	f2 = tanf(f1)	f2 = tangent of f1
	f2 = tanhf(f1)	f2 = hyperbolic tangent of f1

3 Module Structure

In some special cases (linking problems, perhaps) it might be useful to know how the library is packaged. The table below shows where the functions reside. "NN" indicates the processor word size (i.e. 16, 32, or 64-bit). The usual extensions are "s" for source and "o" for object.

module	**contents**
arcNN	atan atan2 asin acos
ceilNN	ceil
dpNN	dptoli dptoul litodp fptodp dptofp dpsub dpadd dpmul dpdiv dpcmp dpinc dpdec
dpcmp	eqdf2 nedf2 gtdf2 gedf2 ledf2 ltdf2
expNN	exp
floorNN	floor
fpNN	fptoli fptoul litofp ultofp fpsub fpadd fpmul fpdiv fpcmp
fpcmp	eqsf2 nesf2 gtsf2 gesf2 lesf2 ltsf2
fparcNN	atanf atan2f asinf acosf
fpceilNN	ceilf
fpexpNN	expf
fpflooNN	floorf
fphypNN	sinhf coshf tanhf
fpllNN	fptoull fptoll lltofp ulltofp
fplogNN	logf log10f
fpmodNN	fmodf frexpf ldexpf modff
fppowNN	powf
fpsqrtNN	sqrtf
fptrigNN	sinf cosf tanf
funcNN	internal functions
hypNN	sinh cosh tanh
llNN	dptoull dptoll lltodp ulltodp
logNN	log log10
modNN	fmod frexp ldexp modf
powNN	pow
sqrtNN	sqrt
trigNN	sin cos tan

4 Testing

4.1 GFTEST

GFTEST serves as a desktop calculator and as a test script validator. Both these functions require support for keyboard input and display output. The validation also needs the capability to read from a file, either directly or through some kind of redirection. Unfortunately not all embedded test boards offer the required software support. (That's perhaps the chicken-and-the-egg dilemma: no software if no sales – but who would use a CPU with poor support.) Still, most will handle character input and output, so you can almost certainly run the desktop calculator, and check the GoFast functions by hand.

If you compile GFTEST with the option –DOS (define variable OS), it will use ANSI C functions for input and output. If this doesn't work, leave out the option, and link in your own versions of routines **putchr and getchr**. (You can probably forget about the file input in this case.)

GFTEST has a 4-element stack. Any number you enter gets pushed on the stack. Initially you work in double-precision mode; you can flip the precision with the command M. (The following examples assume double precision.) The number can be given in different forms:

n.d	number with optional sign and decimals examples: 1 23.776 -12
n.dEm	exponential representation examples: 1e-100 -5.6e8 123e-7
hxxxxxxxxx	hexadecimal representation missing trailing digits become 0 examples: h3ff8 = 1.5 h7ff0 = +INF hfff8 = NAN

You can also enter operators and function names. A two-argument function will perform the operation:

pop stack into op2
pop stack into op1
push FUNC(op1, op2)

This uses the arguments in the order in which you entered them, removing them from the stack. For instance the input "1 2 /" will place the value 1/2 on the stack. (As this example shows, you can enter several parameters on one command line.) A one-argument function will just replace top-of-stack with the result, for instance "6.25 sqrt" will place 6.25 on the stack and then replace it with 2.5.

Below is a list of the GFTEST functions. In this, x means top of stack; y means the second from the top. The commands are not case-sensitive.

Q	quit
+	x = x + y
-	x = x - y
*	x = x * y
/	x = x / y
_	x = y - x
\\	x = y / x
?	x = -1 if x < y, 0 if x == y, 1 if x > y
=	compare x and y, quit if not equal
INT	convert x to integer and back to real
UINT	convert x to unsigned integer and back to real
func	any C function: x = func(x) or func(x,y)
M	change mode between single and double precision
R	roll stack
X	exchange x and y
D	convert single x to double
S	convert double x to single
Pn	put x into register n (0-31)
Gn	get x from register n (0-31)
B f	start reading input file f

In validation mode, GFTEST reads a test script, performs the defined calculations, and compares the result to the given value. If there's a mistake, it displays a message and stops. You can start the scripted test from the command line; just give the name of the script file as a parameter. If there's no command line, you can get the script going with the "B f" command. The script files are:

file name	**tests**
DPCNVT.TST	double-precision conversions
DPFNCS.TST	double-precision functions
DPOPNS.TST	double-precision operations
FPCNVT.TST	single-precision conversions
FPFNCS.TST	single-precision functions
FPOPNS.TST	single-precision operations
LLCNVT.TST	64-bit conversions

If you can't get file input to work, should you run all the scripts through GFTEST by hand? No, but check every function at least once, paying particular attention to compare and to divide. How to get test values: print the scripts and borrow from them.

