SPECIAL COMPUTATIONAL INSTRUCTIONS

The instructions presented in the preceding chapter dealt with data transfers (loading and storing information) and elementary arithmetic operations (addition and subtraction). This chapter describes more advanced computational instructions such as square root and exponentiation. These instructions perform operations on the top-of-stack register(s) and thus always treat the 8087 as a simple stack.

The special computational instructions can be subdivided into *special arithmetic instructions* and *elementary transcendental instructions* as shown in table 6.1. The formats for these instructions are shown in figure 6.1.

Special Arithmetic Instructions

The special arithmetic instructons are *absolute-value*, *change-sign*, *round-to-integer*, *extract*, *square-root*, *scale*, and *remainder*.

The absolute-value instruction (FABS) changes the sign bit of the top-of-stack register to 0 (non-negative), and the change-sign instruction (FCHS) inverts the sign of the top-of-stack register.

The round-to-integer instruction (FRNDINT) does nothing if the stack top is an integral value; otherwise, the instruction converts the stack-top value to one of the two integral values bounding it, as determined by the rounding mode.

The extract instruction (FXTRACT) decomposes the stack-top value into its exponent and significand. The stack-top value is removed from the stack, and the exponent followed by the significand are then pushed on the stack. For example, if

Table 6.1 Special Computational Instructions

Special Arithmetic Instructions

mnemonic	*function*
FABS	absolute value
FCHS	change sign
FRNDINT	round to integer
FXTRACT	extract exponent and significand
FSQRT	square root
FSCALE	scale
FPREM	partial remainder

Elementary Transcendental Instructions

mnemonic	*function*
FPTAN	partial tangent
FPATAN	partial arctangent
FYL2X	$y*\log_2(x)$
FYL2XP1	$y*\log_2(x+1)$
F2XM1	2^x-1

the stack top has the value $1.5*2^{-201}$, FXTRACT would create $1.5*2^0$ and $-201*2^0$ as the new stack-top and next-to-top values. FXTRACT is useful whenever the exponent and significand must be operated on separately as, for example, when performing binary to decimal conversions.

The 8087 square-root instruction (FSQRT) replaces the value in the top-of-stack register with its square root. This instruction is unique to commercial floating-point implementations in several ways. For one, it is as fast as a divide instruction (in fact, it is slightly faster because the 8087 does not have to check for overflow or underflow since neither can occur). For another, it is as accurate (to within one-half a unit in the last place of the significand) as the primitive functions of add, subtract, multiply, and divide. And, finally, the square root is implemented so that all rounding modes are available. Because of its speed and accuracy, users of the 8087 can think of square root as if it were one of the primitive arithmetic functions and not an operation to avoid.

The scale instruction (FSCALE) is obviously used to scale variables. It adds the value (assumed to be integral) in the register below the stack top (this value is called the *scale factor*) to the exponent of the stack-top value. This is a very fast way of multiplying (dividing if scale facter is negative) by a power of two (scaling). Note that the scale factor was not put in the top-of-stack register, but in the next register below it. This makes it easy to scale a sequence of values, such as an array, by the same factor; we simply load, scale, and store each array element with no need for intermediate stack manipulations.

1	1	0	1	1	0	0	1		1	1	1	1	1	0	1	0

FSQRT — square root of ST(0)

1	1	0	1	1	0	0	1		1	1	1	1	1	1	0	1

FSCALE — scale ST(0) by ST(1)

1	1	0	1	1	0	0	1		1	1	1	1	1	0	0	0

FPREM — partial remainder of ST(0) ÷ ST(1)

1	1	0	1	1	0	0	1		1	1	1	1	1	1	0	0

FRNDINT — round ST(0) to integer

1	1	0	1	1	0	0	1		1	1	1	1	0	1	0	0

FXTRACT — extract components of ST(0)

1	1	0	1	1	0	0	1		1	1	1	0	0	0	0	1

FABS — absolute value of ST(0)

1	1	0	1	1	0	0	1		1	1	1	0	0	0	0	0

FCHS — change sign of ST(0)

1	1	0	1	1	0	0	1		1	1	1	1	0	0	1	0

FPTAN — partial tangent of ST(0)

1	1	0	1	1	0	0	1		1	1	1	1	0	0	1	1

FPATAN — partial arctangent of ST(0) ÷ ST(1)

