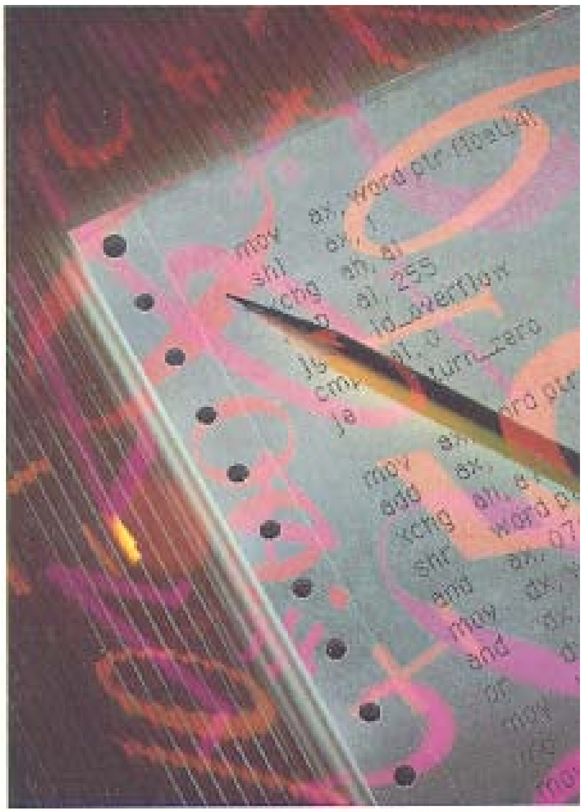
Numerical Methods

DISK INCLUDED

Real-Time and Embedded Systems Programming

Featuring in-depth coverage of:

- Fixed and floating point mathematical techniques without a coprocessor
- Numerical I/O for embedded systems
- Data conversion methods



M&T BOOKS

Don Morgan

Numerical Methods

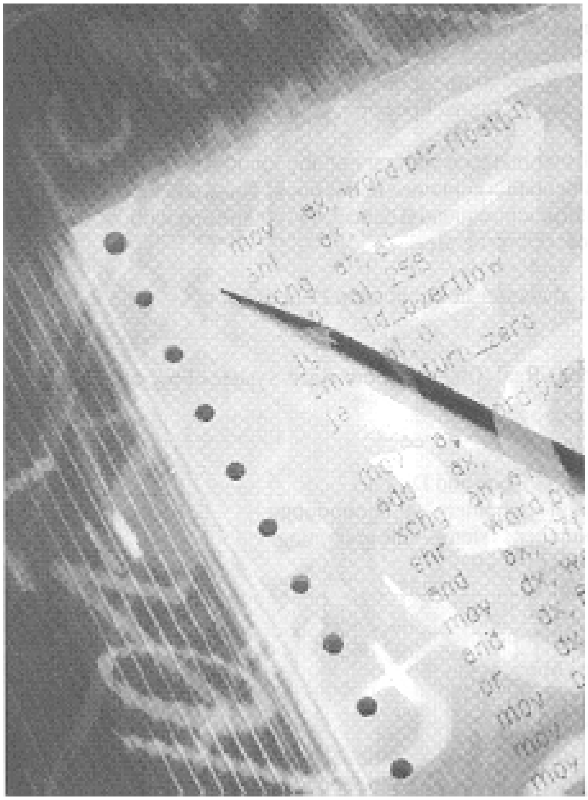
Real-Time and Embedded Systems Programming

Numerical Methods

Real-Time and Embedded Systems Programming

Featuring in-depth coverage of:

- Fixed and floating point mathematical techniques without a coprocessor
- Numerical I/O for embedded systems
- Data conversion methods



Don Morgan

M&T BOOKS

M&T Books
A Division of M&T Publishing, Inc.
411 BOREL AVE.
SAN MATEO, CA 94402

Printed in the United States of America

Library of Congress Cataloging-in-Publication Data

Morgan, Don 1948-
Numerical Methods/Real-Time and Embedded Systems Programming
by Don Morgan
p. cm.
Includes Index
ISBN l-5585l-232-2 Book and Disk set
1. Electronic digital computers—Programming. 2. Real-time data processing.
3. Embedded computer systems—Programming. I. Title
QA76.6.M669 1992
513.2 ' 0285—dc20 91-47911
CIP

Project Editor: Sherri Morningstar **Cover Design:** Lauren Smith Design

95 94 93 92 4 3 2 1

Acknowledgment

Thank you Anita, Donald and Rachel
for your love and forbearance.

Contents

APPENDIXES:

Additional Disk

Just in case you need an additional disk, simply call the toll-free number listed below. The disk contains all the routines in the book along with a simple C shell that can be used to exercise them. This allows you to walk through the routines to see how they work and test any changes you might make to them. Once you understand how the routine works, you can port it to another processor. Only $10.00 postage-paid.

To order with your credit card, call Toll-Free l-800-533-4372 (in CA 1-800-356-2002). Mention code 7137. Or mail your payment to M&T Books, 411 Bore1 Ave., Suite 100, San Mateo, CA 94402-3522. California residents please add applicable sales tax.

Why This Book Is For You

The ability to write efficient, high-speed arithmetic routines ultimately depends upon your knowledge of the elements of arithmetic as they exist on a computer. That conclusion and this book are the result of a long and frustrating search for information on writing arithmetic routines for real-time embedded systems.

With instruction cycle times coming down and clock rates going up, it would seem that speed is not a problem in writing fast routines. In addition, math coprocessors are becoming more popular and less expensive than ever before and are readily available. These factors make arithmetic easier and faster to use and implement. However, for many of you the systems that you are working on do not include the latest chips or the faster processors. Some of the most widely used microcontrollers used today are not Digital Signal Processors (DSP), but simple eight-bit controllers such as the Intel 8051 or 8048 microprocessors.

Whether you are using one on the newer, faster machines or using a simple eight-bit one, your familiarity with its foundation will influence the architecture of the application and every program you write. Fast, efficient code requires an understanding of the underlying nature of the machine you are writing for. Your knowledge and understanding will help you in areas other than simply implementing the operations of arithmetic and mathematics. For example, you may want the ability to use decimal arithmetic directly to control peripherals such as displays and thumbwheel switches. You may want to use fractional binary arithmetic for more efficient handling of D/A converters or you may wish to create buffers and arrays that wrap by themselves because they use the word size of your machine as a modulus.

The intention in writing this book is to present a broad approach to microprocessor arithmetic ranging from data on the positional number system to algorithms for developing many elementary functions with examples in 8086 assembler and pseudocode. The chapters cover positional number theory, the basic arithmetic operations to numerical I/O, and advanced topics are examined in fixed and floating point arithmetic. In each subject area, you will find many approaches to the same problem; some are more appropriate for nonarithmetic, general purpose machines such as the 8051 and 8048, and others for the more powerful processors like the Tandy TMS34010 and the Intel 80386. Along the way, a package of fixed-point and floating-point routines are developed and explained. Besides these basic numerical algorithms, there are routines for converting into and out of any of the formats used, as well as base conversions and table driven translations. By the end of the book, readers will have code they can control and modify for their applications.

This book concentrates on the methods involved in the computational process, not necessarily optimization or even speed, these come through an understanding of numerical methods and the target processor and application. The goal is to move the reader closer to an understanding of the microcomputer by presenting enough explanation, pseudocode, and examples to make the concepts understandable. It is an aid that will allow engineers, with their familiarity and understanding of the target, to write the fastest, most efficient code they can for the application.

Introduction

If you work with microprocessors or microcontrollers, you work with numbers. Whether it is a simple embedded machine-tool controller that does little more than drive displays, or interpret thumbwheel settings, or is a DSP functioning in a real-time system, you must deal with some form of numerics. Even an application that lacks special requirements for code size or speed might need to perform an occasional fractional multiply or divide for a D/A converter or another peripheral accepting binary parameters. And though the real bit twiddling may hide under the hood of a higher-level language, the individual responsible for that code must know how that operation differs from other forms of arithmetic to perform it correctly.

Embedded systems work involves all kinds of microprocessors and microcontrollers, and much of the programming is done in assembler because of the speed benefits or the resulting smaller code size. Unfortunately, few references are written to specifically address assembly language programming. One of the major reasons for this might be that assembly-language routines are not easily ported from one processor to another. As a result, most of the material devoted to assembler programming is written by the companies that make the processors. The code and algorithms in these cases are then tailored to the particular advantages (or to overcoming the particular disadvantages) of the product. The documentation that does exist contains very little about writing floating-point routines or elementary functions.

This book has two purposes. The first and primary aim is to present a spectrum of topics involving numerics and provide the information necessary to understand the fundamentals as well as write the routines themselves. Along with this information are examples of their implementation in 8086 assembler and pseudocode that show each algorithm in component steps, so you can port the operation to another target. A secondary, but by no means minor, goal is to introduce you to the benefits of binary arithmetic on a binary machine. The decimal numbering system is so pervasive that it is often difficult to think of numbers in any other format, but doing arithmetic in decimal on a binary machine can mean an enormous number of wasted machine cycles, undue complexity, and bloated programs. As you proceed through this book, you should become less dependent on blind libraries and more able to write fast, efficient routines in the native base of your machine.

Each chapter of this book provides the foundation for the next chapter. At the code level, each new routine builds on the preceeding algorithms and routines. Algorithms are presented with an accompanying example showing one way to implement them. There are, quite often, many ways that you could solve the algorithm. Feel free to experiment and modify to fit your environment.

Chapter 1 covers positional number theory, bases, and signed arithmetic. The information here provides the necessary foundation to understand both decimal and binary arithmetic. That understanding can often mean faster more compact routines using the elements of binary arithmetic-in other words, shifts, additions, and subtractions rather than complex scaling and extensive routines.

Chapter 2 focuses on integer arithmetic, presenting algorithms for performing addition, subtraction, multiplication, and division. These algorithms apply to machines that have hardware instructions and those capable of only shifts, additions, and subtractions.

Real numbers (those with fractional extension) are often expressed in floating point, but fixed point can also be used. Chapter 3 explores some of the qualities of real numbers and explains how the radix point affects the four basic arithmetic functions. Because the subject of fractions is covered, several rounding techniques are also examined. Some interesting techniques for performing division, one using multiplication and the other inversion, are also presented. These routines are interesting because they involve division with very long operands as well as from a purely conceptual viewpoint. At the end of the chapter, there is an example of an algorithm that will draw a circle in a two dimensional space, such as a graphics monitor, using only shifts, additions and subtractions.

Chapter 4 covers the basics of floating-point arithmetic and shows how scaling is done. The four basic arithmetic functions are developed into floating-point routines using the fixed point methods given in earlier chapters.

Chapter 5 discusses input and output routines for numerics. These routines deal with radix conversion, such as decimal to binary, and format conversions, such as ASCII to floating point. The conversion methods presented use both computational and table-driven techniques.

Finally, the elementary functions are discussed in Chapter 6. These include table-driven techniques for fast lookup and routines that rely on the fundamental binary nature of the machine to compute fast logarithms and powers. The CORDIC functions which deliver very high-quality transcendentals with only a few shifts and additions, are covered, as are the Taylor expansions and Horner's Rule. The chapter ends with an implementation of a floating-point sine/cosine algorithm based upon a minimax approximation and a floating-point square root.

Following the chapters, the appendices comprise additional information and reference materials. Appendix A presents and explains the pseudo-random number generator developed to test many of the routines in the book and includes SPECTRAL.C, a C program useful in testing the functions described in this book. This program was originally created for the pseudo-random number generator and incorporates a visual check and Chi-square statistical test on the function. Appendix B offers a small set of constants commonly used.

The source code for all the arithmetic functions, along with many ancillary routines and examples, is in appendices C through F.

Integer and fixed-point routines are in Appendix C. Here are the classical routines for multiplication and division, handling signs, along with some of the more complex fixed-point operations, such as the Newton Raphson iteration and linear interpolation for division.

Appendix D consists of the basic floating-point routines for addition, subtraction, multiplication, and division, Floor, ceiling, and absolute value functions are included here, as well as many other functions important to the more advanced math in Chapter 6.

The conversion routines are in Appendix E. These cover the format and numerical conversions in Chapter 5

In Appendix F, there are two source files. TRANS.ASM contains the elementary functions described in Chapter 6, and TABLE.ASM that holds the tables, equates and constants used in TRANS.ASM and many of the other modules.

MATH.C in Appendix F is a C program useful in testing the functions described in this book. It is a simple shell with the defines and prototypes necessary to perform tests on the routines in the various modules.

Because processors and microcontrollers differ in architecture and instruction set, algorithmic solutions to numeric problems are provided throughout the book for machines with no hardware primitives for multiplication and division as well as for those that have such primitives.

Assembly language by nature isn't very portable, but the ideas involved in numeric processing are. For that reason, each algorithm includes an explanation that enables you to understand the ideas independently of the code. This explanation is complemented by step-by-step pseudocode and at least one example in 8086 assembler. All the routines in this book are also available on a disk along with a simple C shell that can be used to exercise them. This allows you to walk through the routines to see how they work and test any changes you might make to them. Once you understand how the routine works, you can port it to another processor. The routines as presented in the book are formatted differently from the same routines on the disk. This is done to accommodate the page size. Any last minute changes to the source code are documented in the Readme file on the disk.

There is no single solution for all applications; there may not even be a single solution for a particular application. The final decision is always left to the individual programmer, whose skills and knowledge of the application are what make the software work. I hope this book is of some help.

CHAPTER 1

Numbers

Numbers are pervasive; we use them in almost everything we do, from counting the feet in a line of poetry to determining the component frequencies in the periods of earthquakes. Religions and philosophies even use them to predict the future. The wonderful abstraction of numbers makes them useful in any situation. Actually, what we find so useful aren't the numbers themselves (numbers being merely a representation), but the concept of ***numeration:*** counting, ordering, and grouping.

Our numbering system has humble beginnings, arising from the need to quantify objects-five horses, ten people, two goats, and so on-the sort of calculations that can be done with strokes of a sharp stone or root on another stone. These are ***natural*** numbers-positive, whole numbers, each defined as having one and only one immediate predecessor. These numbers make up the ***number ray,*** which stretches from zero to infinity (see Figure 1- 1).

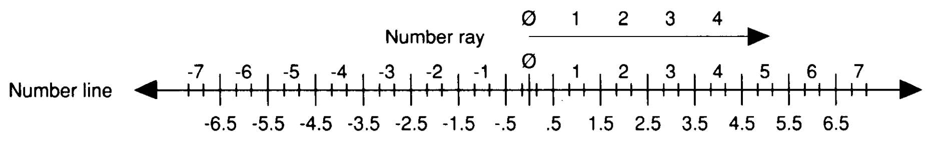


Figure 1-1. The number line.

The calculations performed with natural numbers consist primarily of addition and subtraction, though natural numbers can also be used for multiplication (iterative addition) and, to some degree, for division. Natural numbers don't always suffice, however; how can you divide three by two and get a natural number as the result? What happens when you subtract 5 from 3? Without decimal fractions, the results of many divisions have to remain symbolic. The expression "5 from 3" meant nothing until the Hindus created a symbol to show that money was owed. The words *positive* and *negative* are derived from the Hindu words for credit and debit'.

The number ray-all natural numbers-became part of a much greater schema known as the ***number line,*** which comprises all numbers (positive, negative, and fractional) and stretches from a negative infinity through zero to a positive infinity with infinite resolution*. Numbers on this line can be positive or negative so that 3-5 can exist as a representable value, and the line can be divided into smaller and smaller parts, no part so small that it cannot be subdivided. This number line extends the idea of numbers considerably, creating a continuous weave of ever-smaller pieces (you would need something like this to describe a universe) that finally give meaning to calculations such as 3/2 in the form of real numbers (those with decimal fractional extensions).

This is undeniably a valuable and useful concept, but it doesn't translate so cleanly into the mechanics of a machine made of finite pieces.

Systems of Representation

The Romans used an ***additional system*** of representation, in which the symbols are added or subtracted from one another based on their position. Nine becomes *IX* in Roman numerals (a single count is subtracted from the group of 10, equaling nine; if the stroke were on the other side of the symbol for 10, the number would be 11). This meant that when the representation reached a new power of 10 or just became too large, larger numbers could be created by concatenating symbols. The problem here is that each time the numbers got larger, new symbols had to be invented.

Another form, known as ***positional representation,*** dates back to the Babylonians, who used a sort of floating point with a base of 60.[3] With this system, each successively larger member of a group has a different symbol. These symbols are then arranged serially to grow more significant as they progress to the left. The position of the symbol within this representation determines its value. This makes for a very compact system that can be used to approximate any value without the need to invent new symbols. Positional numbering systems also allow another freedom: Numbers can be regrouped into coefficients and powers, as with polynomials, for some alternate approaches to multiplication and division, as you will see in the following chapters.

If ***b*** is our base and ***a*** an integer within that base, any positive integer may be represented as:

$$A = \sum_{i=0}^{n-1} a_i b^i$$

or as:

$$a_i * b^i + a_{i-1} * b^{i-1} + ... + a_0 * b_0$$

As you can see, the value of each position is an integer multiplied by the base taken to the power of that integer relative to the origin or zero. In base 10, that polynomial looks like this:

$$a_i * 10^i + a_{i-1} * 10^{i-1} + ... + a_0 * 10^0$$

and the value 329 takes the form:

$$3 * 10 + 2 * 10 + * 10$$

Of course, since the number line goes negative, so must our polynomial:

$$a_i * b_i + a_{i-1} * b^{i-1} + ... + a_0 * b^0 + a_{-1} * b^{-1} + a_{-2} * b^{-2} + ... + a_{-i} * {}^{b-i}$$

Bases

Children, and often adults, count by simply making a mark on a piece of paper for each item in the set they're quantifying. There are obvious limits to the numbers that can be conveniently represented this way, but the solution is simple: When the numbers get too big to store easily as strokes, place them in groups of equal size and count only the groups and those that are left over. This makes counting easier because we are no longer concerned with individual strokes but with groups of strokes and then groups of groups of strokes. Clearly, we must make the size of each group greater than one or we are still counting strokes. This is the concept of ***base.*** ***(See*** Figure l-2.) If we choose to group in l0s, we are adopting 10 as our base. In base 10, they are gathered in groups of 10; each position can have between zero and nine things in it. In base 2, each position can have either a one or a zero. Base 8 is zero through seven. Base 16 uses zero through nine and *a* through *f.* Throughout this book, unless the base is stated in the text, a *B* appended to the number indicates base 2, an *O* indicates base 8, a ***D*** indicates base 10, and an ***H*** indicates base 16.

Regardless of the base in which you are working, each successive position to the left is a positive increase in the power of the position.

In base 2, 999 looks like:

```
lllll00lllB
```

If we add a subscript to note the position, it becomes:

$$1_9 1_8 1_7 1_6 1_5 0_4 0_3 1_2 1_1 1_0$$

This has the value:

$$1*2^9 \ +1*2^8 \ +1*2^7 \ +1*2^6 \ +1*2^5 \ +1*2^4 \ +1*2^3 \ +1*2^2 \ +1*2^1 \ +1*2^0$$

which is the same as:

```
1*512 + 1*256 + 1*128 + 1*64 + 1*32 + 0*16 + 0*8 + 1*4 + 1*2 + 1*1
```

Multiplying it out, we get:

```
512 + 256 + 128 + 64 + 32 + 0 + 0 + 4 + 2 + 1 = 999
```

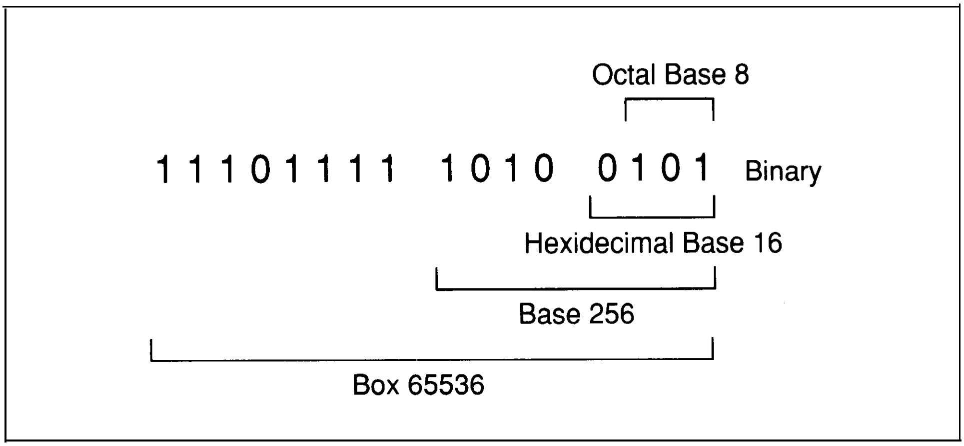


Figure 1-2. The base of a number system defines the number of unique digits available in each position.

Octal, as the name implies, is based on the count of eight. The number 999 is 1747 in octal representation, which is the same as writing:

$1*8^3 + 7*8^2 + 4*8^1 + 7*8^0$

or

```
1*512 + 7*64 + 4*8 + 7*1
```

When we work with bases larger than 10, the convention is to use the letters of the alphabet to represent values equal to or greater than 10. In base 16 (hexadecimal), therefore, the set of numbers is 0 1 2 3 4 5 6 7 8 9 a b c d e f, where *a = 10* and *f* = 15. If you wanted to represent the decimal number 999 in hexadecimal, it would be 3e7H, which in decimal becomes:

$3*16^2 + 14*16^1 + 7*16^0$

Multiplying it out gives us:

```
3*256 + 14*16 + 7*1
```

Obviously, a larger base requires fewer digits to represent the same value.

Any number greater than one can be used as a base. It could be base 2, base 10, or the number of bits in the data type you are working with. Base 60, which is used for timekeeping and trigonometry, is attractive because numbers such as l/3 can be expressed exactly. Bases 16, 8, and 2 are used everywhere in computing machines, along with base 10. And one contingent believes that base 12 best meets our mathematical needs.

The Radix Point, Fixed and Floating

Since the physical world cannot be described in simple whole numbers, we need a way to express fractions. If all we wish to do is represent the truth, a symbol will do. A number such as 2/3 in all its simplicity is a symbol-a perfect symbol, because it can represent something unrepresentable in decimal notation. That number translated to decimal fractional representation is ***irrational;*** that is, it becomes an endless series of digits that can only approximate the original. The only way to express an irrational number in finite terms is to truncate it, with a corresponding loss of accuracy and precision from the actual value.

Given enough storage any number, no matter how large, can be expressed as ones and zeros. The bigger the number, the more bits we need. Fractions present a similar but not identical barrier. When we're building an integer we start with unity, the smallest possible building block we have, and add progressively greater powers (and multiples thereof) of whatever base we're in until that number is represented. We represent it to the least significant bit (LSB), its smallest part.

The same isn't true of fractions. Here, we're starting at the other end of the spectrum; we must express a value by adding successively smaller parts. The trouble is, we don't always have access to the smallest part. Depending on the amount of storage available, we may be nowhere near the smallest part and have, instead of a complete representation of a number, only an approximation. Many common values can never be represented exactly in binary arithmetic. The decimal 0.1 or one 10th, for example, becomes an infinite series of ones and zeros in binary (1100110011001100 ... B). The difficulties in expressing fractional parts completely can lead to unacceptable errors in the result if you're not careful.

The radix point (the point of origin for the base, like the decimal point) exists on the number line at zero and separates whole numbers from fractional numbers. As we move through the positions to the left of the radix point, according to the rules of positional notation, we pass through successively greater positive powers of that base; as we move to the right, we pass through successively greater negative powers of the base.

In the decimal system, the number 999.999 in positional notation is

$9_2 9_1 9_0 . 9_{-1} 9_{-2} 9_{-3}$

And we know that base 10

$$10^2 = 100$$
$$10^1 = 10$$
$$10^0 = 1$$

It is also true that

$$10^{-1} = .1$$
$$10^{-2} = .01$$
$$10^{-3} = .001$$

We can rewrite the number as a polynomial

$9*10^2 + 9*10^1 + 9*10^0 + 9*10^{-1} + 9*10^{-2} + 9*10^{-3}$

Multiplying it out, we get

900 +90 + 9 + .9 + .09 + .009

which equals exactly 999.999.

Suppose we wish to express the same value in base 2. According to the previous example, 999 is represented in binary as 1111100111B. To represent 999.999, we need to know the negative powers of two as well. The first few are as follows:

$$
\begin{aligned}
2^{-1} &= .5D \\
2^{-2} &= .25D \\
2^{-3} &= .125D \\
2^{-4} &= .0625D \\
2^{-5} &= .03125D \\
2^{-6} &= .015625D \\
2^{-7} &= .0078125D \\
2^{-8} &= .00390625D \\
2^{-9} &= .001953125D \\
2^{-10} &= .0009765625D \\
2^{-11} &= .00048828125D \\
2^{-12} &= .000244140625D
\end{aligned}
$$

Twelve binary digits are more than enough to approximate the decimal fraction .999. Ten digits produce

$$1111100111.1111111111 = 999.9990234375$$

which is accurate to three decimal places.

Representing 999.999 in other bases results in similar problems. In base 5, the decimal number 999.999 is noted

$$
\begin{gathered}
12444.4444141414 = \\
1*5^4 + 2*5^3 + 4*5^2 + 4*5^1 + 4*5^0. + 4*5^{-1} + 4*5^{-2} + 4*5^{-3} + 4*5^{-4} + 1*5^{-5} + \\
4*5^{-6} + 1*5^{-7} + 4*5^{-8} + 1*5^{-9} + 4*5^{-10} = \\
1*625 + 2*125 + 4*25 + 4*5 + 4+ 4*.2 + 4*.04 + 4*.008 + 4*.0016 \\
+ 1*.00032 + 4*.000065 + 1*.0000125 + 4*.00000256 \\
+ 1*.000000512 + 4*.0000001024
\end{gathered}
$$

or

$$
\begin{gathered}
625+ +250 + 100 + 20 + 4 + .8 + .16 + .032 + .0064 + .00032 + .000256 + \\
.0000128 + .00001024 + .000000512 + .00004096 = \\
999.9990045696
\end{gathered}
$$

But in base 20, which is a multiple of 10 and two, the expression is rational. (Note that digits in bases that exceed 10 are usually denoted by alphabetical characters; for example, the digits of base 20 would be 0 l 2 3 4 5 6 7 8 9 A B C D E F G H I J .)

```
29J.JJC
2x20^2 + 9x20^1 + 19x20^0. + 19x20^-1 + 19x20^-2 + 12x20^-3 =
2x400 +  9x20 + 19x1. + 19x.05 + 19x.0025 + 12x.000125
```

or

```
800 + 180 + 19. + .95 + .0475 + .0015 =
999.999
```

As you can see, it isn't always easy to approximate a fraction. Fractions are a sum of the value of each position in the data type. A rational fraction is one whose sum precisely matches the value you are trying to approximate. Unfortunately, the exact combination of parts necessary to represent a fraction ***exactly*** may not be available within the data type you choose. In cases such as these, you must settle for the accuracy obtainable within the precision of the data type you are using.

Types of Arithmetic

This book covers three basic types of arithmetic: fixed point (including integer-only arithmetic and modular) and floating point.

Fixed Point

Fixed-point implies that the radix point is in a fixed place within the representation. When we're working exclusively with integers, the radix point is always to the right of the rightmost digit or bit. When the radix point is to the left of the leftmost digit, we're dealing with fractional arithmetic. The radix point can rest anywhere within the number without changing the mechanics of the operation. In fact, using fixed-point arithmetic in place of floating point, where possible, can speed up any arithmetic operation. Everything we have covered thus far applies to fixed-point arithmetic and its representation.

Though fixed-point arithmetic can result in the shortest, fastest programs, it shouldn't be used in all cases. The larger or smaller a number gets, the more storage is required to represent it. There are alternatives; modular arithmetic, for example, can, with an increase in complexity, preserve much of an operation's speed.

Modular arithmetic is what people use every day to tell time or to determine the day of the week at some future point. Time is calculated either modulo 12 or 24—that is, if it is 9:00 and six hours pass on a 12-hour clock, it is now 3:00, not 15:00:

$$9 + 6 = 3$$

This is true if all multiples of 12 are removed. In proper modular notation, this would be written:

$$9 + 6 \cong 3, \ \text{mod } 12.$$

In this equation, the sign $\cong$ means *congruence*. In this way, we can make large numbers congruent to smaller numbers by removing multiples of another number (in the case of time, 12 or 24). These multiples are often removed by subtraction or division, with the smaller number actually being the remainder.

If all operands in an arithmetic operation are divided by the same value, the result of the operation is unaffected. This means that, with some care, arithmetic operations performed on the remainders can have the same result as those performed on the whole number. Sines and cosines are calculated mod 360 degrees (or mod 2π radians). Actually, the input argument is usually taken mod $\pi/2$ or 90 degrees, depending on whether you are using degrees or radians. Along with some method for determining which quadrant the angle is in, the result is computed from the congruence (see Chapter 6).

Random number generators based on the Linear Congruential Method use modular arithmetic to develop the output number as one of the final steps.[4] Assembly-language programmers can facilitate their work by choosing a modulus that's as large as the word size of the machine they are working on. It is then a simple matter to calculate the congruence, keeping those lower bits that will fit within the word size of the computer. For example, assume we have a hexadecimal doubleword:

```
12345678H
```

and the word size of our machine is 16 bits

```
12345678H = 5678 mod 10000H
```

For more information on random number generators, see Appendix A.

One final and valuable use for modular arithmetic is in the construction of self-maintaining buffers and arrays. If a buffer containing 256 bytes is page aligned-the last eight bits of the starting address are zero-and an 8-bit variable is declared to count the number of entries, a pointer can be incremented through the buffer simply by adding one to the counting variable, then adding that to the address of the base of the buffer. When the pointer reaches 255, it will indicate the last byte in the buffer; when it is incremented one more time, it will wrap to zero and point once again at the initial byte in the buffer.

Floating Point

Floating point is a way of coding fixed-point numbers in which the number of significant digits is constant per type but whose range is enormously increased because an exponent and sign are embedded in the number. Floating-point arithmetic is certainly no more accurate than fixed point-and it has a number of problems, including those present in fixed point as well as some of its own-but it is convenient and, used judiciously, will produce valid results.

The floating-point representations used most commonly today conform, to some degree, to the IEEE 754 and 854 specifications. The two main forms, the *long real* and the *short real,* differ in the range and amount of storage they require. Under the IEEE specifications, a long real is an 8-byte entity consisting of a sign bit, an 11-bit exponent, and a 53-bit significand, which mean the significant bits of the floating-point number, including the fraction to the right of the radix point and the leading one to the left. A short real is a 4-byte entity consisting of a sign bit, an 8-bit exponent, and a 24-bit significand.

To form a binary floating-point number, shift the value to the left (multiply by two) or to the right (divide by two) until the result is between 1.0 and 2.0. Concatenate the sign, the number of shifts (exponent), and the mantissa to form the float.

Doing calculations in floating point is very convenient. A short real can express a value in the range 10^{38} to 10^{-38} in a doubleword, while a long real can handle values ranging from 10^{308} to 10^{-308} in a quadword. And most of the work of maintaining the numbers is done by your floating-point package or library.

As noted earlier, some problems in the system of precision and exponentiation result in a representation that is not truly "real"—namely, gaps in the number line and loss of significance. Another problem is that each developer of numerical software adheres to the standards in his or her own fashion, which means that an equation that produced one result on one machine may not produce the same result on another machine or the same machine running a different software package. This compatibility problem has been partially alleviated by the widespread use of coprocessors.

Positive and Negative Numbers

The most common methods of representing positive and negative numbers in a positional number system are sign magnitude, diminished-radix complement, and radix complement (see Table 1- 1).

With the sign-magnitude method, the most significant bit (MSB) is used to indicate the sign of the number: zero for plus and one for minus. The number itself is represented as usual—that is, the only difference between a positive and a negative representation is the sign bit. For example, the positive value 4 might be expressed as 0l00B in a 4-bit binary format using sign magnitude, while -4 would be represented as 1100B.

This form of notation has two possible drawbacks. The first is something it has in common with the diminished-radix complement method: It yields two forms of zero, 0000B and 1000B (assuming three bits for the number and one for the sign). Second, adding sign-magnitude values with opposite signs requires that the magnitudes of the numbers be consulted to determine the sign of the result. An example of sign magnitude can be found in the IEEE 754 specification for floating-point representation.

The diminished-radix complement is also known as the ***one's complement*** in binary notation. The MSB contains the sign bit, as with sign magnitude, while the rest of the number is either the absolute value of the number or its bit-by-bit complement. The decimal number 4 would appear as 0100 and -4 as 1011. As in the foregoing method, two forms of zero would result: 0000 and 1111.

The radix complement, or *two's complement,* is the most widely used notation in microprocessor arithmetic. It involves using the MSB to denote the sign, as in the other two methods, with zero indicating a positive value and one meaning negative. You derive it simply by adding one to the one's-complement representation of the same negative value. Using this method, 4 is still 0100, but -4 becomes 1100. Recall that one's complement is a bit-by-bit complement, so that all ones become zeros and all zeros become ones. The two's complement is obtained by adding a one to the one's complement.

This method eliminates the dual representation of zero-zero is only 0000 (represented as a three-bit signed binary number)-but one quirk is that the range of values that can be represented is slightly more negative than positive (see the chart below). That is not the case with the other two methods described. For example, the largest positive value that can be represented as a signed 4-bit number is 0111B, or 7D, while the largest negative number is 1000B, or -8D.

	One's complement	Two's complement	Sign complement
0111	7	7	7
0110	6	6	6
0101	5	5	5
0100	4	4	4
0011	3	3	3
0010	2	2	2
0001	1	1	1
0000	0	0	0
1111	-0	-1	-7
1110	-1	-2	-6
1101	-2	-3	-5
1100	-3	-4	-4
1011	-4	-5	-3
1010	-5	-6	-2
1001	-6	-7	-1
1000	-7	-8	-0

Table 1-1. Signed Numbers.

Decimal integers require more storage and are far more complicated to work with than binary; however, numeric I/O commonly occurs in decimal, a more familiar notation than binary. For the three forms of signed representation already discussed, positive values are represented much the same as in binary (the leftmost bit being zero). In sign-magnitude representation, however, the sign ***digit*** is nine followed by the absolute value of the number. For nine's complement, the sign digit is nine and the value of the number is in nine's complement. As you might expect, 10's complement is the same as nine's complement except that a one is added to the low-order (rightmost) digit.

Fundamental Arithmetic Principles

So far we've covered the basics of positional notation and bases. While this book is not about mathematics but about the implementation of basic arithmetic operations on a computer, we should take a brief look at those operations.

1. *Addition* is defined as $a + b = c$ and obeys the commutative rules described below.
2. *Subtraction* is the inverse of addition and is defined as $b = c - a$.
3. *Multiplication* is defined as $ab = c$ and conforms to the commutative, associative, and distributive rules described below.
4. *Division* is the inverse of multiplication and is shown by $b = c/a$.
5. A *power* is $ba=c$.
6. A *root* is $b = {}^{a}\sqrt{c}$
7. A *logarithm* is $a = log_b c$.

Addition and subtraction must also satisfy the following rules:[5]

8. *Commutative:*

   a + b = b + a

   ab = ba

9. *Associative:*

   a = (b + c) = (a + b) + c

   a(bc) = (ab)c

10. *Distributive:*

    a(b + c) = ab + ac

From these rules, we can derive the following relations:[6]

11. $(ab)^c = a^c b^c$

12. $a^b a^c = ac^{(b+c)}$
13. $(a^b)^c = a^{(bc)}$
14. $a + 0 = a$
15. $a \times 1 = a$
16. $a^1 = a$
17. a/0 is undefined

These agreements form the basis for the arithmetic we will be using in upcoming chapters.

Microprocessors

The key to an application's success is the person who writes it. This statement is no less true for arithmetic. But it's also true that the functionality and power of the underlying hardware can greatly affect the software development process.

Table l-2 is a short list of processors and microcontrollers currently in use, along with some issues relevant to writing arithmetic code for them (such as the instruction set, and bus width). Although any one of these devices, with some ingenuity and effort, can be pushed through most common math functions, some are more capable than others. These processors are only a sample of what is available. In the rest of this text, we'll be dealing primarily with 8086 code because of its broad familiarity. Examples from other processors on the list will be included where appropriate.

Before we discuss the devices themselves, perhaps an explanation of the categories would be helpful.

Buswidth

The wider bus generally results in a processor with a wider bandwidth because it can access more data and instruction elements. Many popular microprocessors have a wider internal bus than external, which puts a burden on the cache (storage internal to the microprocessor where data and code are kept before execution) to keep up with the processing. The 8088 is an example of this in operation, but improvements in the 80x86 family (including larger cache sizes and pipelining to allow some parallel processing) have helped alleviate the problem.

	Add	Add with Carry	Subtract	Subtract with Carry	Divide	Divide (signed)	Modular	Multiply	Multiply (signed)	1's Complement	2's Complememt	Sign Extension	Rotate	Rotate through Carry	Arithmetic Shift	Normalization	Decimal Adjust (add)	Decimal Adjust (subtract)	ASCII Adjust	Basic Data Type: 8-bit	16-bit	32-bit	64-bit	Compare	Combined Jump	Flags: Zero	Carry	Sign	Overflow	Auxilary Carry	Sticky Bit	Overflow Trap
8048	✓	✓								✓			✓	✓			✓			✓							✓			✓		
8051	✓	✓		✓	✓			✓		✓			✓	✓			✓			✓					✓	✓*	✓		✓	✓		
Z80	✓	✓	✓	✓						✓	✓		✓	✓	✓		✓	✓		✓					✓	✓	✓	✓	✓	✓		
8086	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓		✓	✓		✓	✓			✓	✓	✓	✓	✓	✓	✓		
80386	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓					✓	✓		✓	✓	✓	✓	✓	✓							
80C196	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓		✓	✓				✓	✓	✓		✓	✓	✓	✓	✓	✓		✓	✓
TMS34010	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓				✓	✓	✓	✓	✓	✓	✓	✓	✓	✓			

*Note: Actually means that the accumulator is = zero

Table 1-2. Instructions and flags.

Data type

The larger the word size of your machine, the larger the numbers you can process with single instructions. Adding two doubleword operands on an 8051 is a multiprecision operation requiring several steps. It can be done with a single ADD on a TMS34010 or 80386. In division, the word size often dictates the maximum size of the quotient. A larger word size allows for larger quotients and dividends.

Flags

The effects of a processor's operation on the flags can sometimes be subtle. The following comments are generally true, but it is always wise to study the data sheets closely for specific cases.

- *Zero.* This flag is set to indicate that an operation has resulted in zero. This can occur when two operands compare the same or when two equal values are subtracted from one another. Simple move instructions generally do not affect the state of the flag.

- *Carry.* Whether this flag is set or reset after a certain operation varies from processor to processor. On the 8086, the carry will be set if an addition overflows or a subtraction underflows. On the 80C196, the carry will be set if that addition overflows but cleared if the subtraction underflows. Be careful with this one. Logical instructions will usually reset the flag and arithmetic instructions as well as those that use arithmetic elements (such as *compare)* will set it or reset it based on the results.

- *Sign.* Sometimes known as the *negative* flag, it is set if the MSB of the data type is set following an operation.

- *Overflow.* If the result of an arithmetic operation exceeds the data type meant to contain it, an overflow has occurred. This flag usually only works predictably with addition and subtraction. The overflow flag is used to indicate that the result of a signed arithmetic operation is too large for the destination operand. It will be set if, after two numbers of like sign are added or subtracted, the sign of the result changes or the carry into the MSB of an operand and the carry out don't match.

- *Overflow Trap.* If an overflow occurred at any time during an arithmetic operation, the overflow trap will be set if not already set. This flag bit must be cleared explicitly. It allows you to check the validity of a series of operations.
- *Auxiliary Carry.* The decimal-adjust instructions use this flag to correct the accumulator after a *decimal* addition or subtraction. This flag allows the processor to perform a limited amount of decimal arithmetic.
- *Parity.* The parity flag is set or reset according to the number of bits in the lower byte of the destination register after an operation. It is set if even and reset if odd.
- *Sticky Bit.* This useful flag can obviate the need for guard digits on certain arithmetic operations. Among the processors in Table l-2, it is found only on the 80C196. It is set if, during a multiple right shift, more than one bit was shifted into the carry with a one in the carry at the end of the shift.

Rounding and the Sticky Bit

A multiple shift to the right is a divide by some power of two. If the carry is set, the result is equal to the integer result plus l/2, but should we round up or down? This problem is encountered frequently in integer arithmetic and floating point. Most floating-point routines have some form of extended precision so that rounding can be performed. This requires storage, which usually defaults to some minimal data type (the routines in Chapter 4 use a word). The sticky bit reduces the need for such extended precision. It indicates that during a right shift, a one was shifted into the carry flag and then shifted out.

Along with the carry flag, the sticky bit can be used for rounding. For example, suppose we wish to divide the hex value 99H by 16D. We can do this easily with a four-bit right shift. Before the shift, we have:

```
Operand        Carry flag        Sticky bit
10011001          0                  0
```

We shift the operand right four times with the following instruction:

```
shr       ax, #4
```

During the shift, the Least Significant Bit (LSB) of the operand (a one) is shifted into the carry and then out again, setting the sticky bit followed by two zeros and a final one. The operand now has the following form:

```
Operand      Carry flag              Sticky bit
00001001     1 (from the last shift) 1 (because of the first one
                                     shifted in and out of the carry)
```

To round the result, check the carry flag. If it's clear, the bits shifted out were less than half of the LSB, and rounding can be done by truncation. If the carry is set, the bits shifted out were at least half of the LSB. Now, with the sticky bit, we can see if any other bits shifted out during the divide were ones; if so, the sticky bit is set and we can round up.

Rounding doesn't have to be done as described here, but however you do it the sticky bit can make your work easier. Too bad it's not available on more machines.

Branching

Your ability to do combined jumps depends on the flags. All the processors listed in the table have the ability to branch, but some implement that ability on more sophisticated relationships. Instead of a simple "jump on carry," you might find "jump if greater," "jump if less than or equal," and signed and unsigned operations. These extra instructions can cut the size and complexity of your programs.

Of the devices listed, the TMS34010, 80x86 and 80C196 have the richest set of branching instructions. These include branches on signed and unsigned comparisons as well as branches on the state of the flags alone.

Instructions

Addition

- *Add.* Of course, to perform any useful arithmetic, the processor must be capable of some form of addition. This instruction adds two operands, signaling any overflow from the result by setting the carry.

- *Add-with-Carry.* The ability to add with a carry bit allows streamlined, multiprecision additions. In multibyte or multiword additions, the *add* instruction is usually used first; the *add-with-carry* instruction is used in each succeeding addition. In this way, overflows from each one addition can ripple through to the next.

Subtraction

- *Subtract.* All the devices in Table l-2 can subtract except the 8048 and 8051. The 8051 uses the *subtract-with-carry* instruction to fill this need. On the 8048, to subtract one quantity (the *subtrahend)* from another (the *minuend),* you must complement the subtrahend and increment it, then add it to the minuend-in other words, add the two's complement to the minuend.
- *Subtract-with-Carry.* Again, the 8048 does not support this instruction, while all the others do. Since the 8051 has *only* the *subtract-with-carry* instruction, it is important to see that the carry is clear before a subtraction is performed unless it is a multiprecision operation. The *subtract-with-carry* is used in multiprecision subtraction in the same manner as the *add-with-carry* is used in addition.
- *Compare.* This instruction is useful for boundary, magnitude and equality checks. Most implementations perform a comparison by subtracting one value from another. This process affects neither operand, but sets the appropriate flags. Many microprocessors allow either signed or unsigned comparisons.

Multiplication

- *Multiply*. This instruction performs a standard unsigned multiply based on the word size of the particular microprocessor or microcontroller. Hardware can make life easier. On the 8088 and 8086, this instruction was embarrassingly slow and not that much of a challenge to shift and add routines. On later members of the 80x86 family, it takes a fraction of the number of cycles to perform, making it very useful for multiprecision and single-precision work.
- *Signed Multiply.* The signed multiply, like the signed divide (which we'll

discuss in a moment), has limited use. It produces a signed product from two signed operands on all data types up to and including the word size of the machine. This is fine for tame applications, but makes the instruction unsuitable for multiprecision work. Except for special jobs, it might be wise to employ a generic routine for handling signed arithmetic. One is described in the next chapter.

Division

- *Divide.* A hardware divide simplifies much of the programmer's work unless it is very, very slow (as it is on the 8088 and 8086). A multiply canextend the useful range of the divide considerably. The following chapter gives examples of how to do this.
- *Signed Divide.* Except in specialized and controlled circumstances, the signed divide may not be of much benefit. It is often easier and more expeditious to handle signed arithmetic yourself, as will be shown in Chapter 2.
- *Modulus.* This handy instruction returns the *remainder* from the division of two operands in the destination register. As the name implies, this instruction is very useful when doing modular arithmetic. This and *signed modulus* are available on the TMS34010.
- *Signed Modulus*. This is the signed version of the earlier *modulus* instruction, here the remainder bears the sign of the dividend.

Negation and Signs

- *One's Complement.* The one's complement is useful for logical operations and diminished radix arithmetic (see Positive and Negative Numbers, earlier in this chapter). This instruction performs a bit-by-bit complement of the input argument; that is, it makes each one a zero and each zero a one.
- *Two's Complement.* The argument is one's complemented, then incremented by

one. This is how negative numbers are usually handled on microcomputers.

- *Sign Extension.* This instruction repeats the value of the MSB of the byte or word through the next byte, word, or doubleword so the proper results can be obtained from an arithmetic operation. This is useful for converting a small data type to a larger data type of the same sign for such operations as multiplication and division.

Shifts, Rotates and Normalization

- *Rotate.* This simple operation can occur to the right or the left. In the case of a ROTATE to the right, each bit of the data type is shifted to the right; the LSB is deposited in the carry, while a zero is shifted in on the left. If the rotate is to the left, each bit is moved to occupy the position of the next higher bit in the data type until the last bit is shifted out into the carry flag (see figure l-3). On the Z80, some shifts put the same bit into the carry and the LSB of the byte you are shifting. Rotation is useful for multiplies and divides as well as normalization.
- *Rotate-through-Carry.* This operation is similar to the ROTATE above, except that the carry is shifted into the LSB (in the case of a left shift), or the MSB (in the case of a right shift). Like the ROTATE, this instruction is useful for multiplies and divides as well as normalization.
- *Arithmetic Shift.* This shift is similar to a right shift. As each bit is shifted, the value of the MSB remains the same, maintaining the value of the sign.
- *Normalization.* This can be either a single instruction, as is the case on the 80C196, or a set of instructions, as on the TMS34010. *NORML* will cause the 80C196 to shift the contents of a doubleword register to the left until the MSB is a one, “normalizing” the value and leaving the number of shifts required in a register. On the TMS34010, *LMO* leaves the number of bits required to shift a doubleword so that its MSB is one. A multibit shift can then be used to normalize it. This mechanism is often used in floating point and as a setup for binary table routines.

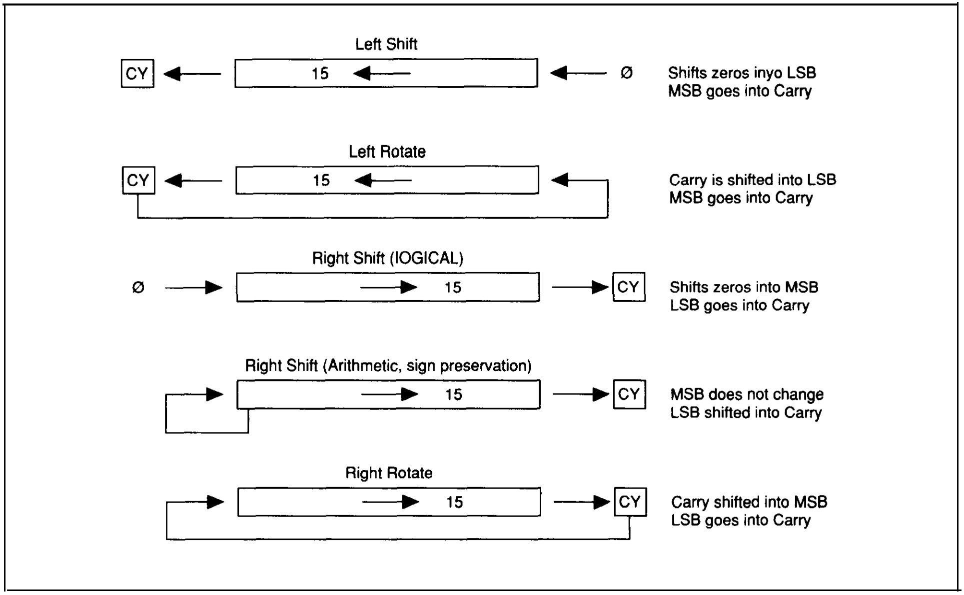


Figure 1-3. Shifts and rotates.

Decimal and ASCII Instructions

- *Decimal Adjust on Add.* This instruction adjusts the results of the addition of two decimal values to decimal. Decimal numbers cannot be added on a binary computer with guaranteed results without taking care of any intrabyte carries that occur when a digit in a position exceeds nine. On the 8086, this instruction affects only the AL register. This and the next instruction can be very useful in an embedded system that receives decimal data and must perform some simple processing before displaying or returning it.

- *Decimal Adjust on Subtract.* This instruction is similar to the preceeding one except that it applies to subtraction.

- *ASCII Adjust.* These instructions prepare either binary data for conversion to ASCII or ASCII data for conversion to binary. Though Motorola processors also implement these instructions, they are found only in the 80x86 series in our list. Used correctly, they can also do a certain amount of arithmetic.

Most of the earlier microprocessors-such as the 8080, 8085, Z80, and 8086—as well as microcontrollers like the 8051 were designed with general applications in mind. While the 8051 is billed as a Boolean processor, it's general set of instructions makes many functions possible and keeps it very popular today.

All these machines can do arithmetic at one level or another. The 8080, 8085, and Z80 are bit-oriented and don't have hardware multiplies and divides, making them somewhat slower and more difficult to use than those that do. The 8086 and 8051 have hardware multiplies and divides but are terribly slow about it. (The timings for the 8086 instructions were cleaned up considerably in subsequent generations of the 286, 386, and 486.) They added some speed to the floating-point routines and decreased code size.

Until a few years ago, the kind of progress usually seen in these machines was an increase in the size of the data types available and the addition of hardware arithmetic. The 386 and 486 can do some 64-bit arithmetic and have nice shift instructions, *SHLD* and *SHRD,* that will happily shift the bits of the second operand into the first and put the number of bits shifted in a third operand. This is done in a single stroke, with the bits of one operand shifted directly into the other, easing normalization of long integers and making for fast binary multiplies and divides. In recent years we've seen the introduction of microprocessors and microcontrollers that are specially designed to handle floating-point as well as fixed-point arithmetic. These processors have significantly enhanced real-time control applications and digital signal processing in general. One such microprocessor is the TMS34010; a microcontroller with a similar aptitude is the 80C196.

1 Kline, Morris. *Mathematics for the Nonmathematician. New York*, NY: Dover Publications, Inc., 1967, Page 72.

2 Gellert, W., S. Gottwald, M. Helwich, H. Kastner, and H. Küstner (eds.). *The VNR Concise Encyclopedia of Mathematics*. New York, NY: Van Nostrand Reinhold, 1989, Page 20.

3 Knuth, D. E. *Seminumerical Algorithms*. Reading, MA: Addison-Wesley Publishing Co., 1980, Page 180.

4 Knuth, D. E. *Seminumerical Algorithms*. Reading, MA: Addison-Wesley Publishing Co., 1981, Pages 1-127.

5 Cavanagh, Joseph J. F. *Digital Computer Arithmetic*. New York, NY: McGraw-Hill Book Co., 1984, Page 2.

6 Pearson, Carl E. (ed.) *Handbook of Applied Mathematics*. New York, NY: Van Nostrand Reinhold, 1983, Page 1.

CHAPTER 2

Integers

Reducing a problem to the integer level wherever possible is certainly one of the fastest and safest ways to solve it. But integer arithmetic is only a subset of fixed-point arithmetic. Fixed-point arithmetic means that the radix point remains in the same place during all calculations. Integer arithmetic is fixed point arithmetic with the radix point consistently to the right of the LSB. Conversely, fractional arithmetic is simply fixed point with the radix point to the left of the MSB. There are no specific requirements regarding placement of the radix point; it depends entirely on the needs of the operation. Sines and cosines may require no integer at all, while a power-series calculation may require an integer portion. You may wish to use two guard digits during multiplication and division for rounding purposes-it depends on you and the application.

To present algorithms for the four basic operations of mathematics-addition, subtraction, multiplication, and division-this chapter will concentrate on integer-only arithmetic. The operations for fixed-point and integer-only arithmetic are essentially the same; the former simply involves some attention to the placement of the radix point.

This chapter begins with the basic operations and the classic algorithms for them, followed by some more advanced algorithms. The classic algorithms aren't necessarily the fastest, but they are so elegant and reflect the character of the binary numbering system and the nature of arithmetic itself so well that they are worth knowing. Besides, on any binary machine, they'll always work.

Addition and Subtraction

Unsigned Addition and Subtraction

Simply put, addition is the joining of two sets of numbers or *quantities*, into one set. We could also say that when we add we're really incrementing one value, the *augend,* by another value, the *addend.* Subtraction is the inverse of addition, with one number being reduced, or *decremented,* by another.

For example, the addition operation

```
                 0111
+2        or     0010
 9               1001
```

might be accomplished on the 8086 with this instruction sequence:

```
mov    al,7
add    al,2
```

In positional arithmetic, each position is evaluated $0 \leq x <$ base, with x being the digit in that position, and any excess is carried up to the next position. If the base is 10, no number greater than nine can exist in any position; if an operation results in a value greater than nine, that value is divided by 10, the quotient is carried into the next position, and the remainder is left in the current position.

The same is true of subtraction except that any underflow in an operation results in a borrow from the next higher position, reducing the strength of that position by one. For example:

```
17                 1   0001
-9        or           1001
 8                     1000
```

In 8086 assembler, this would be:

```
mov    al,llh
Sub    al,9h
```

On a microprocessor, the carry and borrow use the carry flag. If adding any two unsigned numbers results in a value that cannot be contained within the data type we're using, a carry results (the carry flag is set); otherwise, it is reset. To demonstrate this, lets add two bytes, 7H and 9H:

```
     0111
    +1001
1    0000 the result
¦ the carry
```

This addition was unsigned and produced a result that was too large for the data type. In this case, the overflow was an error because the value represented in the result was not the full result. This phenomenon is useful, however, when performing multiprecision arithmetic (discussed in the next section).

Subtraction will produce a carry on an underflow (in this case, it's known as a *borrow):*

```
1     0001
     -1001
0     1000 the result
¦ the borrow
```

Processors use the carry flag to reflect both conditions; the trick is to know how they're representing the borrow. On machines such as the 8086, the carry is set for both overflow from addition and underflow from subtraction. On the 80C196, the carry is set on overflow and reset (cleared) on underflow, so it's important to know what each setting means. Besides being set or reset as the result of an arithmetic operation, the carry flag is usually reset by a logical operation and is unaffected by a move.

Because not every problem can be solved with single precision arithmetic, these flags are often used in multiprecision operations.

Multiprecision Arithmetic

Working with large numbers is much the same as working with small numbers. As you saw in the earlier examples, whenever we *ADDed* a pair of numbers the carry flag was set according to whether or not an overflow occurred. All we do to add a very large number is *ADD* the least significant component and then *ADD* each subsequent component with the carry resulting from the previous addition.

Let's say we want to add two doubleword values, 99999999H and 15324567H. The sequence looks like this:

```
mov  dx,9999h
mov  ax,9999h
add  ax,4567h
adc  dx,1532h
```

DX now contains the most significant word of the result, and AX contains the least. A 64-bit addition is done as follows.

add64: Algorithm

1. A pointer is passed to the result of the addition.
2. The least significant words of *addend0* are loaded into AX:DX.
3. The least significant words of *addend1* are added to these registers, least significant word first, using the *ADD* instruction. The next more significant word uses the *ADC* instruction.
4. The result of this addition is written to result.
5. The upper words of *addend0* are loaded into AX:DX.
6. The upper words of *addend1* are added to the upper words of *addend0* using the *ADC* instruction. (Note that the *MOV* instructions don't change the flags.)
7. The result of this addition is written to the upper words of result.

add64: Listing

```
; *****
;add64 - adds two fixed-point numbers
;the arguments are passed on the stack along with a pointer to storage for the
result
add64 proc uses ax dx es di, addendO:qword, addendl:qword, result:word
      mov         di, word ptr result
      mov         ax, word ptr addend0[0]      ; ax = low word, addend0
      mov         dx, word ptr addend0[2]      ; dx = high word, addend0
      add         ax, word ptr addendl[0]      ; add low word, addend1
      adc         dx, word ptr addendl[2]      ; add high word, addend1
      mov         word ptr [di], ax
      mov         word ptr [di][2], dx
```

```
        mov     ax, word ptr addend0[4]     ; ax = low word, addend0
        mov     dx, word ptr addend0[6]     ; dx = high word, addend0
        adc     ax, word ptr addendl[4]     ; add low word, addend1
        adc     dx, word ptr addendl[6]     ; add high word, addend1
        mov     word ptr [di][4], ax
        mov     word ptr [di] [6], dx
        ret
add64 endp
```

This example only covered 64 bits, but you can see how it might be expanded to deal with operands of any size. Although the word size and mnemonics vary from machine to machine, the concept remains the same.