4.2 BENCH

BENCH measures the speed of some floating-point operations, producing a table like this:

Function	Double	Single
add	3.6	2.3
subtract	6.8	2.8
multiply	9.0	3.0
divide	18.2	6.6
sqrt	24.0	10.1
exp	43.4	10.2
log	80.2	13.4
log10	73.7	14.2
sin	39.8	9.2
cos	34.2	13.6
tan	63.6	13.8
asin	94.6	32.4
acos	122.5	33.5
atan	44.5	13.6
atan2	67.8	18.9
pow	118.1	25.7

The numbers are microseconds. (This example is for an old R3000 board.) BENCH calls routine **clock** to get the elapsed time, and **putchar** to display the results. The clock frequency is set as CLOCKS_PER_SEC, which normally comes from the system header file **time.h**. (In an embedded environment, you may have to use different methods for the timing.) BENCH calculates the overhead separately for the timing loops, so the results should be fairly accurate.

5 Technical Considerations

5.1 Exception Handling

GoFast makes no distinction between quiet and signaling not-a-numbers (NaNs). In an invalid operation, the answer is always a standard quiet NaN, 0xFFF8000000000000 in double precision and 0xFFC00000 in single precision.

The GoFast routines support the IEEE 754 masked exception handling for overflows and invalid operations. An overflow is returned as the special value infinity, and an invalid operation is returned as the special value NaN.

In ANSI C, the error code is stored into the variable **errno**. In the interests of simplifying reentrancy, GoFast does not do this. No unmasked exceptions are supported; there are no exception interrupts. Underflow and loss of precision are not reported. Division by zero is treated as an invalid operation.

5.2 Precision

The basic operations (add, subtract, multiply, divide, square root) and the conversions all use the IEEE 754 "round to nearest or even" rounding exactly. No other rounding modes are supported. These operations are IEEE exact.

The transcendental functions (which are not defined in IEEE 754) are correct to within two mantissa units. However, the trigonometric functions SIN, COS and TAN can lose precision in the argument reduction. For π, GoFast uses 64 bits in single precision and 66 bits in double precision (chosen so that an ordinary PC can be used to verify the results), which is enough for any argument up to about 1000π. Above that, exact multiples of $\pi/4$ start losing precision, until eventually none remains.

Software or hardware that uses fewer than 66 bits of π will give less accurate answers. In most applications this makes no difference, because the arguments stay below 2π.

5.3 Special Values

An overflow returns +INF or -INF, an underflow returns +0 or -0. If an argument is not-a-number (NaN), the result is NaN. The table below gives the GoFast result for some other special situations. It does not include cases that should not cause any confusion.

-	INF-INF = NaN
*	0*INF = NaN
/	0/0 = NaN
	INF/INF = NaN

sqrt	sqrt(-0) = -0
	sqrt(x<0) = NaN
fmod	fmod(INF,y) = NaN
	fmod(x,0) = NaN
	fmod(x,INF) = x
frexp	frexp(INF,x) = NaN
modf	modf(INF,x) = NaN
log/log10	-INF if x=0
	NaN if x<0
sin/cos/tan	NaN if |x| >= 262**π**
acos/asin	NaN if |x| > 1
atan2	atan2(0,0) = NaN
tanh	tanh(+INF) = 1
	tanh(-INF) = -1
pow	pow(0,0) = NaN
	pow(x<0, y not integer) = NaN
	pow(0,INF) = pow(INF,0) = NaN

Most likely, these pathological cases will be of no interest to anyone. It is not at all unusual to find a C library that returns questionable values for one or more. You may also meet someone who proceeds to prove that pow(0,0) is zero, or one, or anything. (The proofs will all be correct – that’s why it’s called NaN.)

5.4 Accuracy in Calculations

Floating-point calculations are in practice always inexact. This is easy to forget because just about everything else in programming is exact, and because the precision seldom becomes a problem. But you forget at your own peril.

There is nothing mysterious about the loss of precision; it's simply the nature of the thing. The following illustrates different faces of the inaccuracy.

5.4.1 Rounding

A floating-point number contains a fixed number of digits. Unless there are a lot of trailing zeroes, an arithmetic operation will very likely produce too many digits to fit in the same space. This of course happens even in normal decimal calculations, for instance:

```
  1234.567
+   12.34567
  1246.91267  ➔ 1246.913
```

Rounding errors as such are unlikely to become noticeable, but they can be enhanced by other effects. Some algorithms are notoriously prone to lose precision.

5.4.2 Base Conversion

Changing the base of a fractional number generally requires approximations. Any application that uses decimal input, decimal constants or decimal output has to perform base conversions. Consider the example

```
float f1;
f1 = 1.1;
printf("%.12f\n", f1);
```

This program will display the value 1.100000023842, not the exact 1.1. What happened?

The root of the problem is that 1 1/10 in base 2 is 1.0001100(1100), i.e. can't be represented exactly. The compiler creates a constant 1.1 with 24 bits:

1.000 1100 1100 1100 1100 1101

This value is obviously larger than 1.1 because we rounded up at bit 24. Printing the value with too many decimals (anything more than 7 in this case) will show the difference.

5.4.3 Difference between Large Numbers

Let's try the program

```
float f1, f2, f3;
f1 = 1234.0;
f2 = 1233.1;
f3 = f1 - f2;
printf("%lf\n", f3);
```

The result is 0.900024: off by quite a bit. The basic effect is the same as explained above: the required base conversion. But the relative error got enlarged in the subtraction of two almost equal numbers:

	1234.0 =	1001 1010 0100 0000 0000 0000
-	1233.1 =	1001 1010 0010 0011 0011 0011
	0.9 =	1100 1100 1101

5.4.4 Irrational Numbers

Values such as sqrt(2) or sin(0.5) have no exact representation in any base. These can still be calculated "exactly" to the value that is mathematically correct considering the rounding rules. IEEE specifically requires an exact square-root, but says nothing about other functions. The GoFast square-root is of course exact.