1	1	0	1	1	0	0	1		1	1	1	1	0	0	0	0

F2XM1 — $2^{ST(0)}-1$

1	1	0	1	1	0	0	1		1	1	1	1	0	0	0	1

FYL2X — $ST(1) \cdot \log_2[ST(0)]$

1	1	0	1	1	0	0	1		1	1	1	1	1	0	0	1

FYL2XP1 — $ST(1) \cdot \log_2 [ST(0) + 1]$

Fig. 6.1 Formats of special computational instructions

The remainder instruction (FPREM) is unique to the 8087. It is used to calculate *exact* remainders. We are emphasizing "exact" because that is what is unique about it; no other floating-point operation is exact over its entire domain, and no other computer provides this operation. The FPREM instruction is intended to be used by system programmers so that they can provide us with reliable transcendental functions over these functions' entire domains.

To see why an exact remainder is useful for providing reliable transcendental functions, consider the problem of calculating sin(1,000,000.5). The transcendental functions (like sin, cos, etc.) are computed by the 8087 using an approximation that is good only when the argument is fairly small. Therefore, you can make use of the fact that sin is periodic according to the relationship:

$$\sin(x + 2n\pi) = \sin(x)$$

to reduce the argument to a small range. This is done by calculating the new variable y as:

$$y = \text{remainder}\left(\frac{x}{2\pi}\right)$$

Now we know that $0 \leq y \leq 2\pi$ and that $\sin(x) = \sin(y)$. The last equality holds only if the remainder is exact and, unfortunately, on all machines except the 8087, it is not.

Often the remainder is calculated as follows:

$$t = \frac{x}{2\pi}$$

$$y = t - \text{(integer part of t)}$$

The problem here is that t has been contaminated by roundoff error and, in fact, if x is very large, t becomes an integer and y becomes zero. This "rounded" remainder causes the periodic functions to lose their periodic properties. For example, the common identities such as "$\cos^2 x + \sin^2 x = 1$" no longer hold. This is why that remainder should be exact.

The remainder instruction produces an exact result because it performs its operation by doing successive subtractions, much the same as we would do in conventional pencil-and-paper long division. This could involve an extremely large number of subtractions and tie the 8087 up for a relatively long time. If the remainder instruction were allowed to complete all its subtractions without being interrupted, a high priority interrupt routine might have to wait an unacceptably long time before gaining access to the 8087. For this reason FPREM is a partial re-

mainder instruction; it does at most 64 subtractions and returns the result obtained to that point even if there are more subtractions still to do. The condition-code bit C2 is set to 1 if more subtractions are necessary and to 0 otherwise. Thus, the following loop is necessary in order to calculate the complete remainder:

```
LOOP:
   FPREM
   if C2 = 1 go to LOOP
```

The number of subtractions was limited to 64 so the FPREM instruction would never be slower than the FDIV instruction.

Note that after executing FPREM, three of the condition-code bits supposedly reflect the least-significant three bits of the quotient (see table 4.3). This was intended to simplify the task of reducing the argument of a function such as sin(x). We would simply divide x by $\pi/4$ and then, depending on the octant that x is in, compute either $+\sin$, $-\sin$, $+\cos$, or $-\cos$ of either the remainder or $\pi/4$ minus the remainder. The octant is obtained from the last three bits of the quotient, hence the use of the condition-code bits.

However, it turned out to be very difficult to implement such condition-code settings correctly, and nobody has been able to prove that the 8087's condition codes do indeed return the correct octant in all cases (for less than 62 subtractions it has been proven that the condition codes are correct). Furthermore, it's really not essential that we know the octant since we could have reduced x by dividing by π, then by 2 if necessary, and then by 2 again if still necessary. So what was thought to be a useful feature early in the design of the 8087, was realized later not to be necessary and is possibly not even implemented reliably. It's interesting to note that none of Intel's floating-point library routines use this feature.

Elementary Transcendental Instructions

The elementary transcendental instructions consist of two trigonometric instructions, two logarithmic instructions, and an exponential instruction. These instructions can be used to compute all the trigonometric, hyperbolic, logarithmic, and exponential functions within a restricted range.

The reason for the restricted range on some instructions is to save microcode space on the 8087 (microcode is a program stored within the 8087 that defines the algorithms used by the processor). It required some special techniques to fit just the restricted functions into the microcode. Thus, the 8087 performs what would otherwise be the most time-consuming portion of the computation and leaves the task of argument reduction to the user's program.