You can perform multiprecision subtraction in a similar fashion. In fact, all you need to do is duplicate the code above, changing only the add-with-carry *(ADC)* instruction to subtract-with-borrow *(SBB)*. Remember, not all processors (the 8048 and 8051, for instance) have a simple subtract instruction; in case of the 8051, you must clear the carry before the first subtraction to simulate the SUB. With the 8048 you must have two's complement the subtrahend and ADD.

sub64: Algorithm

1. A pointer is passed to the result of the subtraction.
2. The least significant words of *sub0* are loaded into AX:DX.
3. The least significant words of *sub1* are subtracted fromthese registers, least significant word first, using the SUB instructions with the next most significant word using the SBB instruction.
4. The result of this subtraction is written to result.
5. The upper words of *sub0* are loaded into AX:DX
6. The upper words of *sub1* are subtracted from the upper words of *sub0* using the SBB intruction. (Note that the MOV instructions don't change the flags.)
7. The result of this subtraction is written to the upper words of result.

sub64: Listing

```
;*****
;sub64
;arguments passed on the stack, pointer returned to result
```

```
sub64 proc uses dx es di,
                sub0:qword,  sub1:qword, result:word
      mov       di, word ptr result
      mov       ax, word ptr sub0[0]          ; ax = low word, sub0
      mov       dx, word ptr sub0[2]          ; dx = high word, sub0
      sub       ax, word ptr sub1[0]          ; subtract low word, sub1
      sbb       dx, word ptr subl[2]          ; subtract high word, sub1
      mov       word ptr [di] [0],ax
      mov       word ptr [di] [2],dx
      mov       ax, word ptr sub0[4]          ; ax = low word, sub0
      mov       dx, word ptr sub0[6]          ; dx = high word, sub0
      sbb       ax, word ptr subl[4]          ; subtract low word, sub1
      sbb       dx, word ptr subl[6]          ; subtract high word, sub1
      mov       word ptr  [di][4],ax
      mov       word ptr [di][6],dx
      ret
sub64 endp
```

For examples of multiprecision addition and subtraction using other processors, see the SAMPLES. module included on the disk.

Signed Addition and Subtraction

We perform signed addition and subtraction on a microcomputer much as we perform their unsigned equivalents. The primary difference (and complication) arises from the MSB, which is the sign bit (zero for positive and one for negative). Most processors perform signed arithmetic in two's complement, the method we'll use in this discussion. The two operations of addition and subtraction are closely related; each can be performed using the logic of the other. For example, subtraction can be performed identically to addition if the subtrahend is two's-complemented before the operation. On the 8048, in fact, it must be done that way due to the absence of a subtraction instruction.

```
15 - 7 = 15 + (-7) = 8

0fH - 7H = 0fh + 0f9H = 8H
```

These operations are accomplished on a microprocessor much as we performed them in school using a pencil and paper.

One aspect of using signed arithmetic is that the range of values that can be expressed in each data type is limited. In two's-complement representation, the range is -2^{n-1} to $2^{n-1}-1$. Use signed arithmetic carefully; ordinary arithmetic processes can result in a sign reversal that invalidates the operation.

Overflow occurs in signed arithmetic when the destination data type is too small to hold the result of a signed operation-that is, a bit is carried into the MSB (the sign bit) during addition and is not propagated through to the carry, or a borrow was made from the MSB during subtraction and is not propagated through to the carry. If either event occurs, the carry flag may not be set correctly because the carry that did occur may not propagate through the sign bit into the carry flag.

Adding 60H and 50H in an eight-bit accumulator results in b0H, a negative number in signed notation even though the original operands were positive. Guard against such overflows when using signed arithmetic.

This is where the overflow flag comes in. Simply put, the overflow flag is used to indicate that the result of a signed arithmetic operation is too large or too small for the destination operand. It is set after two numbers of like sign are added or subtracted if the sign of the result changes or if the carry into the MSB of an operand and the carry out don't match.

When we added 96D (60H) and 80D (50H), we got an overflow into the sign bit but left the carry flag clear:

```
0l0l0000B (+50H)
+01100000B (+60H)
l0ll0000B (b0H, or -80D)
```

The result was a specious negative number. In this case, the overflow flag is set for two reasons: because we're adding two numbers of like sign with a subsequent change in the sign of the result and because the carry into the sign bit and the carry out don't match.

To guard against accidental overflows in addition and subtraction, test the overflow flag at the end of each operation.

Assume a 32-bit signed addition in 8086 assembler. The code might look like this:

```
signed_add:
     mov        ax, word ptr summendl       ;first load one summend
                                            ;into dx:ax
     mov        dx, word ptr summendl[2]
     add        ax, word ptr summend        ;add the two, using the carry flag
     add        dx, word ptr summend2[2]    ;to propagate any carry
     jo         bogus_result                ;out of the lower word;
                                            ;check for a valid result
good-result:
```

When writing math routines, be sure to allocate enough storage for the largest possible result; otherwise, overflows during signed operations are inevitable.

Decimal Addition and Subtraction

Four bits are needed to represent the decimal numbers zero through nine. If the microcomputer we're using has a base 10 architecture rather than one based on binary, we could increment the value 1001 (9D) and get 0 (0D) or decrement 0 and get 1001. We could then add and subtract decimal numbers on our machine and get valid results. Unfortunately, most of the processors in use are base 2, so when we increment 100l (9D) we get 1010 (0AH). This makes performing decimal arithmetic directly on a microcomputer difficult and awkward.

In *packed* binary coded decimal, a digit is stored in each nibble of a byte (as opposed to *unpacked,* in which a byte holds only one digit). Whenever addition or subtraction on packed BCD results in a digit outside the range of normal decimal arithmetic (that is, greater than nine or less than zero), a special flag known as the *auxiliary carry* is set. This indicates that an overflow or underflow has resulted during a particular operation that needs correction. This is analogous to the carry bit being set whenever an overflow occurs. On the 80x86, this flag, in association with the appropriate instruction—*DAA* for addition and *DAS* for subtraction-will produce a decimally correct result on the lower byte of the AX register. Unfortunately, these instructions only work eight bits at a time and even then in only one register, with the operands moved into and out of AL to perform a calculation of any length. As limited as this is, the instructions do allow you to perform a certain amount of decimal arithmetic on a binary machine without converting to binary.

When decimal addition is performed, each addition should be followed by a *DAA* or its equivalent. This instruction forces the CPU to add six to a BCD digit if it is outside the range, zero through nine, or if there has been a carry from the digit. It then passes the resulting carry into the next higher position. This adjusts for decimal overflows and allows normal decimal addition to be performed correctly in a packed format.

As an example, if we add 57D and 25D on a binary machine without converting to binary, we might first store the two values in registers in the following packed format:

```
A = 01010111B(57H)
B = 00100101B(25H)
```

We follow this with an *ADD* instruction (note that the carry is ignored here):

```
add      a,b
```

with the result placed in A:

```
A = 1111100B (7cH)
```

Because a decimal overflow occurred in the first nibble (1100B = 12D), the auxiliary carry flag is set. Now when the *DAA* instruction is executed, a six is added to this nibble and the carry propagated into the next higher nibble:

```
1100B
0ll0B
l00l0B
```

This leaves a two as the least significant digit with a carry into the next higher position, which is the same as adding a one to that digit:

```
0111 (7H)
0001 (1H)
1000 (8H)
```

The final result is 10000010B (82H).

This mechanism is widely implemented on both microprocessors and microcontrollers, such as the 8048, 8051, Z80, 80x86, and 80376. Unfortunately, neither the decimal adjust nor the auxiliary carry flag exists on the 80C196 or the TMS34010.

The *DAA* will work with decimal additions but not with decimal subtractions. Machines such as the Z80 and 80x86 make up for this with additional hardware to support subtraction. The Z80 uses the N and H flags along with *DAA*, while the 80x86 provides the *DAS* instruction.

The 8086 series and the 68000 series of microprocessors provide additional support for ASCII strings. On the 8086, these instructions are *AAM*, *AAS*, *AAA*, and *AAD* (see Chapter 5 for examples and greater detail). Since they do offer some arithmetic help, let's take a brief look at them now.[1]

- *AAA* adjusts the result of an addition to a simple decimal digit (a value from zero through nine). The sum must be in AL; if the result is greater than nine, AH is incremented. This instruction is used primarily for creating ASCII strings.
- *AAD* converts unpacked BCD digits in AH and AL to a binary number in AX. This instruction is also used to convert ASCII strings.
- *AAM* converts a number less than 100 in AL to an unpacked BCD number in AX, the high byte in AH, and the low byte in AL.
- *AAS*, similar to *AAA*, adjusts the result of a subtraction to a single decimal digit (a value from zero through nine).

Multiplication and Division

This group comprises what are known as "arithmetic operations of the second kind," multiplication being iterative addition and division being iterative subtraction. In the sections that follow, you'll see several algorithms for each operation, starting with the classic methods for each.

The classic algorithms, which are based on iterative addition or subtraction, may or may not be the fastest way to execute a particular operation on your target machine.

Though error checking must always be done for correct results, the errors that occur with these routines don't have the same impact on the processor state as those involving hardware instructions. What's more, these algorithms work in any binary environment because they deal with the most fundamental elements of the machine. They often provide fast, economical solutions to specialized situations that might prove awkward or slow with hardware instructions (see the *multen* routine in FXMATH.ASM). Along with the classic algorithms, there will be examples of enhancements to these routines and some algorithms that work best in silicon; nonetheless, they're based on arithmetic viewpoints that you may find interesting.

Signed vs. Unsigned

Without special handling, multiplication or division of signed numbers won't always result in correct answers, even if the operands themselves are sign-extended.

In multiplication, a problem arises in that the number of bits in the result of the multiplication is equal to, at a minimum, the number of bits in the largest operand (if neither operand is zero) and, at a maximum, the sum of bits in both operands (if each operand is equal to or greater than 2^{n-1}, where n is the size of the data type). It is usually wise to provide a result data type equal in size to the number of bits in the multiplicand plus the number of bits in the multiplier, or twice the number of bits in the largest operand. For a signed operation, this can mean the result may not have the sign required by the operands. For example, multiplying the two unsigned integers, ffH(255D) and ffH(255D), produces fe0lH(65025D), which is correct. If, however, two numbers are signed, ffH(-1D) and ffH(-1D), the correct result is 1H(1D), not fe01H(-511D). Further, an ordinary integer multiply knows nothing about sign extension, multiplying ffH(-1D) by 1H(1D) produces ffH(255D) in a 16-bit data type.

Similar problems occur in division. Unlike multiplication, the results of a divide require the difference in the number of bits in the operands. That means two 8-bit operands could require as little as one bit to represent the result of the division, or as many as eight. With division, it is wise to allot storage equal to the size of the dividend to account for any solution. With this in mind, dividing the two signed 8-bit operands, ffH(-1D) by 1H(1D), is no problem-in this case the result is ffH(-1D). But if the divisor is any larger, the result is incorrect—FFH/5H = 33H, when the correct answer is OH.

Many processors offer a signed version of their multiply and divide instructions. On the 8086, those instructions are *IMUL* and *IDIV*. To use them on single-precision operands, be sure both operands are signed and the (byte) word sizes are compatible so the result won't overflow. If you attempt to multiply a signed word operand by an unsigned word operand greater than 7fffH, your result will be in error. Be careful; this problem can go undetected for a long time.

In multiprecision multiplication, the use of *IMUL* and *IDIV* is often impractical, because the operation treats the large numbers as polynomials, breaking them apart into smaller units, or coefficients. These instructions handle all numbers as signed with 2^{n-1} significant bits, where *n* is the size of the data type. This inevitably produces an incorrect result because the instructions can only handle word operands in the range -32,768 to 32,767 and byte operands ranging from -128 to 127, with the MSB of each word or byte treated as a sign bit. Multiplying the numbers 1283H and 1234H will result in one subproduct that is out of range and an incorrect product because any of the submultiplies that involve 83H will incorrectly interpret it as a signed number.

A foolproof way to work with signed multiplies and divides, either single- or multiprecision, is to check the operands for a sign before the multiply or divide. You then handle the operation as unsigned by two's-complementing any negative operands. If necessary, the result can be two's-complemented at the end of the procedure. The algorithm is shown in pseudocode, and the code fragment is an example of how it might be implemented.

sign_operation: Algorithm

```
1. Declare and clear a byte variable, sign.
2. Check the sign of the first operand to see if it's negative.
   If not, go to step 3.
   If so, complement sign, then complement the operand.
3. Check the sign of the second operand to see if it's negative.
   If not, go to step 4.
   If so, complement sign, then complement the operand.
```

4. Perform the multiply or divide.

5. Check ign.

   If it's zero, you're done.

   If it's -1 (0ffH), two's-complement the result and go home.

signed-operation: Listing

```
;******

signed-operation  proc  operand0:dword,  operandl:dword,  result:word
        local          sign:byte
        mov            ax, word ptr operand0L21
        or             =, ax
        jns            check-second                     ;if not sign, it is positive
        not            byte ptr sign
        not            word ptr operand0[2]             ;two's complement of operand
        neg            word ptr operand0
        jc             check-second
        add            word ptr operand0[2],1

check-second:
        mov            ax, word ptr operand1[2]
        or             ax, ax
        jns            done-with-check
        not            byte ptr sign
        not            word ptr operandl[2]
        neg            word ptr operand1
        jc             done_with_check
        add            word ptr operandl[2],1
done_with_check:



;perform operation here



on_the_way_out:
        mov            al, byte ptr sign
        or             al, al
        jns            all-done
        mov            si, word ptr result
        not            word ptr si[6]
```

```
        not         word ptr si[4]
        not         word ptr si[2]
        neg         word ptr si[0]
        jc          all_done
        add         word ptr si[2], 1
        adc         word ptr si[4], 0
        adc         word ptr si[6], 0
all_done:
```

Adding this technique to one of those described below will make it a signed process.

Binary Multiplication

Multiplication in a binary system may generally be represented as the multiplication of polynomials, with the algorithm handling each bit, byte, or word as a coefficient of the power of the bits position or the least significant position within that word or byte:

$$\begin{array}{rl} & a_n * 2^n + \ldots a_1 * 2^1 + a_0 * 2^0 \\ * & b_n * 2^n + \ldots b_1 * 2^1 + b_0 * 2^0 \\ \hline & b_n *(a) * 2^n + \ldots b_1 *(a) * 2^1 + b_0 *(a) * 2^0 \end{array}$$

where n = the bit position. It is the same for bytes and words except that n is then the power of the least significant bit within the word or byte:

$$12345678H = 1234H * 16^4 + 5678H * 16^0 = 1234H * 2^{16} + 5678H * 2^0$$

In the following example involving the multiplication of two 4-bit quantities, you may recognize the pencil-and-paper method you learned in school:

Step 1:

```
            a3x23 + a2x22 + a1x21 + a0x20
   *        b3x23 + b2x22 + b1x21 + b0x20
   -------------------------------------
   b0 * a3 + b0 * a2 + b0 * a1 + b0 * a0
```

Step 2:

```
                              a3x23 + a2x22 + a1x21 + a0x20
                          *   b3x23 + b2x22 + b1x21 + b0x20
                          ---------------------------------
                    b0 * a3 + b0 * a2 + b0 * a1 + b0 * a0
          b1 * a3 t b1 * a2 + b1 * a1 + b1 * a0
```

Step 3:

```
                              a3x23 + a2x22 + a1x21 + a0x20
                            * b3x23 + b2x22 + b1x21 + b0x20
                          ---------------------------------
                    b0 * a3 + b0 * a2 + b0 * a1 + b0 * a0
            b1 * a3 + b1 * a2 + b1 * a1 + b1 * a0
    b2 * a3 + b2 * a2 + b2 * a1 + b2 * a0
```

Step 4:

```
                              a3x23 + a2x22 + a1x21 + a0x20
                            * b3x23 + b2x22 + b1x21 + b0x20
                          ---------------------------------
                    b0 * a3 + b0 * a2 + b0 * a1 + b0 * a0
            b1 * a3 + b1 * a2 + b1 * a1 + b1 * a0
    b2 * a3 + b2 * a2 + b2 * a1 + b2 * a0
b3 * a3 + b3 * a2 + b3 * a1 + b3 * a0
-----------------------------------------------------------------------------
b3 * a3 + ((b2 * a3)+ (b3 * a2))((b0 * a1)+ (b0 * a1) + (b1 * a0))+ b0 * a0
```

An example of this in a four-bit multiply could be shown as:

```
     1100=12D
*    1101=13D
---------
     1100
    0000
   1100
  1100
----------
10011100=156
```

This is also how the basic shift-and-add algorithm for microprocessors is written. This procedure is taken directly from the positional number theory, which simply states that the value of a bit or integer within a number depends on its position. Thus, each pass through the algorithm shifts both the multiplier and the multiplicand through their corresponding positions, adding the multiplicand to the result if the multiplier has a one in the *0*th position. (The right shift is arithmetic; that is, a zero is shifted into the MSB.) As with the pencil-and-paper method, the multiplicand is rotated left and the multiplier is rotated right.

To demonstrate, let's multiply two numbers, 1100 (12D) and 1101 (13D). We must first designate one as the multiplicand and the other as the multiplier and set up registers to hold them. We also need a loop counter to indicate when we have passed through all the bit positions of the multiplier. We can call this variable *cntr* (counter) and a variable to hold the product *prdct*. We'll call 1100 (the multiplicand) *mltpnd* and 1101 (the multiplier) *mltpr*. In the following example, the values in parentheses are all decimal:

0. *mltpnd* = 1100 (12)
   mltpr = 1101 (13)
   cntr = 100 (4)
   prdct = 0

Then, with each pass through the algorithm, the results are:

1. *mltpnd* = 11000 (24)
   mltpr = 0110 (6)
   cntr = 011 (3)
   prdct = 1100 (12)
2. *mltpnd* = 110000 (48)
   mltpr = 0011 (3)
   cntr = 010 (2)
   prdct = 1100 (12)
3. *mltpnd* = 1100000 (96)
   mltpr = 0001 (1)
   cntr = 1 (1)
   prdct = 111100 (60)
4. *mltpnd* = 11000000 (192)
   mltpr = 0000 (0)
   cntr = 00 (0)
   prdct = 10011100 (156)

The following routine is based on this algorithm but expects 32-bit operands.

cmul: Algorithm

1. Allocate enough space to store *multiplicand* and allow for 32 left shifts, set the variable *numbits* to 32, and see that the registers where product is formed contain zeros. (Be certain to provide enough storage for the output, at most *Product_bits* = *Multiplicand_bits* + *Multiplier_bits.* Here, 4 *Multiplicand_bits*+ 4 *Multiplier_bits* = 8 *Product bits.)*
2. Shift *multiplier* right one position and check for a carry.

   If there is not a carry, go to step 3.

   If there is, add the current value in *mltpcnd* to the product registers.
3. Shift *mltpcnd* left one position and decrement the counter variable *numbits.* Test *numbits* for zero.

   If it's zero, go to step 4.

   If not, return to step 2.
4. Write the product registers to *product* and go home.

cmul: Listing

```
; ******
; classic multiply

;

cmul proc uses bx cx dx si di, multiplicand:dword, multiplier:dword,
product:word
        local       numbits:byte, mltpcnd:qword
        pushf
        cld
        sub         ax, ax
        lea         s1, word ptr multiplicand
        lea         di, word ptr mltpcnd
        mov         cx, 2
rep     movsw
        stosw
        stosw                          ;clear upper words
        mov         bx, ax             ;clear register to be used to form product
        mov         cx, ax
                    dx, ax
                    byte ptr numbits, 32
```

```
test-multiplier:
        shr         word ptr multiplier[2], 1
        rcr         word ptr multiplier, 1
        jnc         decrement_counter
        add         ax, word ptr mltpcnd
        adc         bx, word ptr mltpcnd[2]
        adc         cx, word ptr mltpcnd[4]
        adc         dx, word ptr mltpcnd[6]
decrement_counter:
        shl         word ptr mltpcnd, 1
        rcl         word ptr mltpcnd[2], 1
        rcl         word ptr mltpcnd[4], 1
        rcl         word ptr mltpcnd[6], 1
        dec         byte ptr numbits
        jnz         test-multiplier
exit:
        mov         di, word ptr product
        mov         word ptr [di], ax
        mov         word ptr [di] [2], bx
        mov         word ptr [di][4], cx
        mov         word ptr [di][6], dx
        popf
        ret
cmul    endp
```

One possible variation of this example is to employ the "early-out" method. This technique doesn't use a counter to track the multiply but checks the multiplier for zero each time through the loop. If it's zero, you're done. For examples of early-out termination, see the routines in the section "Skipping Ones and Zeros" and others in FXMATH.ASM included on the accompanying disk.

A Faster Shift and Add

The same operation can be performed faster and in a smaller space. For one thing, the shifts being done on the multiplicand and multiplier result in unnecessary double-precision additions. Eliminating any unnecessary additions saves time and space. Arranging any shifts so that they are all in the same direction, means fewer registers or memory variables.

As you may recall, positional notation lends itself quite nicely to polynomial interpretation. Using a binary byte as an example, let's say we have two numbers, *a* and *b:*

$$a_3*2^3 + a_2*2^2 + a_1*2^1 + a_0*2^0 = a$$

and

$$b_3*2^3 + b_2*2^2 + b_1*2^1 + b_0*2^0 = b$$

When we multiply them, we get:

$$b_3*(a_3*2^3 + a_2*2^2 + a_1*2^1 + a_0*a^0)* 2^3 + b_2* (a_3*2 + a_2*2^2 + a_1*2^1 + a_0* 2^0) * 2^2 + b_1 * (a_3* 2^3 + a_2*2^2 + a_1*2^1 + a_0*2^0)* 2^1 + b_0* (a_3*2^3 + a_2*2^2 + a_1*2^1 + a_0 x*2^0)* 2^0 = a * b$$

Assuming an initial division by 2^4 produces a fraction:

$$a * b = [b_3*(a*2^{-1}) + b_2 * (a*2^{-2}) + b_1* (a*2^{-3}) + b_0 * (a*2^{-4})] * 10000H$$

Now we can arrive at the same result as in the previous shift-and-add operation using only right shifts.

In *cmul2,* we'll be using the multiplicand as the product as well. Since the data type is a quadword, the initial division must be by 2^4. Storing the multiplicand in the product variable and concatenating this variable with the internal registers allows us eight words, enough for the largest possible product of two quadwords. As the multiplicand is shifted right and out, the lower bytes of the product are shifted in. This way, we can use one less register (or memory location).

cmul2: Algorithm

1. Move the multiplicand into the lowest four words of the product and load the shift counter *(numbits)* with 64. Clear registers AX, BX, CX, and DX to hold the upper words of the product.
2. Check bit 0 of the multiplicand.

If it's one, go to step 3.

Shift the product and multiplicand right one bit.

Decrement the counter.

If it's not zero, return to the beginning of step 2.

If it's zero, we're done.

3. Add the multiplier to the product.

Shift the product and the multiplicand right one bit.

Decrement the counter.

If it's not zero, return to step 2.

If it's zero, we're done.

cmul2: Listing

```
; ******
;
; A faster shift and add. Multiply one quadword by another,
; passed on the stack, pointers returned to the results.
; Composed of shift and add instructions.
cmul2 proc uses bx cx dx si di, multiplicand:qword, multiplier:qword,
product:word
        local       numbits:byte
        pushf
        cld
        sub         ax, ax
        mov         di, word ptr product
        lea         si, word ptr multiplicand       ;write the multiplicand
                                                    ;to the product
        mov         cx, 4
rep     mov         sw
        sub         di, 8                           ;point to base of product
        lea         si, word ptr multiplier         ;number of bits
        mov         byte ptr numbits, 40h
        sub         ax, ax
        mov         bx, ax
        mov         cx, ax
        mov         dx, ax
test_for_zero:
        test        word ptr [di], 1                ;test the multiplicand for a
                                                    ;one in the LSB
        jne         add multiplier                  ;makes the jump if the
```

```
                                                  ;LSB is a one
        jmp          short  shift
add multiplier
        add          ax, word ptr [si]            ;add the multlplier to
                                                  ;subproduct
        adc          bx, word ptr [si][2]
        adc          cx, word ptr [si][4]
        adc          dx, word ptr [si][6]
shift:
        shr          dx, 1                        ;shift it into the lower
                                                  ;bytes of the product
        rcr          cx, 1
        rcr          bx, 1
        rcr          ax, 1
        rcr          word ptr [di][6], 1
        rcr          word ptr [d1][4], 1
        rcr          word ptr [di][2], 1
        rcr          word ptr [di][0], 1
        dec          byte ptr numbits
        jz           exit
        jmp          short test_for_zero
exit:
        mov          word ptr [di][8], ax         ;move the upper byte of
                                                  ;the product
        mov          word ptr [dil [10], bx
        mov          word ptr [di][12], cx
        mov          word ptr [di][14], dx
        popf
        ret
cmul    endp
```

For an example of a routine written for the Z80 and employing this technique, see the SAMPLES. module on the accompanying disk.

Skipping Ones and Zeros

Anyone who has ever struggled with time-critical code on a bit-oriented machine has probably tried to find a way to lump the groups of ones and zeros in multipliers and multiplicands into one add or shift. A number of methods involve skipping over series of ones and zeros; we'll look at two such procedures. Their efficiency depends on the hardware involved: On machines that provide a sticky bit, such as the 80C196, these routines can provide the most improvement. Unfortunately, the processors that provide that bit also normally have a hardware multiply.

The first technique we'll look at is the Booth algorithm which finds its way around ones and zeros by restating the multiplier.[2] Suppose we want to multiply 1234H by 0fff0H. Studying the multiplier, we find that 0fff0H is equal to 10000H -10H. A long series of rotates and additions can thus be replaced by one subtraction and one addition-that is, subtract 10H x 1234H from the product, then add 10000H x 1234H to the product. The drawback is that the time it takes to execute this operation depends on the data. If the worst-case condition arises-a multiplier with alternating ones and zeros-the procedure can take longer than a standard shift and add.

The trick here is to scan the multiplier looking for changes from ones to zeros and zeros to ones. The way this is done depends on the programmer and the MPU selected. The following table presents the possible combinations of bits examined and the actions taken.

Bit 0	Carry	Action*
0	0	No action
0	1	Add the current state of the multiplicand
1	0	Subtract the current state of the multiplicand
1	1	No action

* This chart assumes that the multiplicand has been rotated along with the multiplier as it is being scanned.

Remember that as the multiplier is scanned from position *0* through position *n*, the multiplicand must also be shifted (multiplied) through these positions.

In its simplest form, the Booth algorithm may be implemented similarly to the shift and add above except that bit 0 of the multiplier is checked along with the carry to determine the appropriate action. As you can see from the table, if you're in the middle of a stream of zeros or ones, you do nothing but shift the multiplier and multiplicand. Depending on the size of the operands involved and the instruction set, it may be faster simply to increment a counter for a multibit shift when the time comes.

The coding for this algorithm is heavily dependent on the device (instruction set), but one possible scheme is as follows.

booth: Algorithm

1. Allocate space for the multiplicand and 32 shifts. Clear the carry and the registers used to form the product.
2. Jump to step 6 if the carry bit is set.
3. Test the 0th bit. If it's not set, jump to step 5.
4. Subtract mltpcnd from the registers used to form the product.
5. Shift mltpcnd left one position.

   Check multiplier to see if it's zero. If so, go to step 8.

   Shift multiplier right one position, shifting the LSB into the carry, and jump to step 2.
6. Test the 0th bit. If it's set, jump to step 5.
7. Add mltpcnd to the product registers and jump to step 5.
8. Write the product registers to product and go home.

booth: Listing

```
; *****
; booth
; unsigned multiplication algorithm
; 16 X 16 multiply

booth proc uses bx cx dx, multiplicand:dword, multiplier:dword, product:word
        local       mltpcnd:qword
        pushf
        cld
        sub         ax, ax
        lea         si, word ptr multiplicand
        lea         di, word ptr mltpcnd
        mov         cx, 2
rep     mov         sw
        stosw
        stosw                                         ;clear upper words
        mov         bx, ax
        mov         cx, ax
        mov         dx, ax
        clc
check_carry:
```

```
        jc          carry_set
        test        word ptr multiplier, 1          ;test bit 0
        jz          shift_multiplicand
sub_multiplicand:
        sub         ax, word ptr mltpcnd
        sbb         bx, word ptr mltpcnd[2]
        sbb         cx, word ptr mltpcnd[4]
        sbb         dx, word ptr mltpcnd[61
shift_multiplicand:
        shl         word ptr mltpcnd, 1
        rcl         word ptr mltpcnd[2], 1
        rcl         word ptr mltpcnd[4], 1
        rcl         word ptr mltpcnd[6], 1
        or          word ptr multiplier[2], 0       ;early-out mechanism
        jnz         shift multiplier
        or          word ptr multiplier, 0
        jnz         shift multiplier
        jw          short exit
shift multiplier:
        shr         word ptr multiplier[2], 1       ;shift multiplier
        rcr         word ptr multiplier, 1
        jmp         short check carry
exit:
        mov         di, word ptr product
        mov         word ptr [di], ax
        mov         word ptr [dil[2], bx
        mov         word ptr [di][4], cx
        mov         word ptr [di][6], dx
        popf
        ret
carry-set:
        test        word ptr multiplier, 1          ;test bit 0
        jnz         shift-multiplicand
add multiplicand:
        add         ax, word ptr mltpcnd
        adc         bx, word ptr mltpcnd[2]
        adc         cx, word ptr mltpcnd[4]
        adc         dx, word ptr mltpcnd[6]
        jmp         short shift_multiplicand
booth endp
```

A corollary to the Booth algorithm is bit pair encoding. The multiplier is scanned, as in the Booth algorithm, but this time three bits are considered at once (see the following chart). This method is attractive because it guarantees that half as many partial products will be required as with the shift and add to produce the result.

Bit n+l	Bit n	Bit n-l	Action*
0	0	0	No action
0	0	1	Add the current state of the multiplicand
0	1	0	Add the current state of the multiplicand
0	1	1	Add twice the current state of the multiplicand
1	0	0	Subtract twice the current state of the multiplicand
1	0	1	Subtract the current state of the multiplicand
1	1	0	Subtract the current state of the multiplicand
1	1	1	No action

* This chart assumes that the multiplicand has been shifted along with the multiplier scanning.

The multiplier is examined two bits at a time relative to the high-order bit of the next lower pair (bit ***n-l*** in the table). First, the multiplier is understood to have a phantom zero to the right of bits 0 and 1; These bits are analyzed accordingly. Second, a phantom zero can be assumed to the left of the multiplier for the purpose of filling out the table. For example, the number 21H would be viewed as:

$$\begin{array}{cccccccc} & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 & \\ \underline{0} & 1 & 0 & 0 & 0 & 0 & 1 & \underline{0} \end{array}$$

The basic approach to implementing this routine is similar to the Booth algorithm.

bit_pair: Algorithm

1. Allocate space to hold the multiplicand plus 32 bits for shifting. Clear the carry and the registers to be used to form the product.
2. If the carry bit is set, jump to step 8.
3. Test the *0*th bit. If it's clear, jump to step 5.

4. Test bit 1.

   If it's set, subtract *mltpcnd* from the product registers and continue with step 7.

   Otherwise, add *mltpcnd* to the product registers and go to step 7.

5. Test bit 1.

   If it's set, jump to step 6.

   Otherwise, continue from step 7.

6. Subtract twice *mltpcnd* from the product registers.

7. Shift *mltpcnd* left two positions.

   Check *multiplier* to see if it's zero. If so, continue at step 13.

   Shift *multiplier* two positions to the right and into the carry.

   Jump to step 2.

8. Test the *0* th bit.

   If it's set, jump to step 11.

   Otherwise, go to step 12.

9. Add the current value of *mltpcnd* to the product registers.

10. Add the current value of *mltpcnd* to the product registers and continue with step 7.

11. Test bit 1.

    If it's set, jump to step 7.

    Otherwise, add twice *mltpcnd* to the product registers and continue from step 7.

12. Test bit 1.

    If it's set, subtract *mltpcnd* from the product registers and continue with step 7.

    Otherwise, add *mltpcnd* to the product registers and go to step 7.

13. Write the product registers to *product* and go home.

bit_pair: Listing

```
; ******
; bit pair encoding
;
;
;
bit_pair proc uses bx cx dx, multiplicand:dword, multiplier:dword, product:word
```

```
        local       mltpcnd:qword
        pushf
        cld
        sub         ax, ax
        lea         si, word ptr multiplicand
        lea         di, word ptr mltpcnd
        mov         cx, 2
rep     movsw
        stosw
        stosw                                           ;clear upper words
        mov         bx, ax
        mov         cx, ax
        mov         dx, ax
        clc
check carry:
        jc          carry set                           ;test bit n-l
        test        word ptr multiplier, 1              ;test bit 0
        jz          shiftorsub
        test        word ptr multiplier, 2              ;test bit 1
        jnz         sub multiplicand
        jmp         add multiplicand
shiftorsub:
        test        word ptr multiplier, 2              ;test bit 1
        jz          shift multiplicand
subx2_multiplicand:                                     ;cheap in-line multiply
        sub         ax, word ptr mltpcnd
        sbb         bx, word ptr mltpcnd[2]
        sbb         cx, word ptr mltpcnd[4]
        sbb         dx, word ptr mltpcnd[6]
sub multiplicand:
        sub         ax, word ptr mltpcnd
        sbb         bx, word ptr mltpcnd[2]
        sbb         cx, word ptr mltpcnd[4]
        sbb         dx, word ptr mltpcnd[6]
shift multiplicand:
        shl         word ptr mltpcnd, 1
        rcl         word ptr mltpcnd[2], 1
        rcl         word ptr mltpcnd[4], 1
        rcl         word ptr mltpcnd[6], 1
        shl         word ptr mltpcnd, 1
        rcl         word ptr mltpcnd[2], 1
        rcl         word ptr mltpcnd[4], 1
        rcl         word ptr mltpcnd[6], 1
        or          word ptr multlplier[2], 0           ;early out if multiplier is zero
```

```
        jnz         shift_multiplier
        or          word ptr multiplier, 0
        jnz         shift  multiplier
        jmp         short exit
shift  multiplier:
        shr         word ptr multiplier[2], 1     ; shift multiplier right twice
        rcr         word ptr multiplier, 1
        shr         word ptr multiplier[2], 1
        rcr         word ptr multiplier, 1
        jmp         short  check_carry
exit:
        mov         di, word ptr product          ;write product out beforeleaving
        mov         word ptr [di], ax
        mov         word ptr [di] [2], bx
        mov         word ptr [di][4], cx
        mov         word ptr [di][6], dx
        popf
        ret
carry_set:
        test        word ptr multiplier, 1
        jnz         addorsubx2
        jmp         short  addorsubx1

addx2_multiplicand:
        add         ax, word ptr mltpcnd          ;cheap  in-line  multiplier
        adc         bx, word ptr mltpcnd[2]
        adc         cx,  word  ptr  mltpcnd[4]
        adc         dx,  word  ptr  mltpcnd[6]
add-multiplicand:
        add         ax, word ptr mltpcnd
        adc         bx, word ptr mltpcnd[2]
        adc         cx,  word  ptr  mltpcnd[4]
        adc         dx,  word  ptr  mltpcnd[6]
        jmp         short  shift_multiplicand
addorsubx2:
        test        word ptr multiplier, 2        ;test bit 1
        jnz         shift-multiplicand
        jmp         short  addx2_multiplicand
addorsubx1:
        test        word ptr multiplier, 2        ;test bit 1
        jnz         sub_multiplicand
        jmp         short  add_multiplicand
```

```
bit_pair            endp
```

Hardware Multiplication: Single and Multiprecision

If the processor you're using has a hardware multiply, you're in luck. Depending on the size of the operands, it's almost always faster than any of the preceeding techniques and can be extended to handle operands of virtually any size. There are exceptions, however; for example, the *MUL* instruction on the 8086 was terribly slow, making it a draw in certain situations. The 80286 was faster in both cycle time and clock speed, and the 80386 was even faster; nevertheless, many examples show that multiplication using the shift and add technique is highly competitive. This is almost never true of multiprecision multiplication, although the double precision shift available on the 80386 and up may be an exception.

In earlier examples involving multiplication, we saw numbers represented as binary polynomials in which each position contained either a zero or a one times base 2 taken to a certain power. To perform that multiplication, we multiplied each bit of the muliplicand by each bit of the multiplier, and summed the subproducts according to their power to form the product (see the section Binary Multiplication). Working with larger numbers is much the same except that the polynomials generally show the operands broken into bytes or words. For example, suppose we needed to multiply two 24-bit quantities, such as 123456H and 654321H. We would want to restate these numbers in terms of a new base, that of our hardware multiply. In this case, we're using an 8086 with a 16-bit multiply, so our base is 21^6 (10000H). First, 123456H becomes three single-byte quantities:

```
12x10000H1 + 3456x10000H0
```

and 654321H becomes:

```
65x10000H1 + 4321x10000H0
```

To better understand this process, let's relabel each byte. The quantity 123456H can be seen as the sum of 120000H + 3456H, which becomes a + b. The quantity

654321H becomes 650000H + 4321H, which then becomes d + e. Now, multiply:

```
    d+e
    a+b
    be
  bd
  ae
 ad
```

With the original numbers, that calculation is:

```
   650000H + 43218
  *120000H + 34568
          0db94116H
        14a5ee0000H
         4b8520000H
       71a00000000H
       7336bf94116H
```

The direction the multiply takes is not significant; that is, the most significant words could have been multiplied first because the final additions align the results. This technique can be extended as far as needed to produce a result. It's also fast, requiring only a few multiplies and divides.

In *mul32,* we multiply two doubleword numbers and arrive at a quadword result.

mul32: Algorithm

1. Use DI to hold the address of the result, a quadword.
2. Move the most significant word of the multiplicand into AX and multiply by the most significant word of the multiplier. The product of this multiplication is written to *result.*
3. The most significant word of the multiplicand is returned to AX and multiplied by the least significant word of the multiplier. The least significant word of this product is *MOV*ed to the second word of *result,* the most significant word of the product is *ADD*ed to the third word of *result,* and any carry is propagated to the most significant word by adding-with-carry a zero.
4. The least significant word of the multiplicand is moved to AX and multiplied by the most significant word of the multiplier. The product

is added to the second word of *result* and added-with-carry to the third word of *result,* with any carry propagated into the most significant word.

5. Finally, the least significant word of the multiplicand is moved into AX and multiplied by the least significant word of the multiplier. The least significant word of this product is moved to the least significant word of result, the most significant word of the product is added to the second word of *result,* and any carry is propagated into the third and then the most significant word of result.

mu132: Listing

```
;*****
;mu132 - Multiplies two unsigned fixed-point values. The
;arguments and a pointer to the result are passed on the stack.
mu132 proc uses dx di,
                   smultiplicand:dword, smultiplier:dword, result:word
       mov         di, word ptr result              ;small model pointer is near
       mov         ax, word ptr smultiplicand[2]    ;multiply multiplicand high
       mul         word ptr smultiplier[2]          ;word by multiplier high word
       mov         word ptr [di][4], ax
       mov         word ptr [di] [6], dx
       mov         ax, word ptr smultiplicand[2]    ;multiply multiplicand high
       mul         word ptr smultiplier[0]          ;word by multiplier low word
       mov         word ptr [di][2], ax
       add         word ptr [di][4], dx
       adc         word ptr [di][6], 0              ;add any remnant carry
       mov         ax, word ptr smultiplicand[0]    ;multiply multiplicand low
       mul         word ptr smultiplier[2]          ;word by multiplier high word
       add         word ptr [di][2], ax
       adc         word ptr [di][4], dx
       adc         word ptr [di][6], 0              ;add any remnant carry
       mov         ax, word ptr smultiplicand[0]    ;multiply multiplicand low
       mul         word ptr smultiplier[0]          ;word by multiplier low word
       mov         word ptr [di] [0], ax
       add         word ptr [di][2], dx
       adc         word ptr [di][4], 0              ;add any remnant carry
       adc         word ptr [di][6], 0
       ret
mu132  endp
```

For additional examples of this technique, see the FXMATH.ASM module.

Binary Division

Error Checking

Division requires more error checking than any of the other basic arithmetic operations. Depending on whether you're using the hardware division instructions or a brew of your own, you'll need to know if a mistake has been made. The primary difference between using a hardware instruction and using your own solution is that an error made during the execution of a hardware instruction can blow up a program quite unaesthetically by invoking an exception or trap.

Three basic errors can occur during division: overflow, division of zero, and an attempt to divide by zero.

You can avoid overflow by checking the dividend and divisor to see whether their quotient will fit in the space provided, or by always breaking the dividend into coefficients of the same size data type as the dividend. An overflow (or underflow) can happen quite easily when the dividend is very large and the divisor is small. If you're using a software algorithm to perform the divide, you may find that you lose part of your data. If you're using a hardware instruction, a hardware exception will be invoked. On the 8086, the largest dividend allowed is 32-bits, the largest divisor is 16 with a 16-bit quotient. In this case, dividing 12345678H by 01DEH results in a quotient of 9bfe9H and a hardware exception (the result too large for the 16 bits the 8086 allows).

If you think such an overflow could occur in your code, it might be wise to include a test before the divide to ascertain how much storage the quotient will require and, therefore, which form of the divide to use. The largest dividend a divisor can divide and store is equal to the size of the data type multiplied by the divisor. By comparing the number obtained from such a multiplication with an arbitrary dividend, you can determine whether the result of that operation will fit in the data type specified.

With binary numbers, this is easy. The largest quotient an 8086 can produce without overflow is 16 bits, which amounts to a left shift of the divisor of 16 bits or a multiplication of 10000H. If the value obtained is greater than or equal to the dividend, the result of the division will fit; if not, it won't. In other words, if you're dividing a 32-bit quantity by a 16-bit quantity, simply comparing the divisor with the upper word of the dividend (dividend/l0000H) will tell you whether the quotient will fit in 16 bits or not. If the upper 16 bits of the dividend are greater than your divisor, the operation will overflow. This test can be extended to 16-bit dividends and eight-bit divisors.

Suppose we wish to divide 12345678H by 1deH. Since this divisor is larger than one byte, we must use 16-bit division. The 1deH need not be multiplied by 10000H or shifted; we only need to compare the upper word of the dividend and the divisor to see which is greater.

```
        mov     dx, dvdnd[2]            ;1234H
        mov     ax, dvdnd[0]            ;5678H
        mov     cx, dvsr                ;1DEH
        cmp     dx, cx                  ;compare
        ja      not_big_enuf            ;the quotient won't fit
div32:
```

Depending on the circumstances, the best method may be to begin any multiprecision divide by clearing DX and loading AX with the most significant word. An overflow is impossible with this technique as long as you have a divisor, since 1H multiplied by 10000H is greater than any one-word dividend.

The other two errors, division of zero and an attempt to divide by zero, are easily detected in the beginning of the routine. If either condition is true, the program can branch to a predetermined error routine and return.

Finally, two conditions are worth checking if your arithmetic gets very big:

- Are the divisor and dividend equal?
- Is the divisor greater than the dividend?

If the two are equal, return a one; if the divisor is greater, return a zero with the dividend in the remainder.

Examples of this kind of checking can be found in the FXMATH.ASM module and later in this chapter in the section Hardware Division.

Software Division

The classic multiplication algorithm is based on the idea of multiplication as iterative addition, so you shouldn't be surprised to learn that the method for division

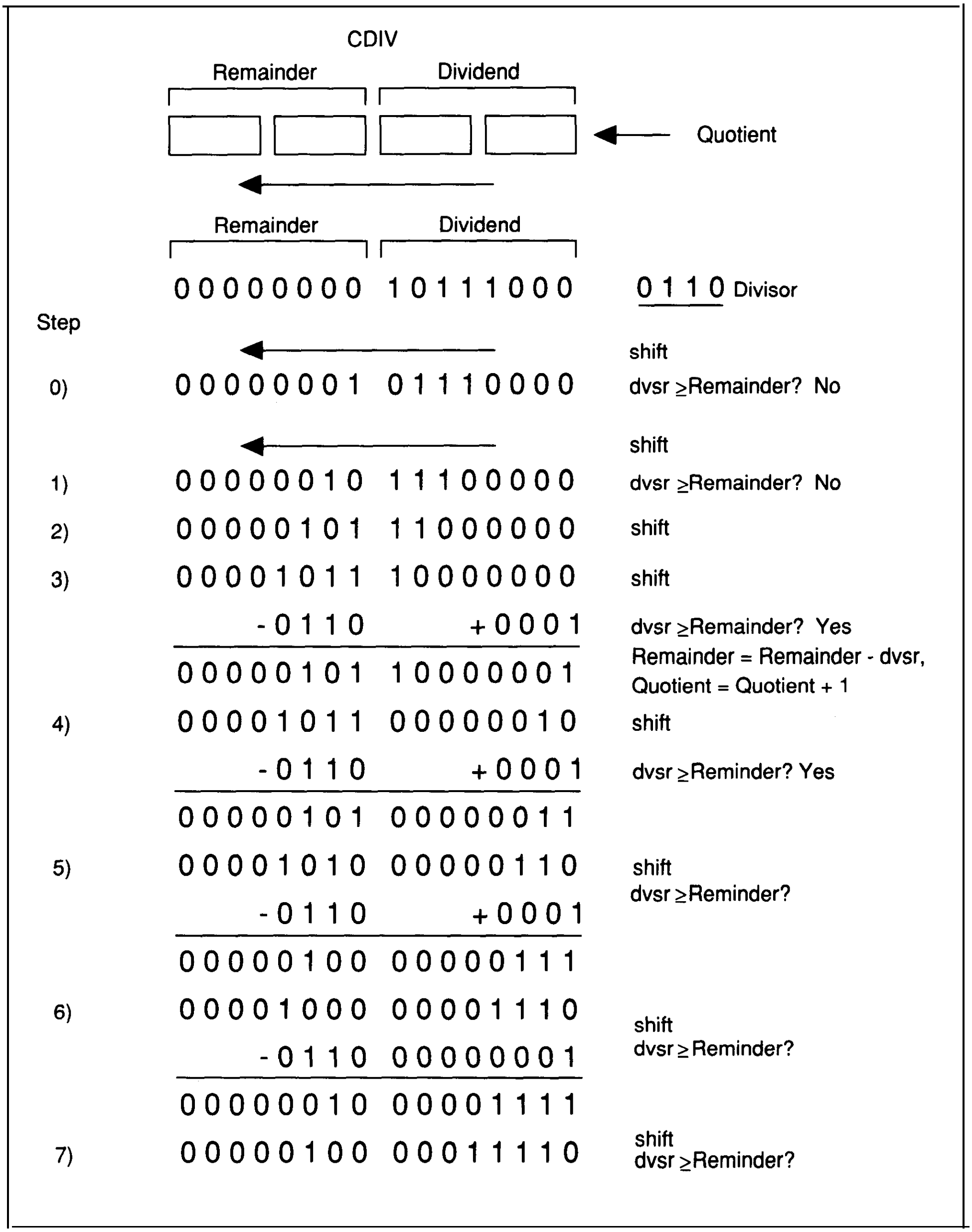


Figure 2-1. Division using shift and subtract.

is based on shift and subtract. This procedure isn't fast, but it's friendly.

The procedure involves shifting the dividend left into a variable, the *remainder*, and comparing this remainder with the divisor. If the remainder is equal to or larger than the divisor, the divisor is subtracted from the remainder and a one is left-shifted into a variable, called the *quotient.* This continues until the requisite number of bits have been shifted. No early out is available here; the number of shifts necessary depends on the size of the operands.

The following variables will be used for the division algorithms: *dvsr, dvdnd, qtnt, cntr,* and *rmndr*. Note that during execution of the algorithm the quotient, dividend, and remainder share memory locations (Figure 2- 1). Shifting the dividend into the remainder leaves the lower bits free to become the quotient. At the end of the routine the dividend is gone, leaving only the quotient and the remainder. For the programmer, this means fewer shifts, some increase in speed, and a slightly smaller routine. The integers these routines are meant to handle are unsigned; the method for signed division is the same as for multiplication which was described earlier (see Signed vs. Unsigned), and is demonstrated in FXMATH.ASM.

cdiv: Algorithm

1. Load the quotient *(qtnt)* with the dividend *(dvdnd);* set an onboard register, si, with the number of bits in the dividend(this will also be the size of our quotient); and clear registers AX, BX, CX, and DX.
2. Left-shift the dividend into the quotient and remainder simultaneously.
3. Compare *rmdr* and *dvsr.*

   If *dvsr* > = *rmndr,* subtract *dvsr* from *rmndr* and increment *qtnt.*

   Otherwise, fall through to the next step.
4. Decrement *si* and test it for zero. If *si* is not 0, return to step 2.
5. Write the remainder and leave.

This will work for any size data type and, as you can see, is basically an iterative subtract.

cdiv: Listing

```
;*****
; classic divide
; one quadword by another, passed on the stack, pointers returned
; to the results
; composed of shift and sub instructions
; returns all zeros in remainder and quotient if attempt is made to divide
; zero. Returns all ffs in quotient and dividend in remainder if divide by
; zero is attempted.
cdiv proc uses bx cx dx si di, dvdnd:qword, dvsr:qword,
        qtnt:word, rmndr:word

        pushf
        cld                                             ;upward
        mov         cx, 4
        lea         si, word ptr dvdnd
        mov         di, word ptr qtnt                   ;move dividend to quotient
rep     movsw                                           ;dvdnd and qtnt share same
                                                        ;memory space
        sub         di, 8                               ;reset pointer
        mov         si, 64                              ;nurrber of bits
        sub         ax, ax
        mov         bx, ax
        mov         cx, ax
        mov         dx, ax
shift:
        shl         word ptr [dil, 1                    ;shift quotient/dividend left
        rcl         word ptr [dil[2], 1                 ;into registers (remainder)
        rcl         word ptr [dil[41, 1
        rcl         word ptr [di] [6], 1
        rcl         ax, 1
        rcl         bx, 1
        rcl         cx, 1
        rcl         dx, 1
compare:
        cmp         dx, word ptr dvsr[6]                ;Compare the remainder and
                                                        ;divisor
        jb          test-for-end
        cmp         cx, word ptr dvsr[4]                ;if remainder ≥ divisor
        jb          test-for-end
        cmp         bx, word ptr dvsr[2]
        jb          test-for-end
        cmp         ax, word ptr dvsr[0]
        jb          test-for-end
```

```
        sub         ax, word ptr dvsr                ;if it is greater than
                                                     ;the divisor
        sbb         bx, word ptr dvsr[2]             ;subtract the divisor and
        sbb         cx, word ptr dvsr[4]
        sbb         dx, word ptr dvsr[6]
        add         word ptr [di], 1                 ;increment the quotient
        adc         word ptr [di][2], 0
        adc         word ptr [di][4], 0
        adc         word ptr [di][6], 0
test_for_end:
        dec         si                               ;decrement the counter
        jnz         shift
        mov         di, word ptr rmndr
        mov         word ptr [di], ax                ;write remainder
        mov         word ptr [di][2], bx
        mov         word ptr [di][4], cx
        mov         word ptr [di][6], dx
exit:
        popf                                         ;to take care of cld
        ret
cdiv    endp
```

Hardware Division

Many microprocessors and microcontrollers offer hardware divide instructions that execute within a few microseconds and produce accurate quotients and remainders. Except in special cases, from the 80286 on up, the divide instructions have an advantage over the shift-and-subtract methods in both code size (degree of complexity) and speed. Techniques that involve inversion and continued multiplication (examples of both are shown in Chapter 3) don't stand a chance when it comes to the shorter divides these machine instructions were designed to handle.

The 8086 offers hardware division for both signed and unsigned types; the 286, 386, and 486 offer larger data types but with the same constraints. The *DIV* instruction is an unsigned divide, in which an implied destination operand is divided by a specific source operand. If the divisor is 16 bits wide, the dividend is assumed to be in DX:AX. The results of the division are returned in the same register pair (the quotient goes in AX and the remainder in DX). If the divisor is only eight bits wide, the dividend is expected to be in AX; at the end of the operation, AL will contain the quotient, while AH will hold the remainder.

As a result, you should make sure the implied operands are set correctly. For example,

```
div     cx
```

says that DX:AX is to be divided by the 16-bit quantity in CX. It also means that DX will then be replaced by the remainder and AX by the quotient. This is important because not all divisions turn out neatly. Suppose you need to divide a 16-bit quantity by a 9-bit quantity. You'll probably want to use the 16-bit form presented in the example. Since your dividend is only a word wide, it will fit neatly in AX. Unless you zero DX, you'll get erroneous results. This instruction divides the entire DX:AX register pair by the 16-bit value in CX. This can be a major annoyance and something you need to be aware of.

As nice as it is to have these instructions available, they do have a limitation; what if you want to perform a divide using operands larger than their largest data type? The 8086 will allow only a 32-bit dividend and a 16-bit divisor. With the 386 and 486, the size of the dividends has grown to 64 bits with divisors of 32; nevertheless, if you intend to do double-precision floating point, these formats are still too small for a single divide.

Several techniques are available for working around these problems. Actually, the hardware divide instructions can be made to work quite well on very large numbers and with divisors that don't fit so neatly in a package.

Division of very large numbers can be handled in much the same fashion as hardware multiplication was on similarly large numbers. Begin by dividing the most significant digits, using the remainders from these divisions as the upper words (or bytes) in the division of the next most significant digits. Store each subquotient as a less and less significant digit in the main quotient.

The number 987654321022H can be divided by a 2987H bit on the 8086 using the 16-bit divide, as follows (also see Figure 2-2):

1. Allocate storage for intermediate results and for the final quotient. Assuming 32 bits for the quotient *(qtnt)* and 16 bits for the remainder *(rmndr),* three words will be required for the dividend *(dvdnd)* and only 16 bits for the divisor

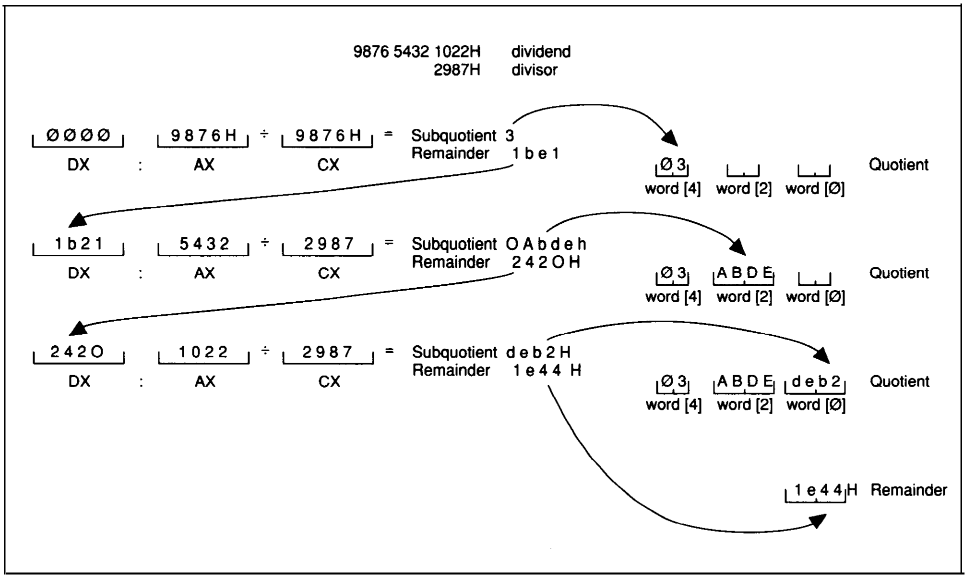


Figure 2-2. Multiprecision division

(dvsr). Actually, the number of bits in the QUOTIENT is equal to the log, DIVIDEND - $\log_2$, DIVISOR, or 34 bits.

2. Clear DX, load the most significant word of the dividend into AX and the divisor into CX, and divide:

```
sub,        dx, dx
mov         ax, word ptr dvdnd[4]          ;9876
div         cx                             ;divide
```

3. At the completion of the operation, AX will hold 3 and DX will hold 1 be 1H.

4. Store AX in the upper word of the quotient:

```
mov         word ptr qtnt[4], ax           ;3H
```

5. With the remainder still in DX as the upper word of the "new" dividend, load

the next most significant word into AX and divide again:

```
mov          ax, word ptr dvdnd[2]          ;5432H
div          cx                             ;recall that the divisor
                                            ;is still in CX
```

6. Now DX holds 2420H and AX holds 0abdeH as the remainder. Store AX in the next most significant word of the quotient and put the least significant word of the dividend into AX.

```
mov          word ptr qtnt[2],ax            ;0abdeH
```

7. Divide DX:AX one final time:

```
mov          ax, word ptr dvdnd[0]
div          cx
```

8. Store the result AX in the least significant word of the quotient and DX in the remainder.

```
mov          word ptr qtnt[0],ax            ;0deb2H
mov          word ptr rmndr,dx              ;le44H
```

This technique can be used on arbitrarily large numbers; it's simply a matter of having enough storage available.