You probably won't find an "exact" implementation of the transcendentals anywhere. The additional error should be of the same order as the rounding error.

5.4.5 Special Functions

As a rule, the relative error of a function is different than the relative error of the argument. In some cases this becomes important. Take the following code:

```
double d1, d2;
d1 = 1.1;
d3 = exp(100*d1);
```

The result will differ from exp(110) by quite a bit. This does not mean that exp(x) is inaccurate; it means that the original inaccuracy of x got magnified. An important special case is

$$z = x^y = e^{y*\log(x)}$$

This formula – definitely not from GoFast – is a bad way to calculate x to the power y.

A point where a function approaches zero for a non-zero argument is especially tricky. As an example, log(0.999998) is close to twice log(0.999999). If your argument is only a little inexact, say due to rounding, the answer may be so wrong as to be meaningless. Again we need to remember that log(x) as such is not the culprit, it is not inaccurate.

The same warning applies whenever significant argument reduction is needed, such as the trigonometric functions for arguments much larger than **π**. Worst of all are cases where these two situations coincide: sin(1000**π**) for instance.

5.4.6 Conversion to Integer

ANSI C specifies that a floating-point number is converted to an integer using truncation: the decimals are discarded. This innocuous rule can cause surprises. Consider the program

```
int i1, i2;
i1 = 256;
i2 = (float)i1 / 2.56;
printf("%d\n", i2);
```

Certainly the correct answer is 100, but you can't count on this; the program as written is unstable. In some cases, the answer will keep jumping between 99 and 100, depending on the compilation options and the exact code used.

The root reason for the instability is not hard to see. The value 2.56 has to be rounded when it is converted to base 2. If this rounding is up, the division will give a value that is slightly less than 100. According to ANSI C rules, this becomes 99. If again 2.56 in base 2 is rounded down, the division will give slightly over 100, and truncates to 100.

IEEE 754 is a very rigorous standard; whether 2.56 is rounded up or down, surely it should be rounded the same way every time. How is it possible that two standard implementations give completely different results? Well, it really isn't. This is an interesting example of what happens when a standard meets an optimizing compiler. How the rounding is done depends on the number of bits in the constant. ANSI C says that a floating-point constant is **double**, and IEEE 754 rules this to have 53 binary digits. Unfortunately

1 Some compilers use **float** constants in float expressions. This difference may be enough to change the direction of the rounding.

2 Some compilers optimize out all divisions by a constant, using instead a multiplication with the inverse value. What happens to the rounding is anybody's guess.

5.4.7 Financial Calculations

You want to make absolutely sure your broker isn't cheating you, so you write a little program to check the commission. The first trade looks fine. The second trade looks fine. The third trade – caught him! Overcharged by a penny!

Well, not really. Financial rounding follows law and custom, knowing (and caring) nothing about IEEE 754 rounding. In some special cases, you have to round up. Even the usual "bank rounding" isn't quite the same as the IEEE default – though you'll have to look hard to catch the difference.

None of this means that there's a problem. Financial institutions just don't use floating-point math.

6 16-Bit Processors

6.1 Motorola 68HC16

6.1.1 Compilers

There are three GoFast versions for the Motorola 68HC16, for three different C compilers:

- Archimedes/Hi-Cross
- Introl
- Whitesmith

The instruction set is of course the same, but the assembly controls differ, and so do the calling conventions. The library links right in, and works with all memory models. (HC16 libraries are not always model-independent, but GoFast qualifies because no function uses more than one pointer as argument, so no pointers are fetched from the stack.)

6.1.2 Test Environment

The libraries have been tested on the Motorola 68HC16 evaluation board. The product includes instructions on how to set up and use the board. There are also various files that might be useful in testing the GoFast library.

6.1.3 Timings

The following table shows the timings (in microseconds) for a few functions on the 16.78 MHz evaluation board.

Function	GoFast	Whitesmith C	Introl C
divide	226	688	844
sqrt	171	3284	4386
exp	1348	2343	4354
log	1177	3015	5437
sin	963	1437	3708

7 32-Bit Processors

7.1 Altera Nios II

7.1.1 Compiler

This GoFast library is for the GNU C compiler. There are two versions: one that uses multiply and divide instructions (NIOSII), and one that uses neither (NIOSIIX).

7.1.2 Timings

The following table shows some times (in microseconds) for the Altera Nios II 1C12 "standard" evaluation board, using no multiply or divide in hardware.