Trigonometric Instructions ▪ All trigonometric and inverse trigonometric functions can be computed from the elementary instructions FPTAN (tangent) and

FPATAN (arctan).

FPTAN computes two values, y and x, from the stack-top value z where y/x = tan(z). After executing FPTAN, z is removed from the top of the stack, and y followed by x are then pushed on the stack.

The FPATAN instruction is the opposite of FPTAN. It calculates z = arctan(y/x), where x is the stack-top value and y is next value on the stack. FPATAN removes both y and x from the stack and then pushes z onto the stack.

The argument for FPTAN must lie in the range $0 < z < \pi/4$. The operands for FPATAN are assumed to obey the relationship $0 < y < x$. Note that the results generated by FPTAN satisfy this relationship (since $\tan(z) = y/x < 1$ for $z < \pi/4$); hence, FPATAN can use as arguments the results generated by FPTAN.

The argument range for FPTAN excludes the value zero. This was done to simplify the microcode. Thus, tan(0) (or of any denormal for that matter) must be detected by software and calculated as a special case. This presents no problems since $\tan(z) \approx z$ for small z.

Both FPTAN and FPATAN are very accurate; their error is confined to a few units in the last place of temporary-real format (i.e., at least 61 out of 64 bits are correct). They are also fast; only about four times slower than divide. Because of their speed and accuracy, these functions can be treated almost as if they were primitive operations. FPTAN and FPATAN, along with FSQRT, form the basis of all other trigonometric and inverse trigonometric functions as shown in table 6.2.

Table 6.2 Trigonometric Functions

A. Normal Trigonometric Functions

z = stack-top value prior to executing FPTAN
x,y = stack-top value and next value after dividing z by 2 and then executing FPTAN

$$\sin(z) = \frac{2(y/x)}{[1 + (y/x)^2]}$$

$$\cos(z) = \frac{[1 - (y/x)^2]}{[1 + (y/x)^2]}$$

$$\tan(z/2) = \frac{y}{x}$$

$$\cot(z/2) = \frac{x}{y}$$

$$\csc(z) = \frac{[1 + (y/x)^2]}{2(y/x)}$$

$$\sec(z) = \frac{[1 + (y/x)^2]}{[1 - (y/x)^2]}$$

B. Inverse Trigonometric Functions

z = argument of desired inverse trigonometric function f
x,y = stack-top value and next value before executing FPATAN so that resulting stack-top value is f(z)

$$\arcsin(z) = \arctan\left(\frac{z}{\sqrt{(1-z)(1+z)}}\right) = \arctan\left(\frac{y}{x}\right)$$

$$\arccos(z) = 2^* \arctan\left(\sqrt{\frac{1-z}{1+z}}\right) = 2^* \arctan\left(\frac{y}{x}\right)$$

$$\arctan(z) = \arctan\left(\frac{z}{1}\right) = \arctan\left(\frac{y}{x}\right)$$

$$\operatorname{arccot}(z) = \arctan\left(\frac{1}{z}\right) = \arctan\left(\frac{y}{x}\right)$$

$$\operatorname{arccsc}(z) = \arctan\left(\frac{\operatorname{sign}(z)}{\sqrt{(z-1)(z+1)}}\right) = \arctan\left(\frac{y}{x}\right)$$

$$\operatorname{arcsec}(z) = 2^* \arctan\left(\sqrt{\frac{z-1}{z+1}}\right) = 2^* \arctan\left(\frac{y}{x}\right)$$

Note: These formulas must be used with care. The required argument reduction is not explicitly stated and special arguments such as NAN, infinity, denormals, and zero must be dealt with separately.

In the table, the trigonometric functions such as sin(z) are expressed in terms of tan(z/2) rather than in terms of tan(z). It can be shown that computing sin(z) in this manner greatly reduces the roundoff error. In fact, all the formulas appearing in this table have been carefully chosen to reduce roundoff error as much as possible.

Note the subtle difference between computing trigonometric functions and inverse trigonometric functions. The former are computed by first executing the FPTAN instruction on the desired argument (after reducing it to the required range) and then performing arithmetic operations on the two results generated by the FPTAN instruction. The latter are computed by first performing arithmetic operations on the desired argument and then executing the FPATAN instuction on the two results generated by these arithmetic operations.