What if both the divisor and the dividend are too large for the hardware to handle by itself? There are at least two ways to handle this. In the case below, the operands are of nearly equal size and only need to be normalized; that is, each must be divided or right-shifted by an amount great enough to bring the divisor into range for a hardware divide (on an 8086, this would be a word). This normalization doesn't affect the integer result, since both operands experience the same number of shifts. Because the divisor is truncated, however, there is a limitation to the accuracy and precision of this method.

If we have good operands, right-shift the divisor, counting each shift, until it fits within the largest data type allowed by the hardware instruction with the MSB a one. Right shift the dividend an equal number of shifts. Once this has been done, divide the resulting values. This approximate quotient is then multiplied by the *original* divisor and subtracted from the *original* dividend. If there is an underflow, the quotient is decremented, the new quotient multiplied by the divisor with that result subtracted from the original dividend to provide the remainder. When there is no underflow, you have the correct quotient and remainder.

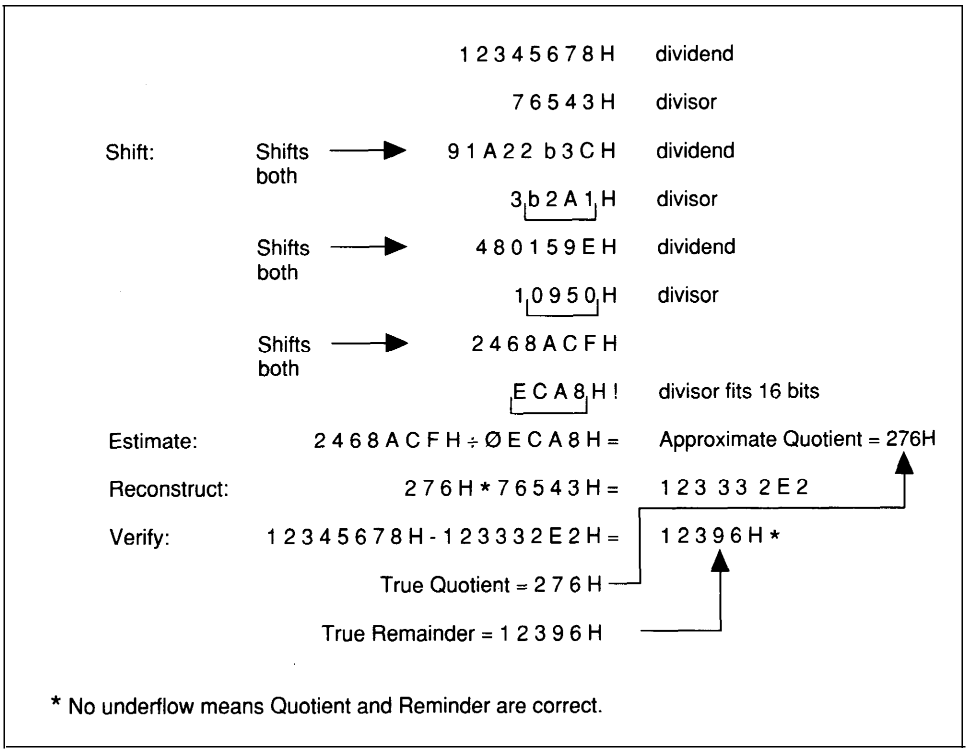


Figure 2-3. Multiword division. This process can continue as long as there is a remainder.

The normalization mentioned earlier is illustrated in Figure 2-3. It requires only that the operands be shifted right until the 16 MSBs of the divisor reside within a word and the MSB of that word is a one.

An example of this technique for 32 bit operands is shown in *div32*.

div32: Algorithm

1. Set aside a workspace of eight words. Load the dividend *(dvdnd)* into the lowest two words and the divisor *(dvsr)* into the next two words. Use DI to point to the quotient.
2. Check to see that the dividend is not zero.

   If it is, clear the quotient, set the carry, and leave.
3. Check for divide by zero.

   If division by zero has occurred, return -1 with the carry set.

   If the divisor is greater than a word, go to step 4.

   Use BX to point at the remainder *(rmndr).*

   Bring the most significant word of the dividend into AX (DX is zero) and divide by the normalized divisor.

   Store the result in the upper word of the quotient.

   Bring the least significant word of the dividend into AX (DX contains the remainder from the last division) and divide again.

   Store the result in the least significant word of the quotient.

   Store DX and clear the upper word of the remainder.
4. Shift both the dividend and the divisor until the upper word of the divisor is zero. This is the normalization.
5. Move the normalized dividend into DX:AX and divide by the normalized divisor.
6. Point to the quotient with BX and the top of the workspace with DI.
7. Multiply the divisor by the approximate quotient and subtract the result from a copy of the original dividend.

   If there is no overflow, you have the correct quotient and remainder.

   Otherwise, decrement the approximate quotient by one and go back to the beginning of step 7. This is necessary to get the correct remainder.
8. Write the remainder, clear the carry for success, and go home.

div32: Listing

```
; *****
;div32
;32-by-32-bit divide
;Arguments are passed on the stack along with pointers to the
;quotient and remainder.

div32 proc uses ax dx di si,
                    dvdnd:dword, dvsr:dword, qtnt:word, rmndr:word
        local       workspace[8]  :word
        sub         ax, ax
        mov         dx, a
        mov         cx, 2
        lea         si, word ptr dvdnd
        lea         di, word ptr workspace
rep     movsw
        mov         cx, 2
        lea         si, word ptr dvsr
        lea         di, word ptr workspace[4]
rep     movsw
        mov         di, word ptr qtnt
        cmP         word ptr dvdnd, ax
        jne         do-divide
        cmp         word ptr dvdnd[2], ax
        jne         do_divide                  ;check for a
        jmp         zero_div                   ;zero dividend
do_divide:
        cmp         word ptr dvsr[2],ax
        jne         shift                      ;see if it is small enough
        cmp         word ptr dvsr, ax          ;check for divide by zero
        je          div_by_zero                ;as long as dx is zero,
                                               ;no overflow is possible
        mov         bx, word ptr rmndr         ;point at remainder
        mov         ax, word ptr dvdnd[2]      ;first divide upper word
        div         word ptr dvsr
        mov         word ptr [di][2],ax        ;and save it
        mov         ax, word ptr dvdnd
        div         word ptr dvsr              ;then the lower word
        mov         word ptr [di],ax           ;and save it
        mov         word ptr [bx],dx           ;save remainder
        xor         ax,ax
        mov         word ptr [bx][2],ax
        jmp         exit
```

```
shift:
       shr          word ptr dvdnd[2], 1        ;normalize both dvsr and
                                                ;dvdnd
       rcr          word ptr dvdnd[0], 1        ;shift both the same number
       shr          word ptr dvsr[2], 1
       rcr          word ptr dvsr[0], 1
       cmp          word ptr dvsr[2],ax         ;shift until last one
                                                ;leaves upper word
       jne          shift
divide:
       mov          ax, word ptr dvdnd          ;since MSB of dvsr is a one, no
       mov          dx, word ptr dvdnd[2]       ;overflow is possible here
       div          word ptr dvsr
       mov          word ptr [di] [0], ax       ;approximate quotient
get-remainder
       mov          bx, di                      ;quotient
       lea          di, word ptr workspace[8]   ;test first approximation of
                                                ;quotient by multiplying it by
reconstruct:                                    ;dvsr and comparing it with dvdnd
       mov          ax, word ptr workspace[4]
       mul          word ptr [bx]               ;low word of multiplicand by
       mov          word ptr [di] [0], ax       ;low word of multiplier
       mov          word ptr [di][2], dx
       mov          ax, word ptr workspace[6]
       mul          word ptr [bx]
       add          word ptr [di][2], ax        ;high word of multiplicand by
       mov          ax, word ptr workspace[0]   ;low word of multiplier
       mov          dx, word ptr workspace[2]
       sub          ax, word ptr [di] [0]       ;compare results of divide
                                                ;approximation
       sbb          dx, word ptr [di][2]
       jnc          div_ex                      ;good or overflows
                                                ;overflow, decrement approx
                                                ;quotient
       mov          ax, word ptr [bx]
       mov          dx, word ptr [bx][2]
       sub          word ptr [bx], 1            ;decrement the quotient
       sbb          word ptr [bx] [2], 0
       jmp          short reconstruct
div_ex:
       mov          di, word ptr rmndr          ;the result is a good quotient
                                                ;and remainder
       mov          word ptr [di], ax
       mov          word ptr [di] [2], dx
```

```
      clc
exit:
      ret
div_by_zero:
      not          ax                           ;division by zero
      mov          word ptr [di][0], ax
      mov          ord ptr [di] [2], ax
      stc
      jmp          exit
zero_div:                                       ;division of zero
      mov          word ptr [di][0], ax
      mov          word ptr [di][21, ax
      stc
      jmp          exit
div32 endp
```

If very large operands are possible and the greatest possible precision and accuracy is required, there is a very good method using a form of linear interpolation. This is very useful on machines of limited word length. In this technique, the division is performed twice, each time by only the Most Significant Word of the divisor, once rounded down and once rounded up to get the two limits between which the actual quotient exists. In order to better understand how this works, take the example, 98765432H/54321111H. The word size of the example machine will be 16 bits, which means that the MSW of the divisor is 5432H * 2^{16}. The remaining divisor bits should be imagined to be a fractional extension of the divisor, in this manner: 5432.1111H.

The first division is of the form:

987654328/54320000H

and produces the result:

1.cf910000H.

Next, increment the divisor, and perform the following division:

```
98765432H/54330000H
```

for the second quotient:

```
1.cf8c0000H.
```

Now, take the difference between these two values:

```
1cf9l0000H - 1cf8c0000H = 50000H.
```

This is the range within which the true quotient exists. To find it, multiply the fraction part of the divisor described in the lines above by this range:

```
50000H * .1111H = 5555H,
```

and subtract this from the first quotient:

```
1cf910000H - 5555H = 1.cf90aaabH.
```

To prove this result is correct, convert this fixed point result to decimal, yielding:

```
1.810801188229D.
```

Convert the operands to decimal, as well:

```
98765432H/54321111H = 2557891634D/1412567313D = 1.81081043746D.
```

This divide does not produce a remainder in the same way the division above does; its result is true fixed point with a fractional part reflecting the remainder. This method can be very useful for reducing the time it takes to perform otherwise time consuming multiple precision divisions. However, for maximum efficiency, it requires that the position of the Most Significant Word of the divisor and dividend be known in advance. If they are not known, the routine is responsible for locating these bits, so that an attempt to divide zero, or divide by zero, does not occur.

The next routine, *div64,* was specially written for the floating point divide in

Chapter Four. This method was chosen, because it can provide distinct advantages in code size and speed in those instances in which the position of the upper bits of each operand is known in advance. In the next chapter, two routines are presented that perform highly accurate division without this need. They, however, have their own complexities.

To begin with, the operands are broken into word sizes (machine dependent), and an initial division on the entire dividend is performed using the MSW of the divisor and saved. The MSW of the divisor is incremented and the same division is performed again, this will, of course result in a quotient smaller than the first division. The two quotients are then subtracted from one another, the second quotient from the first, with the result of this sutraction multiplied by the remaining bits of the divisor as a fractional multiply. This product is subtracted from the first quotient to yield a highly accurate result. The final accuracy of this operation is not to the precision you desire, it can be improved by introducing another different iteration.

div64: Algorithm

1. Clear the result and temporary variables.
2. Divide the entire dividend by the Most Significant Word of the divisor. (The remaining bits will be considered the fractional part.)

   This is the first quotient, the larger of the two, save this result in a temporary variable.
3. Increment the divisor.

   If there is an overflow, the next divide is really by 2^{16}, therefore, shift the dividend by 16 bits and save in a temporary variable.

   Continue with step 5.
4. Divide the entire dividend by the incremented divisor.

   This is the second quotient, the smaller of the two, save this result in a temporary variable.
5. Subtract the second quotient from the first.
6. Multiply the result of this subtraction by the fractional part of the divisor.
7. Subtract the integer portion of this result from the first quotient.
8. Write the result of step 7 to the output and return.

div64: Listing

```
; ******
;div64
;will divide a quad word operand by a divisor
;dividend occupies upper three words of a 6 word array
;divisor occupies lower three words of a 6 word array
;used by floating point division only
div64 proc uses es ds,
          dvdnd:qword, dvsr:qword, qtnt:word

      local      result:tbyte, tmp0:qword,
                 tmpl:qword, opa:qword, opb:qword

      pushf
      cld

      sub        ax, ax
      lea        di, word ptr result
      mov        cx, 4
rep   stosw
      lea        di, word ptr tmp0           ;quotient
      mov        cx, 4
rep   stosw

setup:
      mov        bx, word ptr dvsr[3]
continue_setup:
      lea        si, word ptr dvdnd           ;divisor no higher than
      lea        di, word ptr tmp0            ;receives stuff for quotient
      sub        a, dx

      mov        ax, word ptr [si][3]
      diV        bx
      mov        word ptr [di][4], ax        ;result goes into temporary varriable
      mov        ax, word ptr [si][1]
      div        bx
      mov        word ptr [di][2], ax
      sub        ax, ax
      mov        ah, byte ptr [si]
      div        bx
      mov        word ptr[di] [O], ax        ;entire first approximation
```

```
     lea         si, word ptr dvdnd            ;divisor no higher than
     lea         di, word ptr tmpl             ;receives stuff for quotient
     sub         dx, dx
     add         bx, 1                         ;round divisor up
     jnc         as_before                     ;if the increment results in
                                               overflow
     mov         ax, word ptr [si] [3]         ;there is no divide, only a
                                               ;shift by 2^16
     mov word ptr [di][2], ax
     mov ax, word ptr [si] [l]
     mov         word ptr [di][0], ax
     jmp         find-difference

as-before:
     mov         ax, word ptr [si] [3]         ;divide entire dividend by new
                                               ;divisor
     div         bx
     mov         word ptr [di][4], ax          ;result goes into quotient
     mov         ax, word ptr [si] [l]
     div         bx
     mov         word ptr [di][2], ax          ;result goes into quotient
     sub         a, ax
     mov         ah, byte ptr [si]
     div         bx
     mov         word ptr [di] [0], ax         ;result goes into quotient

find-difference:
     invoke      sub64, tmp0, tmp1, addr opa   ;get the difference between the
                                               ;two extremes
     lea         si, word ptr dvsr
     lea         di, word ptr opb
     mov         cx, 3
rep  movsb
     sub         ax, ax
     stosb
     stosw

     invoke      mu164a, opa, opb, addr result ;fractional multiply to get
                                               ;portion of
     lea         si, word ptr result[3]        ;difference to subtract from
                                               ;initial quotient
```

```
     lea       di, word ptr opb              ;(high quotient)
     mov       cx, 3
rep  movsb
     sub       ax, ax
     stosb
     stosw

     invoke    sub64,tmp0, opb, addr tmp0    ;subtract and write out result
     lea       si, word ptr tmp0

div_exit:
     mov       di, word ptr qtnt
     mov cx, 4
rep  movsw
     popf
     ret
div64 endp
```

When writing arithmetic routines, keep the following in mind:

- Addition can always result in the number of bits in the largest summend plus one.
- Subtraction requires at least the number of bits in the largest operand for the result and possibly a sign.
- The number of bits in the product of a multiplication is always equal to log, multiplier + log, multiplicand. It is safest to allow 2n bits for an n-bit by n-bit multiplication.
- The size, in bits, of the quotient of a division is equal to the difference, log, dividend - $\log_2$, divisor. Allow as many bits in the quotient as in the dividend.

[1] *Microsoft Macro Assembler Reference.* Version 6.0. Redmond, WA: Microsoft Corp., 1991.

[2] Cavanagh, Joseph J. F. *Digital Computer Arithmetic.* New York, NY: McGraw-Hill Book Co., 1984, Page 148.

CHAPTER 3

Real Numbers

There are two kinds of fractions. A symbolic fraction represents a division operation, the numerator being the dividend and the denominator the divisor. Such fractions are often used within the set of natural numbers to represent the result of such an operation. Because this fraction is a symbol, it can represent any value, no matter how irrational or abstruse.

A *real* number offers another kind of fraction, one that expands the number line with negative powers of the base you're working with. Base 10, involves a decimal expansion such that $a_{-1}, * 10^{-l0} + a^{-2} * 10^{-10} + a_{-3} * 10^{-10}$. Nearly all scientific, mathematical, and everyday work is done with real numbers because of their precision (number of representable digits) and ease of use.

The symbolic fraction 1/3 *exactly* represents the division of one by three, but a fractional expansion in any particular base may not, For instance, the symbolic fraction l/3 is irrational in bases 10 and 2 and can only be approximated by .33333333D and .55555555H.

A value in any base can be irrational-that is, you may not be able to represent it perfectly within the base you're using. This is because the positional numbering system we use is modular, which means that for a base to represent a symbolic fraction rationally all the primes of the denominator must divide that base evenly. It's not the value that's irrational; it's just that the terms we wish to use cannot express it exactly. The example given in the previous paragraph, l/3, is irrational in base 10 but is perfectly rational in base 60.

There have been disputes over which base is best for a number system. The decimal system is the most common, and computers must deal with such numbers at some level in any program that performs arithmetic calculations. Unfortunately, most microprocessors are base 2 rather than base 10. We can easily represent decimal integers, or *whole numbers,* by adding increasingly larger powers of two. Decimal fractions, on the other hand, can only be approximated using increasingly larger *negative* powers of two, which means smaller and smaller pieces. If a fraction isn't exactly equal to the sum of the negative powers of two in the word size or data type available, your representation will only be approximate (and in error). Since we have little choice but to deal with decimal reals in base 2, we need to know what that means in terms of the word size required to do arithmetic and maintain accuracy.

The focus of this chapter is fixed-point arithmetic in general and fractional fixed point in particular.

Fixed Point

Embedded systems programmers are often confronted with the task of writing the fastest possible code for real-time operation while keeping code size as small as possible for economy. In these and other situations, they turn to integer and fixed-point arithmetic.

Fixed point is the easiest-to-use and most common form of arithmetic performed on the microcomputer. It requires very little in the way of protocol and is therefore fast-a great deal faster than floating point, which must use the same functions as fixed point but with the added overhead involved in converting the fixed-point number into the proper format. Floating point *is* fixed point, with an exponent for placing the radix and a sign. In fact, within the data types defined for the two standard forms of floating-point numbers, the *long real* and *short real,* fewer significant bits are available than if the same data types were dedicated to fixed-point numbers. In other words, no more precision is available in a floating-point number than in fixed point.

In embedded applications, fixed point is often a must, especially if the system can not afford or support a math coprocessor. Applications such as servo systems, graphics, and measurement, where values are computed on the fly, simply can't wait for floating point to return a value when updating a Proportional-Integral-Derivative (PID) control loop such as might be used in a servo system or with animated graphics. So why not always use integer or fixed-point arithmetic?

Probably the most important reason is range. The short real has a decimal range of approximately 10^{38} to 10^{-38} (this is a *range,* and does not reflect resolution or accuracy; as you'll see in the next chapter, floating-point numbers have a real problem with granularity and significance). To perform fixed-point arithmetic with this range, you would need 256 bits in your data type, or 32 bytes for each operand.

Another reason is convenience. Floating-point packages maintain the position of the radix point, while in fixed point you must do it yourself.

Still another reason to use floating-point arithmetic is the coprocessor. Using a coprocessor can make floating point as fast as or faster than fixed point in many applications.

The list goes on and on. I know of projects in which the host computer communicated with satellites using the IEEE 754 format, even though the satellite had no coprocessor and did not use floating point. There will always be reasons to use floating point and reasons to use fixed point.

Every application is different, but if you're working on a numerically intensive application that requires fast operation or whose code size is limited, and your system doesn't include or guarantee a math coprocessor, you may need to use some form of fixed-point arithmetic.

What range and precision do you need? A 32-bit fixed-point number can represent 2^{32} separate values between zero and 4,294,967,296 for integer-only arithmetic and between zero and 2.3283064365E-10 for fractional arithmetic or a combination of the two. If you use a doubleword as your basic data type, with the upper word for integers and the lower one for fractions, you have a range of 1.52587890625E-5 to 65,535 with a resolution of 4,294,967,296 numbers.

Many of the routines in FXMATH.ASM were written with 64-bit fixed-point numbers in mind: 32 bits for integer and 32 bits for fraction. This allows a range of 2.3283064365E- 10 to 4,294,967,295 and a resolution of 1.84467440737E30 numbers, which is sufficient for most applications. If your project needs a wider range, you can write specialized routines using the same basic operations for all of them; only the placement of the radix point will differ.

Significant Bits

We normally express decimal values in the language of the processor-binary. A 16-bit binary word holds 16 binary digits (one bit per digit) and can represent 65,536 binary numbers, but what does this mean in terms of decimal numbers? To estimate the number of digits of one base that are representable in another, simply find *ceil*($log_a B^n$), where *a* is the target base, *B* is the base you're in, and *n* is the power or word size. For example, we can represent a maximum of

$$\texttt{decimal-digits} = 5 = \texttt{ceil}((\log_{10}(2^{16}))$$

in a 16-bit binary word (a word on the 8086 is 16 bits, 2^{16} = 65536, and $\log_{10}$ 65536 = 4.8, or five decimal digits). Therefore, we can represent 50,000 as c350H, 99.222D as 63.39H, and .87654D as .e065H.

Note: A reference to fixed-point or fractional arithmetic refers to binary fractions. For the purposes of these examples, decimal fractions are converted to hexadecimal by multiplying the decimal fraction by the data type. To represent .5D in a byte, or 256 bits, I multiply .5 by 256. The result is 128, or 80H. These are *binary* fractions in hexadecimal format. Results will be more accurate if guard digits are used for proper rounding. Conversion to and from fixed-point fractions is covered in Chapter 5.

We can express five decimal numbers in 16 bits, but how accurately are we doing it? The Random House dictionary defines *accuracy* as the degree of correctness of a quantity, so we could say that these five decimal digits are accurate to 16 bits. That is, our representation is accurate because it's correct given the 16 bits of precision we're using, though you still may not find it accurate enough for your purposes.

Clearly, if the fraction cannot be expressed directly in the available storage or is irrational, the representation won't be exact. In this case, the error will be in the LSB; in fact, it will be equal to or less than this bit. As a result, the smallest value representable (or the percent of error) is shown as 2^{-m}, where *m* is the number of fraction bits. For instance, 99.123D is 63.1fH accurate to 16 bits, while 63.1f7cH is accurate to 24 bits. Actually, 63.lfH is 99.121D and 63.lf7cH is 99.12298, but each is accurate within the constraints of its precision. Assuming that no extended precision is available for rounding, no closer approximation is possible within 16 or 24 bits. The greater the precision, the better the approximation.

The Radix Point

The radix point can occur anywhere in a number. If our word size is 16 bits, we can generally provide eight bits for the integer portion and eight bits for the fractional portion though special situations might call for other forms. Floating point involves fixed-point numbers between 1.0 and 2.0. In a 24-bit number, this leaves 23 bits for the mantissa. The maximum value for a sine or cosine is 1, which may not even need to be represented, leaving 16 bits of a 16-bit data type for fractions. Perhaps you only need the fraction bits as guard digits to help in rounding; in such cases you might choose to have only two bits, leaving the rest for the integer portion.

Depending on your application, you may want a complete set of fixed-point routines for each data type you use frequently (such as 16- and 32-bit) and use other routines to address specific needs. In any event, maintaining the radix point (scaling) requires more from the programmer than does floating point, but the results, in terms of both speed and code size, are worth the extra effort.

The nature of the arithmetic doesn't change because of the radix point, but the radix point does place more responsibility on the programmer. Your software must know where the radix point is at all times.

Rounding

If all the calculations on a computer were done with symbolic fractions, the error involved in approximating fractions in any particular base would cease to exist. The arithmetic would revolve around multiplications, divisions, additions, and subtractions of numerators and denominators and would always produce rational results. The problem with doing arithmetic this way is that it can be very awkward, time-consuming, and difficult to interpret in symbolic form to any degree of precision.

On the other hand, real numbers involve error because we can't always express a fraction exactly in a target base. In addition, computing with an erroneous value will propagate errors throughout the calculations in which it is used. If a single computation contains several such values, errors can overwhelm the result and render the result meaningless. For example, say you're multipying two 8-bit words and you know that the last two bits of each word are dubious. The result of this operation will be 16 bits, with two error bits in one operand plus two error bits in the other operand. That means the product will contain four erroneous bits.

For this reason, internal calculations are best done with greater precision than you expect in the result. Perform the arithmetic in a precision greater than needed in the result, then represent only the significant bits as the result. This helps eliminate error in your arithmetic but presents another problem: What about the information in the extra bits of precision? This brings us to the subject of rounding.

You can always ignore the extra bits. This is called *truncation* or *chop,* and it simply means that they are left behind. This method is fast, but it actually contributes to the overall error in the system because the representation can only approach the *true* result at exact integer multiples of the LSB. For example, suppose we have a decimal value, 12345D, with extended bits for greater precision. Since 5 is the least significant digit, these extended bits are some fraction thereof and the whole number can be viewed as:

```
12345.XXXXD
         ↑_____extended bits
```

Whenever the result of a computation produces extended bits other than zero, the result, 12345D, is not quite correct. As long as the bits are always less than one-half the LSB, it makes little difference. But what if they exceed one-half? A particular calculation produces 12345.7543D. The true value is closer to 12346D than to 12345D, and if this number is truncated to 12345D the protection allowed by the extended bits is lost. The error from truncation ranges from zero to almost one in the LSB but is definitely biased below the true value.

Another technique, called *jamming,* provides a symmetrical error that causes the true value or result to be approached with an almost equal bias from above and below. It entails almost no loss in speed over truncation. With this technique, you simply set the low-order bit of the significant bits to one. Using the numbers from the previous example, 12345.0000D through 12345.9999D remain 12345.0000D.

And if the result is even, such as 123456.0000D, the LSB is set to make it 123457.0000D. The charm of this technique is that it is fast and is almost equally biased in both directions. With this method, your results revolve symmetrically about the ideal result as with jamming, but with a tighter tolerance (one half the LSB), and, at worst, only contributes a small positive bias.

Perhaps the most common technique for rounding involves testing the extended bits and, if they exceed one-half the value of the LSB, adding one to the LSB and propagating the carry throughout the rest of the number. In this case, the fractional portion of 12345.5678D is compared with .5D. Because it is greater, a one is added to 12345D to make it 12346D.

If you choose this method of rounding to maintain the greatest possible accuracy, you must make still more choices. What do you do if the extended bits are equal to exactly one-half the LSB?

In your application, it may make no difference. Some floating-point techniques for calculating the elementary functions call for a routine that returns an integer closest to a given floating-point number, and it doesn't matter whether that number was rounded up or down on exactly one-half LSB. In this case the rounding technique is unimportant.

If it is important, however, there are a number of options. One method commonly taught in school is to round up and down alternately. This requires some sort of flag to indicate whether it is a positive or negative toggle. This form of rounding maintains the symmetry of the operation but does little for any bias.

Another method, one used as the default in most floating-point packages, is known as *round to nearest.* Here, the extended bits are tested. If they are greater than one-half the LSB, the significant bits are rounded up; if they are less, they are rounded down; and if they are exactly one-half, they are rounded toward even. For example, 12345.5000D would become 12346.0000D and 12346.5000D would remain 12346.0000D. This technique for rounding is probably the most often chosen, by users of software mathematical packages. Round to nearest provides an overall high accuracy with the least bias.

Other rounding techniques involve always rounding up or always rounding down. These are useful in interval arithmetic for assessing the influences of error upon the calculations. Each calculation is performed twice, once rounded up and once rounded down and the results compared to derive the direction and scope of any error. This can be very important for calculations that might suddenly diverge.

At least one bit, aside from the significant bits of the result, is required for rounding. On some machines, this might be the carry flag. This one bit can indicate whether there is an excess of equal to or greater than one-half the LSB. For greater precision, it's better to have at least two bits: one to indicate whether or not the operation resulted in an excess of one-half the LSB, and another, *the sticky bit*, that registers whether or not the excess is actually greater than one-half. These bits are known as *guard bits*. Greater precision provides greater reliability and accuracy. This is especially true in floating point, where the extended bits are often shifted into the significant bits when the radix points are aligned.

Basic Fixed-Point Operations

Fixed-point operations can be performed two ways. The first is used primarily in applications that involve minimal number crunching. Here, scaled decimal values are translated into binary (we'll use hex notation) and handled as though they were decimal, with the result converted from hex to decimal.

To illustrate, let's look at a simple problem: finding the area of a circle ($A = \pi r^2$). If the radius of the circle is 5 inches (and we use 3.14 to approximate it), the solution is 3.14 * (5 * 5), or 78.5 square inches. If we were to code this for the 8086 using the scaled decimal method, it might look like this:

```
mov          ax, 5                    ;the radius
mul          al                       ;square the radius
mov          dx, 13aH                 ;314 = 3.14D * 100D
mul          dx                       ;ax will now hold 1eaaH
```

The value leaaH converted to decimal is 7,850, which is 100D times the actual answer because π was multiplied by 100D to accommodate the fraction. If you only need the integer portion, divide this number by 100D. If you also need the fractional part, convert the remainder from this division to decimal.

The second technique is binary. The difference between purely binary and scaled decimal arithmetic is that instead of multiplying a fraction by a constant to make it an integer, perform the operation, then divide the result by the same constant for the result. We express a binary fraction as the sum of negative powers of two, perform the operation, and then adjust the radix point. Addition, subtraction, multiplication, and division are done just as they are with the integer-only operation; the only additional provision is that you pay attention to the placement of the radix point. If the above solution to the area of a circle were written using binary fixed point, it would look like this:

```
mov        ax, 5          ;the radius
mul        al             ;square the radius
mov        dx, 324H       ;804D = 3.14D * 256D
mul        dx             ;ax will now hold 4e84H
```

The value 4eH is 78D, and 84H is .515D (132D/256D).

Performing the process in base 10 is effective in the short term and easily understood, but it has some drawbacks overall. Both methods require that the program keep track of the radix point, but correcting the radix point in decimal requires multiplies and divides by 10, while in binary these corrections are done by shifts. An added benefit is that the output of a routine using fixed-point fractions can be used to drive D/A converters, counters, and other peripherals directly because the binary fraction and the peripheral have the same base. Using binary arithmetic can lead to some very fast shortcuts; we'll see several examples of these later in this chapter.

Although generalized routines exist for fixed-point arithmetic, it is often possible to replace them with task specific high-speed routines, when the exact boundaries of the input variables are known. This is where thinking in base 2 (even when hex notation is used) can help. *Scaling* by 1,024 or any binary data type instead of 1,000 or any decimal power can mean the difference between a divide or multiply routine and a shift. As you'll see in a routine at the end of this chapter, dividing by a negative power of two in establishing an increment results in a right shift. A negative power of 10, on the other hand, is often irrational in binary and can result in a complex, inaccurate divide.

Before looking at actual code, lets examine the basic arithmetic operations. The conversions used in the following examples were prepared using the computational techniques in Chapter 5.

Note: The "." does not actually appear. They are assumed and added by the author for clarification.

Say we want to add 55.33D to 128.67D. In hex, this is 37.54H + 80.acH, assuming 16 bits of storage for both the integer and the mantissa:

```
     37.54H      (55.33D)
+    80.acH     (128.67D)
-----------------------
     b8.00H     (184.00D)
```

Subtraction is also performed without alteration:

```
     80.acH     (128.67D)
-    37.54H      (55.33D)
-----------------------
     49.58H      (73.34D)

     37.54H      (55.33D)
-    80.acH     (128.67D)
-----------------------
     b6.a8H     (-73.34D)
```

Fixed-point multiplication is similar to its pencil-and-paper counterpart:

```
     80.acH       (128.67D)
x    37.548        (55.33D)
---------------------------
     lbcf.2c70H  (7119.3111D)
```

as is division:

```
     80.acH     (128.67D)
÷    37.54H      (55.33D)
-----------------------
     2.53H        (2.32D)
```

The error in the fractional part of the multiplication problem is due to the lack of precision in the arguments. Perform the identical operation with 32-bit precision, and the answer is more in line: 80.ab85H x 37.547bH = 1bcf.4fad0ce7H.

The division of 80acH by 3754H initially results in an integer quotient, 2H, and a remainder. To derive the fraction from the remainder, continue dividing until you reach the desired precision, as in the following code fragment:

```
sub     a, dx
mov     ax, 80ach
mov     cx, 37548
div     cx                            ;this divide leaves the quotient
                                      ;(2) in ax and the remainder
                                      ;remainder (1204H) in dx
mov     byte ptr quotient[l], al
sub     ax, ax
div     cx                            ;Divide the remainder multiplied
                                      ;by 10000H x to get the fraction
                                      ;bits Overflow is not a danger
                                      ;since a remainder may never be
                                      ;greater than or
                                      ;even equal to the divisor.
mov      byte ptr quotient[0], ah
                                      ;the fraction bits are then 53H,
                                      ;making the answer constrained
                                      ;to a 16-bit word for this
                                      ;example, 2.53H
```

* The 8086 thoughtfully placed the remainder from the previous division in the DX register, effectively multiplying it by 10000H.

Of course, you could do the division once and arrive at both fraction bits and integer bits if the dividend is first multiplied by 10000H (in other words, shifted 16 places to the left). However, the danger of overflow exists if this scaling produces a dividend more than 10000H times greater than the divisor.

The following examples illustrate how fixed-point arithmetic can be used in place of floating point.

A Routine for Drawing Circles

This first routine is credited to Ivan Sutherland, though many others have worked with it.' The algorithm draws a circle incrementally from a starting point on the circle and without reference to sines and cosines, though it's based on those relationships.

To understand how this algorithm works, recall the two trigonometric identities (see Figure 3- 1):

```
sin θ = ordinate / radius vector

cos θ = abscissa / radius vector

(where θ is an angle)
```

Multiplying a fraction by the same value as in the denominator cancels that denominator, leaving only the numerator. Knowing these two relationships, we can derive both the vertical coordinate *(ordinate)* and horizontal coordinate *(abscissa)*

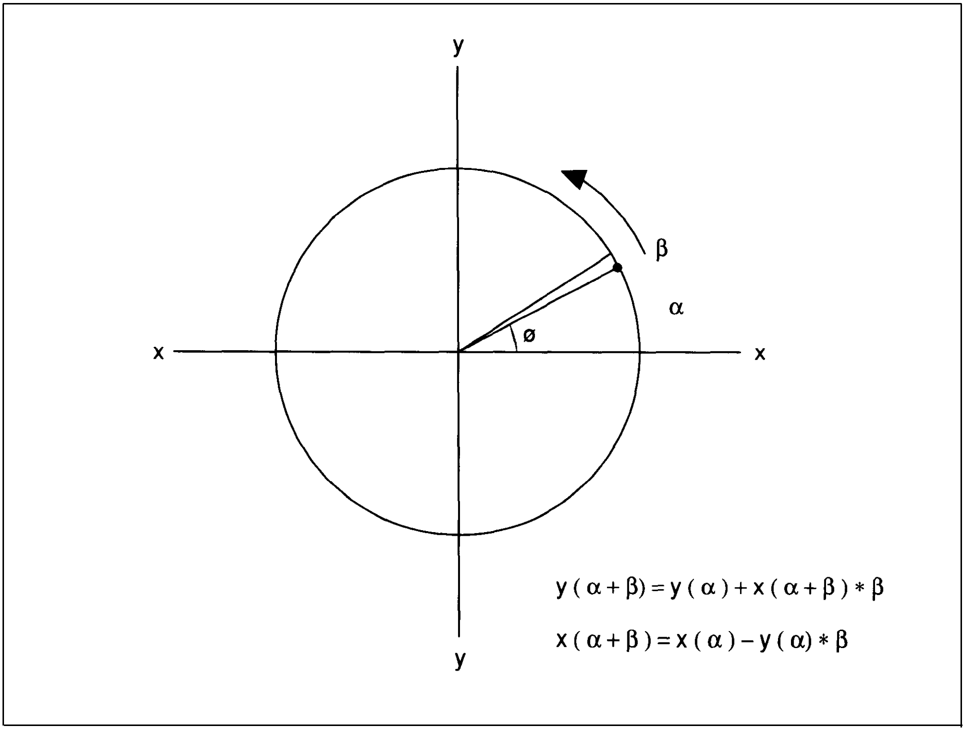


Figure 3-1. A circle drawn using small increments of t.

in a rectangular coordinate system by multiplying sinθ by the radius vector for the ordinate and cos θ by the radius vector for the abscissa. This results in the following polar equations:

```
x(a) = r * cos a

y(a) = r * sin a
```

Increasing values of θ, from zero to 2π radians, will rotate the radius vector through a circle, and these equations will generate points along that circle. The formula for the sine and cosine of the sum of two angles will produce those increasing values of θ by summing the current angle with an increment:

```
sin(α+ß) = sin α cos ß + cos α sin ß

cos(α+ß) = cos α cos ß - sin α sin ß
```

Let *a* be the current angle and *b* the increment. Combining the polar equations for deriving our points with the summing equations we get:

```
y(α+ß) = r * sin α cos ß + r * cos α sin ß

x(α+ß) = r * cos α cos ß - r * sin α sin ß
```

For small angles (much smaller than one), an approximation of the cosine is about one, and the sine is equal to the angle itself. If the increment is small enough, the summing equations become:

```
y(α+ß) = r * sin α + r * cos α * ß

x(α+ß) = r * cos α  - r * sin α * ß,
```

and then:

```
y(α+ß) = y(α) + x(α) * ß

x(α+ß) = x(α) - y(α) * ß
```

Using these formulae, you can roughly generate points along the circumference of a circle. The difficulty is that the circle gradually spirals outward, so a small adjustment is necessary-the one made by Ivan Sutherland:

$$y(\alpha+\beta) = y(\alpha) + x(\alpha+\beta) * \beta$$

$$x(\alpha+\beta) = x(\alpha) - y(\alpha) * \beta$$

When you select an increment that is a negative power of two, the following routine generates a circle using only shifts and adds.

circle: Algorithm

1. Initialize local variables to the appropriate values, making a copy of x and y and clearing *x_point* and *y_point.* Calculate the value for the the loop counter, *count.*
2. Get_x and_y and round to get new values for pixels. Perform a de facto divide by l000H by taking only the value of DX in each case for the points.
3. Call your routine for writing to the graphics screen.
4. Get _y, divide it by the increment, *inc,* and subtract the result from _X.
5. Get _x, divide it by *inc,* and add the result to _y.
6. Decrement *count.*

   If it isn't zero, return to step 2.

   If it is, we're done.

circle: listing

```
; *****
;
;
;
circle proc uses bx cx dx si di, x_ coordinate:dword, y_ coordinate:dword,
increment:word

        local       x:dword, y:dword, x_point:word, y_point:word, count

        mov         ax, word ptr x_ coordinate
        mov         dx, word ptr x_coordinate[2]
```

```
        mov         word ptr x, ax
        mov         word ptr x[2], dx
        mov         ax, word ptr y_coordinate
        mov         dx, word ptr y_coordinate[2]
        mov         word ptr y, ax
        mov         word ptr y[2], dx           ;load local variables

        sub         ax, ax
        mov         x_point, ax                 ;x coordinate
        mov         y__point, ax                ;y coordinate

        mov         ax, 4876h
        mov         dx, 6h                      ;2 * pi
        mov         cx, word ptr increment      ;make this a negative
                                                ;power of two

get_num_points:
        shl         ax, 1                       ;2 * pi radians
        rcl         dx, 1
        loop        get_num_points
        mov         count, dx                   ;divide by 10000h

set_point:
        mov         ax, word ptr x
        mov         dx, word ptr x[2]
        add         ax, 8000h                   ;add .5 to round up
        jnc         store_x                     ;to integers
        adc         dx, 0h
store_x:
        mov         x_point, dx

        mov         ax, word ptr y
        mov         dx, word ptr y[2]
        add         ax, 8000h                   ;add .5
        jnc         store_y
        adc         dx, Oh
store_y:
        mov         y_point, dx

;your routine for writing to the screen goes here and uses x-point and
;y_point as screen coordinates

        mov         ax, word ptr y
        mov         dx, word ptr y[2]
```

```
        mov         cx, word ptr increment
update_x:
        sar         dx 1                        ;arithmetic shift to maintain
sign
        rcr         ax, 1
        loop        update x
        sub         word ptr x, ax              ;new x equals x - y * increment
        sbb         word ptr x[2], dx

        mov         ax, word ptr x
        mov         dx, word ptr x[2]
        mov         cx, word ptr increment
update_y:
        sar         dx, 1                       ;arithmetic shift to maintain
sign
        rcr         ax, 1
        loop        update_y
        add         word ptr y, ax              ;new y equals y + x * increment
        adc         word ptr y[2], dx
        dec         count
        jnz         set_point

        ret
circle  endp
```

Bresenham's Line-Drawing Algorithm

This algorithm is credited to Jack Bresenham, who published a paper in 1965 describing a fast line-drawing routine that used only integer addition and subtraction.[2]

The idea is simple: A line is described by the equation *f(x,y) = y' * x-x' * y* for a line from the origin to an arbitrary point (see Figure 3-2). Points not on the line are either above or below the line. When the point is above the line, *f(x,y)* is negative; when the point is below the line, *f(x,y)* is positive. Pixels that are closest to the line described by the equation are chosen. If a pixel isn't exactly on the line, the routine decides between any two pixels by determining whether the point that lies exactly between them is above or below the line. If above, the lower pixel is chosen; if below, the upper pixel is chosen.

In addition to these points, the routine must determine which axis experiences the greatest move and use that to program diagonal and nondiagonal steps. It

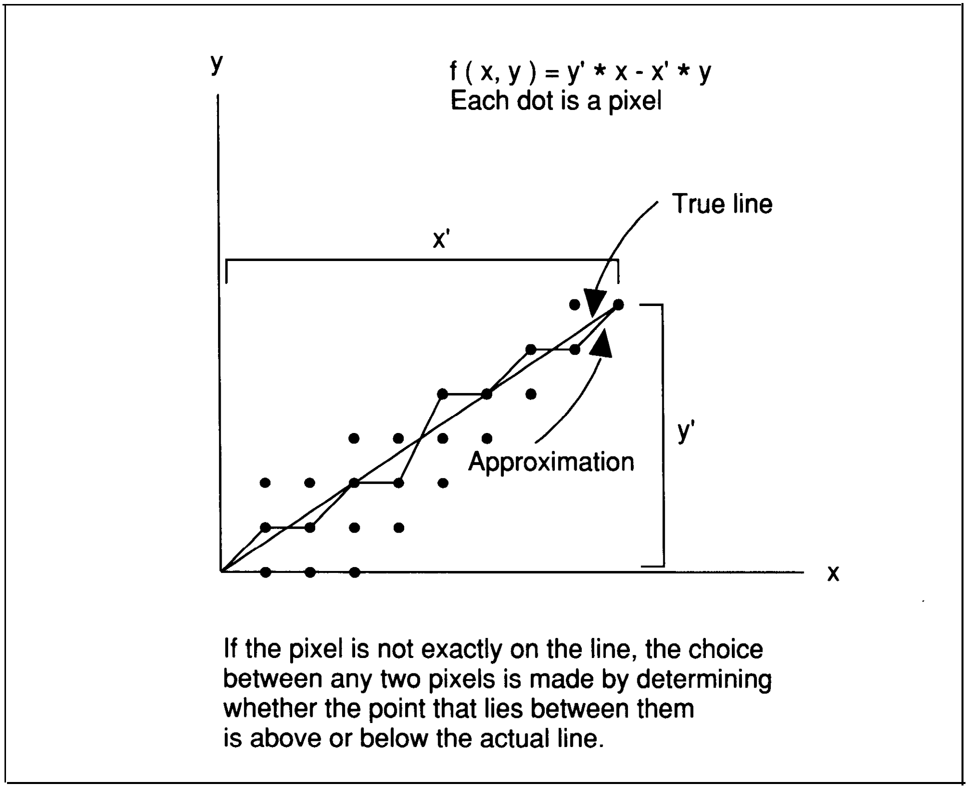


Figure 3-2. Bresenham's line-drawing algorithm.

calculates a decision variable based on the formula *2 * b+a,* where a is the longest interval and *b* the shortest. Finally, it uses *2 * b* for nondiagonal movement and *2 * b - 2 * a* for the diagonal step.

line: Algorithm

1. Set up appropriate variables for the routine. Move *xstart* to x and *ystart* to *y.*
2. Subtract *xstart* from *xend.*

   If the result is positive, make *xstep_diag* 1 and store the result in *x dif.*

   If the result is negative, make *xstep_diag* -1 and two's-complement the result before storing it in *x_dif.*

3. Subtract *ystart* from *yend.*

   If the result is positive, make *ystep_diag* 1 and store the result in *y_dif.*

   If the result is negative, make *ystep_diag* -1 and two's-complement the result before storing it in *y_dif.*

4. Compare *x_dif* with *y_dif.*

   If *x_dif* is smaller, swap *x_dif* and *y_dif,* clear *xstep,* and store the value of *ystep_diag* in *ystep.*

   If *x_dif* is larger, clear *ystep* and store the value of *xstep_diag* in *xstep.*

5. Call your routine for putting a pixel on the screen.

6. Test *decision.*

   If it's negative, add *xstep* to x, *ystep* to y, and *inc* to *decision,* then continue at step 7.

   If it's positive, add *xstep_diag* to *x, ystep_diag* to y, and *diag_inc* to *decision,* then go to step 7.

7. Decrement *x dif.*

   If it's not zero, go to step 5.

   If it is, we're done.

line: Listing

```
;*****
;
line  proc  uses bx cx dx si di, xstart:word, ystart:word, xend:word, yend:word

       local        x:word, y:word, decision:word, x_dif:word, y_dif:word,
xstep_diag:word,
       ystep_diag:word, xstep:word, ystep:word, diag_incr:word, incr:word

       mov          ax, word ptr xstart
       mov          word ptr x, ax              ;initialize local variables
       mov          ax, word ptr ystart
       mov          word ptr y, ax

direction:
       mov          ax, word ptr xend
       sub          ax, word ptr xstart         ;total x distance
       jns          large_x                     ;which direction are we drawing?
```

```
        neg        ax                        ;went negative
        mov        word ptr xstep_diag, -1
        jmp        short store xdif
large_x:
        mov        word ptr xstep_diag, 1
store xdif:
        mov        x dif, ax

        mov        ax, word ptr yend         ;y distance
        sub        ax, word ptr ystart
        jns        large_y                   ;which direction?
        neg        ax
        mov        word ptr ystep_diag, -1
        jmp        short store ydif
large_y:
        mov        word ptr ystep_diag, 1
store ydif:
        mov        word ptr y_dif, ax        ;direction is determined by signs

octant:
        mov        ax, word ptr x dif        ;the axis with greater difference
        mov        bx, word ptr y-dif        ;becomes our reference
        cmp        ax, bx
        jg         bigger x

        mov        y_dif, ax                 ;we have a bigger y move
                                             ;than x
        mov        x dif, bx                 ;x won't change on nondiagonal
        sub        ax, ax                    ;steps, y changes every step
        mov        word ptr xstep, ax
        mov        ax, word ptr ystep_diag
        mov        word ptr ystep, ax
        jmp        setup inc
bigger x:
        mov        ax, word ptr xstep_diag   ;x changes every step
                                             ;y changes only
        mov        word ptr xstep, ax        ;on diagonal steps
        sub        ax, ax
        mov        word ptr ystep, ax
setup inc:
        mov        ax, word ptr y dif        ;calculate decision variable
        shl        ax, 1
        mov        word ptr incr, ax         ;nondiagonal increment
                                             ; = 2 * y_dif
```

```
        sub         ax, word ptr x_dif
        mov         word ptr decision, ax       ;decision variable
                                                ; = 2 * y_dif - x_dif
        sub         ax, word ptr x_dif
        mov         word ptr diag incr, ax      ;diagonal increment
                                                ; = 2 * y_dif - 2 * x_dif

        mov         ax, word ptr decision       ;we will do it all in
                                                ;the registers
        mov         bx, word ptr x
        mov         cx, word ptr x_dif
        mov         dx, word ptr y

line loop:

; Put your routine for turning on pixels here. Be sure to push ax, cx, dx,
; and bx before destroying them; they are used here. The value for the x
; coordinate is in bx, and the value for the y coordinate is in dx.

        or          ax, ax
        jns         dpositive
                                                ;calculate new position and
        add         bx, word ptr xstep          ;update the decision variable
        add         dx, word ptr ystep
        add         ax, incr
        jmp         short chk loop
dpositive:
        add         bx, word ptr xstep_diag
        add         dx, word ptr ystep_diag
        add         ax, word ptr diag incr

chk_loop:
        loop        line loop
        ret
line    endp
```

When fixed-point operands become very large, it sometimes becomes necessary to find alternate ways to perfom arithmetic. Multiplication isn't such a problem; if it exists as a hardware instruction on the processor you are using, it is usually faster than division and is easily extended.

Division is not so straightforward. When the divisors grow larger than the size allowed by any hardware instructions available, the programmer must resort to other methods of division, such as *CDIV* (described in Chapter 2), linear polation (used in the floating-point routines), or one of the two routines presented in the following pages involving Newton-Raphson approximation and iterative multiplication. The first two will produce a quotient and a remainder, the last two return with a fixed point real number. Choose the one that best fits the application.[3, 4]

Division by Inversion

A root of an equation exists whenever $f(x)=0$. Rewriting an equation so that $f(x)=0$ makes it possible to find the value of an unknown by a process known as Newton-Raphson Iteration. This isn't the only method for finding roots of equations and isn't perfect, but, given a reasonably close estimate and a well behaved function, the results are predictable and correct for a prescribed error.

Look at Figure 3-3. The concept is simple enough: Given an initial estimate of a point on the *x* axis where a function crosses, you can arrive at a better estimate by evaluating the function at that point, *f(x,)*, and its first derivative of $f(x_0)$ for the slope of the curve at that point. Following a line tangent to $f(x_0)$ to the *x* axis produces an improved estimate. This process can be iterated until the estimate is within the allowed error.

The slope of a line is determined by dividing the change in the y axis by the corresponding change in the *x* axis: *dy/dx*. In the figure, *dy* is given by $f(x_0)$, the distance from the *x* axis at x_0 to the curve, and *dx* by $(x_1 - x_0)$, which results in

$$f'(x_0) = f(x_0)/(x_1-x_0)$$

Solving for x, gives

$$x_1 = x_0 - f(x_0)/f'(x_0)$$

Substituting the new *x,* for x_0 each time the equation is solved will cause the estimate to close in on the root very quickly, doubling the number of correct significant bits with each pass.

To using this method to find the inverse of a number, *divisor,* requires that the

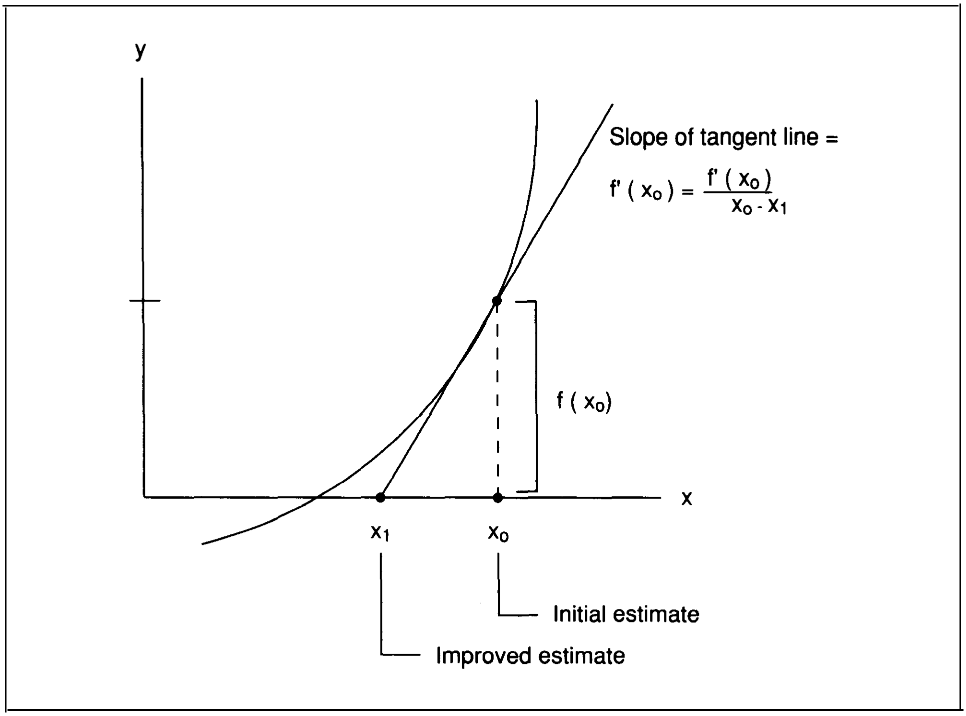


Figure 3-3. Newton-Raphson iteration.

equation be formulated as a root. A simple equation for such a purpose is

$$f(x) = 1/x - \textit{divisor}$$

From there, it's a short step to

$$x = 1/\textit{divisor}$$

Now the equation for finding the root becomes an equation for finding the inverse of a number:

$$x_1, = ((1/x) - \textit{divisor})/(-1/\textit{divisor}^2)$$

which simplifies to:

$$x_{new} = x_{old} * (2 - \mathit{divisor}\ (x)_{old})$$

In his book *Digital Computer Arithmetic,* Joseph Cavanagh suggests that this equation be simplified even further to eliminate the divisor from the iterative process and use two equations instead of one. He makes the term *divisor(x)* equal to one called *unity* (because it will close in on one) in this routine, which reduces the equation to:

$$x_{new} = X_{old} * (2 - \mathit{unity})$$

Multiplying both sides of this equation by the divisor, *divisor,* and substituting again for his equality, *divisor(x) = unity,* he generates another equation to produce new values for *unity* without referring to the original divisor:

$$(\mathit{divisor}\ (x))_{new} = (\mathit{divisor}\ (x))_{old} * (2 - \mathit{unity}) =$$
$$\mathit{unity}_{new} = \mathit{unity}_{old} * (2 - \mathit{unity})$$

This breaks the process down to just two multiplies and a two's complement. When these two equations are used together in each iteration, the algorithm will compute the inverse to an input argument very quickly.

To begin, there must be an initial estimate of the reciprocal. For speed, this can be computed with a hardware instruction or derived from a table if no such instruction exists on your processor. Multiply the initial estimate by the divisor to get the first unity. Then, the two equations are evaluated as a unit, first generating a new divisor and then a new unity, until the desired precision is reached.

The next routine expects the input divisor to be a value with the radix point between the doublewords of a quadword, fixed-point number. The routine finds the most significant word of the divisor, then computes and saves the number of shifts required to normalize the divisor at that point-that is, position the divisor so that its most significant one is the bit just to the right of the implied radix point: .1XXX...

For example, the number 5 is

```
101.000B
   ↑______radix point
```

Normalized, it is:

```
000.101B
   ↑______radix point
```

After that, the actual divisor is normalized within the divisor variable as if the radix point were between the third and fourth words. Since the greatest number *proportion* or *divisor* will see is two or less, there is no danger of losing significant bits. Placing the radix point there also allows for greater precision.

Instead of subtracting the new *proportion* from two, as in the equation, we two's complementproportion and the most significant word is ANDed with 1H to simulate a subtraction from two. This removes the sign bits generated by the two's complement and leaves an integer value of one plus the fraction bits.

Finally, the reciprocal is realigned based on a radix point between the doublewords as the fixed-point format dictates, and multiplied by the dividend.

divnewt: Algorithm

1. Set the loop counter, lp, for three passes. This is a reasonable number since the first estimate is 16-bits. Check the dividend and the divisor for zero.

   If no such error conditions exist, continue with step 2,

   Otherwise, go to step 10.

2. Find the most significant word of the divisor.

   Determine whether it is above or below the fixed-point radix point. In this case, the radix point is between the doublewords.

   Test to see if it is already normalized.

   If so, go to step 5.

3. Shift a copy of the most significant word of the divisor left or right until it is normalized, counting the shifts as you proceed.

4. Shift the actual divisor until its most significant one is the MSB of

the third word of the divisor. This is to provide maximum precision for the operation.

5. Divide 10000000H by the most significant word of the divisor for a first approximation of the reciprocal. The greater the precision of this first estimate, the fewer passes will be required in the algorithm (the result of this division will be between one and two.)