	Double-Precision		Single-Precision	
Function	**GoFast**	**GNU**	**GoFast**	**GNU**
add	0.7	2.1	0.3	1.1
subtract	0.7	2.1	0.3	1.1
multiply	3.2	10.9	0.9	3.0
divide	3.4	7.6	1.0	1.5
sqrt	6.3	7.4	3.2	1.7
exp	21.6	117.1	5.6	37.6
log	18.2	182.4	4.0	54.1
log10	20.4	199.8	4.8	60.8
sin	18.0	105.0	4.0	32.1
cos	16.0	120.9	4.0	38.4
tan	22.0	220.6	5.6	67.1
asin	32.4	213.1	8.8	64.1
acos	32.2	201.5	9.6	59.6
atan	18.8	209.5	4.9	61.4
atan2	21.9	223.4	5.8	64.4
pow	42.5	542.0	10.6	163.6

7.2 ARM

GoFast for ARM is offered for the ARM and Thumb-2 instruction sets, not Thumb. Each is sold separately.

7.2.1 Compiler

GoFast for ARM supports the following C compilers:

- IAR EWARM
- Keil ARM
- Rowley CrossWorks ARM

7.2.2 Timings for IAR EWARM (ARM and Thumb-2)

The following table shows all times (in microseconds) for the indicated processor and evaluation board. The basic operations (add, subtract, multiply, divide, conversions, and comparisons) in the IAR library are hand-coded in assembly and faster than those in GoFast, so the IAR versions are used instead. (If you only need these basic operations, you don't need GoFast.) Thus, the routines linked are a mixture of both libraries, as indicated in **bold** below.

ARM7: LPC2468, 48 MHz, Code Int SRAM, Data Ext SDRAM

	Double-Precision		Single-Precision	
Function	**GoFast**	**IAR**	**GoFast**	**IAR**
add	1.8	**1.2**	1.3	**0.8**
subtract	1.9	**1.3**	1.3	**0.8**
multiply	1.8	**1.4**	1.2	**0.7**
divide	9.2	**6.6**	4.8	**1.6**
sqrt	**17.8**	29.1	9.4	**7.7**
exp	**8.7**	29.9	**2.7**	17.8
log	**19.2**	29.1	**8.0**	9.3
log10	**19.6**	33.0	**8.2**	11.1
sin	**7.2**	21.0	**2.7**	7.9
cos	**7.1**	20.8	**2.7**	7.8
tan	**16.7**	27.6	**6.7**	9.3
asin	**15.8**	66.9	20.0	**18.6**
acos	**16.2**	67.0	22.5	**18.7**
atan	**20.5**	32.5	**8.6**	9.5
atan2	**29.2**	38.0	12.5	**11.0**
pow	**27.6**	83.2	**11.6**	39.7
tanh	**17.3**	35.3	**9.9**	19.0
sinh	**17.0**	37.4	**7.0**	21.2
cosh	**16.9**	36.3	**6.5**	20.6
modf	**2.5**	3.4	**1.5**	2.1
fmod	**6.3**	75.3	**4.8**	48.1
fabs	**0.4**	1.0	**0.3**	0.9
floor	**0.9**	2.4	**0.6**	1.8
ceil	**0.9**	2.4	**0.6**	1.8
ldexp	**0.9**	2.2	**0.8**	1.8
frexp	**0.8**	1.0	**0.7**	0.9
cmp	1.1	**0.8**	0.8	**0.7**
fp to long	0.7	**0.5**	**0.5**	0.5
fp to ulong	0.7	**0.4**	0.5	**0.4**
long to fp	**0.9**	1.1	0.8	**0.5**
ulong to fp	**0.8**	1.2	0.6	**0.5**
sgl to dbl	0.7	**0.5**	—	—
dbl to sgl	0.8	**0.5**	—	—

Times were measured on Embedded Artists LPC2468 OEM board with IAR v5.20.

Cortex-M3: LM3S8962, 50 MHz, Int SRAM

	Double-Precision		Single-Precision	
Function	**GoFast**	**IAR**	**GoFast**	**IAR**
add	2.6	**1.8**	1.8	**1.2**
subtract	2.7	**1.9**	1.9	**1.2**
multiply	2.6	**2.1**	1.6	**1.0**
divide	**7.3**	12.4	3.9	**1.6**
sqrt	**13.7**	53.4	**7.6**	11.3
exp	**12.8**	49.5	**4.2**	32.7
log	**19.9**	50.1	**9.1**	16.6
log10	**20.9**	56.8	**9.3**	20.0
sin	**10.5**	35.1	**4.1**	15.0
cos	**10.3**	34.7	**4.1**	14.8
tan	**17.7**	47.8	**6.9**	16.5
asin	**17.9**	123.6	**15.9**	29.6
acos	**18.2**	123.8	**17.8**	29.9
atan	**19.3**	59.3	**8.0**	15.2
atan2	**25.5**	69.5	**10.8**	17.2
pow	**32.5**	136.3	**13.6**	67.1
tanh	**18.3**	61.0	**9.0**	34.3
sinh	**17.8**	63.8	**7.1**	37.9
cosh	**17.5**	62.6	**6.9**	36.8
modf	3.8	**3.7**	**2.4**	2.4
fmod	**10.4**	104.6	**7.5**	71.6
fabs	**0.4**	1.3	**0.3**	0.9
floor	**1.1**	3.3	**0.7**	2.3
ceil	**1.2**	3.2	**0.8**	2.3
ldexp	**1.1**	2.2	**0.9**	1.9
frexp	**1.0**	1.1	**0.8**	0.9
cmp	1.4	**0.9**	**0.8**	0.8
fp to long	**0.8**	0.8	**0.5**	0.5
fp to ulong	0.9	**0.5**	0.5	**0.3**
long to fp	1.2	**0.6**	1.1	**0.6**
ulong to fp	1.0	**0.4**	0.8	**0.4**
sgl to dbl	0.8	**0.5**	–	–
dbl to sgl	1.1	**0.7**	–	–

Times were measured on Texas Instruments (Luminary Micro) LM3S8962-EK board with IAR v5.20.