For example, sin(z) is computed by (1) placing z/2 on the top of the stack, (2) executing FPTAN, thereby obtaining values called y and x on the top of the stack, and (3) performing arithmetic operations to obtain $2(y/x)/[1+(y/x)^2]$. Conversely, arcsin(z) is computed by (1) performing arithmetic operations to obtain x which is $[(1-z)(1+z)]^{1/2}$ and y which is z itself, (2) placing y and x on the top of the stack, and (3) executing FPATAN.

As stated above, the FPTAN instruction requires that its argument be less than $\pi/4$. The process of turning a larger number into this range is called *argument reduction*. In this case we can reduce a large argument by applying the identity $\tan(x+n\pi)=\tan(x)$. In other words, if $x>\pi$, we first compute $z=x \bmod \pi$, and then compute tan(x) instead of tan(z). Thus, although x can be any value, we only have to be able to calculate tan(x) from 0 to π. Also since $\tan(x+\pi/2)=-\cot(x)$, we can reduce the interval to 0 to $\pi/2$. The identity $\tan(\pi/2-x)=\cot(x)$ where $0\leq x\leq\pi/4$ permits the required range to be reduced to 0 to $\pi/4$.

All these identities are exact, but the computation of x mod π presents a major problem. Since no commercial computer or minicomputer has a true floating-point "mod" capability, the calculation of x mod π is usually done with a divide and subtract; that is, x mod π is approximated by x/π − (integer part of x/π). This means there will be roundoff error in the computation, even if extended precision is used. Thus, not only is the function (tan, sin, etc.) calculated with roundoff error, but so is the argument to the function.

Calculated $(\tan(x))=\tan^*(x^*)$ where * means rounding error

Consequently the calculated tan does not have true periodicity; thus, the trigonometric identities are not even approximately satisfied.

As we have already seen, the 8087 has a true floating-point "mod" function (FPREM) allowing x mod π to be calculated without roundoff error. You may object that, since π is not exactly representable and must be represented by π^*, we still have a roundoff error in the argument reduction. It is true there is an error, but it is not a roundoff error; rather it is a systematic error whose effect is simply to change the period of the calculated tan to π^* instead of π. Since the calculated

trigonometric functions are periodic and calculated to within a small roundoff error, the identities are satisfied to within a small roundoff error. (This is true as long as π does not appear explicitly in the identity.) Therefore, by using temporary-real precision and the "mod" function, the 8087 trigonometric functions remain accurate and reliable.

Logarithmic Instructions • Two logarithmic instructions, FYL2X and FYL2XP1, are provided on the 8087. They are calculated with a binary radix (that is $\log_2$) since, on a binary machine like the 8087, implementing the instructions is much simpler in radix 2 than in radix 10 or e. Logarithms with other radices can be computed from the two instructions provided, as shown in table 6.3(A).

The first logarithmic instruction, FYL2X, takes two operands, x and y, where x is on the stack top and y is next under x. The instruction replaces x and y with $y*\log_2(x)$. This instruction assumes that $x>0$ (since the log function is defined only for positive arguments) and y is any valid floating-point number. The factor y appears as part of the instruction because any practical use of log almost always involves a multiplying factor. Examples illustrating such factors are:

$$\log_e(x) = y*\log_2(x) \text{ where } y = \log_e(2)$$
$$\log_{10}(x) = y*\log_2(x) \text{ where } y = \log_{10}(2)$$
$$x^y = 2^{(y*\log_2(x))}$$

If the above instruction didn't include a multiplying factor, the multiplication could be accomplished by following the logarithmic instruction with a multiplication instruction. The obvious advantages to a single instruction are decreased execution time and decreased program size. A less obvious, but more important, advantage is increased accuracy. The increased accuracy results because internally the factor $\log_2(x)$ is calculated as accurately as possible to 67 bits, then multiplied by y before being rounded to 64 bits. The three extra bits used here are the bits usually used for rounding as discussed in chapter 2.

The implementation of the logarithmic instruction above shows an unusual concern for accuracy. But this concern is justified. A goal in the design of the 8087 was to be able to calculate all the elementary functions on arguments in long-real format and obtain a long-real result with an error of less than one unit in the last place. This goal is one reason temporary-real format was provided for intermediate results. The most difficult elementary function to compute that routinely appears in high-level languages is x^y. We saw in chapter 2 that intermediate results must be represented in a floating-point format with at least a 64-bit significand if we are to be sure of the accuracy of x^y in long-real format. But this is exactly the size of the significand field in temporary-real format; thus, there is no margin for error. Consequently, every extra bit of accuracy in the computation of $y*\log_2(x)$ is precious.