   Shift the result of this division left one hex digit, four bits, to align it as an integer with a three-word fraction part. This initial estimate is stored in the *divisor* variable.

   Divide this first estimate by two to obtain the proportionality variable, *proportion.*

6. Perform a two's complement on *proportion* to simulate subtracting it from two. Multiply *proportion* by *divisor.* Leave the result in *divisor.*

7. Multiply *proportion* by the estimate, storing the results in both *proportion* and *estimate.* Decrement *lp.*

   If it isn't zero, continue with step 6.

   Otherwise, go to step 8.

8. Using the shift counter, *shift,* reposition *divisor* for the final multiplication.

9. Multiply *divisor,* now the reciprocal of the input argument, by the dividend to obtain the quotient. Write the proper number of words to the ouput and exit.

10. An attempt was made to divide zero or divide by zero; exit with error.

divnewt: Listing

```
; *****
;divnewt- division by Raphson-Newton zeros approximation
;
;
;
divnewt             proc uses bx cx dx di si, dividend:qword, divisor:qword,
quotient:word

        local       temp[8]:word, proportion:qword, shift:byte, qtnt_adjust:byte,
                    lp:byte, tmp:qword, unity:qword

        cld                                         ;upward


        sub         cx, cx
```

```
        mov     byte ptr lp, 3              ;should only take three passes
        mov     qtnt_adjust, cl
        or      cx, word ptr dividend[0]
        or      cx, word ptr dividend[2]
        or      cx, word ptr dividend[4]
        or      cx, word ptr dividend[6]
        je      ovrflw                      ;zero dividend

        sub     cx, cx
        or      cxr word ptr divisor [0]
        or      cx, word ptr divisor [2]
        or      cx, word ptr divisor [4]
        or      cx, word ptr divisor [6]
        je      ovrflw                      ;zero divisor

        sub     ax, ax
        mov     bx, 8
find_msb:                                   ;look for MSB of divisor
        lea     di, word ptr divisor
        dec     bx
        dec     bx
        cmp     word ptr [di] [bx], ax      ;di is pointing at divisor
        je      find_msb

        mov     byte ptr gtnt_adjust, bl
        mov     ax, word ptr [di][bx]       ;get most significant word
        sub     cx, cx                      ;save shifts here
        cmp     bx, 2h                      ;see if already normalized
        jb      shift_left
        ja      shift_right
        test    word ptr [di][bx], 8000h    ;normalized?
        jne     norm dvsr                   ;it's already there

shift_left:
        dec     cx
        shl     ax, 1
        test    ah, 80h
        jne     save_shift
        jmp     shift_ left                 ;count the number of shifts to
normalize

shift_right:
        inc     cx
        shr     ax, 1
        or      ax, ax
```

```
        je          save-shift
        jmp         shift right              ;count the number of shifts to
                                             ;normalize

save shift:
        mov         byte ptr shift, cl
        sub         ax, ax
shift back:
        cmp         word ptr [di][6], ax     ;we will put radix point
                                             ;at word three
        je          norm_dvsr
        shr         wordptr [di][6], 1
        rcr         word ptr [di][4], 1
        rcr         word ptr [di][2], 1
        rcr         word ptr [di] [0], 1
        jmp         shift back

norm dvsr:
        test        word ptr [di][4], 8000h
        jne         make_first
        shl         word ptr [di][O], 1      ;the divisor
        rcl         word ptr [di][2], 1      ;truly normalized
        rcl         word ptr [di][4], 1      ;for maximum
        jmp         norm_dvsr                ;this should normalize divisor

make first:
        mov         dx, l000h
        sub         ax, ax
        mov         bx, word ptr [di][4]     ;first approximation
                                             ;could come from a table
        div         bx
        sub         dx, dx                   ;keep only the four least bits
        mov         cx, 4
correct dvsr:
        shl         ax, 1                    ;don't want to waste time with
                                             ;a big shift
        rcl         dx, 1
        loop        correct_dvsr
        mov         word ptr divisor[4], ax
        mov         word ptr divisor[6], dx
        sub         cx, cx
        mov         word ptr divisor[2], cx
        mov         word ptr divisor[0], cx
        shr         dx 1                     ;don't want to waste time
```

```
                                                ;with a big shift
        rcr         ax, 1
        mu1         bx                          ;reconstruct for first attempt
        shl         ax, 1                       ;don't want to waste time
                                                  ;with a big shift
        rcl         dx, 1
        mov         word ptr unity[4], dx
        sub         cx, cx
        mov         word ptr unity[6], cx
        mov         word ptr unity[2], cx
        mov         word ptr unity, cx

makeproportion:                                   ;this could be done with
                                                  ;a table
        mov         word ptr proportion[4], dx
        sub         ax, ax
        mov         word ptr proportion[6], ax
        mov         word ptr proportion[2], ax
        mov         word ptr proportion, ax

invert_proportion:
        not         word ptr proportion[6]
        not         word ptr proportion[4]
        not         word ptr proportion[2]
        neg         word ptr proportion           ;attempt to develop with
                                                  ;2's complement
        jc          mloop
        add         word ptr proportion[2], 1
        adc         word ptr proportion[4], 0
        adc         word ptr proportion[6], 0

mloop:
        and         word ptr proportion[6], 1     ;make it look like it was
                                                  ;subtracted from 2
        invoke      mu164, proportion, divisor, addr temp

        lea         si, word ptr temp[6]
        lea         di, word ptr divisor
        mov         cx, 4
rep     movsw

        invoke      mu164, proportion, unity, addr temp

        lea         si, word ptr temp[6]
```

```
        lea         di, word ptr unity
        mov         cx, 4
rep     movsw
        lea         si, word ptr temp[6]
        lea         di, word ptr proportion
        mov         cx, 4
rep     movsw

        dec         byte ptr lp
        je          divnewt_shift
        jmp         invert_proportion          ;six passes for 64 bits

ovrflw:
        sub         ax, ax
        not         ax
        mov         cx, 4
        mov         di, word ptr quotient
rep     stosw                                  ;make infinite answer
        jmp         divnewt_exit

divnewt_shift:
        lea         di, word ptr divisor
        mov         cl, byte ptr shift         ;get shift count
        or          cl, cl
        js          qtnt_left                  ;positive, shift left
qtnt_right:
        mov         ch, 10h
        sub         ch, cl
        mov         cl, ch
        sub         ch, ch
        jmp         qtlft

qtnt_left:
        neg         Cl
        sub         ch, ch
        add         cl, 10h                     ;we Want to take it to the MSB
qtlft:
        shl         word ptr [dil[0], 1
        rcl         word ptr [di][2], 1
        rcl         word ptr [di1[4], 1
        rcl         word ptr [di][6], 1
        loop        qtlft

divnewt_mult:                                  ;multiply reciprocal by dividend
```

```
        sub         ax, ax                     ;see that temp is clear
        mov         cx, 8
        lea         di, word ptr temp
rep     stosw

        invoke      mul64, dividend, divisor, addr temp
        mov         bx, 4                      ;adjust for magnitude of result
        add         bl, byte ptr qtnt_adjust
        mov         di, word ptr quotient
        lea         si, word ptr temp
        add         si, bx
        cmp         bl, 0ah
        jae         write zero
        mov         cx, 4
rep     movsw
        jmp         divnewt  exit

write_zero:
        mov         cx, 3
rep     movsw
        sub         ax, ax
        stosw
divnewt_exit:
        popf
        ret
divnewt   endp
```

Division by Multiplication

If the denominator and numerator of a fraction are multiplied by the same factor, the ratio does not change. If a factor could be found, such that multiplying it by the denominator causes the denominator to approach one, then multiplying the numerator by the same factor must cause that numerator to approach the quotient of the ratio or simply the result of the division.

In this procedure, as in the last, you normalize the divisor, or numerator-that is, shift it so that its most significant one is to the immediate right of the radix point, creating a number-such that $.5 \geq$ number < 1. To keep the ratio between the denominator and numerator equal to the original fraction, perform the same number of shifts, in the same direction, on the dividend or numerator.

Next, express the divisor, which is equal to or greater than one half and less than one, as one minus some offset:

```
divisor = l- offset
```

To bring this number, 1- offset, closer to one, choose another number by which to multiply it which will retain its original value and increase it by the offset, such as:

```
multiplier = 1 + offset.
```

To derive the first attempt, multiply this multiplier by the divisor:

$$\texttt{multiplier * divisor = (1 - offset) * (1 + offset) = 1 - offset}^2$$

followed by

```
(1 + offset) * dividend
```

As you can see, the result of this single multiplication has brought the divisor closer to one (and the dividend closer to the quotient). For the next iteration, 1 - offset^2 is multiplied by 1 + offset^2 (with a similar multiplication to the dividend). The result is 1 - offset^4, which is closer still to one. Each following iteration of 1 - offset^n is multiplied by 1 + offset^n (with that same 1 + offset^n multiplying the dividend) until the divisor is one, or almost one, which is .11111111...B to the word size of the machine you're working on. Since the same operation was performed on both the dividend and the divisor, the ratio did not change and you need not realign the quotient.

To illustrate, let's look at how this procedure works on the decimal division 12345/1222. Remember that a bit is a digit in binary. Normalizing the denominator in the discussion above required shifting it so that its most significant one was to the immediate right of the radix point. The same thing must be done to the denominator in the decimal fraction 12345/1222D; 1222D becomes .9776D, and performing the same number of shifts (in the same direction) on the numerator, 12345, yields 9.8760D. Since the divisor (.9976D) is equal to 1 - .0224, make the first multiplier equal to 1 + .0224 and multiply divisor * (1. + .0224) = .99949824D. You then take 9.8760D, the dividend, times (1. + .0224) to get 10.0972224D. On the next iteration, the multiplier is ($1 + .0224^2$), or l.000501760D, which multiplied by the denominator is .999999748D and by the numerator is 10.10228878D. Finally, multiplying .999999748D by ($1 + .0224^4$) produces a denominator of .999999999D, and ($1 + .0224^4$) times 10.10228878D equals 10.10229133D, our quotient. The next routine illustrates one implementation of this idea.

clivmul: Algorithm

1. Set pass counter, *lp,* for 6, enough for a 64-bit result. Check both operands for zero,

   If either is zero, go to step 10.

   Otherwise continue with step 2.

2. Find the most significant word of *divisor,* and see whether it is above or below the radix point,

   If it's below, normalization is to the left; go to step 3a.

   If it's above, normalization is to the right; go to step 3b.

   If it's right there, see whether it's already normalized.

   if so, skip to step 4.

   Otherwise, continue with step 3a.

3. a) Shift a copy of the most significant word of the divisor left until the MSB is one, counting the shifts as you go. Continue with step 4.

   b) Shift a copy of the most significant word of the divisor right until it is zero, counting the shifts as you go. Continue with step 4.

4. Shift the actual divisor so that the MSB of the most significant word is one.

5. Shift the dividend right or left the same number of bits as calculated in step 3. This keeps the ratio between the dividend and the divisor the same.

6. Offset = 1 - normalized divisor.

7. Multiply the offset by the divisor, saving the result in a temporary register. Add the divisor to the temporary register to simulate the multiplication of 1 + offset by the divisor. Write the temporary register to the divisor.

8. Multiply the offset by the dividend, saving the result in a temporary

simulate the multiplication of 1 + offset by the dividend. Write the temporary register to the divisor.

9. Decrement *lp,*

   If it's not yet zero, go to step 6.

   Otherwise, the current dividend is the quotient; exit.

10. Overflow exit, leave with an error condition.

divmul: Listing

```
; *****
;divmul-division by iterative multiplication
;Underflow and overflow are determined by shifting. If the dividend shifts out on
;any attempt to normalize, then we have "flowed" in whichever direction it
;shifted out.
;
divmul proc uses bx cx dx di si, dividend:qword, divisor:qword, guotient:word

       local        temp[8]:word, dvdnd:qword, dvsr:qword, delta:qword,
divmsb:byte,
                    lp:byte, tmp:qword

       cld                                          ;upward

       sub          cx, cx
       mov          byte ptr lp, 6                  ;should only take six passes
       lea          di, word ptr dvdnd              ;check for zero
       mov          ax, word ptr dividend[0]
       mov          dx, word ptr dividend[2]
       or           cx, ax
       or           cx, dx
       mov          word ptr [di][0], ax
       mov          word ptr [di][2], dx
       mov          ax, word ptr dividend[4]
       mov          dx, word ptr dividend[6]
       mov          word ptr [di][4], ax
       mov          word ptr [di][6], dx
       or           cx, ax
       or           cx, dx
       je           ovrflw                          ;zero dividend

       sub          cx, cx
       lea          di, word ptr dvsr               ;check for zero
       mov          ax, word ptr divisor[0]
       mov          dx, word ptr divisor[2]
```

```
        or          cx, ax
        or          cx, dx
        mov         word ptr [dil[0], ax
        mov         word ptr [di] [2], dx
        mov         ax, word ptr divisor[4]
        mov         dx, word ptr divisor[6]
        mov         word ptr [di][4], ax
        mov         word ptr [di][6], dx
        or          cx, ax
        or          cx, dx
        je          ovrflw                          ;zero divisor

        sub         ax, ax
        mov         bx, 8
find_MSB:                                           ;look for MSB of divisor
        dec         bx
        dec         bx
        cmp         word ptr [di] [bx], ax          ;di is pointing at dvsr
        je          find msb

        mov         ax, word ptr [di] [bx]          ;get MSW
        sub         cx, cx                          ;save shifts here
        cmp         bx, 2h                          ;see if already
                                                    ;normalized
        jb          shift left
        ja          shift right
        test        word ptr [di][bxl, 8000h        ;normalized?
        jne         norm dvsr                       ;it's already there

shift_left:
        dec         cx
        shl         ax, 1
        test        ah, 80h
        jne         norm_dvsr
        jmp         shift_left                      ;count the number of
                                                    ;shifts to normalize

shift_right:
        inc         cx
        shr         ax, 1
        or          ax, ax
        je          norm_dvsr
        jmp         shift_right                     ;count the number of shifts
                                                    ;to normalize
```

```
norm dvsr:
        test        word ptr [di][6], 8000h
        jne         norm dvdnd                  ;we want to keep
        shl         word ptr [di][0], 1         ;the divisor
        rcl         word ptr [di] [2], 1        ;truly normalized
        rcl         word ptr [di] [4], 1        ;for maximum
        rcl         word ptr [di] [6], 1        ;precision
        jmp         norm_dvsr                   ;this should normalize dvsr

norm dvdnd:
        cmp         bl, 4h                      ;bx still contains pointer
                                                ;to dvsr
        jbe         chk 2
        add         cl, 10h                     ;adjust for word
        jmp         ready_dvdnd
chk_2:
        cmp         bl, 2h
        jae         ready_dvdnd
        sub         cl, 10h                     ;adjusting again for size
                                                ;of shift

ready_dvdnd:
        lea         di, word ptr dvdnd
        or          cl, cl
        je          makedelta                   ;no adjustment necessary
        or          cl, cl
        jns         do_dvdnd_right
        neg         cl
        sub         ch, ch
        jmp         do_dvdnd_left

do_dvdnd_right:
        shr         word ptr [di][6], 1         ;no error on underflow
        rcr         word ptr [di][4], 1         ;unless it becomes zero,
                                                ;there may still be some
useable information
        rcr         word ptr [di][2], 1
        rcr         word ptr [di] [0], 1
        loop        do_dvdnd_right              ;this should normalize dvsr
        sub         ax, ax
        or          ax, word ptr [di][6]
        or          ax, word ptr [di][4]
        or          ax, word ptr [di][2]
        or          ax, word ptr [di][0]
```

```
        jne         setup
        mov         di, word ptr quotient
        mov         cx, 4
rep     stosw
        jmp         divmul exit                 ;if it is now a zero,
                                                ;that is the result

do_dvdnd_left
        shl         word ptr [di] [0], 1
        rcl         word ptr [di][2], 1
        rcl         word ptr [di][4], 1
        rcl         word ptr [di][6], 1
        jc          ovrflw                      ;significant bits
                                                ;shifted out
                                                ;data unusable
        loop        do dvdnd_left               ;this should normalize dvsr

setup:
        mov         si, di
        mov         di, word ptr quotient
        mov         cx, 4
rep     movsw                                   ;put shifted dividend
                                                ;into quotient

makedelta:                                      ;this could be done with
                                                ;a table
        lea         si, word ptr dvsr
        lea         di, word ptr delta
        mov         cx, 4
rep     movsw                                   ;move normalized dvsr
                                                ;into delta

        not         word ptr delta[6]
        not         word ptr delta[4]
        not         word ptr delta[2]
        neg         word ptr delta              ;attempt to develop with
                                                ;2's complement
        jc          mloop
        add         word ptr delta[2], 1
        adc         word ptr delta[4], 0
        adc         word ptr delta[6], 0

mloop:
        invoke      mu164, delta, dvsr, addr temp
```

```
        lea         si, word ptr temp[8]
        lea         di, word ptr tmp
        mov         cx, 4
rep     movsw

        invoke      add64, tmp, dvsr, addr dvsr

        lea         di, word ptr divisor
        mov         si, word ptr quotient
        mov         cx, 4
rep     movsw
        invoke      mu164, delta, divisor, addr temp

        sub         ax, ax
        cmp         word ptr temp[6], 8000h        ;an attempt to round
        jb          no_round                       ;.5 or above rounds up
        add         word ptr temp[8], 1
        adc         word ptr temp[l0], ax
        adc         word ptr temp[l2], ax
        adc         word ptr temp[l4], ax
no_round:

        lea         si, word ptr temp[8]
        lea         di, word ptr tmp               ;double duty
        mov         cx, 4
rep     movsw
        invoke      add64, divisor, tmp, quotient

        dec         byte ptr lp
        je          divmul_exit
        jmp         makedelta                      ;six passes for 64 bits

ovrflw:
        sub         ax, ax
        not         ax
        mov         cx, 4
        mov         di, word ptr quotient
rep     stosw                                      ;make infinite answer
        jmp         divmul exit

divmul_exit:
        popf
        ret
divmul              endp
```

1. Van Aken, Jerry, and Ray Simar. *A Conic Spline Algorithm.* Texas Instruments, 1988.
2. Van Aken, Jerry, and Carrel Killebrew Jr. *The Bresenham Line Algorithm.* Texas Instruments, 1988.
3. Cavanagh, Joseph J. F. *Digital Computer Arithmetic.* New York, NY: McGraw-HillBook Co., 1984, Pages 278-284. Also see Knuth, D. E. *Seminumerical Algorithms.* Reading, MA: Addison-Wesley Publishing Co., 1981, Pages. 295-297.
4. Cavanagh, Joseph J. F. *Digital Computer Arithmetic.* New York, NY: McGraw-HillBook Co., 1984, Pages 284-289.

CHAPTER 4

Floating-Point Arithmetic

Floating-point libraries and packages offer the software engineer a number of advantages. One of these is a complete set of transcendental functions, logarithms, powers, and square-root and modular functions. These routines handle the decimal-point placement and housekeeping associated with arithmetic and even provide some rudimentary handles for numerical exceptions.

For the range of representable values, the IEEE 754 standard format is compact and efficient. A single-precision real requires only 32 bits and will handle a decimal range of 10^{38} to 10^{-38}, while a double-precision float needs only 64 bits and has a range of 10^{308} to 10^{-308}. Fixed-point representations of the same ranges can require a great deal more storage.

This system of handling real numbers is compact yet has an extremely wide dynamic range, and it's standardized so that it can be used for interapplication communication, storage, and calculation. It is slower than fixed point, but if a math coprocessor is available or the application doesn't demand speed, it can be the most satisfactory answer to arithmetic problems.

The advantages of floating point do come with some problems. Floating-point libraries handle so much for the programmer, quietly and automatically generating 8 to 15 decimal digits in response to input arguments, that it's easy to forget that those digits may be in error. After all, floating point is just fixed point wrapped up with an exponent and a sign; it has all the proclivities of fixed point to produce erroneous results due to rounding, loss of significance, or inexact representation. But that's true of any form of finite expression-precautions must always be taken to avoid errors. Floating-point arithmetic is still a valuable tool, and you can use it safely if you understand the limitations of arithmetic in general and of floating-point format, in particular.

What To Expect

Do you know what kind of accuracy your application needs? What is the accuracy of your input? Do you require only slide rule accuracy for fast plotting to screen? Or do you need the greatest possible accuracy and precision for iterative or nonlinear calculation?

These are important questions, and their answers can make the difference between calculations that succeed and those that fail. Here are a few things to keep in mind when using floating-point arithmetic.

- No mathematical operation produces a result more accurate than its weakest input. It's fine to see a string of numbers following a decimal point, but if that's the result of multiplying pi by a number accurate to two decimal places, you have two decimal places of accuracy at best.
- Floating point suffers from some of the very conveniences it offers the developer. Though most floating-point libraries use some form of extended precision, that's still a finite number of significant bits and may not be enough to represent the input to or result of a calculation. In an iterative loop, you can lose a bit more precision each time through an operation, this is especially true of subtraction.
- Floating point's ability to cover a wide range of values also leads to inaccuracies. Again, this is because the number of significant bits is finite: 24 for a short real and 53 for a long real. That means a short real can only represent 2^{23} possible combinations for *every power of two.*

  To get the greatest possible precision into the double- and quadword formats of the short and long real, the integer 1 that must always exist in a number coerced to a value between 1.0 and 2.0 is omitted. This is called the *hidden bit,* and using its location for the LSB of the exponent byte allows an extra bit of precision. Both single and double-precision formats include the hidden bit.

  Between 2^1(2D) and 2^2 (4D), 2^{23} individual numbers are available in a short real. That leaves room for two cardinals (counting numbers, such as 1, 2, and 3) and a host of fractions-not an infinite number, as the number line allows, but still quite a few. These powers of two increase (1, 2, 4, 8...) and decrease (.5, .25, .125...), but the number of significant bits remains the same (except for denormal arithmetic); each power of two can only provide 2^{23} individual numbers. This means that between two consecutive powers of two, such as 2^{32} and 2^{33}, on the number line are 4,294,967,296 whole numbers and an infinite number of fractions thereof. However, a single-precision float will only represent 2^{23} unique values. So what happens if your result isn't one of those numbers? It becomes one of those that 23 bits can represent.

Around 0.0 is a band that floating-point packages and coprocessors handle differently. The smallest legal short real has an exponent of 2^{-126}. The significand is zero, with only the implied one remaining (the hidden bit). That still leaves 8,388,607 numbers known as *denormals* to absolute zero. As the magnitude of these numbers decreases from 2^{-126} to 2^{-149}, the implied one is gone and a bit of precision is lost for every power of two until the entire significand is zero. This is described in the IEEE 854 specification as "gradual underflow" and isn't supported by all processors and packages. Use caution when using denormals; multiplication with them can result in so little significance that it may not be worth continuing, and division can blow up.

- It's easy to lose significance with floating-point arithmetic, and the biggest offender is subtraction. Subtracting two numbers that are close in value can remove most of the significance from your result, perhaps rendering your result meaningless as well. The lost information can be estimated according to the formula *Significance_lost = -ln(1 -minuend/subtrahend)/ln(2),* but this is of little value after the loss.'

Assume for a moment that you're using a floating-point format with seven significant decimal digits (a short real). If you subtract. 1234567 from .1234000, the result is -5.67E-5. You have lost four decimal digits of significance. Instances of such loss are common in function calls involving transcendentals, where the operands are between 0.0 and 1.0.

This loss of significance can occur in other settings as well, such as those that involve modularity. Sines and cosines have a modularity based on $\pi/2$ or 90 degrees. Assuming a computation of harmonic displacement, x = L $\sin(\omega t)$, if ωt gets very large, which can happen if we are dealing with a high enough frequency or a long enough time, very little significance will be left for calculating the sine. The equation x = L sin$(\omega t), \omega = 2\pi f$, , with *f* being frequency and *t* being time, will calculate the angular displacement of a reference on a sinusoid in a given period of time. If the frequency of the sinusoid is 10 MHz and the time is 60 seconds, the result is an ωt of 3769911184.31. If this is expressed as a short real without extended precision, however, we'll have only enough bits to express the eight most significant digits (3.769911lE9). The very information necessary to compute the sine and quadrant is truncated. The sine of 3769911184.31 is 2.24811195116E-3, and the sine of 3.769911lE9 is -.492752198651. Computing with long reals will help, but it's limited to 15 decimal digits.

- The normal associative laws of addition and multiplication don't always function as expected with floating point. The associative law states:

```
(A+B)+C = A+(B+C)
```

Look at the following seven-digit example of floating point:

```
(7.654321 + (-1234567)) + 1234568 =
        -1234559 + 1234568 =
                 9
```

while

```
7.654321 + (-1234567 + 1234568) =
        7.654321 + 1 =
           8.654321
```

Note that the results aren't the same. Of course, using double-precision arguments will minimize such problems, but it won't eliminate them. The number of significant bits available in each precision heavily affects the accuracy of what you're representing.

- It is hard to find any true equalities in floating-point arithmetic. It's nearly impossible to find any exactitudes, without which there can be no equalities. D. E. Knuth suggested a new set of symbols for floating point.[2] These symbols were "round" (because the arithmetic was only approximate) and included a round plus, a round minus, a round multiply, and a round divide. Following from that set was a floating-point compare that assessed the relative values of the numbers. These operations included conditions in which one number was definitely less than, approximately equal to, or definitely greater than another.

Most floating-point packages compare the arguments exactly. This usually works in greater-than/less-than situations, depending on the amount of significance left in the number, but almost never works in equal-to comparisons.

What this means is that the results of a computation depend on the precision and range of the input and the kind of arithmetic performed. When you prepare operations and operands, make sure the precision you choose is appropriate.

Though single-precision arguments will be used in this discussion, the same problems exist in double precision.

A Small Floating-Point Package

The ability to perform floating-point arithmetic is a big advantage. Without a coprocessor to accelerate the calculations it may never equal fixed points for speed, but the automatic scaling, convenient storage, standardized format, and the math routines are nice to have at your fingertips. Unfortunately, even in systems where speed isn't a problem, code size can make the inclusion of a complete floating-point package impossible.

Your system may not require double-precision support-it might not need the trigonometric or power functions—but could benefit from the ability to input and process real-world numbers that fixed point can't handle comfortably. Unfortunately, most packages are like black boxes that require the entire library or nothing; this is especially true of the more exotic processors that have little third-party support. It's hard to justify an entire package when only a few routines are necessary. At times like this, you might consider developing your own.

The rest of this chapter introduces the four basic arithmetic operations in floating point. The routines do not conform to IEEE 754 in all ways-most notably the numeric exceptions, many of which are a bit dubious in an embedded application—

but they do deliver the necessary accuracy and resolution and show the inner workings of floating point. With the routines presented later in the book, they're also the basis for a more complete library that can be tailored to the system designer's needs.

The following procedures are single-precision (short real) floating-point routines written in 80x86 assembler, small model, with step-by-step pseudocode so you can adapt them to other processors.

Only the techniques of addition, subtraction, multiplication, and division will be described in this chapter; refer to FPMATH.ASM for complementary and support functions.

The Elements of a Floating-Point Number

To convert a standard fixed-point value to one of the two floating-point formats, you must first *normalize* it; that is, force it through successive shifts to a number between 1.0 and 2.0. (Note that this differs from the normalization described earlier in the fixed-point routines, which involved coercing the number to a value greater than or equal to one-half and less than one. This results in a representation that consists of: a sign, a number in fixed-point notation between 1.0 and 2.0, and an exponent representing the number of shifts required. Mathematically, this can be expressed as[3]

$$\sum_{k=1}^{24} f_k * 2^{-k}$$

for $-125 \leq \text{exponent} \leq 128$ in single precision and

$$\text{sign} * 2^{\text{exponent}} * \sum_{k=1}^{53} f_k * 2^{-k}$$

for $-1{,}021 \leq \text{exponent} \leq 1{,}024$ in double precision.

Since the exponent is really the signed (int) $\log_2$ of the number you're representing, only numbers greater than 2.0 or less than 1.0 have an exponent other than zero and require any shifts at all. Very small numbers (less than one) must be normalized using left shifts, while large numbers with or without fractional extensions require right shifts. As the number is shifted to the right, the exponent is incremented; as the number is shifted to the left, the exponent is decremented.

The IEEE 754 standard dictates that the exponent take a certain form for some errors and for zero. For a Not a Number (NAN) and infinity, the exponent is all ones; for zero, it is all zeros. For this to be the case, the exponent is biased—127 bits for single precision, 1023 for double precision.

Figure 4-l shows the components of a floating-point number: a single bit representing the sign of the number (signed magnitude), an exponent (8 bits for single precision and 11 bits for double precision, and a mantissa (23 bits for single precision, 52 bits for double).

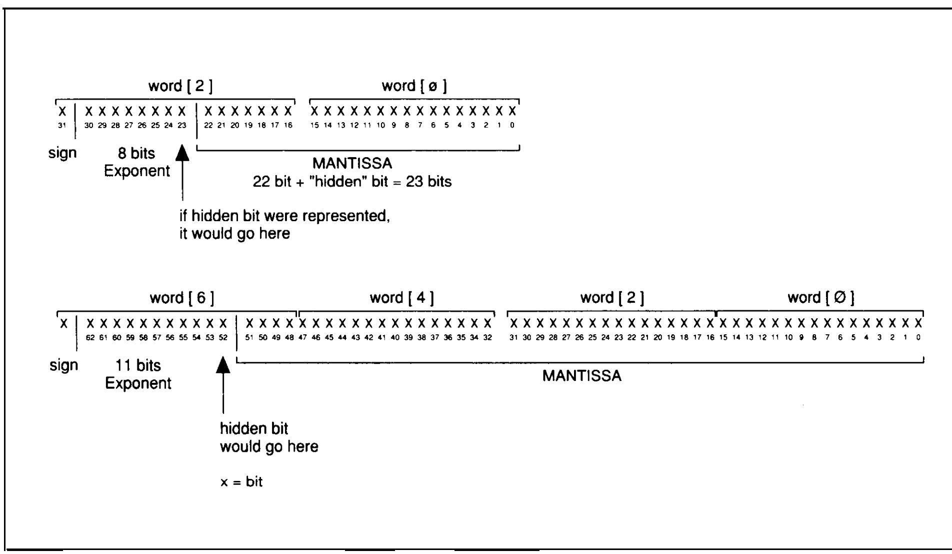


Figure 4-1. Single and double-precision floating-point numbers.

Let's look at an example of a single-precision float. The decimal number 14.92 has the following binary fixed-point representation (the decimal point is shown for clarity):

```
1110.11101011100001010010
```

We need three right shifts to normalize it:

$$1.11011101011100001010010 \times 2^3$$

We add 127D to the exponent to make it 130D (127 + 3):

```
10000010B
```

Because this is a positive number, the sign bit is 0 (the bit in parentheses is the hidden bit):

```
0+10000010+(1)1101110l0lll0000l0l00l0B
```

or

```
416eb852H
```

The expression of the fractional part of the number depends on the precision used. This example used 24 bits to conform to the number of bits in the single-precision format. If the number had been converted from a 16-bit fixed-point word, the single-precision version would be 416eb000H. Note the loss of significance.

Retrieving the fixed-point number from the float is simply a matter of extracting the exponent, subtracting the bias, restoring the implied leading bit, and performing the required number of shifts. The bandwidth of the single-precision float if fairly high-approximately 3.8 db—so having a data type to handle this range would require more than 256 bits. Therefore we need some restrictions on the size of the fixed-point value to which we can legally convert. (For more information, refer to Chapter 5 .)

Extended Precision

If all floating-point computations were carried out in the precision dictated by the format, calculations such as those required by a square-root routine, a polynomial evaluation, or an infinite series could quickly lose accuracy. In some cases, the results would be rendered meaningless. Therefore, IEEE 754 also specifies an extended format for use in intermediate calculations.[3] This format increases both the number of significant bits and the exponent size. Single-precision extended is increased from 24 significant bits to at least 32, with the exponent increased to at least 11 bits. The number of significant bits for double precision can be greater than 79 and the exponent equal to or greater than 15 bits.

This extended precision is invisible to users, who benefit from added accuracy in their results. Those results are still in the standard single or double-precision format, necessitating a set of core routines (using extended precision) that are generally unavailable to the normal user. Another set of routines is needed to convert standard format into extended format and back into standard format, with the results rounded at the end of a calculation.

The routines described later in this chapter take two forms. Some were written, for debugging purposes, to be called by a higher-level language (C); they expect single-precision input and return single-precision output. They simply convert to and from single precision to extended format, passing and receiving arguments from the core routines. These external routines have names that begin with fp_, such as *fp_mul*. The core routines operate only with extended precision and have names beginning with *fl,* such as *flmul;* these routines cannot be called from C.[4]

The extended-precision level in these routines uses a quadword for simple parameter passing and offers at least a 32-bit significand. This simplifies the translation from extended to standard format, but it affords less immunity to loss of significance at the extremes of the single precision floating range than would a greater number of exponent bits. If your application requires a greater range, make the exponent larger—15-bits is recommended-in the higher level routines before passing the values to the core routines. This can actually simplify exponent handling during intermediate calculations.

Use the core routines for as much of your work as you can; use the external routines when the standard format is needed by a higher-level language or for communications with another device. An example of a routine, *cylinder*, that uses these core routines to compute the volume of a cylinder appears in the module FPMATH.ASM, and many more appear in TRANS.ASM.

The External Routines

This group includes the basic arithmetic procedures—*fp_mul, fp_div, fp_add*, and *fp_sub*. Written as an interface to C, they pass arguments and pointers on the stack and write the return values to static variables.

In *fp_add*, two single-precision floating-point numbers and a pointer to the result arrive on the stack. Local variables of the correct precision are created for each of the floats, and memory is reserved for the extended result of the core routines. A quadword is reserved for each of the extended variables, including the return; the single-precision float is written starting at the second word, leaving the least significant word for any extended bits that result from intermediate calculations.

After the variables are cleared, the doubleword floats are written to them and the core routine, *fladd*, is called. Upon return from *fladd*, the routine extracts the single-precision float (part of the extended internal float) from the result variable, rounds it, and writes it out with the pointer passed from the calling routine.

fp_add: Algorithm

1. Allocate and clear storage for three quadwords, one for each operand and one for the extended-precision result.
2. Align the 32-bit operands within the extendedvariables so that the least significant byte is at the boundary of the second word.
3. Invoke the core addition routine with both operands and a pointer to the quadword result.
4. Invoke the rounding routine with the result of the previous operation and a pointer to that storage.
5. Pull the 32-bit float out of the extended variable, write it to the static variable, and return.

fp-add: Listing

```
; *****
;
fp_add  proc  uses bx cx dx si di,
                      fp0:dword, fpl:dword, rptr:word
        local         flp0:qword, flpl:qword, result:qword
        pushf
        cld
        xor           ax,ax
        lea           di,word ptr result
        mov           cx,4
  rep   stosw

        lea           di,word ptr flp0
        mov           cx,4
  rep   stosw

        lea           di,word ptr flpl                  ;clear variables for
                                                        ;the core routine
        mov           cx,4
  rep   stosw

        lea           si,word ptr fp0
        lea           di,word ptr flp0[2]
        mov           cx,2
  rep   movsw
        lea           si,word ptr fpl
        lea           di,word ptr flp1[2]               ;align the floats
                                                        ;within the extended
                                                        ;variables
        mov           cx,2
  rep   movsw
        invoke        fladd, flp0, flpl, addr result    ;do the add
        invoke        round, result, addr result        ;round the result
        lea           si, word ptr result[2]            ;make it a standard float

        mov           di,rptr
        mov           cx,2
  rep   movsw
        popf
        ret
fp_add  endp
```

This interface is consistent throughout the external routines. The prototypes for these basic routines and *fp_comp* are:

```
fp_add  proto  c  fp0:dword,  fpl:dword,  rptr:word
fp_sub  proto  c  fp0:dword,  fpl:dword,  rptr:word
fp_mu1  proto  c  fp0:dword,  fpl:dword,  rptr:word
fp_div  proto  c  fp0:dword,  fpl:dword,  rptr:word
fp_camp proto c fp:dword, fpl:dword
```

Fp_comp compares two floating-point values and returns a flag in AX specifying whether *fp0* is greater than *fpl* (1), equal to *fpl* (0), or less than *fpl* (-1). The comparison assumes the number is rounded and does not include extended-precision bits. (FPMATH.ASM contains the other support routines.)

The Core Routines

Because these routines must prepare the operands and pass their arguments to the appropriate fixed-point functions, they're a bit more complex and require more explanation. They disassemble the floating-point number, extract the exponent, and align the radix points to allow a fixed-point operation to take place. They then take the results of these calculations and reconstruct the float.

The basic arithmetic routines in this group include:

```
fladd    proto      flp0:qword,  flpl:qword,  rptr:word
     -flp0 is addend0; flpl is addend1

flsub    proto      flp0:qword,  flpl:qword,  rptr:word
     -flp0 is the minuend; flpl is the subtrahend

flmul    proto      flp0:qword,  flpl:qword,  rptr:word
     -flp0 is the multiplicand; flpl is the multiplier

fldiv    proto      flp0:qword,  flpl:qword,  rptr:word
     -flp0 is the dividend; flpl is the divisor
```

For pedagogical and portability reasons, these routines are consistent in terms of how they prepare the data passed to them.

Briefly, each floating-point routine must do the following:

1. Set up any variables required for the arguments that are passed and for the results of the current computations.
2. Check for initial errors and unusual conditions.
   - Division:

     divisor == zero: return divide by zero error

     divisor == infinite: return zero

     dividend == zero: return infinity error

     dividend == infinite: return infinite

     dividend == divisor: return one
   - Multiplication:

     either operand == zero: return zero

     either operand == infinite: return infinite
   - Subtraction:

     minuend == zero: do two's complement of subtrahend

     subtrahend == zero: return minuend unchanged

     operands cannot align: return largest with appropriate sign
   - Addition:

     either addend == zero: return the other addend unchanged

     operands cannot align: return largest
3. Get the signs of the operands. These are especially useful in determining what action to take during addition and subtraction.
4. Extract the exponents, subtracting the bias. Perform whatever handling is required by that procedure. Calculate the approximate exponent of the result.
5. Get the mantissa.
6. Align radix points for fixed-point routines.

7. Perform fixed-point arithmetic.
8. Check for initial conditions upon return. If a zero is returned, this is a shortcut exit from the routine.
9. Renormalize, handling any underflow or overflow.
10. Reassert the sign.
11. Write the result and return.

Fitting These Routines to an Application

One of the primary purposes of the floating-point routines in this book is to illustrate the inner workings of floating-point arithmetic; they are not assumed to be the best fit for your system. In addition, all the routines are written as *near* calls. This is adequate for many systems, but you may require *far* calls (which would require *far* pointers for the arguments). The functions write their return values to static variables, an undesireable action in multithreaded systems because these values can be overwritten by another thread. Though the core routines use extended precision, the exponents are not extended; if you choose to extend them, 15 bits are recommended. This way, the exponent and sign bit can fit neatly within one word, allowing as many as 49 bits of precision in a quadword format. The exceptions are not fully implemented. If your system needs to detect situations in which the mathematical operation results in something that cannot be interpreted as a number, such as Signaling or Quiet NANS, you will have to write that code. Many of the in-line utility functions in the core and external routines may also be rewritten as stand alone subroutines. Doing so can make handling of the numerics a bit more complex but will reduce the size of the package.

These routines work well, but feel free to make any changes you wish to fit your target. A program on the disk, MATH.C, may help you debug any modifications; I used this technique to prepare the math routines for this book.

Addition and Subtraction: FLADD

Fladd, the core routine for addition and subtraction, is the longest and most complex routine in FPMATH.ASM (and perhaps the most interesting). We'll use it as an example, dissecting it into the prologue, the addition, and the epilogue.

The routine for addition can be used without penalty for subtraction because the sign in the IEEE 754 specification for floating point is signed magnitude. The MSB of the short or long real is a 1 for negative and a 0 for positive. The higher-level subtraction routine need only *XOR* the MSB of the subtrahend before passing the parameters to *fladd* to make it a subtraction.

Addition differs from multiplication or division in at least two respects. First, one operand may be so much smaller than the other that it will contribute no significance to the result. It can save steps to detect this condition early in the operation. Second, addition can occur anywhere in four quadrants: both operands can be positive or both negative, the summend can be negative, or the addend can be negative.

The first problem is resolved by comparing the difference in the exponents of the two operands against the number of significant bits available. Since these routines use 40 bits of precision, including extended precision, the difference between the exponents can be no greater than 40. Otherwise no overlap will occur and the answer will be the greater of the two operands no matter what. (Imagine adding .00000001 to 100.0 and expressing the result in eight decimal digits). Therefore, if the difference between the exponents is greater than 40, the larger of the two numbers is the result and the routine is exited at that point. If the difference is less than 40, the smaller operand is shifted until the exponents of both operands are equal.

If the larger of the two numbers is known, the problem of signs becomes trivial. Whatever the sign of the larger, the smaller operand can never change it through subtraction or addition, and the sign of the larger number will be the sign of the result. If the signs of both operands are the same, addition takes place normally; if they differ, the smaller of the two is two's complemented before the addition, making it a subtraction.

The *fladd* routine is broken into four logical sections, so each part of the operation can be explained more clearly. Each section comprises a pseudocode description followed by the actual assembly code listing.

FLADD: The Prologue.

1. Two quadword variables, *opa* and *opb*, are allocated and cleared for use later in the routine. Byte variables for the sign of each operand and a general sign byte are also cleared.
2. Each operand is checked for zero.

   If either is zero, the routine exits with the other argument as its answer.
3. The upper word of each float is loaded into a register and shifted left once into the sign byte assigned to that operand. The exponent is then moved to the exponent byte of that operand, *exp0* and *expl*. Finally, the exponent of the second operand is subtracted from the exponent of the first and the difference placed in a variable, *diff*.
4. The upper words of the floats are *AND*ed with 7fH to clear the sign and exponent bits. They're then *OR*ed with 80H to restore the hidden bit.

We now have a fixed-point number in the form 1.xxx.

FLADD: The Prologue

```
; *****
;
;
fladd   proc   uses bx cx dx si di,
                    fp:qword,  fpl:qword,  rptr:word
        local       opa:qword, opb:qword, signa:byte,
        signb:byte, exponent:byte, sign:byte,
                    diff:byte, sign0:byte, sign1:byte,
                    exp0:byte, exp1:byte
        pushf
        std                                        ;decrement
        xor         ax,ax                          ;clear appropriate  variables
        lea         di,word ptr opa[6]             ;larger  operand
        mov         cx,4
rep     stosw       word ptr [di]
        lea         di,word ptr opb[6]             ;smaller  operand
        mov         cx,4
rep     stosw       word ptr [di]
        mov         byte ptr sign0, al
        mov         byte ptr sign1, al
        mov         byte ptr sign, al              ;clear sign
```

```
chk_fp0:
        mov         cx, 3                       ;check for zero
        lea         di,word ptr fp0[4]
        repe        scasw                       ;di will point to the first
nonzero
        jnz         chk_fpl
        lea         si,word ptr fp1[4]          ;return other addend
        jmp         short leave with other

chk_fpl:
        mov         cx, 3
        lea         di,word ptr fp1[4]
repe                scasw                       ;di will point to the
                                                ;first nonzero
        jnz         do add
        lea         si,word ptr fp0[4]          ;return other addend

; *****
leave with other:
        mov         di,word ptr rptr
        add         di,4
        mov         cx,3
rep     movsw
        jmp         fp_addex

; *****
do_add:
        lea         si,word ptr fp0
        lea         bx,word ptr fpl
        mov         ax,word ptr [si][4]         ;fp0
        shl         ax,1                        ;dump the sign
        rcl         byte ptr sign0, 1           ;collect the sign
        mov         byte ptr exp0, ah           ;get the exponent
        mov         dx,word ptr [bx][4]         ;fpl
        shl         dx,l                        ;get sign
        rcl         byte ptr sign1, 1
        mov         byte ptr exp1, dh           ;and the exponent
        sub         ah, dh
        mov         byte ptr diff, ah           ;and now the difference

restore-missing-bit:                            ;set up operands
        and         word ptr fp0[4], 7fh
        or          word ptr fp0[4], 80h
```

```
mov     ax, word ptr fpl        ;load these into registers;
                                ;we'll use them
mov     bx, word ptr fp1[2]
mov     dx, word ptr fp1[4]
and     dx,7fh
or      dx,80h
mov     word ptr fp1[4], dx
```

The FLADD Routine:

5. Compare the difference between the exponents.

   If they're equal, continue with step 6.

   If the difference is negative, take the second operand as the largest and continue with step 7.

   If the difference is positive, assume that the first operand is largest and continue with step 8.

6. Continue comparing the two operands, most significant words first.

   If, on any compare except the last, the second operand proves the largest, continue with step 7.

   If, on any compare except the last, the first operand proves the largest, continue with step 8.

   If neither is larger to the last compare, continue with step 8 if the second operand is larger and step 7 if the first is equal or larger.

7. Two's-complement the variable *diff* and compare it with 40D to determine whether to go on.

   If it's out of range, write the value of the second operand to the result and leave.

   If it's in range, move the exponent of the second operand to exponent, move the sign of this operand to the variable holding the sign of the largest operand, and move the sign of the other operand to the variable holding the sign of the smaller operand.

   Load this fixed-point operand into *opa* and continue with step 9.

8. Compare *diff* with 40D to determine whether it's in range.

   If not, write the value of the first operand to the result and leave.

   If so, move the exponent of the first operand to *exponent,* move the sign of this operand to the variable holding the sign of the largest operand, and move the sign of the other operand to the variable holding the sign of the smaller operand. Load this fixed-point operand into *opa* and continue with step 9.

The FLADD Routine: Which Operand is Largest?

```
find_largest:
        cmp         byte ptr diff,0
        je          cmp_rest
        test        byte ptr diff,80h          ;test for negative
        je          numa_bigger
        jmp         short numb_bigger

cmp_rest:
        cmp         dx, word ptr fp0[4]
        ja          numb_bigger                ;if above
        jb          numa_bigger                ;if below
        cmp         bx, word ptr fp0[2]
        ja          numb_bigger
        jb          numa_bigger
        cmp         ax, word ptr fp0[0]
        jb          numa_bigger
                                               ;defaults to numb
numb_bigger:
        sub         ax, ax
        mov         al,byte ptr diff
        neg         al
        mov         byte ptr diff,al           ;save difference
        cmp         al,60                      ;do range test
        jna         in range

; *****
                                               ;this is an exit!!!!!
        lea         si,word ptr fp1[6]         ;this is a range error
leave_with_largest:                            ;operands will not
                                               ;line up
        mov         di,word ptr rptr           ;for a valid addition
        add         di,6                       ;leave with largest
                                               ;operand
        mov         cx,4                       ;that is where the
rep     movsw                                  ;significance is anyway
        jw          fp_addex
range_errora:
        lea         si,word ptr fp0[6]
        jmp         short leave_with_largest


; *****
```

```
in range:
        mov         al,byte ptr expl
        mov         byte ptr exponent,al       ;save exponent of largest
                                               ;value
        mov         al, byte ptr sign1
        mov         signa, al
        mov         al, byte ptr sign0
        mov         byte ptr signb, al
        lea         si, word ptr fp1[6]        ;load opa with largest operand
        lea         di,word ptr opa[6]
        mov         cx,4
rep     movsw

signb_positive:
        lea         si, word ptr fp0[4]        ;set to load opb
        jmp         shift_into_position

numa_bigger:
        sub         ax, ax
        mov         al,byte ptr diff
        cmp         al,60
        jae         range errora               ;do range test
        mov         al,byte ptr exp0
        mov         byte ptr exponent,al       ;save exponent of largest value
        mov         al, byte ptr sign1
        mov         byte ptr signb, al
        mov         al, byte ptr sign0
        mov         byte ptr signa, al
        lea         si, word ptr fp0[6]        ;load opa with largest
                                               ;operand
        lea         di,word ptr opa[6]
        mov         cx,4
rep     movsw
        lea         si, word ptr fp1[4]        ;set to load opb
```

The FLADD Routine: Aligning the Radix Points.

9. Divide *diff* by eight to determine how many bytes to shift the smaller operand so it aligns with the larger operand.

   Adjust the remainder to reflect the number of bits yet to be shifted, and store it in AL.

   Subtract the number of bytes to be shifted from a maximum of four and add this to a pointer to *opb.* That gives us a starting place to write the most significant word of the smaller operand (we're writing downward).

10. Write as many bytes as will fit in the remaining bytes of *opb.* Move the adjusted remainder from step 9 to CL and test for zero.

    If the remainder is zero, no more shifting is required; continue with step 12.

    Otherwise, continue at step 11.

11. Shift the smaller operand bit by bit until it's in position.

12. Compare the signs of the larger and smaller operands.

    If they're the same, continue with step 14.

    If the larger operand is negative, continue with step 13.

    Otherwise, subtract the smaller operand from the larger and continue with step 15.

13. Two's-complement the larger operand.

14. Add the smaller operand to the larger and return a pointer to the result.

The FLADD Routine: Aligning the Radix Point

```
shift_into_position:                          ;align operands
       xor          ax,ax
       mov          bx,4
       mov          cl,3
       mov          ah,byte ptr diff
       shr          ax,cl                     ;ah contains # of bytes,
                                              ;a1 # of bits
       mov          cx,5h
       shr          al,cl
       sub          bl,ah                     ;reset pointer below initial
                                              ;zeros
       lea          di,byte ptr opb
       add          di,bx
       mov          cx,bx
       inc          cx
load_operand:
       movsb
       loop         load_operand
       mov          cl,al
       xor          ch,ch
       or           cx, cx
```

```
        je          end shift
shift_operand:
        shr         word ptr opb[6],1
        rcr         word ptr opb[4],1
        rcr         word ptr opb[2],1
        rcr         word ptr opb[0],l
        loop        shift_operand
end_shift:
        mov         al, byte ptr signa
        cmp         al, byte ptr signb
        je          just_add                    ;signs alike, just add
opb_negative:
        not         word ptr opb[6]             ;do two's complement
        not         word ptr opb[4]
        not         word ptr opb[2]
        neg         word ptr opb[0]
        jc          just_add
        adc         word ptr opb[2],0
        adc         word ptr opb[41,0
        adc         word ptr opb[6],0

just_add:
        invoke      add64, opa, opb, rptr
```

FLADD: The Epilogue.

15. Test the result of the fixed-point addition for zero. If it's zero, leave the routine and write a floating-point zero to output.

16. Determine whether normalization is necessary and, if so, whichdirection to shift.

    If the most significant word of the result is zero, continue with step 18.

    If the MSB of the most significant word is zero, continue with step 17.

    If the MSB of the second most significant byte of the result (the hidden bit) isn't set, continue with step 18.

    Otherwise, no shifting is necessary; continue with step 19.

17. Shift the result right, incrementing the exponent as you go, until the second most significant byte of the most significant word is set. This will be the hidden bit. Continue with step 19.

18. Shift the result left, decrementing the exponent as you go, until the second most significant byte of the most significant word is set. This

will be the hidden bit. Continue with step 19.

19. Shift the most significant word left once to insert the exponent. Shift it back when you're done, then or in the sign.

20. Write the result to the output and return.

FLADD: The Epilogue

```
handle_sign:
        mov         si, word ptr rptr
        mov         dx, word ptr [si][4]
        mov         bx, word ptr [si][2]
        mov         ax, word ptr [si][0]

norm:
        sub         cx,cx
        cmp         ax, cx
        jne         not_zero
        cmp         bx,cx
        jne         not_zero
        cmp         dx,cx
        jne         not_zero
        jmp         write_result                ;exit with a zero
not_zero:
        mov         cx,64
        cmp         dx,0h
        je          rotate_result_left
        cmp         dh,00h
        jne         rotate_result_right
        test        d1,80h
        je          rotate_result_left
        jmp         short_done_rotate
rotate_result_right:
        shr         dx,1
        rcr         bx,l
        rcr         ax,1
        inc         byte ptr exponent           ;decrement exponent with
                                                ;each shift
        test        dx,0ff00h
        je          done rotate
        loop        rotate result right
rotate_result_left:
        shl         ax,1
        rcl         bx,l
```

```
        rcl         dx,1
        dec         byte ptr exponent           ;decrement exponent with
                                                ;each shift
        test        dx,80h
        jne         done rotate
        loop        rotate result left
done rotate:
        and         dx,7fh
        shl         dx 1
        or          dh, byte ptr exponent       ;insert exponent
        shr         dx, 1
        mov         cl, byte ptr sign           ;sign of result of
                                                ;computation
        or          cl, cl
        je          fix sign
        or          dx,8000h
fix-sign:
        mov         cl, byte ptr signa          ;sign of larger operand
        or          cl, cl
        je          write result
        or          dx,80;0h
write result:
        mov         di,word ptr rptr
        mov         word  ptr [di],ax
        mov         word ptr [di][2],bx
        mov         word ptr [di][4],dx
        sub         ax,ax
        mov         word ptr [di][6],ax
fp_addex:
        popf
        ret
fladd endp
```

At the core of this routine is a fixed-point subroutine that actually does the arithmetic. Everything else involves extracting the fixed-point numbers from the floating-point format and aligning. *Fladd* calls *add64* to perform the actual addition. (This is the same routine described in Chapter 2 and contained in the FXMATH.ASM listing in Appendix C and included on the accompanying disk). It adds two quadword variables passed on the stack with multiprecision arithmetic and writes the output to a static variable, *result.*

Multiplication and Division

One minor difference between the multiplication and division algorithms is the error checking at the entry point. Not only are zero and infinity tested in the prologue to division as they are in the prologue to multiplication, we also check to determine whether the operands are the same. If they are identical, a one is automatically returned, thereby avoiding an unnecessary operation.

Floating point treats multiplication and division in a manner similar to logarithmic operations. These are essentially the same algorithms taught in school for doing multiplication and division with logarithms except that instead of log,,, these routines use $\log_2$. (The exponent in these routines is the log, of the value being represented.) To multiply, the exponents of the two operands are added, and to divide the difference between the exponents is taken. Fixed point arithmetic performs the multiplication or division, any overflow or underflow is handled by adjusting the exponents, and the results are renormalized with the new exponents. Note that the values added and subtracted as the biases aren't exactly 127D. This is because of the manner in which normalization is accomplished in these routines. Instead of orienting the hidden bit at the MSB of the most significant word, it is at the MSB of the penultimate byte (in this case DL). This shifts the number by 8 bits, so the bias that is added or subtracted is 119D.

FLMUL

The pseudocode for floating point multiplication is as follows:

flmul: Algorithm

1. Check each operand for zero and infinity.

   If one is found to be infinite, exit through step 9.

   If one is found to be zero, exit through step 10.

2. Extract the exponent of each operand, subtract 77H (119D) from one of them, and add to form the approximate exponent of the result.