7.2.3 Timings for Keil ARM

The following table shows all times (in microseconds) for the indicated processor and evaluation board. The basic operations (add, subtract, multiply, divide, conversions, and comparisons) in the Keil library are hand-coded in assembly and faster than those in GoFast, so the Keil versions are used instead. (If you only need these basic operations, you don't need GoFast.) Thus, the routines linked are a mixture of both libraries, as indicated in **bold** below.

ARM7: LPC2468, 48 MHz, Code Int SRAM, Data Ext SDRAM

	Double-Precision		Single-Precision	
Function	**GoFast**	**Keil**	**GoFast**	**Keil**
add	1.7	**1.0**	1.2	**0.6**
subtract	1.8	**1.0**	1.3	**0.7**
multiply	1.8	**1.5**	1.1	**0.7**
divide	9.2	**3.6**	4.8	**1.1**
sqrt	17.8	**7.7**	9.4	**3.4**
exp	**8.7**	26.7	**2.6**	21.0
log	**19.3**	28.8	**7.9**	20.4
log10	**19.5**	32.8	**8.1**	20.7
sin	**7.2**	22.8	**2.7**	14.4
cos	**7.1**	22.9	**2.7**	14.4
tan	**16.6**	45.8	**6.6**	15.1
asin	**15.8**	33.9	20.0	**18.8**
acos	**16.2**	31.4	22.5	**19.7**
atan	**20.5**	28.2	**8.6**	15.3
atan2	**28.5**	36.1	**12.4**	16.0
pow	**27.5**	105.3	**11.5**	99.6
tanh	**17.2**	42.1	**9.9**	20.7
sinh	**17.0**	44.2	**6.9**	19.5
cosh	**16.9**	30.6	**6.6**	19.4
modf	**2.3**	2.5	1.4	**0.9**
fmod	**6.2**	8.1	**4.7**	7.3
fabs	**0.3**	0.3	**0.3**	0.3
floor	**0.8**	1.7	**0.6**	1.1
ceil	**0.9**	1.7	**0.6**	1.1
ldexp	**0.9**	2.0	**0.7**	1.1
frexp	**0.8**	0.8	**0.6**	0.7
cmp	1.0	**0.7**	0.8	**0.7**
fp to long	0.7	**0.5**	0.5	**0.4**
fp to ulong	0.6	**0.5**	**0.4**	0.4
long to fp	0.9	**0.5**	0.7	**0.5**
ulong to fp	0.8	**0.5**	**0.5**	0.5
sgl to dbl	0.6	**0.4**	–	–
dbl to sgl	0.7	**0.5**	–	–

Times were measured on Embedded Artists LPC2468 OEM board with Keil v3.85.

7.2.4 Timings for Rowley CrossWorks ARM

The following table shows all times (in microseconds) for the indicated processor and evaluation board. The basic operations (add, subtract, multiply, divide, conversions, and comparisons) in the CrossWorks library are hand-coded in assembly and faster than those in GoFast, so the CrossWorks versions are used instead. (If you only need these basic operations, you don’t need GoFast.) Thus, the routines linked are a mixture of both libraries, as indicated in **bold** below.

ARM7: AT91SAM7X256-EK, 48 MHz, RAM

	Double-Precision		Single-Precision	
Function	**GoFast**	**CWK**	**GoFast**	**CWK**
add	2.5	**1.6**	1.6	**1.0**
subtract	3.0	**2.0**	1.9	**1.4**
multiply	2.7	**2.2**	1.6	**1.0**
divide	16.5	**9.2**	8.3	**2.3**
sqrt	**32.3**	46.6	16.7	**15.3**
exp	**13.1**	46.6	**4.3**	32.4
log	**27.2**	60.2	**11.3**	28.9
log10	**28.3**	62.4	**11.6**	30.0
sin	**11.6**	41.9	**4.0**	21.0
cos	**10.6**	53.1	**4.0**	25.2
tan	**28.2**	59.9	**11.3**	28.5
asin	**45.1**	109.2	**34.8**	48.2
acos	**43.8**	107.6	**42.2**	46.8
atan	**27.1**	64.4	**11.6**	30.2
atan2	**42.8**	75.5	**19.2**	34.6
pow	**40.5**	112.9	**16.7**	65.8
tanh	**30.7**	62.5	**19.2**	40.1
sinh	**30.4**	65.0	**12.1**	39.0
cosh	**29.8**	63.9	**11.6**	40.6
modf	**1.4**	1.9	**1.1**	1.3
fmod	**1.6**	16.5	**1.0**	8.0
fabs	**0.5**	0.5	**0.3**	0.3
floor	**1.1**	1.9	**0.7**	1.2
ceil	**1.2**	1.9	**0.7**	1.2
ldexp	**2.2**	2.3	**1.4**	1.6
frexp	**1.3**	1.8	**0.8**	1.3
cmp	1.7	**1.0**	1.2	**0.8**
fp to long	1.0	**0.6**	0.7	**0.5**
fp to ulong	1.0	**0.5**	0.7	**0.5**
long to fp	4.6	**0.8**	4.3	**0.7**
ulong to fp	4.5	**0.7**	4.2	**0.7**
sgl to dbl	0.9	**0.5**	–	–
dbl to sgl	1.1	**0.6**	–	–

7.3 ColdFire

7.3.1 Compiler

This GoFast library is for the Freescale CodeWarrior C/C++ compiler.