The other logarithmic instruction, FYL2XP1, computes $y*\log_2(1+x)$. To understand why such a function is useful, let's consider interest-rate calculations. Here we frequently encounter expressions of the form $(1+i)^n$, where i is much

Table 6.3 Formula for Computing Logarithmic, Exponentiation, and Hyperbolic Functions

A. Logarithmic Functions

$\log_2(x) = FYL2X(x)$

$\log_e(x) = \log_e(2)*\log_2(x) = FYL2X(\log_e(2),x) = FYL2X(FLDLN2,x)$

$\log_{10}(x) = \log_{10}(2)*\log_2(x) = FYL2X(\log_{10}(2),x) = FYL2X(FLDLG2,x)$

B. Exponentiation Functions

$2^x = (2^x - 1) + 1 = F2XM1(x) + 1$

$e^x = 1 + (2^{x*\log_2(e)} - 1) = 1 + F2XM1(x*\log_2(e)) = 1 + F2XM1(x*FLDL2E)$

$10^x = 1 + (2^{x*\log_2(10)} - 1) = 1 + F2XM1(x*\log_{2(10)} = 1 + F2XM1(x*FLDL2T)$

$x^y = 1 + (2^{y*\log_2(x)} - 1) = 1 + F2XM1(y*\log_2(x)) = 1 + F2XM1(FYL2X(y,x))$

C. Hyperbolic Functions

$$\sinh(x) = \frac{\text{sign}(x)}{2}\left[(e^{|x|} - 1) + \frac{e^{|x|} - 1}{e^{|x|}}\right]$$

$$\cosh(x) = \frac{1}{2}\left[e^{|x|} + \frac{1}{e^{|x|}}\right]$$

$$\tanh(x) = \text{sign}(x) - \left[\frac{e^{2*|x|} - 1}{e^{2*|x|} + 1}\right]$$

$\coth(x) = 1/\tanh(x)$

$\text{csch}(x) = 1/\sinh(x)$

$\text{sech}(x) = 1/\cosh(x)$

D. Inverse Hyperbolic Functions

$\text{arcsinh}(x) = [\text{sign}(x)]\ [\log_e(2)]\ [\log_2(1+z)] = \text{FYL2XP1}(\text{sign}(x)*\text{FLDLN2},z)$

$$\text{where } z = |x| + \frac{|x|}{\frac{1}{|x|} + \sqrt{1+\left(\frac{1}{|x|}\right)^2}}$$

$\text{arccosh}(x) = [\log_e(2)]\ [\log_2(1+2)] + \text{FYL2XP1}(\text{FLDLN2},z)$

where $z = x - 1 + \sqrt{(x-1)\ (x+1)}$

and $x \geq 1$

$\text{arctanh}(x) = [\text{sign}(x)]\ [\log_e(2)]\ [\log_2(1+z)]$

$= \text{FYL2XP1}(\text{FLDLN2}*\text{sign}(x),z)$

$$\text{where } z = \frac{2|x|}{1-|x|}$$

and $-1 < x < 1$

$\text{arccoth}(x) = \text{arctanh}(1/x)$

$\text{arccsch}(x) = \text{arcsinh}(1/x)$

$\text{arcsech}(x) = \text{arccosh}(1/x)$

E. Miscellaneous

The following function appears frequently in financial computations:

$$\frac{(1+z)^m - 1}{z} = \frac{2^{m \log_2(1+z)} - 1}{z} = \frac{\text{F2XM1}(\text{FYL2XP1}(m,z))}{z} = \text{(for } z \text{ non = zero)}$$

$$= m \text{ (for } z = 0)$$

where $z > -1$

Note: These formulas must be used with care. The required argument reduction is not explicitly stated and special arguments such as NAN, infinity, denormals, and zero must be dealt with separately.

smaller than 1. Evaluating such an expression involves computing the log of 1 + i. But to add i to 1 before taking the log, the 8087 would have to first denormalize i (so that the binary points line up). This results in losing valuable bits of i as they are shifted off the right end. Thus, if i_1 and i_2 are small but different numbers, the calculated values $\log(1+i_1)$ and $\log(1+i_2)$ could easily be the same. By using the 8087 function $y*\log_2(1+x)$, the value 1 + x is never actually formed; instead the algorithm implicitly takes the 1 into account, and no bits of x are lost. Although y can be any number, x is required to lie in a restricted range, namely $|x| < 1 - \sqrt{1/2}$. This restriction saves microcode space and does not diminish the utility of the function since it is needed only when x is small.