3. Test each operand for sign and set the *sign* variable accordingly.

4. Restore the hiddenbit in eachoperand. Now each operand is a fixed-point number in the form 1.XXXX...

5. Multiply the two numbers with the fixed-point routine *mu164a.*
6. Check the result for zero. If it is zero, exit through step 10.
7. Renormalize the result, incrementing or decrementing the exponent at the same time. This accounts for any overflows in the result.
8. Replace the exponent, set the sign, and exit.
9. Infinity exit.
10. Zero exit.

flmul: Listing

```
; *****
;
;
flmul proc          c uses bx cx dx si di,
                              fp0:qword, fp1:qword, rptr:word

      local         result[8]:word, sign:byte, exponent:byte

      pushf
      std
      sub           ax,ax
      mov           byte ptr sign,al              ;clear sign variable
      lea           di,word ptr result[14]
      mov           cx,8
rep   stosw                                       ;and result variable

;
      lea           si,word ptr fp0               ;name a pointer to each fp
      lea           bx,word ptr fp1
      mov           ax,word ptr [si][4]
      shl           ax,1
      and           ax,0ff00h                     ;check for zero
      jne           is_a_inf
      jmp           make_zero                     ;zero exponent
is_a_inf:
      cmp           ax, 0ff00h
      jne           is b zero
      jmp           return_infinite               ;multiplicand is infinite
is_b_zero:
      mov           dx,word ptr [bx][4]
      shl           dx,1
      and           dx,0ff00h                     ;check for zero
```

```
        jnz         is b inf
        jmp         make_zero                      ;zero exponent
is_b_inf:
        cmp         dx,0ff00h
        jne         get_exp
        jmp         return infinite                ;multiplicand is infinite
;
get_exp:
        sub         ah, 77h
        add         ah, dh                         ;add exponents
        mov         byte ptr exponent,ah           ;save exponent
;
        mov         dx,word ptr [si][4]            ;set sign variable according
        or          dx, dx                         ;to msb of float
        jns         a_plus
        not         byte ptr sign
a_plus:
        mov         dx,word ptr [bx][4]            ;set sign according to msb
                                                   ;of float
        or          dx, dx
        jns         restore missing bit
        not         byte ptr sign

restore missing bit:
        and         word ptr fp0[4], 7fh           ;remove the sign and exponent
        or          word ptr fp0[4], 80h           ;and restore the hidden bit
        and         word ptr fp1[4], 7fh
        or          word ptr fp1[4], 80h

        invoke      mu164a, fp0, fpl, addr result  ;multiply with fixed point
                                                   ;routine

        mov         dx, word ptr result [10]       ;check for zeros on return
        mov         bx, word, ptr result[8]
        mov         ax, word, ptr results[6]

        sub         cx, cx
        cmp         ax,cx
        jne         not_zero
        cmp         bx,cx
        jne         not_zero
        cmp         dx,cx
        jne         not_zero
        jmp         fix_sign                       ;exit with a zero
```

```
not_zero:
        mov         cx,64                         ;should never go to zero
        cmp         dx,0h                         ;realign float
        je          rotate_result_left
        cmp         dh,00h
        jne         rotate_result_right
        test        d1,80h
        je          rotate_result_left
        jmp         short_done_rotate
rotate result right:
        shr         dx,l
                    rcr bx,l
        rcr         ax,1
        test        dx,0ff00h
        je          done_rotate
        inc         byte ptr exponent             ;decrement exponent with
                                                  ;each shift
        loop         rotate_result_right
rotate_result_left:
        shl         word ptr result[2],1
        rcl         word ptr result[4],1
        rcl         ax,1
                    rcl bx,l
        rcl         dx,l
        test        dx,80h
        jne         done rotate
        dec         byte ptr exponent             ;decrement exponent with
                                                  ;each shift
        loop        rotate result left
done_rotate:
        and         dx,7fh                        ;clear sign bit
                    shl dx, 1
        or          dh, byte ptr exponent         ;insert exponent
                    shr dx, 1
        mov         cl,byte ptr sign              ;set sign of float based on
                                                  ;sign flag
        or          cl,cl
        je          fix-sign
        or          dx,8000h
fix_sign:
        mov         di,word ptr rptr              ;write to the output
        mov         word ptr [di], ax
        mov         word ptr [di][2],bx
        mov         word ptr [di][4],dx
```

```
        sub         ax,ax
        mov         word ptr [di][6],ax
fp_mulex:
        popf
        ret
;
return  infinite:
        sub         ax, ax
        mov         bx, ax
        not         ax
        mov         dx, ax
        and         dx, 0f80h                      ;infinity
        jmp         short fix_sign

make_zero:
        xor         ax,ax
finish_error:
        mov         di,word ptr rptr
        add         di,6
        mov         cx, 4
rep     stos        word ptr [di]
        jmp         short fp_mulex
flmul  endp
```

The multiplication in this routine was performed by the fixed-point routine *mul64a.* This is a specially-written form of *mul64* which appears in FXMATH.ASM on the included disk. It takes as operands, 5-byte integers, the size of the mantissa plus extended bits in this format, and returns a 10-byte result. Knowing the size of the operands, means the routine can be sized exactly for that result, making it faster and smaller.

mul64a: Algorithm

1. Use DI to hold the address of the result, a quadword.
2. Move the most significant word of the multiplicand into AX and multiply by the most significant word of the multiplier. The product of this multiplication is written to the most significant word result.
3. The most significant word of the multiplicand is returned to AX and multiplied by the second most significant word of the multiplier. The least significant word of the product is MOVed to the second most significant word of result, the most significant word of the product is

ADDed to the most significant word of result.

4. The most significant word of the multiplicand is returned to AX and multiplied by the least significant word of the multiplier. The least significant word of this product is MOVed to the third most significant word of result, the most significant word of the product is ADDed to the second most significant word of *result,* any carries are propagated through with an ADC instruction.
5. The second most significant word of the multiplicand is MOVed to AX and multiplied by the most significant word of the multiplier. The lower word of the product is ADDed to the second most significant word of *result* and the upper word is added-with-carry (ADC) to the second most significant word of *result.*
6. The second most significant word of the multiplicand is again MOVed to AX and multiplied by the secondmost significant word of the multiplier. The lower word of the product is ADDed to the third most significant word of result and the upper word is added-with-carry (ADC) to the secondmost significant word of result with any carries propagated to the MSW with an ADC.
7. The second most significant word of the multiplicand is again MOVed to AX and multiplied by the least significant word of the multiplier. The lower word of the product is MOVed to the fourth most significant word of result and the upper word is added-with-carry (AX) to the thirdmost significant word of *result* with any carries propagated through to the MSW with an ADC.
8. The least significant word of the multiplicand is MOVed into AX and multiplied by the MSW of the multiplier. The least significant word of this product is ADDed to the third most significant word of *result,* the MSW of the product is ADCed to the second most significant word of *result,* and any carry is propagated into the most significant word of *result* with an ADC.

mul64a: Listing

```
; *****
;* mu164a - Multiplies two unsigned 5-byte integers. The
;* procedure allows for a product of twice the length of the multipliers,
;* thus preventing overflows.
mu164a proc uses ax dx,
                    multiplicand:qword, multiplier:qword, result:word

       mov          di,word ptr result
       sub          cx, cx
```

```
mov     ax, word ptr multiplicand[4]    ;multiply multiplicand high
mul     word ptr multiplier [4]         ;word by multiplier high word
mov     word ptr [di][8], ax

mov     ax, word ptr multiplicand[4]    ;multiply multiplicand high
mul     word ptr multiplier [2]         ;word by second MSW
                                        ;of multiplier
mov     word ptr [di][6], ax
add     word ptr [di][8], dx

mov     ax, word ptr multiplicand[4]    ;multiply multiplicand high
                                        ;word by third MSW
mul     word ptr multiplier [0]         ;of multiplier
mov     word ptr [di] [4], ax
add     word ptr [di][6], dx
adc     word ptr [di][8], cx            ;propagate carry


mov     ax, word ptr multiplicand[2]    ;multiply second MSW
mul     word ptr multiplier [4]         ;of multiplicand by MSW
add     word ptr [di][6], ax            ;of multiplier
adc     word ptr [di][8], dx

mov     ax, word ptr multiplicand[2]    ;multiply second MSW of
mul     word ptr multiplier[2]          ;multiplicand by second MSW
add     word ptr [di][4], ax            ;of multiplier
adc     word ptr [di][6], dx
adc     word ptr [di][8], cx            ;add any remnant carry

mov     ax, word ptr multiplicand[2]    ;multiply second MSW of
mul     word ptr multiplier[0]          ;multiplicand by least
mov     word ptr [di][2], ax            ;significant word of
                                        ;multiplier
add     word ptr [di][4], dx
adc     word ptr [di][6], cx
adc     word ptr [di][8], cx            ;add any remnant carry

mov     ax, word ptr multiplicand[0]    ;multiply multiplicand low
mul     word ptr multiplier[4]          ;word by MSW of multiplier
add     word ptr [di][4], ax
adc     word ptr [di][6], dx
adc     word ptr [di] [8], cx           ;add any remnant carry

mov     ax, word ptr multiplicand[0]    ;multiply multiplicand low
```

```
        mul          word ptr multiplier[2]        ;word by second MSW of
                                                   ;multiplier
        add          word ptr [di][2], ax
        adc          word ptr [di][4], dx
        adc          word ptr [di][6], cx          ;add any remnant carry
        adc          word ptr [di][8], cx          ;add any remnant carry

        mov          ax, word ptr multiplicand[0]  ;multiply multiplicand low
        mul          word ptr multiplier[0]        ;word by multiplier low word
        mov          word ptr [di][0], ax
        add          word ptr [di][2], dx
        adc          word ptr [di][4], cx          ;add any remnant carry
        adc          word ptr [di][61, cx          ;add any remnant carry
        adc          word ptr [di][8], cx          ;add any remnant carry

        ret
mu164a  endp
```

FLDIV

The divide is similar to the multiply except that the exponents are subtracted instead of added and the alignment is adjusted just before the fixed-point divide. This adjustment prevents an overflow in the divide that could cause the most significant word to contain a one. If we divide by two and increment the exponent, *div64* returns a quotient that is properly aligned for the renormalization process that follows.

The division could have been left as it was and the renormalization changed, but since it made little difference in code size or speed, it was left. This extra division does not change the result.

fldiv: Algorithm

1. Check the operands for zero and infinity.

   If one is found to be infinite, exit through step 11.

   If one is found to be zero, exit through step 12.

2. Test the divisor and dividend to see whether they are equal. If they are, exit now with a floating-point 1.0 as the result.

3. Get the exponents, find the difference and subtract 77H (119D). This is the approximate exponent of the result.

4. Check the signs of the operands and set the sign variable accordingly.
5. Restore the hidden bit.
6. Check the dividend to see if the most significant word is less than the divisor to align the quotient. If it's greater, divide it by two and increment the difference between the exponents by one.
7. Use *div64* to perform the division.
8. Check for a zero result upon return and exit with a floating-point 0.0 if so.
9. Renormalize the result.
10. Insert the exponent and sign and exit.
11. Infinite exit.
12. Zero exit.

fldiv: Listing

```
; *****
;
;
fldiv  proc        C  uses bx cx dx si di,
                              fp0:gword, fpl:qword, rptr:word

       local       qtnt:qword, sign:byte, exponent:byte, rmndr:gword

       pushf
       std
       xor         ax,ax

       mov         byte ptr sign, al               ;begin error and situation
                                                   ;checking
       lea         si,word ptr fp0                 ;name a pointer to each fp
       lea         bx,word ptr fpl

       mov         ax,word ptr [si][4]
       shl         ax,1
       and         ax,0ff00h                       ;check for zero
       jne         chk_b
       jmp         return infinite                 :infinity
chk_b:
       mov         dx,word ptr [bx][4]
       shl         a,1
       and         dx,0ff00h
       jne         b_notz
```

```
        jmp      divide-by-zero               ;infinity, divide by zero
                                              ;is undefined
b_notz:
        cmp      dx,0ff00h
        jne      check_identity
        jmp      make_zero                    ;divisor is infinite
check_identity:
        mov      di,bx
        add      di,4                         ;will decrement selves
        add      si,4
        mov      cx,3
repe    cmpsw
        jne      not same                     ;these guys are the same
        mov      ax,word ptr dgt[8]           ;return a one
        mov      bx,word ptr dgt[10]
        mov      dx,word ptr dgt[12]
        mov      di,word ptr rptr
        mov      word ptr [di],ax
        mov      word ptr [di][2],bx
        mov      word ptr [di][4],dx
        sub      ax,ax
        mov      word ptr [di][6],ax
        jmp      fldivex

not same:                                     ;get exponents
        lea      si,word ptr fp0              ;reset pointers
        lea      bx,word ptr fp1

        sub      ah,dh                        subtract exponents
        add      ah,77h                       ;subtract bias minus two
                                              ;digits
        mov      byte ptr exponent,ah         ;save exponent

        mov      dx, word ptr [si][4]         ;check sign
        or       h, dx
        jns      a_plus
        not      byte ptr sign
a_plus:
        mov      dx,word ptr [bx][4]
        or       dx, dx
        jns      restore_missing_bit
        not      byte ptr sign

restore_missing_bit:                          ;line up operands for division
```

```
        and         word ptr fp0[4], 7fh
        or          word ptr fp0[4], 80h

        mov         dx, word ptr fp1[4]
        and         dx, 7fh
        or          dx, 80h
        cmp         dx,word ptr fp0[4]          ;see if divisor is greater than
        ja          store dvsr                  ;dividend then divide by 2
        inc         byte ptr exponent
        shl         word ptr fp1[0], 1
        rcl         word ptr fp1[2], 1
        rcl         dx, 1
store_dvsr:
        mov         word ptr fp1[41, dx

divide:
        invoke      divmul, fp0, fpl, addr fp0  ;perform fixed point division

        mov         dx,word ptr fp0[4]          ;check for zeros on return
        mov         bx,word ptr fp0[2]
        mov         ax,word ptr fp0[0]
        sub         cx, cx
        cmp         ax,cx
        jne         not zero
        cmp         bx,cx
        jne         not zero
        cmp         dx,cx
        jne         not zero
        jmp         fix-sign                    ;exit with a zero
not_zero:
        mov         cx,64                       ;should never go to zero
        cmp         dx,0h
        je          rotate_result_left          ;realign float
        cmp         dh,00h
        jne         rotate_result_right
        test        d1,80h
        je          rotate_result_left
        jmp         short done_rotate
rotate_result_right:
        shr         dx,1
        rcr         bx,1
        rcr         ax,1
        test        dx,0ff00h
        je          done_rotate
```

```
        inc         byte ptr exponent          ;decrement exponent with
                                               ;each shift
        loop        rotate result right
rotate_result_left:
        shl         word ptr qtnt,l
        rcl         ax,1
        rcl         bx,l
        rcl         dx,l
        test        dx,80h
        jne         done_rotate
        dec         byte ptr exponent          ;decrement exponent with
                                               ;each shift
        loop        rotate_result_left
done rotate:
        and         dx, 7fh
        shl         dx, 1
        or          dh, byte ptr exponent      ;insert exponent
        shr         dx, 1
        mov         cl,byte ptr sign           ;set sign flag according
                                               ;to variable
        or          cl,cl
        je          fix_sign
        or          dx,8000h
fix_sign:
        mov         di,word ptr rptr
        mov          word ptr [di],ax
        mov         word ptr [di][2],bx
        mov         word ptr [di][4],dx
        sub         ax,ax
        mov         word ptr [di][6l,ax
fldivex:
        popf
        ret

return infinite:
        sub         ax, ax
        mov         bx, ax
        not         ax
        mov         dx, ax
        and         dx, 0f80h                      ;infinity
        jmp         short fix_sign
divide_by_zero:
        sub         ax,ax
        not         ax
```

```
        jmp        short finish error

make_zero:
        xor        ax,ax                          ;positive zero
finish_error:
        mov        di,word ptr rptr
        add        di,6
        mov        cx,4
rep     stos       word ptr [di]
        jmp        short fldivex
fldiv   endp
```

In order to produce the accuracy required for this floating-point routine with the greatest speed, use div64 from Chapter 2. This routine was specifically written to perform the fixed-point divide.

Rounding

Rounding is included in this discussion on floating point because it's used in the external routines.

IEEE 754 says that the default rounding shall "round to nearest," with the option to choose one of three other forms: round toward positive infinity, round to zero, and round toward negative infinity. Several methods are available in the rounding routine, as you'll see in the comments of this routine.

The default method in *round* is "round to nearest." This involves checking the extended bits of the floating-point number. If they're less than half the LSB of the actual float, clear them and exit. If the extended bits are greater than half the LSB, add a one to the least significant word and propagate the carries through to the most significant word. If the extended bits are equal to exactly one-half the LSB, then round toward the nearest zero. If either of the last two cases results in an overflow, increment the exponent. Clear AX (the extended bits) and exit *round*. If a fault occurs, AX contains -1 on exit.

round: *Algorithm*

1. Load the complete float, including extended bits, into the microprocessor's

registers.

2. Compare the least significant word with one-half (8000H).

   If the extended bits are less than one-half, exit through step 5.

   If the extended bits aren't equal to one-half, continue with step 3.

   If the extended bits are equal to one-half, test the LSB of the representable portion of the float.

   If it's zero, exit through step 5.

   If it's one, continue with step 3.

3. Strip the sign and exponent from the most significant word of the float and add one to the least significant word. Propagate the carry by adding zero to the upper word and test what might have been the hidden bit for a one.

   A zero indicates that no overflow occurred; continue with step 4.

   A one indicates overflow from the addition. Get the most significant word of the float, extract the sign and exponent, and add one to the exponent.

   If this addition resulted in an overflow, exit through step 5.

   Insert the new exponent into the float and exit through step 5.

4. Load the MSW of the float. Get the exponent and sign and insert them into the rounded fixed-point number; exit through step 5.

5. Clear AX to indicate success and write the rounded float to the output.

6. Return a -1 in AX to indicate failure. Make the float a Quiet NAN (positive overflow) and exit through step 5.

Round: Listing

```
; *****
;
round proto flp0:qword, rptr:word

round proc  uses bx dx di, fp:qword, rptr:word
      mov         ax,word ptr fp[0]
      mov         bx,word ptr fp[2]
      mov         dx,word ptr fp[4]
      cmp         ax,8000h
      jb          round ex                    ;less than half
      jne         needs_rounding
      test        bx,l                        ;put your rounding scheme
                                              ;here, as in the
      je          round_ex                    ;commented-out code below
```

```
        jmp         short needs rounding
;       xor         x, 1                            ;round to even if odd
                                                    ;and odd if even
;       or          bx,1                            ;round down if odd and up if
                                                    ;even (jam)
        jmp         round_ex
needs rounding:
        and         dx,7fh
        add         bx,1h
        adc         dx,0
        test        dx,80h                          ;if this is a one, there will
        je          renorm                          ;be an overflow
        mov         ax,word ptr fp[4]
        and         dx, 7fh
        and         ax,0ff80h                       ;get exponent and sign
        add         ax,80h                          ;kick it up one
        jo          over flow
        or          dx,ax
        jmp         short round_ex
renorm:
        mov         ax,word ptr fp[4]
        and         ax,0ff80h                       ;get exponent and sign
        or          dx,ax
round_ex:
        sub         ax,ax
round_exl:
        mov         di,word ptr rptr
        mov         word ptr [di][0],ax
        mov         word ptr [di][Z],bx
        mov         word ptr [di][4],dx
        sub         ax,ax
        mov         word ptr [di][6],ax
        ret

over_flow:
        xor         ax,ax
        mov         bx,ax
        not         ax
        mov         dx,ax
        xor         dx,7fH                          ;indicate overflow with an
                                                    ;infinity

        jmp         short round ex1
round endp
```

1 Ochs, Tom. “A Rotten Foundation,” *Computer Language 8/2:* Page 107. Feb. 1991.

2 Knuth, D. E. *Seminumerical Algorithms.* Reading, MA: Addison-Wesley Publishing Co., 1981, Pages 213-223.

3 *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Std 754, 1985).

4 Plauger, P.J. “Floating-Point Arithmetic,” *Embedded Systems Programming 4/8:* Pages 95-100. Aug. 1991.

CHAPTER 5

Input, Output, and Conversion

To be useful, an embedded system must communicate with the outside world. How that communication proceeds can strongly influence the system's speed and efficiency.

Often, the nature of the system and the applications program that drives it defines the form of the commands that flow between an embedded system and the host. If it's a graphics card or servo controller embedded in a PC, the fastest way to communicate is pure binary for both commands and data. Depending on the availability of a math chip, the numerics are in either fixed or floating-point notation.

Even so, it's quite common to find such systems using ASCII strings and decimal arithmetic to interface with the user. That's because binary communication can be fast, even though it has its problems. The General Purpose Interface Bus (GPIB) has some of the advantages and speed of the hardware interface, but the binary command and data set can sometimes imitate its own bus-control commands and cause trouble. Binary information on RS232, perhaps the most commonly used interface, has similar problems with binary data aliasing commands and delimiters. Packet-based communications schemes are available, however, they can be slow and clumsy. Any problem can be solved on closed systems under controlled circumstances, but rigorous, simple communication schemes often default to ASCII or EBCDIC for ease of debugging and user familiarity.

Whatever choices you make for your system, it will almost always have to communicate with the outside world. This often means accepting and working with formats that are quite foreign to the binary on the microprocessor bus. What's more, the numerics will most likely be decimal and not binary or hex, since that's how most of us view the world.

Decimal Arithmetic

If your system does very little calculation or just drives a display, it may not be worth converting the incoming decimal data to another format. The Z80, 8085, and 8051 allow limited addition and subtraction in the form of the *DAA* or *DA* instruction. On the Intel parts, this instruction really only helps during addition; the Z80 can handle decimal correction in both addition and subtraction. The 80x86 family offers instructions aimed at packing and unpacking decimal data, along with adjustments for a limited set of basic arithmetic operations. The data type is no greater than a byte, however, making the operation long and cumbersome to implement. The 8096 family lacks any form of decimal instructions such as the *DAA* or *auxiliary carry flag.*

Binary-based microprocessors do not work easily with decimal numbers because base 2, which is one bit per digit, and even base 16, which is four bits per digit, are incompatible with base 10; they have a different modulus. The *DAA* instruction corrects for this by adding six to any result greater than nine (or on an auxiliary carry), thereby producing a proper carry out to the next digit.

A few other instructions are available on the 80x86 for performing decimal arithmetic and converting to and from ASCII:

- *AAA* stands for ASCII Adjust After Addition. Add two unpacked (one decimal digit per byte) 8-bit decimal values and put the sum in AL. If the sum is greater than nine, this instruction will add six but propagate the carry into AH. That leaves you with an unpacked decimal value perfectly suited for conversion to ASCII.
- *AAD* stands for ASCII Adjust before Division, takes an unpacked value in AX and performs an automatic conversion to binary, placing the resulting value in AL. This instruction can help convert ASCII BCD to binary by handling part of the process for you.
- The *AAM* instruction, which stands for ASCII Adjust After Multiply, unpacks an 8-bit binary number less than 100 into AL, placing the most significant digit in AH and the least significant in AL. This instruction allows for a fast, easy conversion back to ASCII after a multiplication or division operation.
- *AAS* stands for ASCII Adjust After Subtraction, corrects radix misalignment

after a subtraction. If the result of the subtraction, which must be in AL, is greater than nine, AH is decremented and six is subtracted from AL. If AH is zero, it becomes -1 (0ffH).

The purpose of these functions is to allow a small amount of decimal arithmetic in ASCII form for I/O. They may be sufficient to drive displays and do simple string or keyboard handling, but if your application does enough number crunching-and it doesn't take much-you'll probably want to do it in binary; it's much faster and easier to work with.

Radix Conversions

In his book *Seminumerical Algorithms,* Donald Knuth writes about a number of methods for radix conversion[1]. Four of these involve fundamental principles and are well worth examining.

These methods are divided into two groups: one for integers and one for fractions. Note: properly implemented, the fractional conversions will work with integers and the integer conversions with the fractions. In the descriptions that follow, we'll convert between base 10 and base 2. Base A will be the base we're converting from, and base B will be the base we're converting to. The code in this section uses binary arithmetic, often with hexadecimal notation, because that's the native radix of most computers and microprocessors. In addition, all conversions are between ASCII BCD and binary, but you can use these algorithms with any radix.

Integer Conversion by Division

In this case, base A (2) is converted to base *B* (10) by division using binary arithmetic. We divide the number to be converted by the base we're converting to and place it in a variable. This is a modular operation, and the remainder of the division is the converted digit. The numbers are converted least significant digit first.

If we were to convert 0ffH (255D) to decimal, for example, we would first divide by 0aH (10D). This operation would produce a quotient of 19H and a remainder of 5H (the 5 is our first converted digit). We would then divide 19H by 0aH, for a resulting quotient of 2H and a remainder of 5H (the next converted digit). Finally, we would divide 2H by 0aH, with a 0H result and a 2H remainder (the final digit).

Briefly, this method works as follows:

1. Shift the bits of the variable *decimal_accumulator* right four bits to make room for the next digit.
2. Load an accumulator with the value to be converted and divide by 0aH (10D).
3. *OR* the least significant nibble of *decimal_accumulator* with the four-bit remainder just produced.
4. Check the quotient to see that there is something left to divide.

   If so, continue with step 1 above.

   If not, return *decimal_ accumulator* as the result.

The routine *bn_dnt* converts a 32-bit binary number to an ASCII string. The routine expects no more than eight digits of decimal data (you can change this, of course).

This routine loads the number to be converted into AX, checks it for zero, and if possible divides it by 10. Because the remainder from the first division is already in DX, we don't have to move it to prepare for the second division (on the LSW). The remainder generated by these divisions is *OR*ed with zero, resulting in an ASCII character that's placed in a string. The conversion is unsigned (we'll see examples of signed conversions later in this chapter).

bn_dnt: Algorithm

1. Point to the binary value, *binary,* to be converted and to the output string, *decptr.* Load the loop counter for maximum string size.
2. Get the MSW of *binary* and check for zero.

   If it's zero, continue with step 6.

   If not, divide by 10.

   Return the quotient to the MSW of *binary.*

   Check the remainder for zero.

   If it's zero, continue with step 6.

   If not, go on to step 3.
3. Get the LSW of binary and check for zero.

If it's zero, check the remainder from the last division. If it's also zero, continue with step 5.

Otherwise, continue with step 4.

4. Divide by 10 and return the quotient to the LSW of *binary.*
5. Make the result ASCII by *OR*ing zero (30H). Write it to the string, increment the pointer, and decrement the loop pointer,

   If the loop counter isn't zero, continue with step 2.

   Otherwise, exit with an error.
6. Test the upper word of *binary* for zero.

   If it's not zero, go to step 3.

   If it's, check the LSW of the binary variable.

   If it's not zero, go to step 4.

   If it's, we're done; go to step 7.
7. Realign the string and return with the carry clear.

bn-dnt: Listing

```
;*****
; bn_dnt - a routine that converts binary data to decimal
;
;A doubleword is converted. Up to eight decimal digits are
;placed in the array pointed to by decptr. If more are required to
;convert this number, the attempt is aborted and an error flagged.
;
bn_dnt proc uses bx cx dx si di, binary:dword, decptr: word

        lea     si,word ptr binary          ;get pointer to MSB of
                                            ;decimal value
        mov     di,word ptr decptr          ;string of decimal ASCII digits
        mov     cx, 9
        add     di,cx                       ;point to end of string
                                            ;this is for correct ordering
        sub     bx,bx
        mov     dx,bx
        mov     byte ptr [di],bl            ;see that string is zero-
                                            ;terminated
        dec     di

binary_conversion:
        sub     dx, dx
        mov     ax,word ptr [si][2]         ;get upper word
```

```
        or          ax,ax                   ;see if it is zero
        je          chk_empty               ;if so, check empty
        div         iten                    ;divide by 10
        mov         word ptr [si][2],ax
        or          dx, dx
        je          chk_empty               ;check for zeros

divide_lower:
        mov         ax, word ptr [si]       ;always checking the least
                                            ;significant word
        or          ax,ax                   ;of the binary accumulator
                                            ;for zero
        jne         not_zero
        or          dx, ax
        je          put_zero
not_zero:
        div         iten                    ;divide lower word
put_zero:
        mov         word ptr [si],ax        ;save quotient
        or          dl,'0'                  ;make the remainder an ASCII
                                            ;digit
        mov         bytr, [di], dl          ;write it to a string
        dec         di
        loop        binary_conversion

oops:                                       ;too many characters; just leave
        mov         ax,-1
        stc
        ret

chk_empty:
        or          dx,ax                   ;we are done if the variable
                                            ;ls empty
        je          still_nothing
        jmp         short divide_lower
still_nothing
        mov         ax, word ptr [si]       ;check least significant word of
binary
        or          ax, ax                  ;variable for zero
        je          empty
        jmp         short not_zero
empty:
        inc         di                      ;realign string
        mov         si, di                  ;trade pointers
```

```
        mov         di, word ptr decptr
        mov         cx, 9
rep     movsw

finished:
        sub         ax,ax                       ;success
        clc                                     ;no carry = success!
        ret
bn_dnt  endp
```

Integer Conversion by Multiplication

In this case, base A (10) is converted to base B (2) by multiplication using binary arithmetic. We convert the number by multiplying the result variable, called *binary-accumulator,* by base A (10), before adding each new decimal digit.

To see how this is done, we can reverse the conversion we just completed. This time, we wish to convert 255D to binary. First we create an accumulator, *binvar,* to hold the result (which is initially set to 0) and a source variable, *decvar,* to hold the decimal value. We then add decimal digits to *binvar* one at a time from *decvar* which is set to 255D. The first iteration places 2D in *binvar;* we multiply this by 0aH (10D) to make room for the next addition. (Recall that the arithmetic is binary.) *Binvar* is now 14H. The next step is to add 5D. The result, 19H, is then multiplied by 0aH to equal 0faH. To this value we add the final digit, 5D, to arrive at the result 0ffH (255D). This is the last digit, so no further multiplications are necessary.

Assume a word variable, *decvar,* holds four packed decimal digits. The following pseudocode illustrates how these digits are converted to binary and the result placed in another word variable, *binvar.*

1. Assume *binvar* and *decvar* are word variables located somewhere in RAM.
2. Multiply *binvar* by base A (10), the routine is converting from base A to base B.
3. Shift a digit (starting with the most significant) from *decvar* into *binvar.*
4. Test *decvar* to see whether it is zero yet.

   If it is, we are done and write *binvar* to memory or return it as the result.

   If not, continue from step 2.

In the following code, a pointer to a string of ASCII decimal digits is passed to a subroutine that, in turn, returns a pointer to a doubleword containing the binary conversion. The routine checks each digit for integrity before processing it. If it encounters a nondecimal character, it assumes that it has reached the end of the string. Multiplication by 10 is performed in-line to save time.

dnt_bn: Algorithm

1. Point at the base of the BCD ASCII string (the most significant decimal digit), clear the binary accumulator, and load the loop counter with the maximum string length.
2. Get the ASCII digit and test to see whether it is between 0 and 9,

   If not, we are done; exit through step 4.

   If so, call step 5 to multiply the binary accumulator by 10. Coerce the ASCII digit to binary, add that digit to the binary accumulator, increment the string pointer, and decrement the loop counter.

   If the loop counter is zero, go to step 3.

   If not, continue with step 2
3. Exit with error.
4. Write the binary accumulator to output and leave with the carry clear.
5. Execute in-line code to multiply DX:BX by 10.

dnt_bn: Listing

```
; *****
; dnt_bn - decimal integer to binary conversion routine
;unsigned
;It is expected that decptr points at a string of ASCII decimal digits.
;Each digit is taken in turn and converted until eight have been converted
;or until a nondecimal number is encountered.
;This might be used to pull a number from a communications buffer.
;Returns with no carry if successful and carry set if not.

dnt bn proc         uses bx cx dx si di, decptr:word, binary:word


        mov         si,word ptr decptr          ;get pointer to beginning of
                                                ;BCD ASCII string
        sub         ax,ax                       ;clear some registers
        mov         bx,ax
```

```
        mov         dx,bx
        mov         cx, 9

decimal_conversion:
        mov         al,byte ptr [si]
        cmp         al,'O'                    ;check for decimal digit
        jb          work_done
        cmp         al, '9'
        ja          work_done                 ;if it gets past here, it
                                              ;must be OK
        call        near ptr times_ten        ;in-line multiply
        xor         a1,'O'                    ;convert to number
        add         bx,ax                     ;add next digit
        adc         dx,0                      ;propagate any carries
        inc         si
        loop        decimal_conversion
oops:
        stc                                   ;more than eight digits
        ret

work_done:
        mov         di, word ptr binary
        mov         word ptr [di],bx
        mov         word ptr [dil[2],dx       ;store result
        clc                                   ;success
        ret

times_ten:
        push        ax                        ;save these, they contain
                                              ;information
        push        cx
        shl         bx,l                      ;l0 = three left shifts and
                                              ;an add
        rcl         dx,1
        mov         ax,bx                     ;this is the multiply by two
                                              ;keep it
        mov         cx, dx

        shl         bx,l
        rcl         dx,l

        shl         bx,l
        rcl         dx,l
```

```
        add                                     ;this is the multiply by eight
                                                ;add the multiply by two to
        adc         dx,cx                       ;get 10

        pop         cx                          ;get it back
        pop         ax
        retn
dnt_bn    endp
```

Fraction Conversion by Multiplication

The next algorithm converts a fraction in base A (2) to base B (10) by successive multiplications of the number to be converted by the base to which we're converting.

First, let's look at a simple example. Assume we need to convert 8cH to a decimal fraction. The converted digit is produced as the overflow from the data type, in this case a byte. We multiply 8cH by 0aH, again using binary arithmetic, to get 578H (the five is the overflow). This conversion may actually occur between the low byte and high byte of a word register, such as the AX register in the 8086. We remove the first digit, 5, from the calculation and place it in an accumulator as the most significant digit. Next, we multiply 78H by 0aH, for a result of 4b0H. Before placing this digit in the accumulator, we shift the accumulator four bits to make room for it. This procedure continues until the required precision is reached or until the initial binary value is exhausted.

Round with care and only if you must. There are two ways to round a number. One is to truncate the conversion at the desired precision plus one digit, n_{-k+1}, where n is a converted digit and k is positional notation. A one is then added to the least significant digit plus one, n_{-k}, if the least significant digit n_{-k+1}, is greater than one-half of n_{-k}. This propagates any carries that might occur in the conversion. The other method involves rounding the fraction in the source base and then converting, but this can lead to error if the original fraction cannot be represented exactly.

To use this procedure, we must create certain registers or variables. First, we create the working variable *bfrac* to hold the binary fraction to be converted. Because multiplication requires a result register as wide as the sum of the bits of the multiplicand and multiplier, we need a variable as wide as the original fraction plus four bits. If the original fraction is a byte, as above, a word variable or register is more than sufficient. Next, we create *dfrac* to accumulate the result starting with the most significant decimal digit (the one closest to the radix point). This variable needs to be as large as the desired precision.

1. Clear *dfrac* and load *bfrac* with the binary fraction we're converting.
2. Check *bfrac* to see if it's exhausted or if we've reached our desired precision. If either is true, we're done.
3. Multiply *bfrac* by the base to which we're converting (in this case, 0aH).
4. Take the upper byte of *bfrac* as the result of the conversion and place it in *dfrac* as the next less significant digit. Zero the upper byte of *bfrac*.
5. Continue from step 2.

The following routine accepts a pointer to a 32-bit binary fraction and a pointer to a string. The converted decimal numbers will be placed in that string as ASCII characters.

bfc_dc: Algorithm

1. Point to the output string, load the binary fraction in DX:BX, set the loop counter to eight (the maximum length of the string), and initialize the string with a period.
2. Check the binary fraction for zero.

   If it's zero, exit through step 3.

   If not, clear AX to receive the overflow. Multiply the binary fraction by 10, using AX for overflow. Coerce AX to ASCII and write it to the string. Decrement the loop counter.

   If the counter is zero, leave through step 3.

   Otherwise, clear the overflow variable and continue with step 2.
3. Exit with the carry clear.

bfc-dc: Listing

```
; *****


; bfc_dc - a conversion routine that converts a binary fraction
; (doubleword) to decimal ASCII representation pointed to by the string
```

```
;pointer decptr. Set for eight digits, but it could be longer.

bfc_dc proc        uses bx cx dx si di bp, fraction:dword, decptr:word

       local        sva:word, svb:word, svd:word


       mov          di,word ptr decptr          ;point to ASCII output string
       mov          bx,word ptr fraction
       mov          dx,word ptr fraction[2]     ;get fractional part

       mov          cx, 8                       ;digit counter
       sub          ax,ax

       mov          byte ptr [di], '.'          ;to begin the ASCII fraction
       inc          di

decimal_conversion:
       or           ax,dx                       ;check for zero operand
       or           ax,bx                       ;check for zero operand
       jz           work_done
       sub          ax,ax

       shl          bx,1                        ;multiply fraction by 10
       rcl          dx,1
       rcl          ax,1                        ;times 2 multiple
       mov          word ptr svb,bx
       mov          word ptr svd,dx
       mov          word ptr sva,ax

       shl          bx,1
       rcl          dx,1
       rcl          ax,1

       shl          bx,1
       rcl          dx,1
       rcl          ax,1

       add          bx,word ptr svb
       adc          dx,word ptr svd             ;multiply by 10
       adc          ax,word ptr sva             ;the converted value is
                                                ;placed in AL
```

```
        or          al,'0'                  ;this result is ASCIIized and
        mov         byte ptr [di],al        ;placed in a string
        inc         di
        sub         ax,ax
        loop        decimal conversion

work done:
        mov         byte ptr [di],al        ;end string with a null
        clc
        ret
bfc_dc  endp
```

Fraction Conversion by Division

Like conversion of integers by multiplication, this procedure is performed as a polynomial evaluation. With this method, base A (10) is converted to base B (2) by successively dividing of the accumulated value by base A using the arithmetic of base B. This is the reverse of the procedure we just discussed.

For example, lets convert .66D to binary. We use a word variable to perform the conversion and place the decimal value into the upper byte, one digit at a time, starting with the least significant. We then divide by the base from which we're converting. Starting with the least significant decimal digit, we divide 6.00H (the radix point defines the division between the upper and lower bytes) by 0aH to get .99H. This fraction is concatenated with the next most significant decimal digit, yielding 6.99H. We divide this number by 0aH, for a result of .a8H. Both divisions in this example resulted in remainders; the first was less than one-half the LSB and could be forgotten, but the second was more than one-half the LSB and could have been used for rounding.

Create a fixed-point representation large enough to contain the fraction, with an integer portion large enough to hold a decimal digit. In the previous example, a byte was large enough to contain the result of the conversion ($\log_{10} 256$ is approximately 2.4) with four bits for each decimal digit. Based on that, the variable *bfrac* should be at least 12 bits wide. Next, a byte variable dfrac is necessary to hold the two decimal digits. Finally, a counter *(dcntr)* is set to the number of decimal digits to be converted.

1. Clear *bfrac* and load *dcntr* with the number of digits to be converted.
2. Check to see that *dcntr* is not yet zero and that there are digits yet to convert. If not, the conversion is done.
3. Shift the least significant digit of *dfrac* into the position of the least significant integer in the fixed-point fraction *bfrac*.
4. Divide *bfrac* by 0aH, clear the integer portion to zero, and continue with step 2.

The following example takes a string of ASCII decimal characters and converts them to an equivalent binary fraction. An invisible radix point is assumed to exist immediately preceding the start of the string.

Dfc_bn: Algorithm

1. Find least significant ASCII BCD digit. Point to the binary fraction variable and clear it. Clear DX to act as the MSW of the dividend and set the loop counter to eight (the maximum number of characters to convert).
2. Put the MSW of the binary result variable in AX and the least significant ASCII BCD digit in DL. Check to see if the latter is a decimal digit.

   If not, exit through step 6.

   If so, force it to binary. Decrement the string pointer and check the dividend (32-bit) for zero.

   If it's zero, go to step 3.

   Otherwise, divide DX:AX by 10.
3. Put AX in the MSW of the binary result variable and get the LSW. Check DX:AX for zero.

   If it's zero, go to step 4.

   Otherwise, divide DX:AX by 10.
4. Put AX in the LSW of the binary result variable. Clear DX for the next conversion. Decrement the loop variable and check for zero.

   If it's zero, go to step 5.

   Otherwise, continue with step 2.
5. Exit with the carry clear.
6. Exit with the carry set.

Dfc-bn: Listing

```
;*****
; dfc_bn - A conversion routine that converts an ASCII decimal fraction
;to binary representation.  decptr points to the decimal string to be
;converted. The conversion will produce a doubleword result. The
;fraction is expected to be padded to the right if it does not fill eight
;digits.

dfc_bn  proc  uses bx cx dx si di, decptr:word, fraction:word

        pushf
        cld

        mov         di, word ptr decptr          ;point to decimal string
        sub         ax,ax
        mov         cx, 9
        repne       scasb                        ;find end of string
        dec         di
        dec         di                           ;point to least significant
                                                 ;byte
        mov         si,di

        mov         di, word ptr fraction        ;point of binary fraction
        mov         word ptr [di], ax
        mov         word ptr [di][2], ax

        mov         cx, 8                        ;maximum number of
                                                 ;characters
        sub         dx, dx



binary_conversion:
        mov         ax, word ptr [di][2]         ;get high word of result
                                                 ;variable
        mov         dl, byte ptr [si]            ;concatenate ASCII input
                                                 ;with binary fraction
        cmP         dl, '0'                      ;check for decimal digit
        jb          oops
        cmp         dl, '9'
        ja          oops                         ;if it gets past here,
                                                 ;it must be OK
        xor         dl, '0'                      ;deASCIIize
```

```
        dec          si

        sub          bx,bx
        or           bx,dx
        or           bx,ax
        jz           no_div0                      ;prevent a divide by zero
        div          iten                         ;divide by 10
no_div0:
        mov          word ptr [di][2],ax

        mov          ax,word ptr [di]
        sub          bx,bx
        or           bx,dx
        or           bx,ax
        jz           no_div1                      ;prevent a divide by zero
        div          iten
no_divl:
        mov          word ptr [di],ax

        sub          dx,dx
        loop         binary-conversion            ;loop will terminate
                                                  ;automatically

work_done:
        sub          ax,ax
        clc                                       ;no carry =success!
        ret

oops:
        mov          ax,-1                        ;bad character
        stc
        ret
dfc_bn  endp
```

As you may have noticed from the fractional conversion techniques, truncating or rounding your results may introduce errors. You can, however, continue the conversion as long as you like. Given a third argument representing allowable error, you could write an algorithm that would produce the exact number of digits required to represent your fraction to within that error margin. This facility may or may not be necessary in your application.

Table-Driven Conversions

Tables are often used to convert from one type to another because they often offer better speed and code size over computational methods. This chapter covers the simpler lookup table conversions used to move between bases and formats, such as ASCII to binary. These techniques are used for other conversions as well, such as converting between English and metric units or between system-dependent factors such as revolutions and frequency. Tables are also used for such things as facilitating decimal arithmetic, multiplication, and division; on a binary machine, these operations suffer increased code size but make up for that in speed.

For all their positive attributes, table-driven conversions have a major drawback: a table is finite and therefore has a finite resolution. Your results depend upon the resolution of the table alone. For example, if you have a table-driven routine for converting days to seconds using a table that has a resolution of one second, an input argument such as 16.1795 days, which yields 1,397,908.8 seconds will only result in only 1,397,908 seconds. In this case, your result is almost a full second off the actual value.

Such problems can be overcome with a knowledge of what input the routine will receive and a suitable resolution. Another solution, discussed in the next chapter, is linear interpolation; however, even this won't correct inexactitudes in the tables themselves. Just as fractions that are rational in one base can be irrational in another, any translation may involve inexact approximations that can compound the error in whatever arithmetic the routine performs upon the table entry. The lesson is to construct your tables with enough resolution to supply the accuracy you need with the precision required.

The following covers conversion from hex to ASCII, decimal to binary, and binary to decimal using tables.

Hex to ASCII

The first routine, *hexasc,* is a very simple example of a table-driven conversion: from hex to ASCII.

The procedure is simple and straightforward. The table, *hextab,* contains the ASCII representations of each of the hex digits from *0* through *f* in order. This is an improvement over the ASCII convention, where the numbers and alphabet are not contiguous. In the order we're using, the hex number itself can be used as an index to select the appropriate ASCII representation.

Because it uses *XLAT*, an 8086-specific instruction, this version of the routine isn't very portable, though it could conceivably be replaced with a move involving an index register and an offset (the index itself). Before executing *XLAT*, the user places the address of the table in BX and an index in AL. After *XLAT* is executed, AL contains the item from the table pointed to by the index. This instruction is useful but limited. In the radix conversion examples that follow, other ways of indexing tables will be presented.

Hexasc takes the following steps to convert a binary quadword to ASCII hex.

hexasc: Algorithm

1. SI points to the most significant byte of the binary quadword, DI points to the output string, BX points to the base of *hextab,* and CX holds the number of bytes to be converted.
2. The byte indicated by SI is pulled into AL and copied to AH.
3. Since each nibble contains a hex digit, AH is shifted right four times to obtain the upper nibble. Mask AL to recover the lower nibble.
4. Exchange AH and AL so that the more significant digit is translated first.
5. Execute *XLAT.* AL now contains the ASCII equivalent of whatever hex digit was in AL.
6. Write the ASCII character to the string and increment DI.
7. Exchange AH and AL again and execute *XLAT.*
8. Write the new character in AL to the string and increment DI.
9. Decrement SI to point to the next lesser significant byte of the hex number.
10. Execute the loop. When CX is 0, it will automatically exit and return.

hexascs Listing

```
; ******
; hex-to-ASCII conversion using xlat
; simple and common table-driven routine to convert from hexadecimal
; notation to ASCII
; quadword argument is passed on the stack, with the result returned
```

```
; in a string pointed to by sptr

        .data

hextab byte     '0', '1', '2', '3', '4', '5', '6', '7',
                '8', '9', 'a', 'b', 'c', 'd', 'e', 'f'  ;table of ASCII
                                                        ;characters

        .code

hexasc proc         uses bx cx dx si di, hexval:qword, sptr:word


        lea         si, byte ptr hexval[7]       ;point to MSB of hex value
        mov         di, word ptr sptr            ;point to ASCII string
        mov         bx, offset byte ptr hextab   ;offset of table
        mov         cx, 8                        ;number of bytes to be
                                                 ;converted

make ascii:
        mov         al, byte ptr [si]            ;get hex byte
        mov         ah, al                       ;copy to ah to unpack
        shr         ah,1                         ;shift out lower nibble
        shr         ah,1
        shr         ah,1
        shr         ah,1
        and         al, 0fh                      ;strip higher nibble
        xchg        al,ah                        ;high nibble first

        xlat
        mov         byte ptr [di],al             ;write ASCII byte to string
        inc         di
        xchg        al, ah                       ;now the lower nibble
        xlat
        mov         byte ptr [di],al             ;write to string
        inc         di                           ;increment string pointer
        dec         si                           ;decrement hex byte pointer
        loop        make ascii

        sub         al, al
        mov         byte ptr [di],al             ;NULL at the end of the
                                                 ;string
```

```
        ret

hexasc  endp
```

Decimal to Binary

Clearly, the table is important in any table-driven routine. The next two conversion routines use the same resource: a table of binary equivalents to the powers of 10, from 10^9 to 10^{-10}. The problem with table-driven radix conversion routines, especially when they involve fractions, is that they can be inaccurate. Many of the negative powers of base 10 are irrational in base 2 and make any attempt at conversion merely an approximation. Nevertheless, tables are commonly used for such conversions because they allow a direct translation without requiring the processor to have a great deal of computational power.

The first example, *tb_dcbn*, uses the tables *int_tab* and *frac_tab* to convert an input ASCII string to a quadword fixed-point number. It uses the string's positional data-the length of the integer portion, for instance— to create a pointer to the correct power of 10 in the table. After the integer is converted to binary, it is multiplied by the appropriate power of 10. The product of this multiplication is added to a quadword accumulator. When the integer is processed, the product is added to the most significant doubleword; when the fraction is processed, the product is added to the least significant doubleword (the radix point is between the doublewords).

Before the actual conversion can begin, the routine must determine the power of 10 occupied by the most significant decimal digit. It does this by testing each digit in turn and counting it until it finds a decimal point or the end of the string. It uses the number of characters it counted to point to the correct power in the table. After determining the pointer's initial position, the routine can safely increment it in the rest of the table by multiplying consecutive numbers by consecutive powers of 10.

tb-dcbn: Algorithm

1. Form a pointer to the fixed-point result (the ASCII string) and to the base address of the integer portion of the table. Clear the variable to hold the fixed-point result and the sign flag. Set the maximum number of integers to nine.

2. Examine the first character.

If it's a hyphen, set the sign flag, increment the string pointer past that point, and reset the pointer to this value. Get the next character and continue with step 3.

If it's a "+," increment the string pointer past that point and reset the pointer to this value. Get the next character and continue with step 3.

3. If it's a period save the current integer count, set a new count for the fractional portion, and continue with step 2.

4. If it's the end of the string, continue with step 5.

   If it's not a number, exit through step 10.

   if it's a number, increment the number counter.

5. Test the counter to see how many integers have been processed.

   If we have exceeded the maximum, exit through step 12.

   If the count is equal to or less than the maximum, increment the string pointer, get a new character and test to see whether we are counting integers or fractions,

   If we are counting integers, continue with step 3.

   If we are counting fractional numbers, continue with step 4.

6. Get the integer count, convert it to hex, and multiply it by four (each entry is four bytes long) to index **into** the table.

7. Get a character.

   If it's a period continue with step 8.

   If it's the end of the string, continue with step 10.

   If it's a number, deASCIIize it, multiply it by the four-byte table entry, and add the result to the integer portion of the fixed-point result variable. Increment the string pointer and increment the table pointer by the size of the data type.Continue with step 7.

8. Increment the string pointer past the period.

9. Get the next character.

   If it's the end of the string, continue with step 10.

   If not, deASCIIize it and multiply it by the next table entry, adding the result to the fixed-point variable. Increment the string pointer and increment the table pointer by the size of the data type. Continue with step 9.

10. Check the sign flag.

    If it's set, two's-complement the fixed-point result and exit with success.

    If it's clear, exit with success.

11. Not a number: set AX to -1 and continue with step 12.

12. Too big. Set the carry and exit.

tb-dcbn: Listing

```
; ******
; table-conversion routines


        .data

int tab         dword           3b9aca00h, 05f5e100h, 00989680h, 000f4240h
                                000186a0h, 00002710h, 000003e8h, 00000064h
                                0000000ah, 0000000lh
frac_tab        dword           1999999ah, 028f5c29h, 00418937h, 00068db9h
                                0000a7c5h, 0000l0c6h, 00000ladh, 0000002ah
                                00000004h
tab_end         dword           00000000h

;
        .code

; converts ASCII decimal to fixed-point binary
;
tb_dcbn              proc       uses bx cx dx si di.
                                sptr:word, fxptr:word
        local        sign:byte

        mov          di, word ptr sptr          ;point to result
        mov          si, word ptr fxptr         ;point to ascii string
        lea          bx, word ptr frac_tab      ;point into table

        mov          cx,4                       ;clear the target variable
        sub          ax,ax
        sub          dx,dx
rep     stosw

        mov          di, word ptr sptr          ;point to result
```

```
        mov         cl,al                      ;to count integers
        mov         ch,9h                      ;max int digits
        mov         byte ptr sign, al          ;assume positive

        mov         al, byte ptr [si]          ;get character
        cmp         al '-'                     ;check for sign
        je          negative
        cmp         al,'+'
        je          positive                   ;count:

                                               ;count the number of
                                               ;characters in the string
        cmp         al,'.'
        je          fnd_dot
chk_frac:
        cmp         al,0                       ;end of string?
        je          gotnumber
        cmp         a1,'0'                     ;is it a number then?
        jb          not_a_number
        cmp         a1,'9'
        ja          not_a_number
cntnu:
        inc         cl                         ;count
        cmp         cl,ch                      ;check size
        ja          too_big
        inc         si                         ;next character
        mov         al, byte ptr [si]          ;get character
        or          dh,dh                      ;are we counting int
                                               ;or frac?
        jne         chk_frac
        jmp         short count                ;count characters in int
fnd_dot:
        mov         dh,cl                      ;switch to counting fractions
        inc         dh                         ;can't be zero
        mov         d1,13h                     ;includes decimal point
        xchg        ch,dl
        jmp         short cntnu
negative:
        not         sign                       ;make it negative
positive:
        inc         si
        mov         word ptr fxptr,si
        mov         al, byte ptr [si]          ;get a character
```

```
        jmp         short count

gotnumber:
        sub         ch,ch
        xchg        cl,dh                       ;get int count
        dec         cl
        shl         word ptr cx,l               ;multiply by four
        shl         word ptr cx,l
        sub         bx,cx                       ;index into table
        sub         cx, cx                      ;don't need integer count
                                                ;anymore
        mov         si,word ptr fxptr           ;point at string again
cnvrt_int:
        mov         cl,byte ptr [si]            ;get first character
        cmp         cl,'.'
        je          handle_fraction             ;go do fraction, if any
        cmp         cl,0
        je          do_sign                     ;end of string
        sub         cl, '0'
        mov         ax,word ptr [bx][2]
        mul         cx                          ;multiply by deASCIIized
                                                -input
        add         word ptr [di][4],ax
        adc         word
        mov         ax,word ptr [bx]
        mul         cx                          ;multiply by deASCIIized
                                                 input
        add         word ptr [dil[4],ax
        adc         word ptr [di][6],dx
        add         bx,4                        ;drop table pointer
        inc         si
        jmp         short cnvrt_int

handle_fraction:
        inc         si                          ;skip decimal point
cnvrt_frac:
        mov         cl,byte ptr [si]            ;get first character
        cmp         cl,0
        je          do_sign                     ;end of string
        sub         cl,'O'
        mov         ax,word ptr [bx][2]         ;this can never result
                                                ;in a carry
        mul         cx                          ;multiply by deASCIIized
                                                ;input
```

```
        add         word ptr [di][2],ax
        mov         ax,word ptr [bx]
        mul         cx                          ;multiply by deASCIIized
                                                ;input
        add         word ptr [di][0],ax
        adc         word ptr [di][2],dx
        add         bx,4                        ;drop table pointer
        inc         si
        jmp         short cnvrt_frac

do_sign:
        mov         al,byte ptr sign            ;check sign
        or          al,al
        je          exit                        ;it is positive
        not         word ptr [di][6]
        not         word ptr [di][4]
        not         word ptr [di][2]
        neg         word ptr [di]
        jc          exit
        add         word ptr [di] [2],1
        adc         word ptr [di] [4],0
        adc         word ptr [di] [61,0
exit:
        ret
;
not_a_number
        sub         ax,ax
        not         ax                          ;-1
too_big:
        stc                                     ;failure
        jmp         short exit
tb_dcbn             endp
```

Binary to Decimal

The binary-to-decimal conversion, *tb_bndc,* could have been written in the same manner as *tb_dcbn*—using a separate table with decimal equivalents to hex positional data. That would have required long and awkward decimal additions, however, and would hardly have been worth the effort.

The idea is to divide the input argument by successively smaller powers of 10, converting the quotient of every division to ASCII and writing it to a string. This is done until the routine reaches the end of the table. To use the same table and keep the arithmetic binary, take the integer portion of the binary part of the fixed-point variable to be converted and, beginning at the top of the table, compare each entry until you find one that's less than the integer you're trying to convert. This is where you start. Successively subtract that entry from the integer, counting as you go until you get an underflow indicating you've gone too far. You then add the table entry back into the number and decrease the counter. This is called *restoring division;* it was chosen over other forms because some of the divisors would be two words long. That would mean using a division routine that would take more time than the simple subtraction here. The number of times the table entry could divide the input variable is forced to ASCII and written to the next location in the string.

Tb-bndc is an example of how this might be done.

tb_bndc: Algorithm

1. Point to the fixed-point variable, the output ASCII string, and the top of the table. Clear the leading-zeros flag.
2. Test the MSB of the fixed-point variable for sign. If it's negative, set the sign flag and two's-complement the fixed-point variable.
3. Get the integer portion of the fixed-point variable. Compare the integer portion to that of the current table entry.

   If the integer is larger than the table entry, continue with step 5.

   If the integer is less than the table entry, check the leading-zeros flag.

   If it's nonzero, output a zero to the string and continue with step 4.

   If it's zero, continue with step 4.
4. Increment the string pointer, increment the table pointer by the size of the data type, and compare the table pointer with the offset of *fractab,* 10^0.

   If the table pointer is greater than or equal to the offset of *fractab,* continue with step 3.

   If the table pointer is less than the offset of *fractab,* continue with step 6.
5. Increment the leading-zeros flag, call step 10, and continue with step

4 upon return.

6. If the leading-zeros flag is clear, write a zero to the string, increment the string pointer, issue a period, increment the string pointer again, and get the fractional portion of the fixed-point variable.

7. Load the fractional portion into the DX:AX registers.

7a. Compare the current table entry with DX:AX.

    If the MSW of the fractional portion is greater, continue with step 9.

    If the MSW of the fractional portion is less, continue with step 8.

8. Write a zero to the string.

8a. Increment the string and table pointers and test for the end of the table.

    If it's the end, continue with step 11.

    If it's not the end, continue with step 7a.

9. Call step 10 and continue with step 8a.

10. Subtract the table entry from the remaining fraction, counting each subtraction. When an underflow occurs, add the table entry back in and decrement the count. Convert the count to an ASCII character and write it to the string. Return to the caller.