This version of GoFast uses the ColdFire DIV instruction, which first appeared on the 5206e. If you are using an older processor, we can supply an older library we created for Diab that we tested on 5204.

7.3.2 Timings for Freescale CodeWarrior

The following table shows all times (in microseconds) for the indicated processor and evaluation board. The basic operations (add, subtract, multiply, divide, conversions, and comparisons) in the CodeWarrior library are hand-coded in assembly and some are faster than those in GoFast, so the CodeWarrior versions are used instead. (If you only need these basic operations, you don't need GoFast.) Thus, the routines linked are a mixture of both libraries, as indicated in **bold** below.

M5275EVB, 150MHz, External SDRAM

	Double-Precision		Single-Precision	
Function	**GoFast**	**CW**	**GoFast**	**CW**
add	14.54	**12.50**	**8.81**	8.91
subtract	15.42	**12.35**	**9.27**	9.41
multiply	**18.16**	19.29	**9.29**	10.74
divide	28.3	**20.61**	13.75	**10.36**
sqrt	**49.30**	148.52	**26.95**	158.78
exp	**92.48**	357.79	**21.93**	376.37
log	**110.17**	383.32	**36.56**	406.54
log10	**116.96**	469.26	**38.21**	477.85
pow	**206.19**	1257.25	**61.17**	1321.70
sin	**68.98**	366.36	**22.38**	375.83
cos	**69.02**	374.61	**22.16**	383.00
tan	**116.54**	662.16	**29.97**	656.48
asin	**143.08**	494.11	**55.67**	501.04
acos	**160.84**	439.00	**61.81**	448.61
atan	**91.73**	423.69	**31.03**	423.63
atan2	**112.69**	486.46	**39.69**	506.11
sinh	**112.08**	565.64	**31.73**	592.64
cosh	**110.15**	406.27	**29.29**	431.53
tanh	**110.33**	546.20	**36.16**	575.13
modf	**16.50**	17.26	**10.20**	28.78
fmod	**34.95**	109.36	**29.43**	121.83
fabs	4.58	**3.18**	**3.44**	11.90
floor	**7.08**	22.36	**5.44**	29.71
ceil	**7.18**	22.21	**5.46**	29.62
ldexp	**7.19**	18.08	**5.76**	24.13
frexp	**6.14**	6.84	**5.06**	15.89
cmp	7.78	**7.12**	5.67	**4.66**
feq/gt/lt...	7.57	**6.91**	5.63	**4.64**
fp to long	5.59	**5.24**	4.75	**3.82**
fp to ulong	5.61	**4.46**	4.59	**3.05**
long to fp	6.73	**4.95**	5.86	**5.47**
ulong to fp	9.00	**6.77**	7.78	**6.88**
fp to llong	**7.13**	941.82	**6.36**	883.37
fp to ullong	**7.11**	578.60	**6.08**	549.23
llong to fp	**12.88**	697.78	**13.73**	753.11
ullong to fp	**13.69**	722.47	**12.15**	676.72
sgl to dbl	6.00	**4.00**	–	–
dbl to sgl	6.72	**6.00**	–	–

7.4 Hitachi SH Family

7.4.1 Compilers

There's a GoFast version for SH1, SH2, and SH3 that is compatible with GNU C. In addition, there's an SH3 version for the Hitachi C compiler.

7.4.2 Timings

The following table gives the timing, in microseconds, of some floating-point operations for the SH2. The test used a 6.144 MHz E7000 emulator; the data and the instruction cache were enabled.

function	double	single
add	57.3	36.0
subtract	61.5	38.3
multiply	62.0	33.9
divide	84.5	44.6
sqrt	225.1	76.2
exp	314.7	75.7
log	380.2	91.1
log10	390.8	94.3
sin	273.5	66.6
cos	272.0	65.3
tan	373.0	77.5
asin	520.6	195.4
acos	586.0	216.0
atan	341.9	109.4
atan2	400.2	139.7
pow	577.6	179.7

These times, in microseconds again, measure speed using a 7708 processor running at 60 MHz on a 15 MHz board.