Exponential Instruction ▪ The 8087 provides a single exponential instruction, F2XM1. This instruction computes $2^x - 1$. It obtains its operand from the top-of-stack register and replaces that operand with the computed result. The argument x is restricted to $0 \leq x \leq 1/2$.

The reason for providing such a function rather than providing 2^x is similar to the reason just given above, namely loss of precision when calculating $2^x - 1$ for small x by explicitly subtracting 1 from 2^x. (The hyperbolic functions in table 6.3(C) illustrate the need for $2^x - 1$ with small x.) On the other hand, if 2^x is the value desired, we can obtain this by simply adding 1 to the result of $2^x - 1$ with no loss of precision.

We usually want either e^x or 10^x instead of 2^x. The algorithms for computing these exponential functions as well as the hyperbolic functions are given in table 6.3. Also present in this table is a miscellaneous function often appearing in financial computations. The formulas appearing in this table have been carefully chosen to reduce roundoff error as much as possible.

The binary radix is used in the exponential instruction for the same reason it was used in the logarithm instructions: because functions with a binary radix are easier to implement on a binary machine. Exponents to other bases can be computed using the given exponential instruction as shown in table 6.3(B).

The exponential instruction requires that x be in a particular range to save space in the microcode. This poses no problem since an arbitrary x can be reduced to that range by applying the relation:

$$2^x = (2^i)*(2^f)$$

where *i* is the nearest integer to x
and *f* is a fraction that is the difference between x and i.

It follows that the magnitude of f is less than 1/2. Thus, we can calculate $2^x - 1$ for arbitrary x as follows:

1. Obtain i and f from i = FRNDINT(x) and f = x − i.
2a. If f is non-negative, evaluate $2^f - 1$ using F2XM1.

2b. If f is negative, evaluate $2^{(-f)}-1$ using F2XM1 and then obtain $2^{f}-1$ from the relation:

$$2^{f}-1 = -\left[\frac{2^{(-f)}-1}{2^{(-f)}}\right]$$

3. If i = 0, we're finished. Otherwise, add 1 to the value of $2^{f}-1$ just obtained, multiply by 2^{i} using the FSCALE instruction, then subtract 1.

Thus, the instruction is applied on the operand f, and then i is inserted into the exponent field of the result.

It's All Done with Mirrors ▪ The 8087 transcendental instructions are implemented using an argument reduction related to the CORDIC (COordinate Rotation DIgital Computer) algorithm. This algorithm is described in a paper by J.S. Walther entitled "A Unified Algorithm for Elementary Functions," published in the Proceedings of the Spring 1971 Joint Computer Conference.

The algorithm relies upon a table of constants. The number of constants required by CORDIC type methods is equal to the number of bits in the result; in the case of the 8087, that would mean 64 constants of 64 bits each. Since two distinct calculations are involved (one for logarithmic functions and one for trigonometric functions), the CORDIC would require 128 64-bit constants, which would necessitate an 8192-bit memory on the 8087 chip.

Considering all the other logic required by the 8087 and the technology available in 1978 (HMOS I), the number of constants had to be reduced significantly if transcendental functions were to be included on the 8087. By modifying the CORDIC, the number of bits needed were reduced to about 2000 with some compromise in speed, but very little in accuracy. These modifications essentially consist of stopping the CORDIC early and using rational approximations at the end.

Summary

The 8087 is a very powerful and complex device. Besides offering all the standard capabilities of a floating-point engine, it has several unique features. These include extended precision and correct rounding, fast and accurate elementary functions and square root, ability to simulate multiple stacks, automatic mixed modes, exact floating-point remainder, and automatic exception handling. However, the 8087 is more than a collection of features; it is an attempt to implement the spirit

of the IEEE floating-point standard faithfully, thereby facilitating reliable numerical computations.

In the past three chapters we have explained the 8087 architecture and the rationale for it. We have seen how the 8087 was designed to make accurate, reliable computations easy in most cases and possible in difficult cases. By using the extended precision provided, we can take double-precision operands and usually return accurate double-precision results without extensive analysis of the computation. With appropriately selected algorithms and judicious use of extended precision, we can get results that are usually much better than in any other familiar computing environment.

Now that you know what an 8087 is, it's time to learn how to write programs for it. This is the topic of the following chapters.