11. Write a NULL to the next location in the string and exit.

tb_bndc: Listing

```
;
; converts binary to ASCII decimal
;
tb_bndc         proc        uses bx cx dx si di
                            sptr:word, fxptr:word

        local   leading_zeros:byte

        mov     si, word ptr fxptr          ;point to input fixed-point
                                            ;argument
        mov     di, word ptr sptr           ;point to ASCII string
        lea     bx, word ptr int tab        ;point into table

        sub     ax,ax
        mov     byte ptr leading_zeros, al  ;assume positive

        mov     ax, word ptr [si][6]        ;test for sign
        or      ax,ax
        jns     positive
```

```
        mov     byte ptr [di],'-'               ;write hyphen to output
                                                ;string
        inc     di
        not     word ptr [si][6]                ;two's complement
        not     word ptr [si][41
        not     word ptr [si] [2]
        neg     word ptr [si][0]
        jc      positive
        add     word ptr [si] [2],1
        adc     word ptr [si][4],0
        adc     word ptr [si][6],0

positive:
        mov     dx, word ptr [si][6]
        mov     ax, word ptr [si][4]            ;get integer portion
        sub     cx, cx
walk_tab:
        cmp     dx, word ptr [bx] [2]           ;find table entry smaller
                                                ;than integer
        ja      gotnumber                       ;entry smaller
        jb      pushptr                         ;integer smaller
        cmp     ax, word ptr [bx]
        jae     gotnumber
pushptr:
        cmp     byte ptr cl, leading_zeros      ;have we written a number
                                                ;yet?
        je      skip_zero
        mov     word ptr count : [di],'O'       ;write a '0' to the string

cntnu:
        inc     di                              ;next character
skip_zero:
        inc     bx                              ;next table entry
        inc     bx
        inc     bx
        inc     bx
        cmp     bx, offset word ptr frac_tab ;done with integers?
        jae     handle-fraction                 ;yes, do fractions
        jmp     short walk_tab

gotnumber:
        sub     cx, cx
        inc     leading zeros                   ;shut off leading zeros bypass
```

```
cnvrt_int:
        call        near ptr index              ;calculate and write to string
        jmp         short cntnu

handle_fraction:
        cmp         byte ptr leading_zeros,0    ;written anything yet?
        jne         do frac
        mov         byte ptr [di],'O'
        inc         di
do_frac:
        mov         word ptr [di],'.'           ;put decimal point
        inc         di
get_frac:
        mov         dx, word ptr [si][2]        ;move fraction to registers
                    ax, word ptr [si][O]
        sub         cx, cx
walk_tabl:
        cmp         dx, word ptr [bx] [2]       ;find suitable table entry
        ja          small_enuf
        jb          pushptr1
        cmp         ax, word ptr [bx]
        jae         small_enuf
pushptrl:
        mov         byte ptr [di],'0'           ;write '0'
skip_zero1:
        inc         di                          ;next character
        inc         bx                          ;next entry
        inc         bx
        inc         bx
        inc         bx
        cmp         bx, offset word ptr tab_end
        jae         exit
        jmp         short walk_tab1

small_enuf:
        sub         cx, cx
small_enufl:
        call        near ptr index             ;calculate and write
        jmp         short skip_zero1

exit:
        inc         di
```

```
        sub         cl,cl                       ;put NULL at
        mov         byte ptr [si],cl            ;end of string
        ret

index:
        inc         cx                          ;count subtractions
        sub         ax, word ptr [bx]
        sbb         dx, word ptr [bx] [2]
        jnc         index                       ;subtract until a carry
        dec         cx
        add         ax, word ptr [bx]           ;put it back
        adc         dx, word ptr [bx][2]
        or          cl,'0'                      ;make it ascii
        mov         byte ptr [di],cl            ;write to string
        retn
tb_bndc             endp
```

Floating-Point Conversions

This next group of conversion routines involves converting ASCII and fixed point to floating point and back again. These are specialized routines, but you'll notice that they employ many of the same techniques just covered, both table-driven and computational.

The conversions discussed in this section are ASCII to single-precision float, single-precision float to ASCII, fixed point to single-precision floating point, and single-precision floating point to fixed point.

You can convert ASCII numbers to single-precision floating point by first converting from ASCII to a fixed-point value, normalizing that number, and computing the shifts for the exponent, or you can do the conversion *in* floating point. This section gives examples of both; the next routine uses floating point to do the conversion.

ASCII to Single-Precision Float

Simply put, each ASCII character is converted to hex and used as a pointer to a table of extended-precision floating-point equivalents for the decimal digits 0 through 10. As each equivalent is retrieved, a floating point accumulator is multiplied by 10, and the equivalent is added, similar to the process described earlier for integer conversion by multiplication.

The core of the conversion is simple. We need three things: a place to put our result *flaccum,* a flag indicating that we have passed a decimal point *dpflag,* and a counter for the number of decimal places encountered *dpcntr.*

1. Clear *flaccum* and *dpcntr.*
2. Multiply *flaccum* by 10.0.
3. Fetch the next character.

   If it's a decimal point, set *dpflag* and continue with step 3.

   If *dpflag* is set, increment *dpcntr.*

   If it's a number, convert it to binary and use it as an index into a table of extended floats to get its appropriate equivalent.
4. Add the number retrieved from the table to *flaccum.*
5. See if any digits remain to be converted. If so, continue from step 2.
6. If *dpflag* is set, divide *flaccum* by 10.0 *dpcntr* times.
7. Exit with the result in *flaccum.*

   The routine *atf* performs the conversion described in the pseudocode. It will convert signed numbers complete with signed exponents.

atf: Algorithm

1. Clear the floating-point variable, point to the input ASCI1 string, clear local variables associated with the conversion, and set the digit counter to 8.
2. Get a character from the string and check for a hyphen.

   If the character is a hyphen, complement *numsin* and get the next character. Go to step 3.

   If not, see if the character is "+."

   If not, go to step 3.

   If so, get the next character and go to step 3.
3. See if the next character is "."

   If so, test *dp,* the decimal-point flag.

   If it's negative, we have gone beyond the end; go to step 7.

   If not, invert *dp,* get the next character, and go to to step 4.

   If not, go to step 4.

4. See if the character is an ASCII decimal digit.

   If it isn't, we may be done; go to step 5.

   If it is, multiply the floating-point accumulator by 10 to make room for the new digit.

   Force the digit to binary.

   Multiply the result by eight to form a pointer into a table of extended floats.

   Add this new floating-point digit to the accumulator.

   Check *dp_flag* to determine whether we have passed a decimal point and should be decrementing *dp* to count the fractional digits. If so, decrement *dp.*

   Decrement the digit counter, *digits.*

   Get the next character.

   Return to the beginning of step 3.

5. Get the next character and force it to lowercase.

   Check to see whether it's an "e"; if not, go to step 7.

   Otherwise, get the next character and check it to see whether it's a hyphen.

   If so, complement *expsin,* the exponent sign, and go to step 6.

   Otherwise, check for a "+."

   If it's not a "+," go to step 6a.

   If it is, go to step 6.

6. Get the next character.

   6a. See if the character is a decimal digit.

       If not, go to step 7.

       Otherwise, multiply the exponent by 10 and save the result.

       Subtract 30H from the character to force it to binary and *OR* it with the exponent.

       Continue with step 6.

7. See if *expsin* is negative.

   If it is, subtract *exponent* from dp and leave the result in the CL register.

   If not, add *exponent* to dp and leave the result in CL.

8. If the sign of the number, *numsin,* is negative, force the extended float

to negative.

If the sign of the number in CL is positive, go to step 10.

Otherwise, two's-complement CL and go to step 9.

9. Test CL for zero.

   If it's zero, go to step 11.

   If not, increment a loop counter. Test CL to see whether its LSB has a zero.

   If so, multiply the value of the loop counter by eight to point to the proper power of 10. Divide the floating-point accumulator by that power of 10 and shift CL right once. Continue with the beginning of step 9. (These powers of 10 are located in a table labeled 10. For this scheme to work, these powers of 10 follow the binary order 1, 2, 4, 8, as shown in the table immediately preceding the code.)

   If not, shift CL right once for next power of two and continue at the beginning of step 9.

10. Test CL for zero.

    If it's zero, go to step 11.

    If not, increment a loop counter and test CL to see whether its LSB is a zero.

    If so, multiply the value of the loop counter by eight to point to the properpower of 10. Multiply the floating-point accumulator by that power of 10, shift CL right once, and continue with the beginning of step 10. (These powers of 10 are located in a table labeled '10'. Again, for this scheme to work, these powers of 10 must follow the binary order 1, 2, 4, 8, as shown in the table immediately preceding the code.)

    If not, shift CL to the right once and continue with the beginning of step 10.

11. Round the new float, write it to the output, and leave.

atf: Listing

```
;*****
;
        .data
dst     qword       000000000000h, 3f8000000000h, 400000000000h, 404000000000h,
                    408000000000h, 40a000000000h, 40c000000000h, 40e000000000h,
                    410000000000h,  411000000000h
one     qword       3f8000000000h
ten     qword       412000000000h,  42c800000000h,
```

```
                    461c40000000h,
                    4cbebc200000h,  5a0elbc9bf00h,  749dc5ada82bh
        .code

;unsigned conversion from ASCII string to short real
atf     proc uses si di, string:word, rptr:word ;one word for near pointer


        local exponent:byte, fp:qword, numsin:byte, expsin:byte.
                    dp_flag:byte, digits:byte, dp:byte

        pushf
        std
        xor         ax,ax
        lea         di,word ptr fp[6]           ;clear the floating
                                                ;variable
        mov         cx,8
rep     stosw       word ptr [di]

        mov         si,string                   ;pointer to string

do_numbers:
        mov         byte ptr [exponent],al      ;initialize variables
        mov         byte ptr dp_flag,al
        mov         byte ptr numsin,al
        mov         byte ptr expsin,al
        mov         byte ptr dp,al
        mov         byte ptr digits,8h          ;count of total digits rounding
                                                ;digit is eight

                                                ;begin by checking for a
                                                ;sign, a number, or a
                                                ;period
;
do_num:
        mov         bl, [si]                    ;get next char
        cmp         bl,'-'
        jne         not minus                   ;it is a negative number
        not         [numsin]                    ;set negative flag
        inc         si
        mov         bl,es:[si]                  ;get next char
        jmp         not sign

not_minus:
        cmp         bl,'+'                      ;is it plus?
```

```
        jne         not_sign
        inc         si
        mov         al, [si]                   ;get next char

not_sign:
        cmp         bl,'.'                     ;check for decimal point
        jne         not_dot
        test        byte ptr [dp],80h          ;negative?
        jne         end_o_cnvt                 ;end of conversion
        not         dp_flag                    ;set decimal point flag
        inc         si
        mov         bl, [si]                   ;get next char

not_dot:
        cmp         bl,'0'                     ;get legitimate number
        jb          not_a_num
        cmp         bl,'9'
        ja          not_a_num
        invoke      flmul, fp, ten, addr fp    ;multiply floating-point
        mov         bl, [si]                   ;accumulator by 10.0
        sub         bl,30h                     ;make it hex
        sub         bh,bh                      ;clear upper byte
        shl         bx,1                       ;multiply index for
                                               ;proper offset
        shl         bx,1
        shl         bx,1
        invoke      fladd, fp, dgt[bx], addr fp ;add to floating-point
                                               ;accumulator
        test        byte ptr [dp_flag],0ffh    ;have we encountered a
        je          no_dot_yet                 ;decimal point yet?
        dec         [dp]                       ;count fractional digits
no_dot_yet:
        inc         si                         ;increment pointer
        dec         byte ptr digits            ;one less digit
        jc          not_a_num                  ;at our limit?
        mov         bl,es:[si]                 ;next char
        jmp         not_sign

not_a_num:
        mov         bl, [si]                   ;next char
        or          bl,lower_case
        cmp         bl,'e'                     ;check for exponent
        je          chk_exp                    ;looks like we may have
                                               ;an exponent
```

```
        jmp             end_o_cnvt
chk_exp:
        inc             si
        mov             bl, [si]                        ;next char
        cmp             bl,'-'                          ;negative exponent
        jne             chk_plus
        not             [expsin]                        ;set exponent sign
        jmp             short chk_expl
chk_plus:
        cmp             bl,'+'                          ;maybe a plus?
        jne             short chk_exp2
chk_expl:
        inc             si
        mov             bl, [si]                        ;next char
chk_exp2:
        cmp             bl,'0'
        jb              end_o_cnvt
        cmp             bl,'9'
        ja              end_o_cnvt
        sub             ax, ax
        mov             al, byte ptr [exponent]
        mul             iten                            ;do conversion of
                                                        ;exponent as in
        mov             byte ptr [exponent],al          ;integer conversion
                                                        ;by multiplication
        mov             bl, [si]                        ;next char
        sub             bl,30h                          ;make hex
        or              byte ptr [exponent],bl          ;or into accumulator
        jmp             short chk_expl

end_o_cnvt:
        sub             cx, cx                          ;calculate exponent
        mov             al,byte ptr [expsin]
        mov             cl,byte ptr [dp]
        or              al,al                           ;is exponent negative?
        jns             Pos_exp
        sub             cl,byte ptr [exponent]          ; subtract exponent from
                                                        ;fractional count
        jmp             short chk_numsin
pas_exp:
        add             cl,byte ptr [exponent]          *exponent to fractional
                                                        ;count
chk_numsin:
        cmp             word ptr numsin,0ffh            ;test sign
```

```
        jne         chk_expsin
        or          word ptr fp[4],8000h      ;if exponent negative,
chk_expsin:                                   ;so is number
        xor         ax,ax
        or          cl,cl
        jns         do_pospow                 ;make exponent positive
        neg         cl
do_negpow:
        or          cl,cl                     ;is exponent zero yet?
        je          atf_ex
        inc         ax
        test        cx,1h                     ;check for one in LSB
        je          do_negpowa
        mov         bx,ax
        push        ax
        shl         bx,1                      ;make pointer
        shl         bx,1
        shl         bx,1
        invoke      fldiv, fp, powers[bx], addr fp
                                              ;divide by power of 10
        pop         ax
do_negpowa:
        shr         cx, 1
        jmp         short do_negpow

do_pospow:
        or          cl,cl                     ;is exponent zero yet?
        je          atf ex
        inc         ax
        test        cx,lh                     ;check for one in LSB
        je          do_pospowa
        mov         bx,ax
        push        ax
        shl         bx,l
        shl         bx,l
        shl         bx,l                      ;make pointer
        invoke      flmul, fp, powers[bx], addr fp
                                              ;multiply by power often
        pop         ax
do_pospowa:
        shr         cx,1
        jmp         short do_pospow
atf_ex:
        invoke      round, fp, addr fp        ;round the new float
```

```
        mov      di,word ptr rptr          ;write it out
        mov      ax,word ptr fp
        mov      bx,word ptr fp[2]
        mov      dx,word ptr fp[4]
        mov      word ptr [di],bx
        mov      word ptr [di][2],dx
        popf
        ret
atf     endp
```

Single-Precision Float to ASCII

This function is usually handled in C with *fcvt()* plus some ancillary routines that format the resulting string. The function presented here goes a bit further; its purpose is to convert the float from binary to an ASCII string expressed in decimal scientific format.

Scientific notation requires its own form of normalization: a single leading integer digit between 1.0 and 10.0. The float is compared to 1.0 and 10.0 upon entry to the routine and successively multiplied by 10.0 or divided by 10.0 to bring it into the proper range. Each time it is multiplied or divided, the exponent is adjusted to reflect the correct value.

When this normalization is complete, the float is disassembled and converted to fixed point. The sign, which was determined earlier in the algorithm, is positioned as the first character in the string and is either a hyphen or a space. Each byte of the fixed-point number is then converted to an ASCII character and placed in the string. After converting the significand, the routine writes the value of the exponent to the string.

In pseudocode, the procedure might look like this.

fta: Algorithm

1. Clear a variable, *fixptr,* large enough to hold the fixed-point conversion. Allocate and clear a sign flag, *sinptr.* Do the same for a flag to suppress leading zeros *(leading zeros),* a byte to hold the exponent, and a byte to count the number of multiplies or divides it takes to normalize the number, *ndg.*

2. Test the sign bit of the input float. If it's negative, set *sinptr* and make the float positive.

3. Compare the input float to 1.0.

   If it's greater, go to step 4.

   If it's less, multiply it by 10.0. Decrement *ndg* and check for underflow.

   If underflow occurred, go to step 18.

   If not, return to the beginning of step 3.

4. Compare the float resulting from step 3 to 10.0.

   If it's less, go to step 5.

   If it's greater, divide by 10.0. Increment *ndg* and check for overflow.

   If overflow occurred, go to step 17.

   If not, return to the beginning of step 4.

5. Round the result.

6. Extract the exponent, subtract the bias, and check for zero. If we underflow here, we have an infinite result; go to step 17.

7. Restore the hidden bit. Using the value resulting from step 6, align the significand and store it in the fixed-point field pointed to by *fixptr.* We should now have a fixed-point value with the radix point aligned correctly for scientific notation.

8. Start the process of writing out the ASCII string by checking the sign and printing hyphen if *sinptr* is -1 and a space otherwise.

9. Convert the fixed-point value to ASCII with the help of *AAM* and call step 19 to write out the integer.

10. Write the radix point.

11. Write each decimal digit as it's converted from the binary fractional portion of the fixed-point number until eight characters have been printed.

12. Check *ndg* to see whether any multiplications or divisions were necessary to force the number into scientific format.

    If *ndg* is zero, we're done; terminate the string and exit through step 16.

    If *ndg* is not zero, continue with step 13.

13. Print the "e."

14. Examine the exponent for the appropriate sign. If it's negative, print hyphen and two's-complement *ndg.*

15. Convert the exponent to ASCII format, writing each digit to the output.
16. Put a NULL at the end of the string and exit.
17. Print "infinite" to the string and return with failure, AX = -1.
18. Print "zero" to the string and return with failure, AX = -1.
19. Test to see whether or not any zero is leading.
    If so, don't print-just return.
    If not, write it to the string.

Fta: Listing

```
; *****

; conversion of floating point to ASCII


fta     proc uses bx cx dx si di, fp:qword, sptr:word
        local        sinptr:byte, fixptr:qword, exponent:byte.
                     leading_zeros:byte, ndg:byte

        pushf
        std

        xor          ax,ax                          ;clear fixed-point
                                                    ;variable
        lea          di,word ptr fixptr[6]
        mov          cx,4
rep     stosw
        mov          byte ptr [sinptr],al           ;clear the sign
        mov          byte ptr [leadin_zeros],al     ;and other variables
        mov          byte ptr [ndg],al
        mov          byte ptr [exponent],al

ck_neg:
        test         word ptr fp[4],8000h           ;get the sign
        je           gtr_0
        xor          word ptr fp[4],8000h           ;make float positive
        not          byte ptr [sinptr]              ;set sign
                                                    ;negative
;
; ***
gtr_0:                                              ;compare input with 1.0
        invoke       flcomp, fp, one                ;still another kind of
```

```
                                                ;normalization
        cmp        ax,1h                        ;argument reduction
        je         less_than_ten
        dec        byte ptr [ndg]               ;decimal counter
        cmp        byte ptr [ndg],-37           ;range of single-
                                                ;precision float
        jl         zero_result
        invoke     flmul, fp, ten, addr fp      ;multiply by 10.0
        jmp        short gtr_0

less_than_ten:                                  ;compare with 10.0
        invoke     flcomp, fp, ten
        cmp        ax,-1
        je         norm fix
        inc        byte ptr [ndg]               ;decimal counter
        cmp        byte ptr [ndg],37            ;orange of single-
                                                ;precision float
        jg         infinite result
        invoke     fldiv, fp, ten, addr fp      ;divide by 10.0
        jmp        short less_than_ten

Rnd:
        invoke     round, fp, addr fp           ;fixup for translation

norm_fix:                                       ;this is for ASCII
                                                ;conversion
        mov        ax,word ptr fp[0]            ;dump the sign bit
        mov        bx,word ptr fp[2]
        mov        dx,word ptr fp[4]
        shl        dx, 1

get_exp:
        mov        byte ptr exponent, dh
        sub        byte ptr exponent, 7fh       ;remove bias

        mov        cx,sh
        sub        cl,byte ptr exponent
        js         infinite_result              ;could come out zero
                                                ;but this is as far as I
        lea        di,word ptr fixptr           ;can go
do_shift:
        stc                                     ;restore hidden bit
        rcr        dl,1
        sub        cx, 1
```

```
        je          put_upper
shift_fraction:
        shr         dl,1                        ;shift significand into
        rcr         bx,1                        ;fractional part
        rcr         ax,1
        loop        shift fraction

put_upper:
        mov         word ptr [di], ax           ;write to fixed-point
                                                ;variable
        mov         word ptr [di][2],bx
        mov         al,dl
        mov         byte ptr fixptr[4],dl       ;write integer portion
        xchg        ah,al

        sub         dx,dx
        mov         di,word ptr sptr
        cld                                     ;reverse direction of
                                                ;write
        inc         dx
        mov         al,' '
        cmp         byte ptr sinptr,0ffh        ;is it a minus?
        jne         put_sign
        mov         al,'-'
put_sign:
        stosb

        lea         si, byte ptr fixptr[3]

write_integer:
        xchg        ah,al                       ;AL contains integer
                                                ;portion
        aam
        xchg        al,ah                       ;use AAM to convert to
                                                ;decimal
        or          al,'0'                      ;then ASCII
        call        near ptr str wrt            ;then write to string
        xchg        al,ah                       ;and repeat
        or          al,'0'
        call        near ptr str_wrt
        inc         dx                          ;max char count
        dec         si

do_decimal:
```

```
        mov         al,'.'                      ;decimal point
        stosb
do_decimall:
        invoke      multen, addr fixptr         ;convert binary fraction
        or          al,'0'                      ;to decimal fraction
        call        near ptr str wrt            ;write to string
        inc         dx
        cmp         dx,maxchar                  ;have we written our
                                                ;maximum?
        je          do_exp
        jw          short do_decimal1
do_exp:
        sub         ax,ax
        cmp         al,byte ptr ndg             ;is there an exponent?
        jne         write exponent
        jmp         short last_byte

write_exponent:
        mov         al,'e'                      ;put the 'e'
        stosb
        mov         al,byte ptr ndg             ;with ndg calculate
        or          al,al                       ;exponent
        jns         finish_exponent
        xchg        al,ah
        mo          val,'-'                     *negative exponent
        stosb
        neg         ah
        xchg        al,ah
        sub         ah,ah
finish exponent
        cbw                                     ;sign extension
        aam                                     ;cheap conversion
        xchg        ah,al
        or          al,'0'
        stosb
        xchg        ah,al
        or          al,'0'                      ;make ASCII
        stosb
last_byte:
        sub         al,al                       ;write NULL to the end
                                                ;of the string
        stosb
        popf
fta_ex:
```

```
        ret

infinite_result:
        mov          di,word ptr sptr              ;actually writes
                                                   ;'infinite'
        mov          si,offset inf
        mov          cx, 9
rep     movsb
        mov          ax,-1
        jmp          short fta_ex

zero_result:
        mov          di,word ptr sptr              ;actually writes 'zero'
        mov          si,offset zro
        mov          cx, 9
rep     movsb
        mov          ax,-1
        jmp          short fta_ex

str_wrt:                                           ;subroutine for writing
                                                   ;characters to output
        cmp          al,'0'                        ;string
        jne          putt                          ;check whether leading
                                                   ;zero or not
        test         byte ptr leading_zeros,-1     ;don't want any leading
                                                   ;zeros
        je           nope
putt:
        test         byte ptr leading_zeros,-1
        jne          prnt
        not          leading_zeros
prnt:
        stosb
nope:
        retn
fta     endp
```

Fixed Point to Single-Precision Floating Point

For this conversion, assume a fixed-point format with a 32-bit integer and a 32-

bit fraction. This allows the conversion of *longs* and *ints* as well as purely fractional values; the number to be converted need only be aligned within the fixed-point field.

The extended-precision format these routines use requires three words for the significand, exponent, and sign; therefore, if we shift the fixed-point number so that its topmost one is in bit 7 of the fourth byte, we're almost there. We simply need to count the shifts, adding in an appropriate offset and sign. Here's the procedure.

ftf: Algorithm

1. The fixed-point value *(binary)* is on the stack along with a pointer *(rptr)* to the float to be created. Flags for the exponent *(exponent)* and sign *(nmsin)* are cleared.
2. Check the fixed-point number for sign. If it's negative, two's-complement it.
3. Scan the fixed-point number to find out which byte contains the most significant nonzero bit.

   If the number is all zeros, return the appropriate float at step 9.

   If the byte found is greater than the fourth, continue with step 5.

   If the byte is the fourth, continue with step 6.

   If the byte is less than the fourth, continue with step 4.
4. The most significant non zero bit is in the first, second, or third byte.

   Subtract our current position within the number from four to find out how many bytes away from the fourth we are.

   Multiply this number by eight to get the number of bits in the shift and put this value in the exponent.

   Move as many bytes as are available up to the fourth position, zeroing those lower values that have been moved and not replaced.

   Continue with step 6.
5. The most significant nonzero bit is located in a byte position greater than four.

   Subtract four from our current position within the number to find how many bytes away from the fourth we are.

   Multiply this number by eight to get the number of bits in the shift and put this value in the exponent.

   Move these bytes back so that the most significant nonzero byte is in the fourth position.

Continue with step 6.

6. Test the byte in the fourth position to see whether bit 7 contains a one.

   If so, continue with step 7.

   If not, shift the first three words left one bit and decrement the exponent; continue at the start of step 6.

7. Clear bit 7 of the byte in the fourth position (this is the hidden bit).

   Add 86H to *exponent* to get the correct offset for the number; place this in byte 5.

   Test *numsin* to see whether the number is positive or negative.

   If it's positive, shift a zero into bit 7 of byte 5 and continue with step 8.

   If it's negative, shift a one into bit 7 of byte 5 and continue with step 8.

8. Place bytes 4 and 5 in the floating-point variable, round it, and exit.

9. Write zeros to every word and exit.

ftf; Listing

```
; *****
;
;
;unsigned conversion from quadword fixed point to short real
;The intention is to accommodate long and int conversions as well.
;Binary is passed on the stack and rptr is a pointer to the result.

ftf     proc uses si di, binary:qword, rptr:word     ;one word for near
                                                     ;pointer

        local exponent:byte, numsin:byte

        pushf
        xor         ax, ax

        mov         di, word ptr rptr                ;point at future float
        add         di, 6
        lea         si, byte ptr binary[0]           ;point to quadword
        mov         bx, 7                            ;index

do_numbers:
        mov         byte ptr [exponent], al          ;clear flags
```

```
        mov         byte ptr nurnsin, al
        mov         dx, ax
;
do_num:
        mov         al, byte ptr [si] [bx]
        or          al, al                      ;record sign
        jns         find top
        not         byte ptr numsin             ;this one is negative
        not         word ptr binary[6]
        not         word ptr binary[4]
        not         word ptr binary[2]
        neg         word ptr binary[0]
        jc          find_top
        add         word ptr binary[2], 1
        adc         word ptr binary[4], 0
        adc         word ptr binary[6], 0

find_top:
        cmp         bl, dl                      ;compare index with 0
        je          make zero                   ;we traversed the entire
                                                ;number
        mov         al, byte ptr [si][bx]       ;get next byte
        or          al,al                       ;anything there?
        jne         found it
        dec         bx                          ;move index
        jw          short find_top

found_it:
        mov         dl, 80h                     ;test for MSB
        cmp         bl, 4                       ;radix point
        ja          shift right                 ;above
        je          final right                 ;equal

shift_left                                      ;or below?
        std
        mov         cx, 4                       ;points to MSB
        sub         cx, bx                      ;target
        shl         cx, 1
        shl         cx, 1
        shl         cx, 1                       ;times 8
        neg         cx
        mov         byte ptr [exponent], cl     ;calculate exponent

        lea         di, byte ptr binary[4]
        lea         si, byte ptr binary
```

```
        add     si, bx
        mov     cx, bx
        inc     cx
rep     movsb                               ;move number for nomalization

        mov     cx, 4
        sub     cx, bx
        sub     ax, ax
rep     stosb                               ;clear unused bytes
        jmp     short final_right

shift_right:
        cld
        mov     cx, bx                      ;points to MSB
        sub     cx, 4                       ;target
        lea     si, byte ptr binary[4]
        mov     di, si
        sub     di, cx

        shl     cl, 1
        shl     cl, 1
        shl     cl, 1                       ;times 8
        mov     byte ptr [exponent], cl     ;calculate exponent

        mov     cx, bx
        sub     cx, 4
        inc     cx
rep     movsb
        sub     bx, 4
        mov     cx, 4
        sub     cx, bx
        sub     ax, ax
        lea     di, word ptr binary
rep     stosb                               ;clear bytes

final_right:
        lea     si, byte ptr binary[4]      ;get most significant one into
                                            ;MSB
final_rightl:
        mov     al, byte ptr [si]
        test    al, dl                      ;are we there yet?
        jne     aligned
        dec     byte ptr exponent
```

```
        shl         word ptr binary[0], 1
        rcl         word ptr binary[2], 1
        rcl         word ptr binary[41, 1
        jmp         short final_right1

aligned:
        shl         al, 1                       ;clear bit
        mov         ah, 86h                     ;offset so that exponent will be
                                                ;right after addition
        add         ah, byte ptr exponent
        cmp         numsin, dh
        je          positive
        stc
        jmp         short get_ready_to_go
positive:
        clc

get_ready_to_go:                                ;shift carry into MSB
        rcr         ax, 1                       ;put it all back the way it
                                                ;should be
        mov         word ptr binary[4], ax
ftf_ex:
        invoke round, binary, rptr              ;round the float

exit:
        popf
        ret
;
make_zero:                                      ;nothing but zeros
        std
        sub         ax, ax                      ;zero it all out
        mov         cx, 4
rep     stosw
        jmp         short exit
ftf     endp
```

Single-Precision Floating Point to Fixed Point

Ftfx simply extracts the fixed-point number from the IEEE 754 floating-point format and places it, if possible, in a fixed-point field composed of a 32-bit integer and a 32-bit fraction. The only problem is that the exponent might put the significand outside our fixed-point field. Of course, the field can always be changed; for this routine, it's the quadword format with the radix point between the doublewords.

fttx Algorithm

1. Clear the sign flag, *sinptr,* and the fixed-point field it points to.
2. Round the incoming floating-point number.
3. Set the sign flag through *sinptr* by testing the MSB of the float.
4. Extract the exponent and subtract the bias. Restore the hidden bit.
5. Test the exponent to see whether it's positive or negative.

   If it's negative, two's complement the exponent and test the range to see if it's within 28H.

   If it's greater, go to step 9.

   If it's not greater, continue with step 7.

   If positive, test the range to see if it's within 18H.

   If it's less, go to step 10.

   If not, continue with step 6.
6. Shift the integer into position and go to step 8.
7. Shift the fraction into position and continue with step 8.
8. See if the sign flag is set.

   If not, exit with success.

   If it is, two's-complement the fixed-point number and exit.
9. Error. Write zeros to all words and exit.
10. Write a 0ffH to the exponent and exit.

fftx: Listing

```
;*****
; conversion of floating point to fixed point
; Float enters as quadword.
; Pointer, sptr, points to result.
; This could use an external routine as well. When the float
; enters here, it is in extended format.
```

```
ftfx    proc uses bx cx dx si di, fp:qword, sptr:word

        local       sinptr:byte, exponent:byte

        pushf
        std
;
        xor         ax,ax
        mov         byte ptr [sinptr],al            ;clear the sign
        mov         byte ptr [exponent],al
        mov         di,word ptr sptr                ;point to result
;
;***
;
do_rnd:
        invoke      round, fp, addr fp              ;fixup for translation
;
set_sign:
        mov         ax,word ptr fp[0]               ;get float
        mov         bx,word ptr fp[2]
        mov         dx,word ptr fp[41
        or          dx,dx                           ;test exponent for sign
        jns         get_exponent
        not         byte ptr [sinptr]               ;it is negative
;
get_exponent:
        sub         cx,cx
        shl         dx,1                            ;dump sign
        sub         dh,86h                          ;remove bias from exponent
        mov         byte ptr exponent, dh           ;store exponent
        mov         cl,dh
        and         dx,0ffh                         ;save number portion
        stc
        rcr         dl,l                            ;restore hidden bit
;
which_way:
        or          cl,cl                           ;test for sign of exponent
        jns         shift_left
        neg         cl                              ;two's complement if negative

shift_right:
        cmp         cl,28h                          ;range of fixed-point number
        ja          make_zero                       ;no significance too small
make_fraction:
```

```
        shr         dx,1                         ;shift fraction into position in
fixed                                            ;point variable
        rcr         bx,1
        rcr         ax,1
        loop        make fraction
        mov         word ptr [di] [0],ax         ;and write result
        mov         word ptr [di] [2],bx
        mov         word ptr [di][4],dx
        imp         short  print_result

shift_left:
        cmP         cl,18h                       ;range of fixed point
                                                 ;(with  significand)
        ja          make_max                     ;failed  significance too big
make_integer
        shr         bx,1                         ;shift  into  position
        rcr         dx,1
        rcr         ax,1
        loop        make_integer
        mov         word ptr [di][6],ax          ;write out
        mov         word ptr [di][4],dx
        mov         word ptr [di][2],bx
print_result
        test        byte ptr  [sinptr],  0ffh    ;check  for  proper  sign
        ie          exit
        not         word ptr  [di] [6]           ;two's  complement
        not         word ptr  [di][4]
        not         word ptr  [di] [2]
        neg         word ptr  [di] [0]
        ic          exit
        add         word ptr  [di]  [2],1
        adc         word ptr  [di]  [4],0
        adc         word  ptr  [di]  [6],0

exit:
        popf
        ret
;
make_zero:                                       ;error  make  zero
        sub         ax,ax
        mov         cx,4
rep     stosw
        imp         short  exit
;
```

```
make_max:                                           ;error too big
        sub         ax,ax
        mov         cx,2
rep     stosw
        not         ax
        stosw
        and         word ptr [di][4], 7f80h         ;infinite
        not         ax
        stosw
        jmp         short exit


ftfx    endp
```

[1] Knuth, D. E. *Seminumerical Algorithms.* Reading, MA: Addison-Wesley Publishing Co., 1981, Pages 300-312.

CHAPTER 6

The Elementary Functions

According to the *American Heritage Dictionary,* elementary means essential, fundamental, or irreducible, but not necessarily simple. The elementary functions comprise algebraic, trigonometric, and transcendental functions basic to many embedded applications. This chapter will focus on three ways of dealing with these functions: simple table-driven techniques that use linear interpolation; computational fixed-point, including the use of tables, CORDIC functions, and the Taylor Series; and finally floating-point approximations. We'll cover sines and cosines along with higher arithmetic processes, such as logarithms and powers.

We'll begin with a fundamental technique of fixed-point arithmetic—table lookup—and examine different computational methods, ending with the more complex floating-point approximations of the elementary functions.

Fixed Point Algorithms

Lookup Tables and Linear Interpolation

In one way or another, many of the techniques in this chapter employ tables. The algorithms in this section derive their results almost exclusively through table lookup. In fact, you could rewrite these routines to do only table lookup, if that is all you require.

Some of the fastest techniques for deriving values involve look-up tables. As mentioned in Chapter 5, the main disadvantage to table-driven routines is that the tables are finite. Therefore, the results of any table-driven routine depends upon the table's resolution. The routines in this section involve an additional step to help alleviate these problems: linear interpolation.

The idea behind interpolation is to approximate unknown data points based upon information provided by known data points. *Linear interpolation* attempts to do this by bridging two known points with a straight line as shown in Figure 6-1. Using this technique, we replace the actual value with the function y=f(x), where $y = y_0+(x-x_0)(y_1-y_0)/(x_1-x_0)$. This formula represents the *slope* of the line between the two known data points, with $[f(x_1)-f(x_0)]/(x_1-x_0)$ representing an approximation of the first derivative (the finite divided difference approximation). As you can see from Figure 6-1, a straight line is not likely to represent the shape of a function well and can result in a very loose approximation if too great a distance lies between each point of known data.

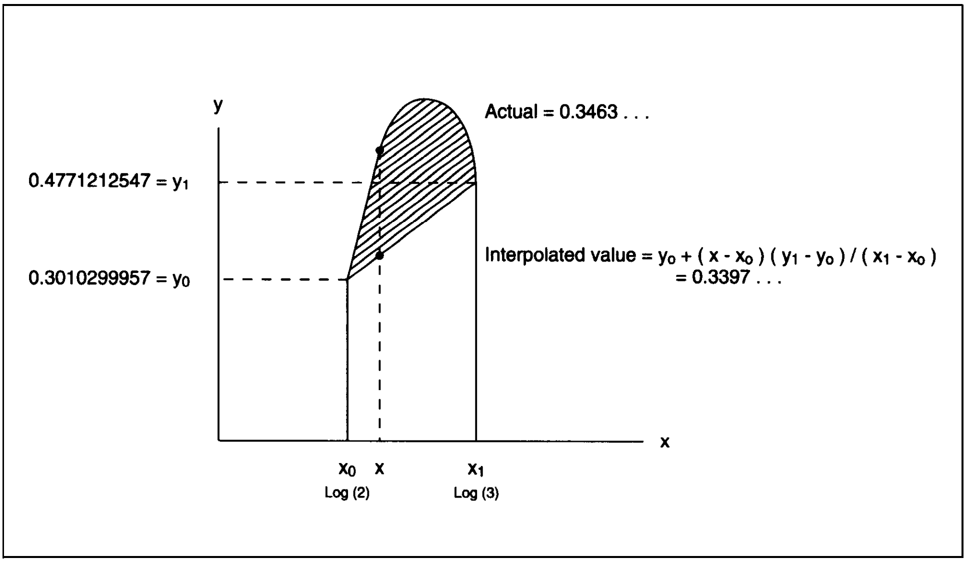


Figure 6-1. Linear interpolation produces an approximate value based on a straight line drawn between known data. The closer the data points, the better the approximation.

Consider the problem of estimating $\log_{10}(2.22)$ based on a table of common logs for integers only. The table indicates that $\log_{10}(2) = 0.3010299957$ and $\log_{10}(3) = 0.4771212547$. Plugging these values into the formula above, we get:

```
y=0.3010299957+(2.22-2.0)(0.4771212547-0. 3010299957) / (3-2
y=0.3010299957 + (.22) (0.1760182552)/(l)
y=0.3397540118.
```

The true value is 0.3463529745, and the error is almost 2%. For more accuracy, we would need more data points on a finer grid.

An example of this type of table-driven approximation using linear interpolation is the function *lg10,* presented later in this section.[1] The derivation of the table used in this routine was suggested by Ira Harden. This function produces $\log_{10}(X)$ of a value based on a table of common logarithms for X/128 and a table of common logarithms for the powers of two. Before looking the numbers up in the table, it normalizes each input argument (in other words, shifts it left until the most significant one of the number is the MSB of the most significant word) to calculate which power of two the number represents. The MSB is then shifted off, leaving an index that is then used to point into the table.

If any fraction bits need to be resolved, linear interpolation is used to calculate a closer approximation of our target value. The log of the power of two is then added in, and the process is complete.

The function *lg10* approximates $\log_{10}(X)$ using a logarithm table and fixed-point arithmetic, as shown in the following pseudocode:

lg10: Algorithm

1. Clear the output variable. Check the input argument for 0.

   If zero, exit

   If not, check for a negative argument.

   If so, exit

   If all OK, continue with step 2.

2. Determine the number of shifts required to normalize the input argument, that is so that the MSB is a one. Perform that shift first by moves and then individual shifts.

3. Perform linear interpolation.

   First get the nominal value for the function according to the table. This is the $f(x_0)$ from the equation above. It must be guaranteed to be equal to or less than the value sought.

   Get the next greater value from the table, $f(x_1)$. This isguaranteed to be greater than the desired point.

   Now multiply by the fraction bits associated with the number we using to point into the table. These fraction bits represent the difference between the nominal data point, x_0, and the desired point.

Add the interpolated value to the nominal value and continue with step 4.

4. Point into another table containing powers of logarithms using the number of shifts required to normalize the number in step 2. Add the logarithm of the power of two required to normalize the number.
5. Exit with the result in quadword fixed-point format.

lg10: Listing

```
; *****
      .data

; log (x/128)     ;To create binary tables from decimal, multiply the decimal
                  ;value you wish to use by one in the data type of your
                  ;fixed-point system. For example, we are using a 64-bit fixed
                  ;point format, a 32-bit fraction and a 32-bit integer. In
                  ;this system, one is 2^32, or 4294967296 (decimal), convert
                  ;the result of that multiplication to hexadecimal and you are
                  ;done. To convert p  to our format we would multiply 3.14 by
                  ;4294967296 with the result 13493037704 (decimal), which we
                  ;then convert to the hexadecimal value 3243f6a89H.

log10 tbl word    00000h, 000ddh, 001b9h, 00293h,
                  0036bh, 00442h, 00517h, 005ebh, 00Gbdh, 0078eh,
                  0085dh, 0092ah, 009f6h, 00ac1h, 00b8ah, 00c51h,
                  00d18h, 00dddh, 00ea0h, 00f63h, 01024h, 0l0e3h,
                  011a2h, 0125fh, 0131bh, 013dSh, 0148fh, 01547h,
                  015feh, 016b4h, 01769h, 0181ch, 018cfh, 01980h

      word        01a30h, 01adfh, 01b8dh, 01c3ah, 01ceGh, 01dg1h,
                  01e3bh, 01ee4h, 01f8ch, 02033h, 020d9h, 0217eh,
                  02222h, 022c5h, 02367h, 02409h, 024a9h, 02548h,
                  025e7h, 02685h, 02721h, 027bdh, 02858h, 028f3h

      word        0298ch, 02a25h, 02abdh, 02b54h, 02beah, 02c7fh,
                  02d14h, 02da8h, 02e3bh, 02ecdh, 02f5fh, 02ff0h,
                  03080h, 0310fh, 0319eh, 0322ch, 032b9h, 03345h,
                  033d1h, 0345ch, 034e7h, 03571h, 035fah, 03682h,
                  0370ah, 03792h, 03818h, 0389eh, 03923h, 039a8h

      word        03a2ch, 03abOh, 03b32h, 03bb5h, 03c36h, 03cb7h,
                  03d38h, 03db8h, 03e37h, 03eb6h, 03f34h, 03fb2h,
```

```
                0402fh, 040ach, 04128h, 041a3h, 0421eh, 04298h,
                04312h, 0438ch, 04405h, 0447dh, 044f5h, 0456ch,
                045e3h, 04659h, 046cfh, 04744h, 047b9h, 0482eh
        word    048a2h, 04915h, 04988h, 049fbh, 04a6dh, 04adeh,
                04b50h, 04bc0h, 04c31h, 04ca0h. 04d10h

;log(2**x)
log10_power dword 000000h, 004d1Oh, 009a20h, 00e730h, 013441h, 018151h,
                01ce61h, 021b72h, 026882h, 02b592h, 0302a3h, 034fb3h,
                039cc3h, 03e9d3h, 0436e4h, 0483f4h, 04d104h, 051e15h,
                056b25h, 05b835h, 060546h, 065256h, 069f66h, 06ec76h,
                073987h, 078697h, 07d3a7h, 0820b8h, 086dc8h. 08bad8h,
                0907e9h, 0954f9h

        .code
;
;Logarithms using a table and linear interpolation.
;Logarithms of negative numbers require imaginary numbers.
;Natural logarithms can be derived by multiplying result by 2.3025.
;Logarithms to any other base are obtained by dividing (or multiplying by the
;inverse of) the log10. of the argument by the log10 of the target base.
lgl0 proc uses bx cx si di, argument:word, 1ogptr:word

        local   powers_of_two:byte

        pushf
        std                                     ;increment down for zero
                                                ;check to come

        sub     ax, ax
        mov     cx, 4
        mov     di, word ptr logptr             ;clear log output
        add     di, 6
rep     stosw

        mov     si, word ptr logptr             ;point at output which is
                                                ;zero
        add     si, 6                           ;most significant word
        mov     di, word ptr argument           ;point at input
        add     di, 6                           ;most significant word
        mov     ax, word ptr [di]
        or      ax, ax
        js      exit                            ;we don't do negatives
        sub     ax, ax
```

```
        mov     cx, 4
repe    cmpsw                                   ;find the first nonzero,
                                                ;or return
                                                ;zero
        je      exit

reposition_argument:                            ;shift so MSB is a one
        mov     si, word ptr argument           ;point at input
        add     si, 6                           ;most significant word
        mov     di, si                          ;shift the one eight times
        inc     cx                              ;make this a one
        mov     ax, 4
        sub     ax, cx                          ;determinenumberof
                                                ;emptywords
        shl     ax, 1                           ;words to bytes
        sub     si, ax                          ;point to first nonnero word
        shl     ax, 1
        shl     ax, 1
        shl     ax, 1                           ;multiply by eight
        mov     bl, al
rep     movsw                                   ;shift

        mov     si, word ptr argument
        mov     ax, word ptr [si][6]
keep_shifting:
        or      ax, ax                          ;shift until MSB is a one
        js      done_with_shift
        shl     word ptr [si][0], 1
        rcl     word ptr [si][2], 1
        rcl     word ptr [si][4], 1
        rcl     ax, 1
        inc     bl                              ;count shifts as powers
                                                ;of two
        jmp     short keep_shifting             ;normalize

done_with_shift
        mov     word ptr [si][6],ax             ;ax will be a pointer
        mov     byte ptr powers_of_two, bl
        sub     bx, bx
        mov     bl, ah                          ;will point into 127-entry
                                                ;table
        shl     bl, 1                           ;get rid of top bit to form
                                                ;actual pointer
        add     bx, offset word ptr log10_tbl
```

```
                                            ;linear interpolation
        mov     ax, word ptr [bx]           ;get first approximation
                                            ;(floor)
        inc     bx
        inc     bx
        mov     bx, word ptr [bx]           ;and following approximation
                                            ;(ceil)
        sub     bx, ax                      ;find difference
        xchg    ax, bx

        mul     byte ptr [si][6]            ;multiply by fraction bits
        mov     al, ah                      ;drop fractional places
        Sub     ah, ah
        add     ax, bx                      ;add interpolated value to
                                            ;original

get_power:
        mov     bl, 31                      ;need to correct for power
                                            ;of two
        sub     bl, byte ptr powers_of_two
        sub     bh, bh
        shl     bx,1
        shl     bx,1                        ;point into this table
        lea     si, word ptr log10_power
        add     si, bx
        sub     dx, dx
        add     ax, word ptr [si]           ;add log of power
        adc     dx, word ptr [si][2]
        mov     di, word ptr logptr
        mov     word ptr [di] [2],ax        ;write result to qword
                                            ;fixed point
        mov     word ptr [di][4],dx
        sub     cx, cx
        mov     word ptr [di],cx
        mov     word ptr [di] [6],cx
exit:
        popf
        ret

lg10 endp
```

An example of linear interpolation appears in the TRANS.ASM module called *sqrtt*.

Another example using a table and linear interpolation involves sines and cosines. Here we have a table that describes a quarter of a circle, or 90 degrees, which the routine uses to find both sines and cosines. Since the only difference is that one is 90 degrees out of phase with the other, we can define one in terms of the other (see Figure 6-2). Using the logic illustrated by this figure, it is possible to calculate sines and cosines using a table for a single quadrant.

To use this method, however, we must have a way to differentiate between the values sought, sine or cosine. We also need a way to determine the quadrant the function is in that fixes the sign (see Figure 6-3).

Dcsin will produce either a sine or cosine depending upon a switch, *cs_flag*.

Dcsin: Algorithm

```
1. Clear sign, clear the output variable, and check the input argument
   for zero.
```

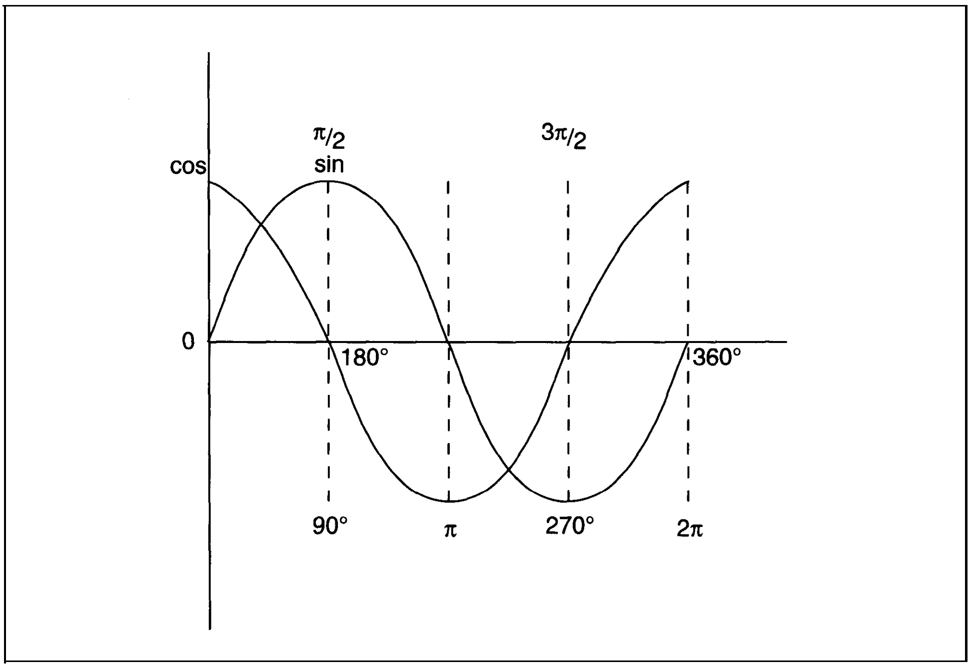


Figure 6-2. Sine and cosine are the same function 90 degrees out of phase.

If it is zero, set the output to 0 for sines and 1 for cosines.

Otherwise, continue with step 2.

2. Reduce the input argument by dividing by 360 (we are dealing in degrees) and take the remainder as our angle.

   If the result is negative, add 360 to make it positive.

3. Save a copy of the angle in a register, divide the original again by 90 to identify which quadrant it is in. The quotient of this division remains in AX.

4. Check *cs-flag* to see whether a sine or cosine is desired.

   A0h requests sine; continue with step 9.

   Anything else means a cosine; continue with step 5.

5. Compare AX with zero.

   If greater, go to step 6.

   Otherwise, continue with step 13.

6. Compare AX with one.

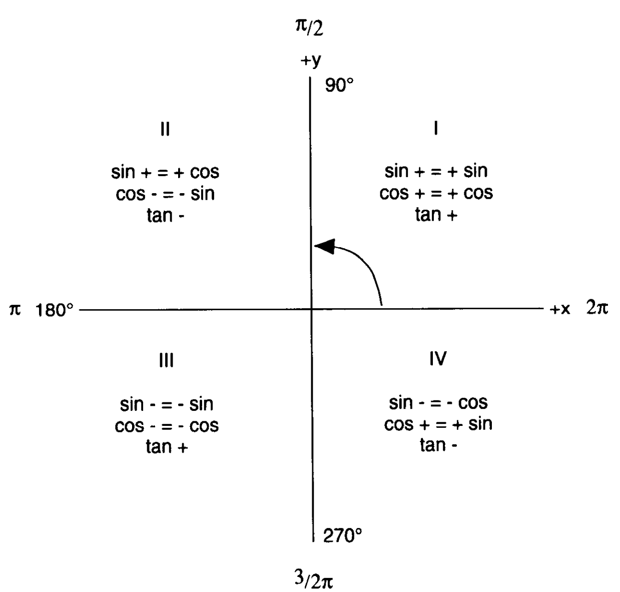


Figure 6-3. Quadrants for sine/cosine approximations.

If greater, go to step 7.

Otherwise, set *sign.*

Two's complement the angle.

Add 180 to make it positive again.

Continue with step 14.

7. Compare AX with two.

   If greater, go to step 8.

   Otherwise, set *sign.*

   Subtract 180 from the angle to bring it back within 90 degrees.

   Continue with step 13.

8. Two's complement the angle.

   Add 360 to point it back into the table.

   Continue with step 14.

9. Compare AX with zero.

   If greater, go to step 10.

   Otherwise, 2's complement the angle.

   Add 90 to point it into the table.

   Continue with step 14.

10. Compare AX with one.

    If greater, go to step ll.

    Otherwise, subtract 90 from the angle to bring it back onto the table.

    Continue with step 13.

11. Compare AX with two.

    If greater, go to step 12.

    Otherwise, two's complement the angle,

    Add 270, so that the angle points at table.

    Set *sign.*

    Continue with step 14.

12. Set *sign.*

    Subtract 270 from the angle.

13. Use the angle to point into the table.

Get $f(x_0)$from the table in the form of the nominal estimation of the sine.

Check to see if any fraction bits require linear interpolation.

If not, continue with step 15.

Get $f(x_1)$ from the table in the form of the next greater approximation.

Subtract $f(x_0)$ from $f(x_1)$ and multiply by the fraction bits.

Add the result of this multiplication to $f(x_0)$.

Continue with step 15.

14. Use the angle to point into the table.

    Get $f(x_0)$ from the table in the form of the nominal estimation of the sine.

    Check to see if any fraction bits requiring linear interpolation.

    If not, continue with step 15.

    Get $f(x_1)$ from the table in the form of the next smaller approximation.

    Subtract $f(x_0)$ from $f(x_1)$ and multiply by the fraction bits.

    Add the result of this multiplication to $f(x_0)$.

    Continue with step 15.

15. Write the data to the output and check *sign.*

    If it's set, two's complement the output and exit.

    Otherwise, exit.

Dcsin: Listing

```
; *****
        .data

;sines(degrees)
sine_tblword        0ffffh, 0fff6h, 0ffd8h, 0ffa6h, 0ff60h, 0ff06h,
                    0fe98h, 0fe17h, 0fd82h, 0fcdgh, 0fclch, 0fb4bh,
                    0fa67h, 0f970h, 0f865h, 0f746h, 0f615h, 0f4d0h,
                    0f378h, 0f20dh, 0f08fh, 0eeffh, 0ed5bh, 0eba6h,
                    0egdeh, 0e803h, 0e617h, 0e419h, 0e208h, 0dfe7h,
                    0ddb3h, 0db6fh, 0d919h, 0d6b3h, 0d43bh, 0d1b3h,
                    0cf1bh, 0cc73h, 0cgbbh, 0c6f3h, 0c4lbh, 0c134h,
                    0be3eh, 0bb39h, 0b826h, 0b504h, 0bld5h, 0ae73h
```

```
         word       0ab4ch, 0a7f3h, 0a48dh, 0a11bh, 09d9bh, 09a10h,
                    09679h, 092d5h, 08f27h, 08b6dh, 087a8h, 083d9h,
                    08000h, 07c1ch, 0782fh, 07438h, 07039h, 06c30h,
                    0681fh, 06406h, 05fe6h, 05bbeh, 0578eh, 05358h,
                    04flbh, 04ad8h, 04690h, 04241h, 03deeh, 03996h,
                    03539h, 030d8h, 02c74h, 0280ch, 023a0h, 01f32h,
                    01ac2h, 0164fh, 011dbh, 00d65h, 008efh, 00477h,
                    0h
        .code


;sines and cosines using a table and linear interpolation
;(degrees)

dscin proc uses bx cx si di, argument:word, cs_ptr:word, cs_flag:byte

        local       powers_of_two:byte, sign:byte

        pushf
        std                                         ;increment down

        sub         ax, ax
        mov         byte ptr sign, al               ;clear sign flag
        mov         cx, 4
        mov         di, wordptr cs_ptr              ;clear sin/cos output
        add         di, 6
rep     stosw
                                                    ;first check arguments
                                                    ;for zero
        add         di, 8                           ;reset pointer
        mov         si, di
        mov         di, word ptr argument
        add         di, 6
        mov         cx, 4
repe                cmpsw                           ;find the first nonzero,or
                                                    ;retum
        je          zero_exit
        jmp         prepare_arguments

zero_exit:
        cmp         byte ptr cs_flag, al            ;ax is zero
        jne         cos_0                           ;sin(0) = 0
        jmp         exit
cos_0:
```

```
        inc     ax
        inc     ax                          ;point di at base of
                                            ;output
        add     si,ax                       ;make ax a one
        dec     ax                          ;cos(0)= 1
        mov     word ptr [si][4],ax         ;one
        jmp     exit

prepare-arguments:
        mov     si, word ptr argument
        mov     ax, word ptr [si][4]        ;get integerportion
                                            ;of angle
        sub     dx, dx
        mov     cx, 360
        idiv    cx                          ;modulararithmeticto
                                            ;reduceangle
        or      dx, dx                      ;we want the remainder
        jns     quadrant
        add     dx, 360                     ;angle has to be
                                            ;positive for this
                                            ;to work
quadrant:
        mov     bx, dx                      ;we will use this to
                                            ;compute the value
                                            ;of the function
        mov     ax, dx                      ;put angle in ax
        sub     dx, dx
        mov     cx, 90
        div     cx                          ;and this to compute
                                            ;the sign ax holds
                                            ;an index to the quadrant
switch:
        cmp     byte ptr cs_flag, 0         ;what do we want?
                                            ;a zero=sine
                                            ;anything else=cosine
        je      do_sin


cos_range:
        cmp     ax, 0
        jg      cchk_l80
        jmp     walk_up                     ;use incrementing method
```

```
cchk_180:
        cmp         ax, 1
        jg          cchk_270
        not         byte ptr sign               ;set sign flag
        neg         bx
        add         bx, 180
        jmp         walk_back                   ;use decrementing method

cchk_270:
        cmp         ax, 2
        jg          clast_90
        not         byte ptr sign               ;set sign flag
        sub         bx, 180
        jmp         walk_up

clast_90:
        neg         bx
        add         bx, 360
        jmp         walk_back
;
;
;
do_sin:                                         ;find the range of the
                                                ;argument
        cmp         ax, 0
        jg          schk_180
        neg         bx
        add         bx, 90
        jmp         walk_back                   ;use decrementing method

schk_180:
        cmp         ax, 1
        jg          schk_270
        sub         bx, 90
        jmp         walk_up                     ;use incrementing method

schk_270:
        cmp         ax, 2
        jg          slast_90
        not         byte ptr sign               ;set sign flag
        neg         bx
        add         bx, 270
        jmp         walk_back
```

```
slast_90:
        not         byte ptr sign                  ;set sign flag
        sub         bx, 270
        jmp         walk_up
;

;
walk_up:
        shl         bx, 1                          ;use angle to point into
                                                   ;the table
        add         bx, offset word ptr sine_tbl
        mov         dx, word ptr [bx]              ;get cos/sine of angle
        mov         ax, word ptr [si][2]           ;get fraction bits
        or          ax, ax
        je          write_result
                                                   ;linear interpolation
        inc         bx                             ;get next approximation
        inc         bx
        mov         cx, dx
        mov         ax, word ptr [bx]
        sub         ax, dx                         ;find difference
        jnc         pos_res0
        neg         ax
        mul         word ptr [si][2]               ;multiply by fractionbits
        not         dx
        neg         ax
        jc          leave_walk_up
        inc         dx
        jmp         leave_walk_up
pos_res0:
        mul         word ptr [si] [2]
leave_walk_up:
        add         dx, cx                         ;multiply by fraction bits
                                                   ;and add in angle
        jmp         write-result

walk_back:
        shl         bx, 1                          ;point into table
        add         bx, offset word ptr sine_tbl
        mov         dx, word ptr [bx]              ;get cos/sine of angle
        mov         ax, word ptr [si][2]           ;get fraction bits
        or          ax, ax
```

```
        je          write_result

        dec         bx
        dec         bx
        mov         cx, dx
        mov         ax, word ptr [bx]           ;get next incremental
                                                ;cos/sine
        sub         ax, dx                      ;get difference
        jnc         pas_resl
        neg         ax
        mul         word ptr [si][2]            ;multiply by fraction bits
        not         dx
        neg         ax
        jc          leave_walk_back
        inc         dx
        jmp         leave_walk_back
pos_resl:
        mul         word ptr [si][2]            ;multiply by fraction bits
leave_walk_back:
        add         dx, cx                      ;multiply by fraction bits
                                                ;and add in angle
write_result:
        mov         di, word ptr cs_ptr
        mov         word ptr [di], ax           ;stuff result into variable
        mov         word ptr [di][2], dx        ;setup output for qword
                                                ;fixed point
        sub         ax, ax                      ;radix point between the
                                                ;double words
        mov         word ptr [di][4], ax
        mov         word ptr [di][6], ax
        cmp         byte ptr sign, al
        je          exit
        not         word ptr [di] [6]
        not         word ptr [di][4]
        not         word ptr [di][2]
        neg         word ptr [di][0]
        jc          exit
        add         word ptr [di][2],1
        adc         word ptr [di][4],ax
        adc         word ptr [di][6],ax
exit:
        popf
        ret
dcsin endp
```

Computing With Tables

Algebra is one of a series of fascinating lectures by the physicist Richard Feynman[2]. In it, he discusses the development of algebra and its connection to geometry. He also develops the basis for a number of very interesting and useful algorithms for logarithms and powers, as well as the algebraic basis for sines and cosines using imaginary numbers.