	GNU		+ GoFast		HITACHI		+ GoFast	
function	double	single	double	single	double	single	double	single
add	54.8	36.4	18.2	12.7	19.6	10.6	17.5	10.8
subtract	60.8	39.6	20.8	13.9	23.1	10.9	19.8	12.4
multiply	170.2	60.2	18.6	11.2	26.1	9.9	17.9	9.7
divide	158.7	63.4	29.8	16.6	79.9	13.8	29.7	15.4
sqrt	240.9	120.6	45.7	27.3	167.5	57.5	44.7	25.9
exp	2126.8	984.6	73.2	19.3	642.5	310.4	65.4	18.1
log	3159.9	1358.8	97.7	27.6	563.9	270.5	88.0	26.4
log10	3508.0	1535.7	101.0	28.6	588.1	286.0	94.7	27.4
sin	1723.7	714.1	70.8	19.4	359.2	169.3	62.0	18.1
cos	2152.9	950.0	65.3	19.6	335.3	157.1	58.0	18.4
tan	3654.6	1538.3	91.0	25.9	476.6	212.5	83.8	24.7
asin	3931.7	1765.6	113.3	58.1	912.5	385.6	106.1	57.1
acos	3623.0	1569.7	107.3	66.4	934.5	401.5	98.6	65.6
atan	3593.8	1589.7	95.7	29.7	560.6	234.4	88.1	28.8
atan2	3871.3	1731.0	118.9	39.7	685.2	265.5	113.5	38.1
pow	10172.7	4498.1	153.0	50.2	1268.4	605.1	146.1	48.8

7.5 Intel x86

GoFast for the Intel x86 processors mostly means emulation, because that is what the compilers require. There is actually a drop-in GoFast library for I386 that works just fine with the Microsoft or the Borland compiler. But there's a catch: you'll have to do all floating-point operations using function calls. As this is not a practical idea, the library only serves as part of a test suite.

However, there is a compiler that will generate emulation-free code for the I386: GNU. Naturally there's also an I386 GoFast library for GNU, the real thing, faster than any emulation.

See the separate GoFast x86 User's Guide.

7.6 MIPS32

GoFast for MIPS32 replaces the old GoFast R3000 version. Improvements have been made. GoFast R4000 (for MIPS64) has been discontinued because there are very few 64-bit cores and the few that exist have a built-in FPU, we are told.

GoFast for MIPS32 runs on all MIPS32 rev1 and rev2 cores. We attempted to create a faster version for rev2 using the new instructions, but there was no benefit. Some functions were slightly faster and others slightly slower.

7.6.1 Compilers

GoFast for MIPS32 supports the following C compilers:

- MIPS SDE (GNU)

The GNU library does not provide single precision versions of most functions, but GoFast does.

7.6.2 Timings

The following table shows all times (in microseconds) for the indicated processor and evaluation board.

MIPS32: PIC32 Starter Kit (PIC32MX360F512L), 80 MHz, RAM

	Double-Precision		Single-Precision	
Function	**GoFast**	**GNU**	**GoFast**	**GNU**
add	0.93	1.53	0.75	0.84
subtract	0.92	1.60	0.70	0.82
multiply	0.97	1.60	0.60	0.77
divide	2.38	7.84	1.12	1.77
sqrt	3.93	8.95	1.67	1.69
exp	4.15	42.22	1.30	–
log	6.05	48.08	2.13	–
log10	6.48	49.64	2.24	–
pow	10.40	135.30	3.66	–
sin	3.77	29.32	1.46	–
cos	3.71	30.05	1.45	–
tan	6.40	59.46	2.10	–
asin	5.33	77.31	4.02	–
acos	5.16	76.13	4.54	–
atan	6.74	48.59	2.35	–
atan2	8.76	67.37	3.16	–
sinh	5.92	70.57	2.06	–
cosh	5.81	54.03	1.96	–
tanh	6.04	70.10	2.53	–
modf	0.55	1.10	0.44	–
fmod	4.83	80.09	2.68	–
fabs	0.14	0.15	0.11	0.11
floor	0.35	6.51	0.23	–
ceil	0.46	5.91	0.24	–
ldexp	0.31	0.80	0.29	0.51
frexp	0.21	0.47	0.19	–
cmp	0.69	0.73	0.56	0.52
fp to long	0.20	0.32	0.18	0.22
fp to ulong	0.20	1.14	0.18	0.75
long to fp	0.24	0.29	0.21	0.67
ulong to fp	0.28	0.40	0.29	0.77
sgl to dbl	0.20	0.24	–	–
dbl to sgl	0.27	0.44	–	–

7.7 Motorola 68000 Family

7.7.1 Compiler

GoFast supports the following Motorola 68k C compilers:

- GNU
- Intermetrics
- Microtec Research

The instruction set is the same, but the calling conventions and the assembly controls differ. There are two versions for each compiler; one supports full 32-bit multiply and divide (68020), the other only 16-bit multiply and divide (68000).

7.7.2 Timings

The following table shows some times (in microseconds) for the 25 MHz 68360 processor using the Microtec compiler.

	Single-Precision		Double-Precision	
function	**Microtec**	**GoFast**	**Microtec**	**GoFast**
add	25	28	51	48
multiply	32	30	73	51
divide	32	28	152	51
sqrt	250	50	712	62
exp	487	62	1550	225
pow	1025	137	3112	437
log	450	62	1587	200
log10	487	62	1650	225
sin	287	62	1187	175
cos	487	75	1675	162
tan	275	62	1212	250
asin	612	100	2337	262
acos	575	100	2237	287
atan	425	75	1737	175

7.8 NEC V830/V850 Families

7.8.1 Compiler Support

The GoFast routines are a direct replacement for the Green Hills C library. There's one version for the V830 family and one for the V850 family.