In algebra, Feynman describes a method of calculating logarithms, and therefore powers, based on 10 successive square roots of 10. The square root of 10 ($10^{.5}$) is 3.16228, which is the same as saying $\log_{10}(3.16228) = .5$. Since $\log_{10}(a*c) = \log_{10}(a) + \log_{10}(c)$, we can approximate any power by adding the appropriate logarithms, or multiplying by the powers they represent. For example, $10^{.875} = 10^{(.5+.25+.125)} = 3.16228 * 1.77828 * 1.33352 = 7.49894207613$.

As shown in Table 6-1, that taking successive roots of any number is the same as taking that number through successively more negative powers of two.

The following algorithm is based on these ideas and was suggested by Donald Knuth[3]. The purpose of *pwrb* is to raise a given base to a power x, $0 \leq x < 1$. This is accomplished in a manner similar to division. We do this by testing the input argument against successively more negative powers of ***b***, and subtracting those that do not drive the input argument negative. Each power whose logarithm is less than the input is added to the output multiplied by that power. If a logarithm of a certain power can not be subtracted, the power is increased and the algorithm continues. The process continues until $x = 0$.

number	power of 10	power of 2
10.0	1	2^{0}
3.16228	l/2	2^{-1}
1.77828	l/4	2^{-2}
1.33352	l/8	2^{-3}
1.15478	l/16	2^{-4}
1.074607	l/32	2^{-5}
1.036633	l/64	2^{-6}
1.018152	l/128	2^{-7}
1.0090350	l/256	2^{-8}
1.0045073	l/512	2^{-9}
1.0022511	1/1024	2^{-10}

Table 6-1. Computing with Tables

In pseudocode:

Pwrb: Algorithm

1. Set the output, *y,* equal to 1, clear the exponent counter, *K.*
2. Test our argument, x, to see if it is zero.

   If so, continue at step 6.

   If not, go to step 3.
3. Use *k* to point into a table of logarithms of the chosen base to successively more negative powers of two. Test $x < \log_b(1+2^{-k})$.

   If so, continue with step 5.

   Else, go to step 4.
4. Subtract the value pointed to in the table from *x.*

   Multiply a copy of the output by the current negative power of two through a series of right shifts.

   Add the result to the output.

   Go to step 2.

5. Bump our exponent counter, *k,* by one,

   Go to step 2.

6. There is nothing left to do, the output is our result,

   Exit.

Pwrb: Listing

```
; *****
        .data

power10         qword       4d104d42h, 2d145116h, 18cf1838h, 0d1854ebh,
                            6bd7e4bh, 36bd211h, 1b9476ah,
                            0dd7ea4h, 6ef67ah, 378915h, 1bc802h, 0de4dfh,
                            6f2a7h, 37961h, 1bcb4h, 0de5bh,
                            6f2eh, 3797h, 1bcbh, 0de6h, 6f3h, 379h, 1bdh,
                            0deh, 6fh, 38h, 1ch, 0eh, 7h, 3h, 2h, 1h

        .code
;
; ******
;pwrb - power to base 2
;input argument must be be 1 <= x < 2
pwrb proc       uses bx cx dx di si, argument:qword, result:word

        local   k:byte, z:qword

        mov     di, word ptr result                 ;Y
        sub     ax, ax
        mov     cx, 2
rep     stosw                                       ;make y = 1
        inc     ax
        stosw
        dec     ax
        stosw
        mov     byte ptr k, al                      ;make k = 0

x0:
        mov     ax, word ptr argument
        mov     cx ptr argument [2]
        mov     dx ptr argument                     ;argument 0 <= x < 1

        sub     bx, bx
```

```
        cmp     ax, bx                          ;test for 0.0
        jne     not_done_yet
        cmp     cx, bx
        jne     not_doneyet
        cmp     dx, bx
        jne     not_doneyet

        jmp     pwrb_exit

not_done_yet
        sub     bx, bx
        mov     bl, byte ptr k                  ;our pointer and exponent
        cmp     bl, 20h                         ;are we done?
        ja      pwrb_exit

        shl     bx, 1
        shl     bx, 1
        shl     bx, 1                           ;point in to table of qwords

        lea     si, word ptr power2

        cmp     dx, word ptr [si] [bx] [4]      ;is this log greater than,
                                                ;equal or less than
        jb      increase                        ;x?
        ja      reduce
        cmp     cx, word ptr [si][bx][2]
        jb      increase
        ja      reduce
        cmp     ax, word ptr [si] [bx]
        jb      increase

reduce:
        sub     ax, word ptr [si] [bx]
        sbb     cx, word ptr [si] [bx] [2]
        sbb     dx, word ptr [si][bx][4]
        mov     word ptr argument, ax           ;x<-x-z
        mov     word ptr argument [2], cx
        mov     word ptr argument[4], dx

        sub     cx, cx
        mov     cl, byte ptr k

        mov     si, word ptr result
        mov     ax, word ptr [si]
```

```
        mov         bx, word ptr [si][2]
        mov         dx, word ptr [si][4]
        cmp         cl, 0                       ;is this shfit necessary?
        je          no_shiftk
shiftk:
        shr         dx, 1
        rcr         bx, 1
        rcr         ax, 1
        loop        shiftk
no_shiftk:
        add         word ptr [si], ax           ;z<-argument>>k
        adc         word ptr [si] [2], bx
        adc         word ptr [si][4], dx

        jmp         x0

increase:

        inc         byte ptr k                  ;bump the counter to the
                                                ;next level
        jmp         x0                          ;and continue
pwrb_exit:
        ret
pwrb    endp
```

There is another, similar, routine in the TRANS.ASM module dealing with logarithms.

CORDIC Algorithms

Cordic is an acronym meaning COordinate, Rotation Digital Computer[4]. It was devised as a way to derive transcendental functions for real-time airborne navigation and has since been used in Intel math coprocessors and Hewlett-Packard calculators. The CORDIC functions are a group of algorithms capable of computing high—quality approximations of the transcendental functions and require very little arithmetic power from the processor. Any functions listed in Table 6-2 can be calculated using only shifts, adds, and subtractions. These functions make very good candidates for the core of a floating-point library for processors with or without hardware multiplication and division.

input:	output:	comments:
circular functions		
x = x rectangular units	l/k(xcos(z)-ysin(z))	in general case
y = y rectangular units	l/k(ycos(z)+xsin(z))	
z = z angle	0	
x = 1	cos(a) multiplier	to compute
y = 0	0	the constant
z = 0	0	circulark
x = circulark(constant)	cos(a)	obtain sine
y = 0	sin(a)	and cosine of
z = a	0	a
inverse circular functions		
x = x rectangular units	$l/k(\div(x^2+y^2)$	in general case
y = y rectangular units	0	
z = z	angle	$z+atan^{-1}(y/x)$
hyperbolic functions		
x = x	rectangular units	l/k(xcosh(z)+ysinh(z)) in general case
y = y	rectangular units	l/k(ycosh(z)+xsinh(z))
z = z	angle	0
inverse hyperbolic functions		
x = x	rectangular units	$l/k(\div(x^2+y^2))$ in general case
y = y	rectangular units	0
z = z	angle	$z+tanh^{-1}(y/x)$

Table 6-2. CORDIC Functions

The CORDIC functions makeup a unified core that can derive many other functions, including circular and hyperbolic, as well as powers and roots (see Table 6-2). This discussion will focus on the circular functions; routines for the hyperbolic functions and inverses for both circular and hyperbolic are in the module TRANS.ASM.

Before getting into the specifics of the routine, let's take some time to understand how the CORDIC functions work. Notice that this algorithm has some things in common with the circle algorithm presented earlier in a Chapter 3. That routine used a modified rotation matrix:

$$R_{a[x,y]} = [\cos(a)-\sin(a), \sin(a)+\cos(a)]$$

and very small values for sine and cosine to draw a circle with only shifts, additions, and subtractions. A similar idea is at work here, but it goes a step farther.

See why the rotation matrix might help derive the functions listed above, look at Figure 6-4. In a Cartesian coordinate system (x,y), you can specify a point on a plane by measuring its position relative to the (x,y) axis. In the figure, point P is at x=20, y=l0. If you draw a line from the origin of the axis to that point, it will form a vector of a certain length offset from the x axis by an angle a. To move this vector about the origin by some amount, in this case $\pi/10$ radians, you can use the rotation matrix as shown in Figure 6-4. First solving for $x = x*\cos(\alpha)-y*\sin(\alpha)$, then $y = x*\sin(\alpha)+y*\cos(\alpha)$, you will develop a new set of coordinates for point P. In this way, you can move around the origin simply by supplying an angle of rotation and the current coordinates. With a few small changes, this same mechanism can deliver the sine and cosine of a desired angle and a number of other functions as well.

To make this work, x and y are needed plus a new argument, z, which will represent the angle or rotation. Next, simplify the equation by factoring out the cosine using the fundamental identity $\tan(a) = \sin(\alpha)/\cos(\alpha)$. This leaves

$$R=_a[x,y] = \cos(\alpha)[l-\tan(\alpha), \tan(a)+1]$$

Writing this out long hand, you have

$$x=\cos(\alpha)[x-x(\tan(\alpha))]$$

and

$$Y = \cos(\alpha)[y(\tan(\alpha))+y]$$

One more step can ease the computing burden even more: replacing the two multiplications, $y(\tan(\alpha))$ and $x(\tan(\alpha))$, with right shifts if a is made the sum of a series of smaller *a*'s and each $\tan(\alpha)$ is chosen to be a negative power of two. If every tan(a) is to become a negative power of two, then the small piece of the angle each represents becomes $\mathrm{atan}(2^{-i})$. This means that we will be breaking the input angle, a,

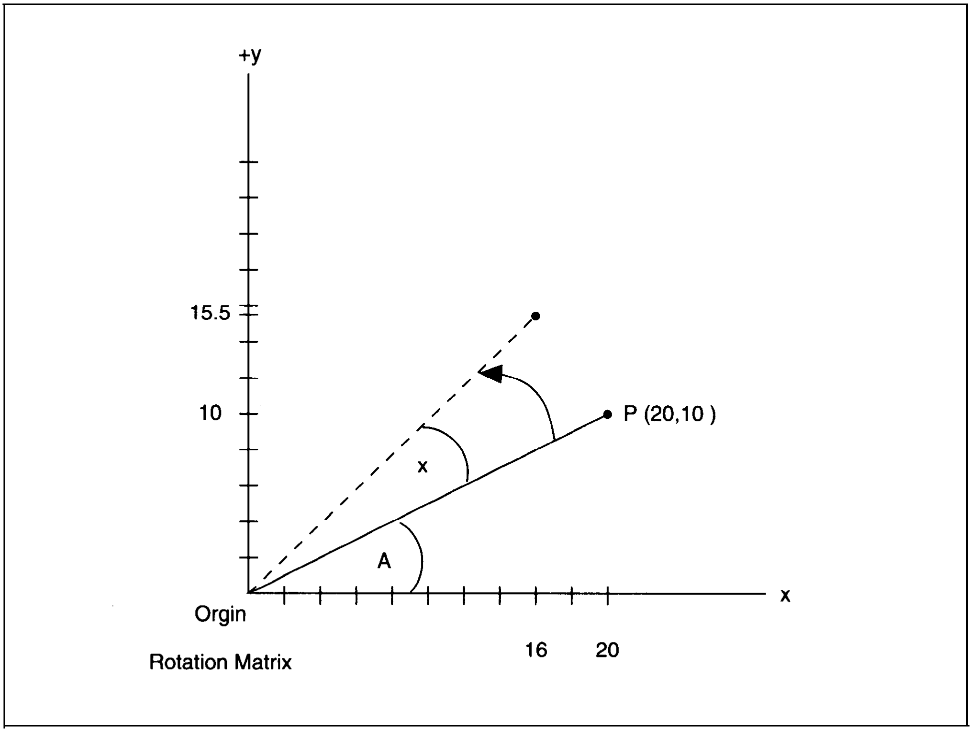


Figure 6-4. Rotation Matrix

into smaller angles equal to atan(2^{-i}) and subtracting each atan(2^{-i}) from the input a after each evaluation of the rotation matrix in an effort to close on zero. This subtraction may involve positive and negative signs depending upon the quadrant we are in as we hover around zero; as the tangent changes sign, so then must the atan. See Figure 6-3 in the previous section for the progression of signs. Now, the formulae become

$$x = \cos(\alpha)\ [x - x\ (2^{-1})]$$

and

$$y = \cos(\alpha)\ [y(2^{-1}) + y]$$

The cos(α) remains, but it is a constant *(circulark)* that has been precomputed and is factored in when needed. Because we are using negative powers of two, each iteration of the algorithm is responsible for a power of two; the result is 32 iterations for 32 fraction bits.

For the routine, *circular,* the table of arctangents was precomputed and stored in the table *atan_array*, as was the constant cos(α), *circulark.* The same was done for the hyperbolic functions with the table *atanh_array* and the constant *hyperk.* To solve for particular functions, see Table 6-2 for the correct inputs and the expected outputs.

As with so many of the functions covered in this chapter, the input argument for the angle must be confined to the first quadrant. *Circular* will solve for both sine and cosine, given X = *circulark* (the constant), Y = 0, and Z = a, if $0 \leq a < \pi/2$. Reducing the argument for these routines can be done in the same manner as in the table driven routine, *dcsin* in an earlier section. Divide the input angle by 2π, to remove unwanted components of π, then divide by $\pi/2$. Take the remainder as your input argument and the quotient as an index to the quadrant the angle is in. See Figure 6-3 for the logic.

Circular: Algorithm

1. The variables X, Y, and Z serve as both input and output variables.

   Load x, y, and z into local variables *smal lx, smal ly,* and *smal lz.*

   Set the exponent counter, i, to 0.

2. Multiply *x* and *y* by 2^{-i} and store in *smal lx* and *smal ly,* respectively. (The multiplication is accomplished by shifting (arithmetically) *x* and *y* to the right by the current count in *i.)*

   Load *z* with table entry pointed at by *atan_array+i.*

3. Test $z \leq 0$.

   If true, go to step 5.

   Else, continue with step 4.

4. Add *smal ly* to *x.*

   Subtract *smal lx* from *y.*

   Add *smal lz* to *z.*

   Continue with step 6.

5. Subtract *smal ly* from *x.*

   Add *smal lx* to *y.*

   Subtract *smal lz* from *z.*

   Continuewith step 6.

6. Bump the exponent counter, *i.*

   Test $i \geq 32$.

   If yes, got to step 7.

   Otherwise, go to step 2.

7. Since we have been using the output variables for intermediate storage of our results, the output is current and we may exit.

Circular: Listing

```
; ******

        .data
atan_array    dword          0c90fdaa2h, 76b19c16h, 3eb6ebf2h, 1fd5ba9bh,
                             0ffaaddch, 7ff556fh, 3ffeaabh, 1fffd55h,
                             0ffffabh, 800000h, 3fffffh, 200000h, 100000h,
                             80000h, 40000h, 20000h, l0000h, 8000h, 4000h,
                             2000h, 1000h, 800h, 400h, 200h, 100h, 80h,
                             40h, 20h, 10h, 8h, 4h, 2h, 1h

        .code
```

```
;
;circular-implementation of the circular routine, a subset of the CORDIC devices
;
;

circular        proc uses bx cx dx di si, x:word, y:word, z:word

        local   smallx:qword, smally:qword, smallz:qword, i:byte,
                shifter:word

        lea     di, word ptr smallx          ;load input x, y, and z
        mov     si, word ptr x
        mov     cx, 4
rep     movsw

        lea     di, word ptr smally
        mov     si, word ptr y
        mov     cx, 4
rep     movsw

        lea     di, word ptr smallz
        mov     si, word ptr z
        mov     cx, 4
rep     movsw


        sub     ax, ax
        mov     byte ptr i, al               ;i=O
        mov     bx, ax
        mov     cx, ax


twist:
        sub     ax, ax
        mov     al, i
        mov     word ptr shifter, ax

        mov     si, word ptr x               ;multiply by 2^-i
        mov     ax, word ptr [si]
        mov     bx, word ptr [si][2]
        mov     cx, word ptr [si][4]
        mov     dx, word ptr [si][6]

        cmp     word ptr shifter, 0
        je      load_smallx
```

```
shiftx:
      sar       dx, 1                       ;note the arithmetic shift
      rcr       cx, 1                       ;for sign extension
      rcr       bx, 1                       ;negative powers of two
                                            ;require right shifts
      rcr       ax, 1
      dec       word ptr shifter
      jnz       shiftx
load_smallx:
      mov       word ptr smallx, ax
      mov       word ptr smallx [2], bx
      mov       word ptr smallx [4], cx
      mov       word ptr smallx [6], dx     ;return x to smallx

      sub       ax, ax
      mov       al, i
      mov       word ptr shifter, ax
      mov       si, word ptry
      mov       ax, word ptr [si]           ;get y
      mov       bx, word ptr [si][2]
      mov       cx, word ptr [si][4]
      mov       dx, word ptr [si][6]

      cmp       word ptr shifter, 0         ;multiply by 2^-i
      je        load_smally
shifty:
      sar       dx, 1                       ;note the arithmetic shift
      rcr       cx, 1                       ;for sign extension
      rcr       bx, 1                       ;take to a negative power
      rcr       ax, 1
      dec       word ptr shifter
      jnz       shifty
load_smally:
      mov       word ptr smally, ax         ;return to smally
      mov       word ptr smally[2], bx
      mov       word ptr smally[4], cx
      mov       word ptr smally[6], dx

get_atan:
      sub       bx, bx
      mov       bl, i
      shl       bx, 1
      shl       bx, 1                       ;have to point into a dword
                                            ;table
```

```
        lea         si, word ptr atan_array
        mov         ax, word ptr [si] [bx]          ;use the negative power
                                                    ;of two as a pointer
        mov         dx, word ptr [si] [bx][2]       ;to get proper atan

        mov         word ptr smallz, ax
        mov         word ptr smallz [2], dx         ;z=atan[i]
        sub         ax, ax
        mov         word ptr smallz [4], ax
        mov         word ptr smallz [6], ax

test_Z:
        mov         si, word ptr z
        mov         ax, word ptr [si][6]            ;the sign of z determines
                                                    ;whether the arguments
        or          ax, ax                          ;are summed or subtracted
        jns         positive

negative:
        mov         ax, word ptr smally
        mov         bx, word ptr smally[2]
        mov         cx, word ptr smally[4]
        mov         dx, word ptr smally[6]

        mov         di, word ptr x
        add         word ptr [di], ax
        adc         word ptr [di][2], bx
        adc         word ptr [di][41, cx
        adc         word ptr [di][6], dx            ;smally is added x

        mov         ax, word ptr smallx
        mov         bx, word ptr smallx[2]
        mov         cx, word ptr smallx[4]
        mov         dx, word ptr smallx[6]

        mov         di, word ptr y
        sub         word ptr [di], ax
        sbb         word ptr [di][2], bx
        sbb         word ptr [di][4], cx
        sbb         word ptr [di][6], dx            ;smallx is added toy

        mov         ax, word ptr small Z
        mov         bx, word ptr small z[2]
```

```
        mov     cx, word ptr smallz [4]
        mov     dx, word ptr smallz [6]

        mov     di, word ptr z
        add     word ptr [di], ax
        adc     word ptr [di][2], bx
        adc     word ptr [di][4], cx
        adc     word ptr [di][6], dx            ;and smallz is added to z


        jmp     for_next

positive:
        mov     ax, word ptr smally             ;z was positive, so
        mov     bx, word ptr smally[2]
        mov     cx, word ptr smally[4]
        mov     dx, word ptr smally[6]
        mov     di, word ptr x
        sub     word ptr [di], ax               ;smally is subtracted
                                                ;from y
        sbb     word ptr [di][2], bx
        sbb     word ptr [di][4], cx
        sbb     word ptr [di][6], dx

        mov     ax, word ptr smallx
        mov     bx, word ptr smallx[2]
        mov     cx, word ptr smallx[4]
        mov     dx, word ptr smallx[6]          ;smallx is added to y
        mov     di, word ptry
        add     word ptr [di], ax
        adc     word ptr [di] [2], bx
        adc     word ptr [di][4], cx
        adc     word ptr [di][6], dx

        mov     ax, word ptr smallz
        mov     bx, word ptr smallz[2]
        mov     cx, word ptr smallz[4]
        mov     dx, word ptr smallz[6]
        mov     di, word ptr z                  ;and smallz is subtracted
                                                ;fromz
        sub     word ptr [di], ax
        sbb     word ptr [di][2], bx
        sbb     word ptr [di][4], cx
```

```
        sbb         word ptr [di][6], dx

for_next:
        inc         byte ptr i                       ;bump exponent on each pass
        cmp         byte ptr i, 32
        ja          circular_exit
        jmp         twist

circular_exit:


        ret
circular            endp
```

Polynomial Evaluations

One of the most popular and most accurate ways to develop the transcendentals is evaluation of a power series. These series are often expressed in the following forms:[5]

$$\sin x = x - x^3/3! + x^5/5! - x^7/7! + x^9/9! \ldots +(-l)^{n+1}x^{2n-1}/(2n-l)!$$

$$\cos x= 1 - x^2/2! + x^4/4! - x^6/6! + x^8/8! \ldots +(-l)^{n+1}x^{2n-2}/(2n-2)!$$

$$\tan x = x + x^3/3 + 2x^5/15 + 17x^7/315 + \ldots$$

$$e^x = 1 + x = x^2/2! + \ldots + x^n/n! + \ldots$$

$$\ln(1 + x) = x - x^2/2 + x^3/3 - \ldots + (-l)^{n+1}+x^n/n + \ldots$$

A power-series polynomial of infinite degree could theoretically accommodate every wrinkle in the shape of a given function within a given domain. But it isn't reasonable to attempt a calculation of a series of infinite degree; instead, some method is used to determine when to truncate the series. Usually this is at the point in the series where the terms fail to contribute significantly to the result. Your application may only require accuracy to 16 bits, such as might be needed for graphics. It may be error limited, which means that the result is calculated using enough precision and to a great enough degree to account for any spikes that might occasionally occur in the more distant terms.

Since the power series are computed in truncated form, they are prone to an error from that truncation as well as any introduced by the arithmetic. A great deal of effort has gone into finding the source of those errors and limiting it.[6] For most embedded applications (such as graphics subsystems, digital filtering and feedback control loops), the truncated Taylor Series provides adequate results.

The quadword fixed-point format used in this section has 32 fraction bits to work with. The terms contributing to bits outside that range (aside from guard digits, if you wish) are not computed even if an occasional spike might influence the rest of the computation. The 13th term of the sine expansion above rounds up to set the least significant of our fraction bits.

An alternative to the doubleword integer and doubleword fraction format could be implemented for each of the functions. At most, sine and cosine functions need a l-bit integer, leaving 63 bits for at least 18 decimal digits. On the other hand, the exponential, e^x, will quickly lose any mantissa bits unless x is less than one. You could rewrite these routines to maximize the precision of the data types you're using and provide greater accuracy; the results could be rounded and realigned for the rest of the fixed-point routines. You can do this without disturbing any de facto format you may have in place by doing the conversions and alignment within the calling function, as *taylorsin* below. Such handling is often the case anyway, since a particular series may require the arguments in a certain format to guarantee convergence. The sine and cosine functions presented here are examples of this: Their arguments should be constrained to $\pi/2$ for the series to function most efficiently and accurately.

Power-series computations are not necessarily table driven, but the execution time of the evaluation is so much faster when you precompute the coefficients that you need a good reason not to. If you wish to compute the coefficients at runtime, it's most efficient if you maintain a copy of the previous powers and factorials and compute each new one based upon that.

Homer's Rule[7] allows us to evaluate an N-degree polynomial with only N-l multiplications and N additions. To use it, we store the coefficients of the polynomial in an array. If a degree or series of degrees is missing from the polynomials, their coefficients automatically become zero. To illustrate, assume a polynomial such as

$$f(x) = 5x^4 + 3x^3 - 4x^2 + 2x + 1$$

We put the coefficients in an array:

```
Poly_array        word        1, 2, -4, 3, 5
```

In the following pseudocode, as in the example, the coefficients of the series (or polynomial) are assumed to be computed in advance, incorporating the sign of the term with the value. They're stored in a table in reverse order of the polynomial expression; that is, the first element in the array is the degree zero term. Evaluation is then simply a matter of processing the polynomial. Upon entry to the algorithm, we make the result variable equal to the coefficient of the highest power (here it's 5). We take a pointer into the array, which is the degree of the polynomial, and use it to select each succeeding coefficient to add to the result variable after multiplying it by the value of x.

taylorsin: Algorithm

```
1. Set an index to the degree of the polynomial (in this case 4).
   Use this to retrieve the coefficient of the highest power and set the
   result variable equal to that.
2. Multiply the value of x by the result variable,
   Decrement the index.
   If it goes negative, exit through step 3
   Retrieve the next coefficient and add it to the result variable,
   Continue at the beginning of step 2.
3. Horner's Method is complete. Exit.
```

In *taylorsin*, the sine approximation given above truncated at the 11th degree for our example:

$$\sin x = x - x^3/3! + x^5/5! - x^7/7! + x^9/9! - x^{11}/11!$$

To process this expansion with Homer's Rule, we need a table of coefficients with 11 terms in it and zeros for those powers not represented in the expansion indecimal:

```
1, 0, -.16666667, 0, .00833333, 0, -.00019841, 0, .00000275, 0,
                       -.00000003
```

Even this can be avoided if we evaluate the expression $x^3/3! + x^5/5! - x^7/7! + x^9/9! - x^{11}/11!$ separately with x^2 instead of x. This eliminates the necessity of processing all the zero coefficients.

With these terms stored in a table, the only thing left to do is evaluate the polynomial.

Actually, two routines are involved: *polyeval* can be made to work with any polynomial, while *taylorsin* is only an entry point. It tells the subroutine *polyeval* which table to use depending on the function to evaluate, the degree of the polynomial, and where to put the results and passes the argument. Each function requiring polynomial evaluation will require a routine such as *taylorsin;* this is where any other fixed-point manipulation-such as scaling, altering the placement of the radix point, or rounding-would be done.

taylorsin: Listing

```
;*****
;taylorsin - Derives a sin by using a infinite series. This is in radians.
;Expects argument in quadword format; expects to returnthe same.
;Input must be  ≤π./2.
taylorsin          proc uses bx cx dx di si, argument:qword, sine:word

       invoke      polyeval, argument, sine, addr polysin, 10

       ret
taylorsin  endp
```

Polyeval does the work and can be made to evaluate any polynomial, given the proper coefficients. Here is how it works:

Polyeval: Algorithm

1. Clear an accumulator and see that the output is clear.
   Set an index equal to the degree of the polynomial.
2. Using the index, point into the table of coefficients.
   Add the value pointed at to the accumulator.
3. Multiply the accumulator by the argument, *x.*
4. Decrement the table pointer.
   If it goes negative, exit through step 5.
   Otherwise, continue with step 2.
5. Write the accumulator to the output and leave.

Polyeval: Listing

```
; *****
        .data
polysin qword       100000000h, 0, 0ffffffffd5555555h, 0, 2222222h, 0,
                    0ffffffffff2ff30h, 0, 2e3ch, 0, 0ffffffffffffff94h

        .code


;
; *****
;polyeval- Evaluates polynomials according to Horner's rule.
;Expects to be passed a pointer to a table of coefficients,
;a number to evaluate, and the degree of the polynomial.
;The argument conforms to the quadword fixed-point format.

polyeval            procuses bx cx dx di si, argument:qword, output:word,
                    coeff:word, n:byte

        local       cf:qword, result[8]:word

        pushf
        cld
        sub         ax, ax
        mov         byte ptr sign, al
        mov         cx, 4
        lea         di, word ptr cf
rep     stosw                                   ;clear the accumulator
```

```
        lea       di, word ptr result
        mov       cx, 8
rep     stosw

eval:
        mov       si, word ptr coeff          ;point at table
        sub       bx, bx
        mov       bl, byte ptr n              ;point at coefficient of
                                              ;n-degree
                                              ;this is the beginning of
                                              ;our approximation
        shl       bx, 1
        shl       bx, 1
        shl       bx, 1                       ;multiply by eight for the
                                              ;quadword
        add       si, bx
        mov       ax, word ptr [si]
        mov       bx, word ptr [si] [2]
        mov       cx, word ptr [si] [4]
        mov       dx, word ptr [si] [6]
        lea       di, word ptr cf
        add       word ptr [di], ax
        adc       word ptr [di] [2], bx       ;add new coefficient to
                                              ;accumulator
        adc       word ptr [di] [4], cx
        adc       word ptr [di] [6], dx
x_by_y:
                                              ;perform a signed multiply
        invoke    smul64, argument, cf, addr result

        lea       si, word ptr result [4]
        lea       di, word ptr cf
        mov       cx, 4
rep     movsw

chk_done:
        dec       byte ptr n                  ;decrement pointer
        jns       eval

polyeval_exit:
        mov       di, word ptr output
        lea       si, word ptr cf
        mov       cx, 4
rep     movsw                                 ;write to the output
```

```
        popf
        ret
polyeval endp
```

Calculating Fixed-Point Square Roots

Finding square roots can be an art. This section presents two techniques. The first, and perhaps the most traditional, is Newton's Method. The other is the technique you learned in school adapted, to binary arithmetic. In this section, we'll examine the square-root approximation in its simplest and most elemental form. Later, in the floating-point section, we'll combine these with other techniques to improve the first estimate and speed the overall convergence of the algorithm. There is no reason those techniques couldn't also be made to fit a fixed-point application.

Newton's Method for finding square roots is a favorite among programmers because of its speed. It's given by the equation $r' = (x/r + r)/2$, with x being our radicand and r the estimate. If you are interested, cube roots may be calculated $r' = (r + (3 * x)/2)/(r * r + x/(2 * r))/2$. It is an iterative approach that eventually finds the root. There is no guarantee how many iterations it might take-that depends upon the quality of the initial guess—but it should about double the number of correct bits on each iteration.

Formulating that initial best guess is the problem. Resolving the routine can require an inordinate number of iterations if the first estimate is very far off. This routine is simple; it only knows that it has a 32-bit input and that the greatest possible root of such an input is 16 bits. To improve first estimate, therefore, the routine shifts the radicand right until it fits within a 16-bit word. Still, there is no way of telling how many iterations will be required. A loop counter with a large enough count would suffice but could easily require more iterations than would otherwise be necessary. Instead, a copy of the last estimate is saved and compared with the current estimate after each iteration. If everything proceeds smoothly, the routine exits when the estimates stop changing.

In some circumstances, however, the routine will hang, toggling between two possible roots. Another escape is provided for that contingency. A counter, *cntr*, is loaded with the maximum number of iterations. If that number is exceeded, the routine leaves with the last best estimate, which is probably close enough. An alternative would be to use another variable to define an error band and compare it with the difference between each new estimate and the last; exiting when the difference is less than the error (this-estimate -last_estimate<error).

fx_sqr: Algorithm

```
1. Establish a limiton the number of iterations possible in cntr.
   Check for negative or zero input.
   If true, exit through step 3.
   leave radicand in the register to be justified and make our first
   estimate.
2. Decrement the limit counter, cntr.
   If there is a carry, exit with current estimate and the carry set
   through step 3.
   If there is no carry, continue.
   Test the estimate to see that it fits within sixteen bits.
   If not, shift right until it does.
   Store the estimate.
   Divide the radicand by the estimate.
   Add the result to the estimate.
   Divide that by two.
   Compare last estimate with current estimate.
   If is different continue with the beginning of step 2.
   Otherwise, go to step 3.
3. Write the result to the output and leave.
```

fx_sqr: Listing

```
; *****
; accepts integers
; Remember that the powers follow the powers of two (the root of a double word
; is a word, the root of a word is a byte, the root of a byte is a nibble, etc.).
; new_estimate = (radicand/last_estimate+last_estimate)/2, last-estimate=
new-estimate.

fx_sqr proc uses bx cx dx di si, radicand:dword, root:word
```

```
        local       estimate:word,  cntr:byte

        mov         byte ptr cntr, 16
        sub         bx, bx                      ;to test radicand
        mov         ax, word ptr radicand
        mov         dx, word ptr radicand [2]
        or          dx, dx
        js          sign_exit
        je          zero_exit
        jmp         find_root                   ;not zero
zero_exit:
        or          ax, ax                      ;no negatives or zeros
        jne         find_root
sigr_exit:                                      ;indicate error in the
                                                ;operation
        stc
        sub         ax, ax
        mov         dx, ax
        jmp         root_exit

find_root:
        sub         byte ptr cntr, 1
        jc          root_exit                   ;will exit with carry
                                                ;set and an approximate
                                                ;root
find_root1:
        or          dx, dx                      ;must be zero
        je          fits                        ;some kind of estimate
        shr         dx, 1
        rcr         ax, 1
        jmp         find_root1                  ;cannot have a root
                                                ;greater than 16 bits
                                                ;for a 32-bit radicand!
fits:
        mov         word ptr estimate, ax       ;store first estimate of root
        sub         dx, dx
        mov         ax, word ptr radicand [2]
        div         word ptr estimate
        mov         bx, ax                      ;save quotient from division
                                                ;of upper word
        mov         ax, word ptr radicand
        div         word ptr estimate           ;divide lower word
        mov         dx, bx                      ;concatenate quotients
```

```
      add         ax, word ptr estimate        ;(radicand/estimate +
                                               ;estimate)/2
      adc         dx, 0
      shr         dx, 1
      rcr         ax, 1

      or          dx, dx                       ;to prevent any modular aliasing
      jne         find_root
      cmp         ax, word ptr estimate        ;is the estimate still changing?
      jne         find_root
      clc                                      ;clear the carry to indicate
                                               ;success

root_exit:
      mov         di, word ptr root
      mov         word ptr [di], ax
      mov         word ptr [di][2], dx
      ret
fx_sqr endp
```

The next approach is based on the technique taught in school for doing square roots by hand. This method turns out to be much simpler in binary than in decimal because of its modulus of 2.[8] It may not be faster than Newton's Method, but it's a good alternative for those processors without hardware division instructions.

school_sqr: Algorithm

```
1. Determine that the radicand is positive and not zero.
   If so, continue with step 2.
   If not, signal the error and exit through step 5.
2. Set bit counter for 16.
   Set buffer to hold radicand and allow for shifts.
   Clear space for root.
3. Shift buffer left twice.
   Shift root left once.
   Subtract 2*root+l from root.
   If there is an underflow, restore the subtraction by means of
   addition and continue with step 4.
   Otherwise, increment the root and continue with step 4.
```

4. Decrement bit counter.

   If zero, exit through step 5.

   Otherwise, continue with step 3.

5. Write root to output and leave.

school_sqr: Listing

```
; ******
;school_sqr
;accepts  integers
school_sqr          proc uses bx cx dx di si, radicand:dword, root:word

        local       estimate:qword, bits:byte

        sub         bx, bx
        mov         ax, word ptr radicand
        mov         dx, word ptr radicand [2]
        or          dx, dx
        js          sign-exit
        je          zero-exit
        jmp         setup                        ;notzero
zero_exit:
        or          ax, ax                       ;no negatives or zeros
        jne         setup
sign_exit:                                       ;indicate error in the
                                                 ;operation; can't do
        sub         ax, ax                       ;negatives
                                                 ;zero for fail
        stc
        jmp         root_exit

setup:
        mov         byte ptr bits, 16
        mov         word ptr estimate, ax
        mov         word ptr estimate [2], dx
        sub         ax, ax
        mov         word ptr estimate [4], ax
        mov         word ptr estimate [6], ax
        mov         bx, ax                       ;root
        mov         cx, ax
        mov         dx, ax                       ;intermediate

findroot:
```

```
        shl         word ptr estimate, 1
        rcl         word ptr estimate[2], 1
        rcl         word ptr estimate[4], 1
        rcl         word ptr estimate[6], 1

        shl         word ptr estimate, 1
        rcl         word ptr estimate[2], 1
        rcl         word ptr estimate[4], 1
        rcl         word ptr estimate[6], 1         ;double-shift  radicand

        shl         ax, 1
        rcl         bx, 1                           ;shift root

        mov         cx, ax
        mov         dx, bx
        shl         cx,  1
        rcl         dx,  1                          ;root*2
        add         cx,  1
        adc         dx,  0                          ;+l

subtract_root:
        sub         word ptr estimate[4], cx        ;accumulator-2*root+l
        sbb         word ptr estimate[6], dx
        jnc         r_plus_one
        add         word ptr estimate[4], cx
        adc         word ptr estimate[6], dx
        jmp         continue_loop
r_plus_one:
        add         ax,  1
        adc         bx,  0                          ;r+=l
continue_loop:
        dec         byte ptr bits
        jne         findroot
        clc

root-exit:
        mov         di, word ptr root
        mov         word ptr [di], ax
        mov         word ptr [di][2], bx

        ret
school_sqr endp
```

Floating-Point Approximations

All the techniques explored so far in fixed point apply to floating-point arithmetic as well. Floating-point arithmetic is similar to fixed point except that it deals with real numbers with far greater range. And because of its extensive use in scientific and engineering applications, greater emphasis is placed on its ability to approximate the real world.

This section presents some concepts that can also be used in fixed-point routines, but they're most valuable in floating point because of its attention to accuracy. Two approximations will also be described- a sine function and square root-based on materials from *Software Manual for the Elementary Functions* by William J. Cody, Jr. and William Waite, published by Prentice-Hall, Inc. This small book is full of valuable information for those writing numerical software. The sine/cosine approximation uses a minimax polynomial approximation, and the square root uses Newton's Method with a much improved initial estimate.

Floating-Point Utilities

The functions in this section use similar techniques to the fixed-point routines; that is, they use tables or arrays of coefficients and Homer's rule for evaluating polynomial approximations to the functions. The floating-point format also has some new tools and requires some new handling.

Many of the manipulations require argument reduction, which takes the floating point word apart and puts it back together again in a different fashion. Some new functions will be presented here for doing that. One is *frxp,* which, when passed a float *(x)* returns its exponent *(n)* and the float *(f)* constrained to a value between $.5 \leq f < 1$, where $f * 2^n = x$. Because it is the power to which the fixed-point mantissa must be raised to represent that number, the exponent is useful in finding the square root of a number, as you'll see in *flsqr*.

frxp: Algorithm

```
1. Point to the variable for the exponent.
2. Test the number to see if it's zero.
   If so, return zero as both the exponent and the mantissa.
```

If not, continue with step 3.

3. Discard the sign bit and subtract 126D to get the exponent, write it to *exptr,* and replace the exponent in the number with 126D.
4. Realign the float and write it to *fraction.*
5. Return.

frxp: Listing

```
; *****

;Frxp performs an operation similar to the C function frexp. Used
;for floating-point math routines.
;Returns the exponent-bias of a floating-point number.
;does not convert to floating point first, but expects a single
;precision number on the stack.
;
;
frxp  proc       uses di, float:qword, fraction:word, exptr:word

      pushf
      cld
      mov        di, word ptr exptr        ;assign pointer to exponent
      mov        ax, word ptr float[4]     ;get upper word of float
      mov        dx, word ptr float[2]
      sub        cx, cx
      or         cx, ax
      or         cx, dx
      je         make_it_zero
      shl        ax, 1                     ;the sign means zero
      sub        ah, 7eh                   ;subtract bias to place
                                           ;float .5<=x<l
      mov        byte ptr [di],ah
      mov        ah,  7eh
      shr        ax, 1                     ;replace sign
      mov        word ptr float[4], ax
      mov        di, word ptr fraction
      lea        si, word ptr float        ;write out new float
      mov        cx, 4
rep   movsw
frxp_exit:
      popf
      ret
make_it_zero
      sub        ax, ax
```

```
        mov         byte ptr [di], al
        mov         di, word ptr fraction
rep     stosw
        jmp         frxp_exit
frxp    endp
```

Ldxp performs essentially the inverse of *frxp*. This routine takes a floating-point number as an argument and replaces the exponent, that is, it raises the mantissa to a new power. It computes *input_float* * $2^{new\text{-}exponent}$ Its operation is simple:

ldxp: Algorithm

```
1. Test the input floating point argument for zero.
   If it's zero, exit with zero as the result through step 6.
2. Save the sign and replace the current exponent with 126D.
3. Add the new exponent and test for overflow.
   If there is an overflow, exit through the overflow error exit, step 7.
4. Shift the sign back into place along with the exponent.
5. Write the new float to the output and leave.
6. Zero error exit; write zero out.
7. Overflow-error exit; write infinite out.
```

ldxp: Listing

```
; *****

;Ldxp is similar to ldexp in C, it is used for math functions.
;Takes from the stack passed with it an input float (extended) and returns a
;pointer to a value to the power of two.



1dxp    proc        uses di, float:qword, power:word, exp:byte

        mov         ax, word ptr float [4]        ;get upper word of float
        mov         dx, word ptr float [2]        ;extended bits are not
                                                  ;checked
        sub         cx, cx
```

```
        or          cx, ax
        or          cx, dx
        je          return_zero
        shl         ax, 1                       ;save the sign
        rcl         cl, 1
        mov         ah, 7eh
        add         ah, byte ptr exp            ;add new exponent
        jc          ld_overflow

        shr         cl, 1                       ;return the sign
        rcr         word ptr ax, 1              ;position exponent
        mov         word ptr float[4], ax

ldxp_exit:
        mov         cx, 4
        mov         di, word ptr power          ;write the result out
        lea         si, word ptr float
rep     movsw
        ret

ld_overflow:
        mov         word ptr float[4], 7f80h
        sub         ax, ax
        mov         word ptr float[2], ax
        mov         word ptr float[0], ax
        jw          ldxp_exit

return_zero:
        sub         ax, ax
        mov         di, word ptr power
        mov         cx, 4
rep     stosw
        jmp         ldxp_exit
ldxp    endp
```

The next three functions are all related. The first, *flr,* implements the C function floor() and returns the largest floating-point mathematical integer not greater than the input.

flr: Algorithm

1. Get the float, extract the exponent, and subtract 126D.

   If there is an underflow, the number must be less than .5; exit through step 5.

2. Subtract the reduced exponent from 40D. This the mantissa portion plus extendedprecision.

   If the result is less than the reduced exponent, we already have the floor (it's all integer); exit through step 3.

   Otherwise, save the number of shifts in shift.

   Shift the float right, shifting off the fraction bits, until the exponent is exhausted. What remains are integer bits.

3. Get the exponent back from shift.

   Shift the float back into its proper position, this time without the fractionbits. This is the floor of the argument.

4. Leave, writing the result to the output.

5. Exit with a result of zero.

flr: Listing

```
; ******
;floor greatest integer less than or equal to x
;single precision


flr     proc        uses bx dx di si, fp:qword, rptr:word

        local       shift :byte

        mov         di, word ptr rptr
        mov         bx, word ptr fp[0]            ;get float with extended
                                                  ;precision
        mov         ax, word ptr fp[2]
        mov         dx, word ptr fp[4]
        mov         cx, dx
        and         cx, 7f80h                     ;get rid of sign and
                                                  ;mantissa portion
        shl         cx, 1
        mov         cl, ch
        sub         ch, ch
        sub         cl, 7eh                       ;subtract bias (-1)from
                                                  ;exponent
```

```
        jbe         leave_with_zero
        mov         ch, 40
        sub         ch, cl                      ;is it greater than the
                                                ;mantissa portion?
        jb          already_floor               ;there is no fractional
                                                ;part
        mov         byte ptr shift, ch
        mov         cl, ch
        sub         ch, ch
fix:
        shr         dx, 1                       ;shift the number the
                                                ;number of times indicated
                                                ;in the exponent
        rcr         ax, 1
        rcr         bx, 1
        loop        fix                         ;position as fixed point

        mov         cl, byte ptr shift
re_position:
        shl         bx, 1                       ;realign float
        rcl         ax, 1
        rcl         dx, 1
        loop        reposition

already-floor:
        mov         word ptr [di][4], dx        ;write to output
        mov         word ptr [di][2], ax
        mov         word ptr [di][0], bx
        sub         ax, ax
        mov         wordptr [di][6], ax

flr_exit:
        ret
leave_with_one:
        lea         si, word ptr one            ;floating-point one
        mov         di, word ptr rptr
        mov         cx, 4
rep     movsw
        jmp         flr_exit

leave_with_zero:
        sub         ax, ax                      -floating-point zero
        mov         cx, 4
```

```
      mov       di, word ptr rptr
rep   stosw
      jmp       short  flr_exit

flr   endp
```

The complement to *flr* is *flceil.* This routine is similar to the C function *ceil()* that returns the smallest floating-point mathematical integer not less than the input argument.

flceil: Algorithm

1. Get the float and check for zero.

   If the input argument is zero, exit through step 6.

   If the input is not zero, continue.

   Extract the exponent and subtract 126D. If there is anunderflow, then the number must be less than .5; exit through step 5.

2. Subtract the reduced exponent from 40D. This is the mantissa portion plus extended precision.

   If the result is less than the reduced exponent, we already have the ceiling (it's all integer); exit through step 3.

   Otherwise, save the number of shifts in *shift.*

   Shift the float right, shifting the fractionbits into the MSW of the floating-point data type until the exponent is exhausted. What remains are integer bits.

   Test the MSW of the floating-point data type.

   If it's zero, go to step 3.

   If it's anything else, round the integer portion up and continue with step 3.

3. Get the exponent back from *shift.*

   Shift the float back into its proper position, this time without the fraction bits. This is the floor of the argument.

4. Leave, writing the result to the output.

5. Exit with a result of one.

6. Exit with a result of zero.

flceil: Listing

```
; *****
;flceil least integer greater than or equal to x
;single precision
;
flceil proc      uses bx dx di si, fp:qword, rptr:word
       local     shift:byte

       mov       di, word ptr rptr
       mov       bx, word ptr fp[0]              ;get float with extended
                                                 ;precision
       mov       ax, word ptr fp[2]
       mov       dx, word ptr fp[4]
       sub       cx, cx
       or        cx, bx
       or        cx, ax
       or        cx, dx
       je        leave_with_zero                 ;this is a zero
       mov       cx, dx
       and       cx, 7f80h                       ;get rid of sign and
                                                 ;mantissa portion
       shl       cx, 1
       mov       cl, ch
       sub       ch, ch
       sub       cl, 7eh                         ;subtract bias (-1) from
                                                 ;exponent
       jb        leave-with-one
       mov       ch, 40
       sub       ch, cl                          ;is it greater than the
                                                 ;mantissa portion?
       jb        already_ceil                    ;there is no fractional
                                                 ;part
       mov       byte ptr shift, ch
       mov       cl, ch
       sub       ch, ch
fix:
       shr       dx, 1                           ;shift the number the
                                                 ;number of times indicated
                                                 ;in the exponent
       rcr       ax, 1
       rcr       bx, 1
       rcr       word ptr [di] [6], 1            ;put guard digits in MSW of
                                                 ;data type
```

```
        loop    fix                         ;position as fixed point

        cmp     word ptr [di][6],0h
        je      not_quite_enough
        add     bx, 1                       ;roundup
        adc     ax, 0
        adc     dx, 0

not_quite_enough:
        mov     cl, byte ptr shift
reposition:
        shl     bx, 1                       ;realign float
        rcl     ax, 1
        rcl     dx, 1
        loop    re_position

already_ceil:
        mov     word ptr [di][4], dx        ;write to output
        mov     word ptr [di][2], ax
        mov     word ptr [di][0], bx
        sub     ax, ax
        mov     word ptr [di][6], ax

ceil-exit:
        ret

leave-with-one:
        lea     si, word ptr one            ;a  floating-point one
        mov     di, word ptr rptr
        mov     cx, 4
rep     movsw
        jmp     ceil-exit

leave_with_zero:
        sub     ax, ax                      ;a floating-point zero
        mov     cx, 4
        mov     di, word ptr rptr
rep     stosw
        jmp     short ceil_exit

flceil  endp
```

Finally, *intrnd* rounds the input argument to its closest integer. As used by Cody and Waite,[9] this function returns an integer representing the mathematical integer closest to the input float. It employs no rounding logic; if the mantissa portion of the input float is greater than .5, the next higher whole integer is returned. In this implementation, however, it returns a floating-point number representing the mathematical integer closest to the input. It was written that way to accommodate other routines in the floating-point package.

intrnd: Algorithm

1. Subtract the value returned by *flr* from the input and take the absolute value of the result.
2. Compare the result with .5.

   If it's greater, get the *flceil* of the input.

   If it's equal to or less than, go to step 3.
3. Write the result to the output and return.

intrnd: Listing

```
; ******
;intrnd is useful for the transcendental functions
;it rounds to the nearest integer according to the logic
;intrnd(x) =if((x-floor(x)) <.5) floor(x);
;          else ceil(x);
intrnd proc       uses bx dx di si, fp:qword, rptr:word

      local temp0:qword, temp1:qword,

      pushf
      cld
      sub       ax, ax
      mov       cx, 4
      lea       di, word ptr temp0             ;prepare intermediate
                                               ;registers
rep   stosw
      mov       cx, 4
      lea       di, word ptr temp1
rep   stosw
      mov       di, word ptr rptr              ;clear the output
```

```
        mov         cx, 4
rep     stosw


        invoke      flr, fp, addr temp0
        invoke      flsub, fp, tempo, addr templ
        and         word ptr temp1[4], 7fffh          ;cheap fabs
        invoke      flcomp, temp1, one-half

        cmp         ax, 1                             ;greater than .5?
        jne         intrnd_exit

do_ceil:
        invoke      flceil, p, addr temp0             ;get the ceiling of the
                                                      ;input

intrnd_exit:
        mov         ax, word ptr temp0[2]
        mov         dx, word ptr temp0[4]
        mov         di, word ptr rptr
        mov         word ptr [di][2], ax
        mov         word ptr [di][4], dx
        popf
        ret
intmd   endp
```

Dealing with real numbers in a finite machine means we must deal with limitations. Two such limitations are *Ymax* and *Eps. Ymax* is the maximum allowable argument for the function that will produce accurate results with minimum error, and *Eps* is the smallest allowable argument. The values for these are chosen based on the size of the data types and the functions being approximated. They're important in the calculation of a number of elementary functions, notably *flsin* (discussed later).

Square Roots

The first function presented here, *flsqr,* computes the square roots of floating-point numbers. Simply, this function finds the square root of the mantissa portion of the float and then the root of $2^{exponent}$. It then reconstructs the float and returns.

To begin with, the function *frxp* is called to constrain the radicand to a small, relatively linear region, $.5 \le x < 1$ (this represents an exponent of -1). Within this region, all square roots adhere to the relationship, *nput_raidcand* < root < 1, precisely, all roots must exist from about .7071067 to 1.0. This makes it much easier to come up with an initial estimate that is very close. Just taking the mid-range value for the first estimate would improve it considerably. Recall that Newton's Method delivers about about twice the number of accurate bits for each iteration; that is, if the initial estimate is accurate to *x* bits, after the first iteration, will have about $2*x+1$ accurate bits. But even this can require an unknown number of iterations to converge, so the estimate must be improved.

The most popular solution is the formula for a straight line, y = m*x+b. Calculating the values for *m* and *b* that provide the best fit to the square-root curve yields slightly different values depending on the approach you take. Cody and Waite use the values .59016 for *m* and .41731 for *b*, which will always produce an initial estimate that's less than one percent in error. Solving for *y* in the equation for a straight line yields the first estimate, and only two passes through Newton's Method produces a result for a 24-bit mantissa.

Finding the root of $2^{exponent}$ is simple if the exponent is even: divide by 2, just as with logarithms. If the exponent is odd, however, it cannot be divided evenly by two, so it must first be incremented by one. To compensate for this adjustment, we divide the root of the input mantissa by sqrt(2). In other words, the exponent represents log, of the input number; to find its root, simply divide by two. If the exponent must be incremented before the division, the root of that additional power must be removed from the mantissa to keep the result correct.

It's then a simple matter of reassembling the float using the new mantissa and exponent.

Flsqr: Algorithm

1. Test input to see whether it is greater than zero.
   If it's equal to zero, or less, exit with error through step 7.
2. Use *frxp* to get the exponent from the input float, and to set its exponent to zero, constraining it to .5 ≤ * ≤ 1. Multiply this number, *f,* by .59016 and add .41731 for our first approximation.
3. Make two passes through r=(x/r+r)/2.

4. Inspect the exponent, *n,* derived earlier with *frxp.*

   If it's odd, multiply our best estimate from Heron's formula by the square root of .5 and increment *n* by 1.

   Even or odd, divide *n* by two.

5. Add *n* back into the exponent of the float.
6. Write the root to the output.
7. Leave.

Flsqr: Listing

```
; ******
; flsqr


flsqr proc        uses bx cx dx si di, fp0:qword, fp1:word

      local       result:qword, temp0:qword, temp1:qword, exp:byte,
                  xn:qword,  f:qword,  y0:qword,m:byte

      pushf
      cld


      lea         di, word ptr xn
      sub         ax, ax
      mov         cx, 4
rep   stosw

      invoke      flcomp, fp0, zero                ;error, entry value too
                                                   ;large
      cmp         ax, 1
      je          ok
      cmp         ax, 0
      je          got-result
      mov         di, word ptr fpl
      sub         ax, ax
      mov         cx, 4
rep   stosw
      not         ax
      and         ax, 7f80h
      mov         word ptr result[4], ax           ;make it plus infinity
      jmp         flsqr_exit
got-result:
```

```
        mov     di, word ptr fpl
        sub     ax, ax
        mov     cx, 4
rep     stosw
        jmp     flsqr_exit

ok:
        invoke  frxp, fp0, addr f, addr exp     ;get exponent


        invoke  flmul, f, y0b, addr temp0
        invoke  fladd, temp0, y0a, addr y0


heron:                                          ;two passes through
        invoke  fldiv, f, y0, addr temp0        ;(x/r+r)/2 is all we need
        invoke  fladd, y0, temp0, addr temp0
        mov     ax, word ptr temp0[4]
        shl     ax, 1
        sub     ah, 1                           ;should always be safe
        shr     ax, 1
        mov     word ptr temp0[4], ax           ;subtracts one half
                                                ;by decrementing the
                                                ;exponent one
        invoke  fldiv, f, temp0, addr temp1
        invoke  fladd, temp0, temp1, addr temp0
        mov     ax, word ptr temp0[4]
        shl     ax, 1
        sub     ah, 1                           ;should always be safe
        shr     ax, 1
        mov     word ptr y0[4], ax              ;subtracts one half
        mov     ax, word ptr temp0[2]           ;by decrementing the
                                                ;exponent one
        mov     word ptr y0[2], ax
        mov     ax, word ptr temp0
        mov     word ptr y0, ax
        sub     ax, ax
        mov     word ptr y0[6], ax

chk_n:
        mov     al, byte ptr exp
        mov     cl, al
        sar     al, 1                           ;arithmetic shift, please
        jnc     evn
```

```
odd:
      invoke     flmul, y0, sqrt_half, addr y0     ;adjustment for uneven
                                                   ;exponent

      mov        al, cl
      inc        al                                ;bump exponent on odd
      sar        al, 1                             ;divide by two
evn:
      mov        cl, al                            ;n/2->m

power:
      mov        ax, word ptr y0[4]
      shl        ax, 1
      add        ah, cl

write_result:
      shr        ax, 1
      mov        word ptr y0[4], ax
      lea        si, word ptr y0
      mov        di, word ptr fpl
      mov        cx, 4
rep   movsw

flsqr_exit:
      popf
      ret

flsqr endp
```

Sines and Cosines

The final routine implements the sine function using a minimax polynomial approximation. A minimax approximation seeks to minimize the maximum error instead of the average square of the error, which can allow isolated error spikes. The minimax method keeps the extreme errors low but can result in a higher average square error. Ultimately, what this means is that the function is resolved using a power series whose coefficients have been specially derived to keep the maximum error to a minimum value.