7.8.2 Timings

The following timings are from a NEC V830 processor running at 33 MHz on an RTE-V830-PC board. The test used the Green Hills compiler version 1.8.8.

function	double	single
add	8.3	5.0
subtract	10.5	6.1
multiply	7.6	13.2
divide	14.4	6.3
sqrt	24.7	13.7
exp	34.4	13.3
log	52.7	16.6
log10	56.1	18.6
sin	32.3	14.5
cos	31.4	13.9
tan	46.9	16.7
asin	68.5	33.6
acos	74.2	37.3
atan	46.9	18.2
atan2	55.0	26.2
pow	78.3	36.2

The next timings apply to a NEC V851 with a 6.6 MHz external crystal and 33 MHz phase-lock loop on an RTE-V851-PC board. The Green Hills compiler version was 1.8.8.

	Single-Precision		Double-Precision	
function	GH	GoFast	GH	GoFast
add	58.0	24.0	99.0	36.0
subtract	82.0	28.0	137.0	45.0
multiply	692.0	126.0	632.0	104.0
divide	564.0	157.0	2076.0	322.0
sqrt	1432.0	344.0	8255.0	683.0
exp	1121.0	210.0	8260.0	923.0
log	2145.0	285.0	10392.0	1015.0
log10	2324.0	306.0	11075.0	1081.0
sin	942.0	169.0	7409.0	669.0
cos	1015.0	151.0	7489.0	669.0
tan	2385.0	317.0	10850.0	1312.0
asin	4993.0	762.0	24382.0	1846.0
acos	4912.0	911.0	18789.0	2179.0
atan	1984.0	283.0	13130.0	847.0
atan2	2680.0	424.0	15428.0	1169.0
pow	3732.0	542.0	19839.0	2003.0

7.9 PowerPC

7.9.1 Compilers

GoFast for PowerPC offers drop-in libraries for the following C compilers:

- Diab Data
- GNU
- Metaware
- Motorola

There's also an emulator interface, used by the IBM compiler. GoFast for PowerPC provides two emulators, for different CPU variants, with some sample code for installing them. Emulation is never a totally painless option in embedded systems; fortunately you are much more likely to need a drop-in library than an emulator.

7.9.2 Timings

The following timings are for the GoFast EABI interface from a PPC860T processor running at 50 MHz (25 MHZ bus), caching disabled. The benchmark program was built with the Diab Data C compiler.

	GoFast		Diab Data	
function	**double**	**single**	**double**	**single**
add	32.6	23.4	97.0	32.6
subtract	38.0	26.1	132.2	39.4
multiply	36.9	22.6	63.5	30.9
divide	61.2	30.0	413.1	131.9
sqrt	110.4	54.7	374.4	98.3
exp	221.3	68.9	1376.0	544.3
log	252.3	59.2	1475.1	585.9
log10	264.9	61.9	1537.1	616.8
sin	177.1	58.4	614.9	439.6
cos	174.3	57.7	732.4	474.4
tan	283.1	66.5	1090.8	507.9
asin	329.7	105.6	1174.0	572.7
acos	394.0	126.8	1303.3	610.0
atan	209.7	61.2	1383.7	622.7
atan2	259.1	80.0	1988.0	842.7
pow	469.9	138.3	8575.7	3166.7

7.10 SPARC

There's a drop-in GoFast library for the V7/V8 (32-bit divide), the SPARClite (1-bit divide), and the Fujitsu MB86934 (1-bit divide, scan). These work with the following compilers:

- GNU
- Microtec
- Sun Microsystems

There's also a floating-point emulator for SPARC. This will allow the same binary program to run both in the V7 and in the V8 reasonably efficiently. The following table gives some times (in microseconds) for emulated floating-point instructions. The platform was the 49 MHz Fujitsu SPARClite evaluation board with static RAM. The measurements were done with and without the cache.

	single		double	
instruction	**SRAM**	**cache**	**SRAM**	**cache**
`branch, not taken`	1.60	0.85	2.55	0.85
`branch, taken`	1.75	0.90	2.70	0.90
`load`	2.70	1.30	2.65	1.35
`store`	2.55	1.30	2.95	1.55
`add`	6.80	3.30	7.90	4.25
`subtract`	6.80	3.30	9.05	4.45
`multiply`	6.60	3.25	8.20	4.30
`divide`	8.35	4.75	12.10	6.10
`compare`	4.75	2.35	4.65	2.25
`integer to real`	4.40	2.10	4.50	2.25
`real to integer`	4.75	2.35	5.40	4.15
`double to single`	5.45	2.95		
`single to double`			4.15	2.30
`square-root`	12.95	7.30	22.60	13.05

8 References

ANSI/IEEE Standard 754-1985: Binary Floating-Point Arithmetic

ANSI Document X3J11/88-159: Draft Proposed American National Standard for Information Systems – Programming Language C

W. Cody, W. Waite: Software Manual for the Elementary Functions, Prentice-Hall, 1980