This routine defines the input argument as some integer times π plus a fraction equal to or less than $\pi/2$. It expects to reduce the argument to the fraction *f*, by removing any multiplies of π It then approximates the sine *(f)* based on the evaluation of a small interval symmetric about the origin, *f*, and puts the number back together as our result. It solves for the cosine by adding $\pi/2$ to the argument and proceeds as with the sine (see Figure 6-3).

In this function we again encounter *Ymax* and *Eps*. These limitations depend on the precision available to the arithmetic in the particular machine and help guarantee the accuracy of your results. According to Cody and Waite, *Ymax* should be no greater than $\pi * 2^{(t/2)}$ and *Eps*, no less than $2^{(-t/2)}$, where *t* is the number of bits available to present the significand[9]. In this example, *t* is 11 bits, but that doesn't take the extended precision into account.

The algorithm is a fairly straightforward implementation. If the input argument is in range, this function initially reduced it to *xn* initially by multiplying by $1/\pi$ (floating-point multiplication is generally faster than division) and calling *intrnd* to get the closest integer. Multiplying *xn* by π and subtracting the result from the absolute value of the input argument extracts a fraction, *f*, which is the actual angle to be evaluated with Cody and Waite's minimax approximation.

The polynomial R(g) is evaluated using a table of precomputed coefficients and Horner's rule, except that in this implementation, the usual loop (see Polyeval in the last section) was unrolled.

```
R(g) = (((r4*g+r3)*g+r2)*g+rl)*g
```

where g = f*f. The values *r*4 through *r*1 are coefficients stored in the table *sincos*.

After the evaluation, R(g) is multiplied by *f* and *f* is added to it. The only thing left to do is adjust the sign of the result according to the quadrant. The pseudocode for this implementation of *flsin* is as follows.

flsin: Algorithm

```
1. See that the input argument is no greater than Ymax.
   If it is, exit with error through step 8.
2. Take the absolute value of the input argument.
   Multiply by 1/π to remove multiple components of π
   Use intrnd to round to the closest integer, xn.
```

Test *xn* to see whether it's odd or even. If it's odd, there is a sign reversal; complement *sign.*

3. Reduce the argument to *f* through (|x|-xn*c1)-xn*c2 (in other words, subtract the rounded value *xn* multiplied by p from the input argument).
4. Compare *f* with Eps.

   If it is less, we have our result, exit through step 8.
5. Square *f,* (f*f->g) and evaluate r(g)
6. Multiply *f* by R(g), then add *f.*
7. Correct for sign; if sign is set, negate result.
8. Write the result to the output and leave.

Flsin: Listing

```
; *****

        .data
sincos  qword       404900000000h, 3a7daa20968bh, 0be2aaaa8fdbeh, 3c088739cb85h,
                    0b94fb2227f1ah, 362e9c5a91d8h
        .code


;
; ******
; flsin


flsin  proc         uses bx cx dx si di, fp0:qword, fp1:word, sign:byte

       local        result:qword, temp0:qword, temp1:qword,
                    y:qword, u:qword

       pushf
       cld


       invoke       flcomp, fp0, ymax                 ;error, entry value too
                                                      ;large
       cmp          ax, 1
       jl           absx

error-exit:
```

```
        lea       di, word ptr result
        sub       ax, ax
        mov       cx, 4
rep     stosw
        jmp       writeout

absx :
        mov       ax, word ptr fp0[4]              ;make absolute
        or        ax, ax
        jns       deconstruct_exponent
        and       ax, 7fffh
        mov       word ptr fp0[4], ax

deconstruct_exponent:
        invoke    flmul, fp0, one_over_pi, addr result
                                                   ;(x/pi)

        invoke    intrnd, result, addr temp0
                                                   ;intrnd(x/pi)

        mov       ax, word ptr temp0[2]            ;determine if integer
                                                   ;has odd or even
        mov       dx, word ptr temp0[4]            ;number of bits
        mov       cx, dx
        and       cx, 7f80h                        ;get rid of sign and
                                                   ;mantissa portion
        shl       cx, 1
        mov       cl, ch
        sub       ch, ch
        sub       cl, 7fh                          ;subtract bias (-1) from
exponent
        js        not-odd
        inc       cl
        or        cl, cl
        je        not-odd
extract_int:
        shl       ax, 1
        rcl       dx, 1
        rcl       word ptr bx, 1
        loop      extract_int                      ;position as fixed point
        test      dh, 1
        je        not_odd
        not       byte ptr sign
not_odd:
```

```
xpi:
                                                        ;extended-precision
                                                        ;multiply by pi
      invoke     flmul, sincos[8*0], temp0, addr result
                                                        ;intrnd(x/pi)*c1

      invoke     flsub, fp0, result, addr result
                                                         ;|x|-intrnd(x/pi)

      invoke     flmul, temp0, sincos[8*1], addr temp1
                                                        ;intrnd(x/pi)*c2

      invoke     flsub, result, temp1, addr y
                                                        ;Y

chk_eps:
      invoke     flabs, y, addr temp0                   ;is the argument less
                                                        ;than eps?
      invoke     flcomp, temp0, eps
      or         ax, ax
      jns        r_g
      lea        di, word ptr result
      sub        ax, ax
      mov        cx, 4
rep   stosw
      jmp        writeout

r_g

      invoke     flmul, y, y, addr u
                                                        ;evaluate r(g)
                                                        ;((r4*g+r3)*g+r2)*g+rl)*g
      invoke     flmul, u, sincos[8*5], addr result
      invoke     fladd, sincos[8*4], result, addr result

      invoke     flmul, u, result, addr result
      invoke     fladd, sincos[8*3], result, addr result

      invoke     flmul, u, result, addr result
      invoke     fladd, sincos[8*2], result, addr result

      invoke     flmul, u, result, addr result
```

```
                                                    ;result== z

fxr:
        invoke      flmul, result, y, addr result

        invoke      fladd, result, y, addr result
                                                    :r*r+f

handle_sign:
        cmp         byte ptr sign, -1
        jne         writeout
        xor         word ptr result[4], 8000h
                                                    ;result * sign
writeout:
        mov         di, word ptr fpl
        lea         si, word ptr result
        mov         cx, 4
rep     movsw

flsin_exit:

        popf
        ret
flsin endp
```

Deriving the elementary functions is both a science and an art. The techniques are given in books, but the art comes from experience with the arithmetic itself. Combining knowledge of how it behaves with science produces the best results.

1 Horden, Ira. *An FFT Algorithm For MCS-96 Products Including Supporting Routines and Examples.* Mt. Prospect, IL:Intel Corp., 1991, AP-275.

2 Feynman, Richard P. *The Feynman Lectures On Physics.* Reading, MA: Addison-Wesley Publishing Co., 1963, Volume I, Chapter 22.

3 Knuth, D. E. *Fundamental Algorithms.* Reading, MA: Addison-Wesley Publishing Co., 1973, Page 26, Exercise 28.

4 Jarvis, Pitts. *Implementing CORDIC Algorithms. Dr. Dobb's Journal,* October 1990, Pages 152-156.

5 Nielsen, Kaj L. *Modern Trigonometry.* New York, NY: Barnes & Noble, 1966, Page 169

6 Acton, Forman ***S.*** *Numerical Methods That Usually Work.* Washington D.C.: Mathematical Association of America, 1990.

Hamming, R. W. *Numerical Methods for Scientists and Engineers. New* York, NY: Dover Publications, 1973.

7 Sedgewick, Robert. *Algorithms in C.* New York, NY: Addison-Welsley Publishing Co., 1990, Page 525.

8 Crenshaw, Jack W. *Square Roots are Simple? Embedded Systems Programming,* Nov. 1991, 4/11, Pages 30-52.

9 Cody, William J. and William Waite. *Software Manual for the Elementary Functions.* Englewood, NJ: Prentice-Hall, Inc., 1980.

APPENDIX A

A Pseudo-Random Number Generator

To test the floating-point routines in this book, I needed something that would generate an unpredictable and fairly uniform series of numbers. These routines are complex enough that a forgotten carry, incorrect two's complement, or occasional overflow could easily hide from an ordinary "peek and poke" test. Even with a random number generator, it took many hours and tests with a number of data ranges to find some of the ugliest bugs.

Of course, the standard C library has a random number generator, *rand()*, but the code for it was unavailable and there were no guarantees as to how it worked. Some random number generators have such a high serial correlation (sequential dependence) that if a sequence of numbers was mapped to x/y locations on a monitor, patterns would appear. With others, users were warned that although each number generated was guaranteed to be random individually, no sequence was guaranteed to be random.

Generating random numbers isn't as easy as it might sound. Random numbers and arbitrary numbers are very different; if you asked a friend for a random number, you would really receive an arbitrarily chosen number. To be truly random, a number must have an equal chance of being chosen out of some known range and precision.

Games of dice, cards, and the lottery all depend on a sequence of random numbers, and most use a means other than computers to generate them. People don't trust machines to generate random numbers because machines can become predictable and repetitive. But with the kind of simulations and testing needed to test the floating-point routines in this book, drawing each number from a pot would take far too long. Some other method had to be devised.

One of the first techniques for generating random numbers was originated by John Von Neumann and called the *middle-square method.* [1] It consisted of taking the *seed,* or previous random number, squaring it, and taking the middle digits. Unfortunately, this method had serious disadvantages that prevented it from being widely used. It didn't take much for it to get into a rut; if a zero found its way into these middle digits, for instance, there it would stay.

A number of pseudo-random number generators are in general use, though not all of them are well tested and not all of them are good. A good random number generator is difficult to define exactly. The one quality that these generators must possess is randomness. An instance of this is given in the chi-square test, presented later. Given a uniformly distributed, pseudo-random sequence of a certain length, *n,* of numbers, all between 0 and some limit, *l,* divided among *l* bins, an equal number of numbers in each bin would be highly suspicious.

The most popular pseudo-random number generator in use, and the one chosen for this book, is the multiplicative congruential method. This technique was first proposed by D. H. Lehmer[1] in 1949. It is based on the formula

$$X_{n+1} = (aX_n + c) \bmod m$$

Each new number is produced from a number, X_n, which is either the seed or the previous number, through multiplication and modular division. It requires a multiplier, *a,* that must be equal to or greater than zero and less than the modulus, an additive or increment, c, that must also be equal to or greater than zero and less than the modulus, and a modulus, *m,* that is greater than zero. Simply supplying numbers for these variables won't result in a good random number generator; the two "bad" generators described earlier were linear congruential generators.

Here are a few guidelines, summarized from the materials of Donald Knuth:

- The seed, X_n, may be arbitrary and may, in fact, be the previously generated number in a pseudo-random sequence. *Irandom,* the pseudo-random number generator created for this book expects a double as the seed; in the demonstration routine *spectral.c,* the DOS timer tick is used.

- The modulus, *m,* should be at least 2^{30}. Very often it is the word size (or a multiple thereof) of the computer, making division and extraction of the remainder trivial. The subroutine that actually produces the random number uses a modulus of 2^{32}. This means that after the seed is multiplied by *a,* the least significant doubleword is the new random number. The result would be the same if the product of *a*X* were divided by l00000000H.
- If you intend to run the random number generator on a binary computer, the multiplier, *a,* should be chosen; *a* mod 8=5. If the target machine is decimal, then a mod 200 = 21. The multiplier and increment determine the period, or the length of the sequence before it starts again, and *potency,* or randomness, of the random number generator. The multiplier, *a,* in *irandom* (presented later in this chapter) is 69069D, which is congruent to 5 mod 8.
- The multiplier should be between .0l*m* and .99*m* and should not involve a regular pattern. The multiplier in *irand* is actually less than .0l*m,* but so was the multiplier in the original psuedo-random number generator proposed by Lehmer. In truth, it was chosen partly because of its size; the arithmetic was easier and faster. In tests described later in this appendix, this multiplier performed as well as those of two other generators.
- If you have a good multiplier, the value of c, the increment, is not important. It may be equal to one or even *a*. In *irandom,* c = 0.
- Beware of the least significant digits. They are not very random and should not be used for decisions. Avoid methods of scaling random numbers that involve modular operations, such as those found in the Microsoft C *getrandom* macro; the modular function will return the least random part of the number. Instead, treat the value returned by the random number generator as a fraction and use it to scale a user-determined maximum.

The technique chosen for the random number generator here is a combination of linear congruential and shuffling. In this sense, *shuffling* means that the random numbers are somehow moved around, or shuffled, before they're generated. This breaks up any serial correlation the sequence might have and provides a much longer, possibly infinite, period.

The pseudo-random number generator here comprises three routines. The first is an initialization, *rinit,* in which an array of 256 doublewords is filled with numbers created using the routine *congruent* and a seed value.

The actual generation is done by *irand.* First, this routine creates a new random number based on the current seed, which is nothing more than the last number generated. It then uses the lower byte of the upper word of this new random number as an index into the array of 256 numbers created at initialization. A new random number is created to replace the one selected and the routine exits, returning the number from the array. The initialization routine, *rinit,* must be called before *irandom* if the user wishes to select their own seeds; otherwise, the value 1 is chosen. Pseudocode for each of the routines is as follows

rinit: Algorithm

1. Point to the double word array in RAM. This will be the initial list of random numbers.
2. Place the input seed in the seed variable. In these routines, the timer tick is used as the seed.
3. Call the routine *congruent* 256 times to fill the array.
4. Exit.

```
; ******
;rinit - initializes random number generator based upon input seed



      .data
a       dword       69069
IMAX  equ           32767
rantop word         IMAX
ran1   dword        256 dup (0)
xsubi  dword        lh                        ;global iterative seed for
                                              ;random number generator, change
                                              ;this value to change default
```

```
init   byte          0h                            ;global variable signaling
                                                   ;whether the generator has been
                                                   ;initialized or not
       .code

rinit proc uses bx cx dx si di, seed:dword


       lea           di, word ptr ran1

       mov           ax, word ptr seed[2]
       mov           word ptr xsubi[2], ax         ;put in seed variable
       mov           ax, word ptr seed             ;get seed
       mov           word ptr xsubi, ax

       mov           cx, 256

fill_array:
       invoke        congruent
       mov           word ptr [di], ax
       mov           word ptr [di][2], dx
       add           di, 4
       loop          fill_array

rinit_exit:
       sub           ax, ax
       not           ax
       mov           byte ptr init, al
       ret

rinit  endp
```

congruent: Algorithm

1. Move the lower word of the seed, xsubi, into AX and multiply by the lower word of the multiplier, a. This will produce a result in DX:AX, with the upperword of the product in DX. (This routine performs a multiple-precisionmultiply. This is a standard polynomial multiply; it is a bit simpler and more direct because the multiplier is known.)
2. Save the lower word of this product in BX and the upper word in CX.

3. Place the upper word of the seed, *xsubi,* in AX and multiply by the lower word of the multiplier, a.
4. Add the lower word of the product of this last multiplication to the upper word of the product from the first multiplication, and propagate any carries.
5. Add to AX the lower word of *xsubi,* and to DX the upper word of *xsubi.* The multiplier used in this routine is 69069D, or 10dcdH. The multiplications performed prior to this step all involved the lower word, 0dcdH. To multiply by 10000H, you need only shift the multiplicand 16 places to the left and add it to the previous subproduct.
6. Replace DX with BX, the LSW of the multiple-precision product. The MSW is discarded because it is purely overflow from any carries that have propagated forward. Instead, the lesser words are used. They might be regarded as the fractional extension of any integer in the MSW.
7. Write BX to the LSW of the seed and AX to the MSW.
8. Return.

```
; ******
;congruent -performs simple congruential algorithm



congruent proc uses bx cx
      mov        ax, word ptr xsubi          ;a*seed (mod2^32)
      mul        word ptr a
      mov        bx, ax                      ;lower word of result
      mov        cx, dx                      ;upper word
      mov        ax, word ptr xsubi[2]
      mul        word ptr a
      add        ax, cx
      adc        dx, 0

      add        ax, word ptr xsubi          ;a multiplication by one is just
                                             ;an add, right?
      adc        dx, word ptr xsubi [2]
      mov        dx, bx
      mov        word ptr xsubi, bx
```

```
      mov        word ptr xsubi [2], ax
      ret
congruent endp
```

irandom: Algorithm

1. Point to the array of random numbers.
2. Call congruent for a new number based upon the last seed.
3. Use the lower byte of the MSW of that number as an index into that array.
4. Get a new random number.
5. Replace the previously selected number with this new number.
6. Replace the seed with the previously selected number.
7. Scale the random number with *rantop,* a variable defining the maximum random number output by the routine.

```
;
; ******
;irandom- generates random floats using the linear congruential method


irandom         proc uses bx cx dx si di

      lea        si, word ptr ran1
      mov        al, byte ptr init        ;check for initialization
      or         al, al
      jne        already_initialized
      invoke     rinit, xsubi             ;default to 1
already_initialized:
      invoke     congruent                ;get a random number
      and        ax, 0ffh                 ;every fourth byte, right?
      shl        ax, 1
      shl        ax, 1                    ;multiply by four
      add        si, ax                   ;point to number in array
      mov        di, si                   ;so we can put one there too
      invoke     congruent
      mov        bx, word ptr [si]
```

```
        mov     cx, word ptr [si][2]        ;get number from array
        mov     word ptr [di], ax
        mov     word ptr [di] [2], dx       ;replace it with another
        mov     word ptr xsubi, bx
        mov     word ptr xsubi[2], cx       ;seed for next random

        mov     ax, bx
        mul     word ptr rantop             ;scale output by rantop, the
                                            ;maximum size of the
                                            ;random number
        mov     ax, dx                      ;if rantop were made 0ffffH,
                                            ;the value could be used
                                            ;directly as a fraction
        ret
irandom         endp
```

The danger with a pseudo-random number generator is that it will look quite acceptable on paper but may fail to produce good numbers. Spectra1.c provides two ways to test irandom or any pseudo-random number generator. One is quite simple, allowing examination of the output in graphic format so that the numbers produced by *irandom* can be checked visually for any patterns or concentrations. Any serial correlations that might arise can be detected using this method, but it is no proof of *k-space*, or multidimensional, randomness.

The other test is the traditional Chi-square statistic. The output of this formula can give a probabilistic indication as to whether your random number generator is truly random. The actual formula is stated:

$$v = \sum_{1 \le s \le k} (Y_s - np_s)^2 / np_s$$

but for the purposes of this algorithm is stated:

$$v = 1/n \sum_{1 \le s \le k} (y_s^2 / p_s) - n$$

This formula merely evaluates a sequence and produces a value indicating how much the sequence diverged from a probable or expected distribution.

Say you generate 1,000 numbers, *a,* all of them less than 100, *b.* You then divide *a* among 100 bins, c, based on the value of the number; in other words, a random number of 55 would go into bin 55, and a number such as 32 would go in bin 32. You would probably expect 10 numbers in each bin. Of course, a *random* number generator will seldom have an absolutely even distribution; it wouldn't be random if it did.

In fact, this statistic is only an indicator and can vary from sampling to sampling on the same generator. Tables can be used to interpret the numbers output by this formula. A good rule of thumb is that the statistic should be close, but not too close, to the number of bins-probably within $2*\sqrt{(b)}$.[2]

This statistic can vary. While you could roll 10 sevens in a row, it simply won't happen very often. A statistic that varies widely from $2*\sqrt{(b)}$, consistently produces the same value, or is extremely close to *b* might be suspect. I tested *Irandom*—along with *rand(),* a "portable" pseudo-random number generator written in C and a third-party routine found little difference in this statistic. It always remained relatively close to *b*, only occasionally straying outside $2*\sqrt{(b)}$.

Both the visual and the Chi-square test are incorporated into a program called *spectral.c* (no relation to Knuth's spectral test; it is so called merely because of its visual aspect). The program is simple: Pairs of random numbers are scaled and used as *x* and *y* coordinates for pixels on a graphics screen; 10,000 pixels are generated this way. Serial correlations can show up as a sawtooth pattern or other concentrations in the display. Otherwise, the display should show a fairly uniform array of white dots similar to a starry night.

After painting the screen, program retrieves the seed to generate a sequence of numbers for the Chi-square statistic. The result is then displayed.

spectral: Algorithm

1. Prepare the screen, turn the cursor off, put the video in EGA graphics mode, and retrieve a structure containing the current video configuration.

2. a) Use the timer tick as a seed and generate 20,000 pseudo-random numbers, using pairs as *x/y* locations on the graphics screen to turn pixels on.
   b) Use the same seed to generate the sequence for Chi-square analysis; output the result to the screen.
   c) Print a message asking for a keystroke to continue, "q" to quit.
3. Return the screen to its previous state and exit to DOS.

```
#include<conio.h>
#include<stdio.h>
#include<graph.h>
#include<stdlib.h>
#include<time.h>


short modes[] = { _TEXTBW40,       _TBXTC40,     _TEXTBW80,
                  _TBXTC80,        _MRBS4COLOR,
_MRESNOCOLOR,
                  _HRESBW,         _TEXTMONO,    _HERCMONO,
                  _MRES16COLOR,    _HRESlGCOLOR,
_ERESNOCOLOR,
                  _ERESCOLOR,      _VRES2COLOR,
_VRES16COLOR,
                  _MRES256COLOR,   _ORESCOLOR
                };

extern int irandom (void);
extern void rinit (int);
extern uraninit (long);
extern double urand (void);

/*this routine scales a random number to a maximum without using a modular
operation*/
int get random (int max)
{
            unsigned long a, b;

              a=irandom();
              b = max*a;
              return(b/32768);
}
```

```
void main ()

        short j, ch, x, Y, row, num = sizeof(modes) /
              size of (modes[0]);
        unsigned int i, e;
        long g, c:
        double rnum;
        double result;
        int n = 20000;
        int r = 100;
        insigned int f[l000];
        unsigned int a, b, d;
        int seed;
        float chi;
        float pi=22.0/7.0;
        struct videoconfig vc;


        _displaycursor(_GCURSOROFF);            /*set up screen*/

        _setvideomode(_ERESNOCOLOR);            /*EGA mode; change this
                                                if you have something
                                                else. The table is above*/

        _getvideoconfig(&vc);                   /*get the video configuration*/

        do{

              do{
              seed=(unsigned)time(NULL);        /*use the timer tick as the
                                                seed*/
              rinit(seed);

              _clearscreen(_GCLEARSCREEN);

              for(i=0;i<l0000;i++) {            /*draw a starry
                                                night on the screen*/
                  x=getrandom(vc.numxpixels);
                  y=getrandom(vc.numypixels);
                  _setpixel(x,y);
```

```
}

      rinit(seed);                         /*calculate x-square based
                                            upon same seed as display*/
      for (a = 0; a < r; a++) f[a] = 0;
      for (a = 0; a < n; a++) {
          f[getrandom(r)]++;

      for (a = 0, c = 0; a < r; a++)
      c += f[a] * f[a];
      chi= ((float)r * (float)c/(float)n)-(float)n;
      printf("\n(irandom) chi-square statistic for this set of
          of random numbers is %f", chi);

      printf("\npress a key to continue...");

      )while((ch=getch()) != 'q');



      }while(ch != 'q');

      _displaycursor(_GCURSORON);
      _setvideomode(_DEFAULTMODE);
}
```

1 Knuth, D. E. *Seminumerical Algorithms.* Reading, MA: Addison-Wesley Publishing Co., 1981, Pages 1-178.

2 Sedgewick, Robert. *Algorithms in C.* Reading, MA: Addison-Wesley Publishing Co., 1990, Page 517.

APPENDIX B

Tables and Equates

Extended Precision Values Used in Elementary Functions

```
zero          qword 000000000000h
one_half      qword 3f0000000000h
one_over_pi   qword 3ea2f9836e4eh
two_over_si   qword 3f22f9836e4eh
half_pi       qword 3fc90fdaa221h
one_over_ln2  qword 3fb8aa3b295ch
ln2           qword 3f317217f7dlh
sqrt_half     qword 3f3504f30000h
expeps        qword 338000000001h
eps           qword 39fffff70000h
ymax          qword 45c90fdb0000h
big-x         qword 42a000000000h
littlex       qword 0c2a000000000h
Y0a           qword 3ed5a9a80000h
Y0b           qword 3f1714ba0000h
quarter       qword 3e8000000000h
```

Constants for Cordic Functions

```
circulark     qword 9b74eda7h
hyperk        qword 1351e8755h
```

Common Values Written for Quadword Fixed Point

1/p	=	0.318309886	=	517cc1b7h
p^2	=	9.869604401	=	9de9e64dfh
÷p	=	1.772453851	=	1c5bf891bh
e	=	2.718281828	=	2b7e15163h
1/e	=	0.367879441	=	5e2d58d9h

e2	=	7.389056099	=	763992e35h
p/180	=	0.017453293	=	477dla9h
÷2	=	1.414213562	=	16a09e668h
ln(p)	=	1.144729886	=	1250d048eh
÷3	=	1.732050808	=	lbb67ae86h
p	=	3.141592654	=	3243f6a89h

Negative Powers of Two in Decimal

2^{-1} = .5D
2^{-2} = .25D
2^{-3} = .125D
2^{-4} = .0625D
2^{-5} = .03125D
2^{-6} = .015625D
2^{-7} = .0078125D
2^{-8} = .00390625D
2^{-9} = .001953125D
2^{-10} = .0009765625D
2^{-l1} = .00048828125D
2^{-12} = .000244140625D

Negative Powers of Ten in 32 bit Hex Format

10^{-l} = .1999999aH
10^{-2} = .028f5c29H
10^{-3} = .00418037H
10^{-4} = .00068db9H
10^{-5} = .0000a7c5H
10^{-6} = .000010c6H
10^{-7} = .00000ladH
$l0^{-8}$ = .0000002aH
10^{-9} = .00000004H

APPENDIX C

FXMATH.ASM

```
.dosseg
.model small, c, os-dos

include math.inc

;
       .code


; ******
; add64 -Adds two fixed-point numbers. The radix point lies between
;word 2 and word 3.
;the arguments are passed on the stack along with a pointer to
;storage for the result
add64 proc uses ax dx es di, addend0:qword, addendl:qword, result:word

       mov          di,word ptr result

       mov          ax, word ptr addend0[0]         ;ax = low word, addend0
       mov          dx, word ptr addend0[2]         ;dx = highword, addend0
       add          ax, word ptr addend1[0]         ;add low word, addend1
       adc          dx, word ptr addendl[2]         ;add high word, addend1
       mov          word ptr [di], ax
       mov          word ptr [di][2], dx

       mov          ax, word ptr addend0[4]         ;ax = low word, addend0
       mov          dx, word ptr addend0[6]         ;dx = high word, addend0
       adc          ax, word ptr addend1[4]         ;add low word, addend1
       adc          dx, word ptr addend1[6]         ;add high word, addend1
       mov          word ptr [di][4], ax
       mov          word ptr [di][6], dx
    ret
add64 endp
```

```
;
;* sub64
;arguments passed on the stack; pointer returned to result
sub64 proc uses dx es di,
                sub0:qword, sub1:qword, result:word

        mov         di,word ptr result

        mov         ax, word ptr sub0 [0]          ;ax = low word, sub0
        mov         dx, word ptr sub0 [2]          ;dx = high word, sub0
        sub         ax, word ptr sub1 [0]          ;subtract low word, ;sub1
        sbb         dx, word ptr sub1 [2]          ;subtract high word, ;sub1
        mov         word ptr [di][0],ax
        mov         word ptr [di] [2],dx

        mov         ax, word ptr sub0 [4]          ;ax = low word, sub0
        mov         dx, word ptr sub0 [6]          ;dx = high word, sub0
        sbb         ax, word ptr sub1 [4]          ;subtract low word,;sub1
        sbb         dx, word ptr sub1 [6]          ;subtract high word, sub1
        mov         word ptr [di][4],ax
        mov         word ptr [di][6],dx

        mov         a,0
        jnc         no-flag
        not         ax
no-flag:

        ret                                        ;result returned as dx:ax
sub64 endp


;
;* sub128
;arguments passed on the stack; pointer returned to result
sub128 proc uses ax dx es di,
                sub0:word, sub1:word, result:word

        mov         di,word ptr sub0
        mov         si,word ptr sub1

        mov         ax, word ptr [di] [0]          ;ax = low word, [di]
        mov         dx, word ptr [di][2]           ;dx = high word, [di]
        sub         ax, word ptr [si] [0]          ;subtract low word, [si]
```

```
      sbb         dx, word ptr [si][2]           ;subtract high word, [si]
      mov         word ptr [di],ax
      mov         word ptr [di][2],dx

      mov         ax, word ptr [di][4]           ;ax = low word, [di]
      mov         dx, word ptr [di][6]           ;dx = high word, [di]
      sbb         ax, word ptr [si][4]           ;subtract low word, [si]
      sbb         dx, word ptr [si] [6]          ;subtract high word, [si]
      mov         word ptr [di][4],ax
      mov         word ptr [di][6],dx

      mov         ax, word ptr [di][8]           ;ax = low word, [di]
      mov         dx, word ptr [di][10]          ;dx = high word, [di]
      sbb         ax, word ptr [si][8]           ;subtract low word, [si]
      sbb         dx, word ptr [si] [10]         ;subtract high word, [si]
      mov         word ptr [di][8],ax
      mov         word ptr [di][10],dx

      mov         ax, word ptr [di][12]          ;ax = low word, [di]
      mov         dx, word ptr [di][14]          ;dx = high word, [di]
      sbb         ax, word ptr [si][12]          ;subtract low word, [si]
      sbb         dx, word ptr [si][14]          ;subtract high word, [si]
      mov         word ptr [di] [12],ax
      mov         word ptr [di][14],dx

      mov         si,di
      mov         di,word ptr result
      mov         cx,8
rep   movsw
      ret                                        ;result returned as dx:ax
sub128 endp


;
;*mullong - Multiplies two unsigned fixed point values. The
;arguments and a pointer to the result are passed on the stack.
mullong proc uses ax dx es di,
                  smultiplicand:dword, smultiplier:dword, result:word

      mov         di,word ptr result             ;small model pointer is
                                                 ;near

      mov         ax, word ptr smultiplicand[2]  ;multiply multiplicand
                                                 ;high word
```

```
      mul       word ptr smultiplier[2]         ;by multiplier high word
      mov       word ptr [di][4], ax
      mov       word ptr [di] [6], dx

      mov       ax, word ptr smultiplicand[2]   ;multiply multiplicand
                                                ;high word
      mul       word ptr smultiplier[0]         ;by multiplier low word
      mov       word ptr [dil[2], ax
      add       word ptr [di][4], dx
      adc       word ptr [di][6], 0             ;add any remnant carry

      mov       ax, word ptr smultiplicand[0]   ;multiply multiplicand
                                                ;low word
      mul       word ptr smultiplier[2]         ;by multiplier high word
      add       word ptr [di][2], ax
      adc       word ptr [di][4], dx
      adc       word ptr [di][6], 0             ;add any remnant carry

      mov       ax, word ptr smultiplicand[0]   ;multiply multiplicand
                                                ;low word
      mul       word ptr smultiplier[0]         ;by multiplier low word
      mov       word ptr [di][0], ax
      add       word ptr [di][2], dx
      adc       word ptr [di][4], 0             ;add any remnant carry
      adc       word ptr [di][6], 0
      ret

mullong endp


; ******
;* Mu164 - Multiplies two unsigned quadword integers. The
;* procedure allows for a product of twice the length of the multipliers,
;* thus preventing overflows.
mu164 proc uses ax dx,
      multiplicand:qword, multiplier:qword, result:word


      mov       di,word ptr result

      mov       ax, word ptr multiplicand[6]    ;multiply multiplicand
                                                ;highword
      mul       word ptr multiplier[6]          ;by multiplier high word
```

```
        mov     word ptr [di][12], ax
        mov     word ptr [di][14], dx

        mov     ax, word ptr multiplicand [6]  ;multiply multiplicand
                                               ;high word
        mov     word ptr multiplier[4]         ;by multiplier low word
        mov     word ptr [di] [10], ax
        add     word ptr [di][12], dx
        adc     word ptr [di][14], 0           ;add any remnant carry

        mov     ax, word ptr multiplicand[6]   ;multiply multiplicand
                                               ;low word
        mov     word ptr multiplier[2]         ;by multiplier high word
        mov     word ptr [di][8], ax
        add     word ptr [di][10], dx
        adc     word ptr [di][12], 0           ;add any remnant carry
        adc     word ptr [di] [14], 0          ;add any remnant carry

        mov     ax, word ptr multiplicand[6]   ;multiply multiplicand
                                               ;low word
        mov     word ptr multiplier[0]         ;by multiplier high word
        mov     word ptr [di][6], ax
        add     word ptr [di][8], dx
        jnc     @f
        adc     word ptr [di][10], 0
        adc     word ptr [di][12], 0
        adc     word ptr [di][14], 0           ;add any remnant carry
;
@@:
        mov     ax, word ptr multiplicand[4]   ;multiply multiplicand
                                               ;low word
        mul     word ptr multiplier[6]         ;by multiplier low word
        add     word ptr [di][10], ax
        adc     word ptr [di][12], dx
        adc     word ptr [di][14], 0

        mov     ax, word ptr multiplicand[4]   ;multiply multiplicand
                                               ;high word
        mul     word ptr multiplier[4]         ;by multiplier high word
        add     word ptr [di][8], ax
        adc     word ptr [di] [10], dx
        adc     word ptr [di] [12], 0
```

```
        adc     word ptr [di] [14], 0

        mov     ax, word ptr multiplicand[4]    ;multiply multiplicand
                                                ;high word
        mul     word ptr multiplier[2]          ;by multiplier low word
        add     word ptr [di][6], ax
        adc     word ptr [di][8], dx
        jnc     @f
        adc     word ptr [di][10], 0
        adc     word ptr [di][12], 0
        adc     word ptr [di][14], 0            ;add any remnant carry

@@:
        mov     ax, word ptr multiplicand[4]    ;multiply multiplicand
                                                ;high word
        mul     word ptr multiplier[0]          ;by multiplier low word
        mov     word ptr [di][4], ax
        add     word ptr [di][6], dx
        jnc     @f
        adc     word ptr [di][8], 0
        adc     word ptr [di][10], 0
        adc     word ptr [di][12], 0
        adc     word ptr [di][14], 0            ;add any remnant carry

@@:
        mov     ax, word ptr multiplicand[2]    ;multiply multiplicand
                                                ;low word
        mul     word ptr multiplier[6]          ;by multiplier high word
        add     word ptr [di][8], ax
        adc     word ptr [di][10], dx
        adc     word ptr [di][12], 0            ;add any remnant carry
        adc     word ptr [di][14], 0            ;add any remnant carry

        mov     ax, word ptr multiplicand[2]    ;multiply multiplicand
                                                ;low word
        mul     word ptr multiplier[4]          ;by multiplier low word
        add     word ptr [di][6], ax
        adc     word ptr [di][8], dx
        jnc     @f
        adc     word ptr [di][10], 0            ;add any remnant carry
        adc     word ptr [di][12], 0            ;add any remnant carry
        adc     word ptr [di][14], 0            ;add any remnant carry
```

```
@@:
     mov     ax, word ptr multiplicand[2]     ;multiply multiplicand
                                              ;low word
     mul     word ptr multiplier[2]           ;by multiplier high word
     add     word ptr [di][4], ax
     adc     word ptr [di][6], dx
     jnc     @f
     adc     word ptr [di][8], 0              ;add any remnant carry
     adc     word ptr [di][10], 0             ;add any remnant carry
     adc     word ptr [di][12], 0             ;add any remnant carry
     adc     word ptr [di][14], 0             ;add any remnant carry

@@:
     mov     ax, word ptr multiplicand[2]     ;multiply multiplicand
                                              ;low word
     mul     word ptr multiplier[0]           ;by multiplier low word
     mov     word ptr [di][2], ax
     add     word ptr [di][4], dx
     jnc     @f
     adc     word ptr [di][6], 0              ;add any remnant carry
     adc     word ptr [di][8], 0              ;add any remnant carry
     adc     word ptr [di][10], 0             ;add any remnant carry
     adc     word ptr [di][12], 0             ;add any remnant carry
     adc     word ptr [di][14], 0             ;add any remnant carry

@@:
     mov     ax, word ptr multiplicand[0]     ;multiply multiplicand
                                              ;low word
     mul     word ptr multiplier[6]           ;by multiplier high word
     add     word ptr [di][6], ax
     adc     word ptr [di][8], dx
     jnc     @f
     adc     word ptr [di][10], 0             ;add any remnant carry
     adc     word ptr [di] [12], 0            ;add any remnant carry
     adc     word ptr [di] [14], 0            ;add any remnant carry

@@:
     mov     ax, word ptr multiplicand[0]     ;multiply multiplicand
                                              ;low word
     mul     word ptr multiplier[4]           ;by multiplier low word
     add     word ptr [di][4], ax
     adc     word ptr [di][6], dx
     jnc     @f
     adc     word ptr [di][8], 0              ;add any remnant carry
```

```
        adc       word ptr [di][10], 0        ;add any remnant carry
        adc       word ptr [di][12], 0        ;add any remnant carry
        adc       word ptr [di][l4], 0        ;add any remnant carry

@@:
        mov       ax, word ptr multiplicand[0];multiply multiplicand
                                              ;low word
        mul       word ptr multiplier[2]      ;by multiplier high word
        add       word ptr [di][2], ax
        adc       word ptr [di][4], dx
        jnc       @f
        adc       word ptr [di][6], 0         ;add any remnant carry
        adc       word ptr [di][8], 0         ;add any remnant carry
        adc       word ptr [di][10], 0        ;add any remnant carry
        adc       word ptr [di][12], 0        ;add any remnant carry
        adc       word ptr [di][14], 0        ;add any remnant carry

@@:
        mov       ax, word ptr multiplicand[0];multiply multiplicand
                                              ;low word
        mul       word ptr multiplier[0]      ;by multiplier low word
        mov       word ptr [di][0], ax
        add       word ptr [di][2], dx
        jnc       @f
        adc       word ptr [di][4], 0         ;add any remnant carry
        adc       word ptr [di][6], 0         ;add any remnant carry
        adc       word ptr [di][8], 0         ;add any remnant carry
        adc       word ptr [di][10], 0        ;add any remnant carry
        adc       word ptr [di][12], 0        ;add any remnant carry
        adc       word ptr [di][14], 0        ;add any remnant carry

@@:
        ret
mu164 endp

;******
; classic multiply


cmul proc uses bx cx dx si di, multiplicand:dword, multiplier:dword,
product:word

        local     numbits:byte,mltpcnd:qword
```

```
        pushf
        cld
        sub         ax, ax
        lea         si, word ptr multiplicand
        lea         di, word ptr mltpcnd
        mov         cx, 2
rep     movsw
        stosw
        stosw                                             ;clear upper words
        mov         bx, ax
        mov         cx, ax
        mov         dx, ax
        mov         byte ptr numbits, 32

test_multiplier:
        shr         word ptr multiplier[2], 1
        rcr         word ptr multiplier,1

        jnc         decrement_counter

        add         ax, word ptr mltpcnd
        adc         bx, word ptr mltpcnd[2]
        adc         cx, word ptr mltpcnd[4]
        adc         dx, word ptr mltpcnd[6]

decrement_counter:
        shl         word ptr mltpcnd, 1
        rcl         word ptr mltpcnd[2], 1
        rcl         word ptr mltpcnd[4], 1
        rcl         word ptr mltpcnd[6],1

        dec         byte ptr numbits
        jnz         test_multiplier

exit:
        mov         di, word ptr product
        mov         word ptr [di], ax
        mov         word ptr [di][2], bx
        mov         word ptr [di][4], cx
        mov         word ptr [di][6], dx
        popf
        ret
cmul    endp
```

```
; ******
; classic multiply (slightly faster)
; one quad word by another, passed on the stack, pointers returned
; to the results.
: composed of shift and add instructions
fast_cmul       proc        uses bx cx dx si di, multiplicand:qword,
multiplier:qword,  product:word

        local       numbits:byte

        pushf
        cld
        sub         ax, ax
        mov         di, word ptr product
        lea         si, word ptr multiplicand
        mov         cx, 4
rep     movsw                                       ;clear the product

        sub         di, 8                           ;point to base of product
        lea         si, word ptr multiplier         ;number of bits
        mov         byte ptr numbits, 40h

        sub         ax, ax
        mov         bx, ax
        mov         cx, ax
        mov         dx. ax

test_for_zero:
        test        word ptr [di], 1
        jne         add-multiplier
        jmp         short shift

add_multiplier:
        add         ax, word ptr [si]
        adc         bx, word ptr [si][2]
        adc         cx, word ptr [si][4]
        adc         dx, word ptr [si][6]

shift:
        shr         dx, 1
        rcr         cx, 1
        rcr         bx, 1
        rcr         ax, 1
```

```
        rcr         word ptr [di][6], 1
        rcr         word ptr [di][4], 1
        rcr         word ptr [di][2], 1
        rcr         word ptr [di] [0], 1

        dec         byte ptr numbits
        jz          exit
        jmp         short test_for_zero

exit:
        mov         word ptr [di][8], ax
        mov         word ptr [di][10], bx
        mov         word ptr [di][12], cx
        mov         word ptr [di][14], dx

        popf
        ret
fast_cmul  endp

; ******
; booth
; unsigned multiplication technique based upon the booth method
;
;
booth proc          uses bx cx dx, multiplicand:dword, multiplier:dword,
product:word

        local       mltpcnd:qword

        pushf
        cld

        sub         ax, ax
        lea         si, word ptr multiplicand
        lea         di, word ptr mltpcnd
        mov         cx, 2
rep     movsw
        stosw
        stosw                                   ;clear upper words

        mov         bx, ax
        mov         cx, ax
        mov         dx, ax
        clc
```

```
check_carry:
      jc          carry_set
      test        word ptr multiplier, 1            ;test bit 0
      jz          shift_multiplicand

sub_multiplicand:
      sub         ax, word ptr mltpcnd
      sbb         bx, word ptr mltpcnd[2]
      sbb         cx, word ptr mltpcnd[4]
      sbb         dx, word ptr mltpcnd[6]

shift_multiplicand:
      shl         word ptr mltpcnd, 1
      rcl         word ptr mltpcnd[2], 1
      rcl         word ptr mltpcnd[4], 1
      rcl         word ptr mltpcnd[6], 1

      or          word ptr multiplier[2], 0         ;early-out mechanism
      jnz         shift_multiplier
      or          word ptr multiplier, 0
      jnz         shift_multiplier
      jmp         short exit

shift_multiplier
      shr         word ptr multiplier[2], 1
      rcr         word ptr multiplier, 1
      jmp         short check_carry

exit:
      mov         di, word ptr product
      mov         word ptr [di], ax
      mov         word ptr [di][2], bx
      mov         word ptr [di][4], cx
      mov         word ptr [di][6], dx

      popf
      ret

carry_set:
      test        word ptr multiplier, 1            ;test bit 0
      jnz         shift-multiplicand

add_multiplicand:
```

```
        add         ax, word ptr mltpcnd
        adc         bx, word ptr mltpcnd[2]
        adc         cx, word ptr mltpcnd[4]
        adc         dx, word ptr mltpcnd[6]
        jmp         short shift-multiplicand

booth endp



; ******
; bit pair encoding
; unsigned corollary to the booth method


bit_pair proc    uses bx cx dx, multiplicand:dword, multiplier:dword,
product:word

        local       mltpcnd:qword

        pushf
        cld

        sub         ax, ax
        lea         si, word ptr multiplicand
        lea         di, word ptr mltpcnd
        mov         cx, 2
rep     movsw
        stosw
        stosw                                         ;clear upper words

        mov         bx, ax
        mov         cx, ax
        mov         dx, ax
        clc

check_carry:
        jc          carry_set                         ;test bit n-1
        test        word ptr multiplier, 1            ;test bit 0
        jz          shiftorsub

        test        word ptr multiplier, 2            ;test bit 1
        jnz         sub_multiplicand
        jmp         add_multiplicand
```

```
shiftorsub:
        test        word ptr multiplier, 2            ;test bit 1
        jz          shift_multiplicand

subx2_multiplicand:                                   ;cheap-inline multiply
        sub         ax, word ptr mltpcnd
        sbb         bx, word ptr mltpcnd[2]
        sbb         cx, word ptr mltpcnd[4]
        sbb         dx, word ptr mltpcnd[6]

sub_multiplicand:
        sub         ax, word ptr mltpcnd
        sbb         bx, word ptr mltpcnd[2]
        sbb         cx, word ptr mltpcnd[4]
        sbb         dx, word ptr mltpcnd[6]

shift_multiplicand:
        shl         word ptr mltpcnd, 1
        rcl         word ptr mltpcnd[2],1
        rcl         word ptr mltpcnd[4], 1
        rcl         word ptr mltpcnd[6], 1

        shl         word ptr mltpcnd, 1
        rcl         word ptr mltpcnd[2], 1
        rcl         word ptr mltpcnd[4], 1
        rcl         word ptr mltpcnd[6], 1

        or          word ptr multiplier[2], 0
        jnz         shift_multiplier
        or          word ptr multiplier, 0
        jnz         shift_multiplier
        jmp         short exit

shift_multiplier:
        shr         word ptr multiplier[2], 1
        rcr         word ptr multiplier, 1
        shr         word ptr multiplier[2],1
        rcr         word ptr multiplier, 1
        jmp         short check_carry

exit:
        mov         di, word ptr product
        mov         wordptr [di], ax
```

```
        mov         word ptr [di][2], bx
        mov         word ptr [di][4], cx
        mov         word ptr [di][6], dx

        popf
        ret

carry_set:
        test        word ptr multiplier, 1
        jnz         addorsubx2
        jmp         short addor subxl

addx2_multiplicand:
        add         ax, word ptr mltpcnd               ;cheap in_line multiply
        adc         bx, word ptr mltpcnd[2]
        adc         cx, word ptr mltpcnd[4]
        adc         dx, word ptr mltpcnd[6]
add_multiplicand:
        add         ax, word ptr mltpcnd
        adc         bx, word ptr mltpcnd[2]
        adc         cx, word ptr mltpcnd[4]
        adc         dx, word ptr mltpcnd[6]
        jmp         short shift_multiplicand

addorsubx2:
        test        word ptr multiplier, 2             ;test bit 1
        jnz         shift_multiplicand
        jmp         short  addx2_multiplicand

addorsubx1:
        test        word ptr multiplier, 2             ;test bit 1
        jnz         sub_multiplicand
        jmp         short add_multiplicand

bit_pair endp


; ******
; classic divide
; One quadword by another, passed on the stack, pointers returned
; to the results.
; Composed of shift and sub instructions.
; Returns all zeros in remainder and quotient if attempt is made to divide
; zero. Returns all ff's inquotient and dividend in remainder if divide by
```

```
;zero is attempted.
cdiv   proc       uses bx cx dx si di, dvdnd:qword, dvsr:qword,
                  qtnt:word,  rmndr:word

       pushf
       cld
       sub        ax, ax
       mov        di, word ptr qtnt
       mov        cx, 4
rep    stosw                                   ;clear the quotient

       mov        cx, 4
       lea        si, word ptr dvdnd
       mov        di, word ptr qtnt
rep    movsw                                   ;dvdnd and qtnt share same
                                               ;memory space
       sub        di, 8
       mov        si, 64                       ;number of bits
       sub        ax, ax
       mov        bx, ax
       mov        cx, ax
       mov        dx, ax

shift:
       shl        word ptr [di], 1
       rcl        word ptr [di][2], 1
       rcl        word ptr [di][4], 1
       rcl        word ptr [di][6], 1          ;shift dividend into
                                               ;the remainder
       rcl        ax, 1
       rcl        bx, 1
       rcl        cx, 1
       rcl        dx, 1

compare:
       cmp        dx, word ptr dvsr[6]
       jb         test_for_end
       cmp        cx, word ptr dvsr[4]
       jb         test_for_end
       cmp        bx, word ptr dvsr[2]
       jb         test_for_end
       cmp        ax, word ptr dvsr[0]
       jb         test_for_end
```

```
        sub         ax, word ptr dvsr
        sbb         bx, word ptr dvsr[2]
        sbb         cx, word ptr dvsr[4]
        sbb         dx, word ptr dvsr[6]
        add         word ptr [di], 1
        adc         word ptr [di][2], 0
        adc         word ptr [di][4], 0
        adc         word ptr [di][6], 0

test_for_end:
        dec         si
        jnz         shift

        mov         di, word ptr rmndr
        mov         word ptr [di], ax
        mov         word ptr [di][2], bx
        mov         word ptr [di][4], cx
        mov         word ptr [di][6], dx

exit:
        popf
        ret
cdiv    endp


;******
;div32
;32 by 32 bit divide
;arguments are passed on the stack along with pointers to the
;quotient and remainder

div32 proc uses ax dx di si,
                    dvdnd:dword, dvsr:dword, qtnt:word, rmndr:word

        local       workspace[8]:word

        sub         ax, ax
        mov         dx, ax
        mov         cx, 2
        lea         si, word ptr dvdnd
        lea         di, word ptr workspace
rep     movsw
        mov         cx, 2
```

```
        lea         si, word ptr dvsr
        lea         di, word ptr workspace[4]
rep     movsw
        mov         di, word ptr qtnt

        cmp         word ptr dvdnd, ax
        jne         do_divide
        cmp         word ptr dvdnd[2],ax
        jne         do_divide
        jmp         zero_div                    ;zero  dividend

do_divide:
        cmp         word ptr dvsr[2],ax
        jne         shift                       ;see if it is small enough
        cmp         word ptr dvsr, ax           ;check for divide by zero
        je          div_by_zero

        mov         bx, word ptr rmndr          ;as long as dx is zero,
                                                ;there is
        mov         ax, word ptr dvdnd[2]       ;no overflow possible in
                                                ;this  division
        div         word ptr dvsr
        mov         word ptr [di][2],ax
        mov         ax, word ptr dvdnd
        div         word ptr dvsr
        mov         word ptr [di],ax
        mov         word ptr [bx],dx
        xor         ax,ax
        mov         word ptr [bx] [2],ax
        jmp         exit

shift:
        shr         word ptr dvdnd[2], 1        ;normalize both dvsr and
                                                ;dvdnd
        rcr         word ptr dvdnd[0], 1
        shr         word ptr dvsr[2], 1
        rcr         word ptr dvsr[0], 1
        cmp         word ptr dvsr[2],ax
        jne         shift

divide:
        mov         ax, word ptr dvdnd          ;since MSB of dvsr is a
                                                ;one,  there
        mov         dx, word ptr dvdnd[2]       ;is no overflow possible
```

```
here
        div         word ptr dvsr
        mov         word ptr [di] [0], ax           ;approximate  quotient

get_remainder:
        mov         bx, di                          ;quotient
        lea         di, word ptr workspace[8]       ;test first approximation
                                                    ;of quotient by multiplying
                                                    ;multiplying it by the dvsr
reconstruct:                                        ;and comparing it with the
                                                    ;dvdnd
        mov         ax, word ptr workspace[4]
        mul         word ptr [bx]                   ;low word of multiplicand
        mov         word ptr [di][0], ax            ;by low word of multiplier
        mov         word ptr [di][2], dx
        mov         ax, word ptr workspace[6]
        mul         word ptr [bx]
        add         word ptr [di][2], ax            ;high word of multiplicand
                                                    ;by

        mov         ax, word ptr workspace[0]
        mov         dx, word ptr workspace[2]
        sub         ax, word ptr [di] [0]
        sbb         dx, word ptr [di] [2]

        jnc         div_ex                          ;good or overflows
                                                    ;overflow, decrement approx
                                                    ;quotient
        mov         ax, word ptr [bx]
        mov         dx, word ptr [bx][2]
        sub         word ptr [bx], 1
        sbb         word ptr [bx][2], 0

        jmp         short reconstruct
div_ex:
        mov         di, word ptr rmndr              ;the result is a good
                                                    ;quotient and remainder
        mov         word ptr [di], ax
        mov         word ptr [di][2], dx
        clc
exit:
        ret

div_by_zero:
```

```
        not         ax
        mov         word ptr [di][0], ax
        mov         word ptr [di][2], ax
        stc
        jmp         exit
zero_div:
        mov         word ptr [di][0], ax
        mov         word ptr [di][2], ax
        stc
        jmp         exit
div32  endp


; ******
;The dividend and divisor are passed on the stack;the doubleword fixed-
;point result is returned in DX:AX. DX contains the integer portion, AX the
;fractional portion.
        .data
roundup             db          3fH, 0fH, 1H, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
        .code

newt    proc uses si di,
                    dividend:word, divisor:word

        local        shifted_bits:byte, pass_count:byte

        sub         ax,ax
        mov         byte ptr shifted_bits, al
        mov         byte ptr pass_count, 4

        mov         cx, ax
        mov         ax, word ptr divisor
normalize:
        or          ax, ax
        js          top_end
        shl         ax, 1
        inc         cl
        cmp         cl, 0fh
        jg          divide_by_zero                  ;the divisor must be
                                                    ;zero
        jmp         short normalize
```

```
top_end:
        mov        byte ptr shifted_bits, cl
        mov        es, ax                          ;store normalized
                                                   ;divisor
        mov        ax, word ptr divisor
        test       ax, 0f8h
        jnz        shift_right
        test       al, 7h
        jz         divide_by_zero
        jmp        short shift_left

shift_right:
        shr        word ptr ax, 1
        test       ax, 0f8h
        je         divisor-justified
        jmp        short shift_right

shift-left:
        test       ax, 4h
        jne        divisor_justified
        shl        word ptr ax, 1
        je         divisor_justified
        jmp        short shift_left

divisor_justified:
        mov        si, offset roundup
        sub        bx, bx
        mov        cx, ax                          ;v
        mov        ax, 32
        div        cl
        sub        ah, ah
        mov        cl, 4
        div        cl
        mov        ch, al
        sub        al, al
        div        cl
        mov        ah, ch
        mov        cx, es
        mov        bx, ax                          ;save z

pass:
        mul        ax                              ;z squared
        mov        al, ah                          ;adjust for 16-bit
                                                   ;fixed point
```

```
        mov          ah, dl

        mul          cx                              ;vkz2
        mov          ax, dx                          ;adjust again

        shl          bx, 1                           ;2*z

        sub          bx, ax                          ;2z - vkz2
        add          bl, byte ptr [si]               ;add rounding bits
        adc          bh, 0

        mov          ax, bx                          ;save z
        inc          si
        dec          byte ptr pass_count
        jnz          pass


prepare_shift:
        mov          ax, word ptr dividend
        mul          bx

        sub          cx, cx
        mov          cl, byte ptr shifted_bits
        sub          cl, 8
        jns          adjust_left

        neg          cl
adjust_right:
        shr          dx, 1
        rcr          ax, 1
        loop         adjust_right
        jmp          short exit

adjust_left:
        shl          ax, 1
        rcl          dx, 1
        loop         adjust_left



exit:
        ret
oops:
divide_by_zero:
```

```
        sub         ax, ax                          ;error of some sort
        not         ax
        jmp         short exit
newt endp

; ******

circle proc uses bx cx dx si di, x-coordinate:dword, y-coordinate:dword,
increment:word

        local       x:dword, y:dword, x_point:word, y_point:word, count

        mov         ax, word ptr x-coordinate
        mov         dx, word ptr x_coordinate[2]
        mov         word ptr x, ax
        mov         word ptr x[2], dx
        mov         ax, word ptr y-coordinate
        mov         dx, word ptr y-coordinate[2]
        mov         word ptr y, ax
        mov         word ptr y[2], dx               ;load local variables

        sub         ax, ax
        mov         x_point, ax                     ;x coordinate
        mov         y_point, ax                     ;y coordinate

        mov         ax, 4876h
        mov         dx, 6h                          ;2*pi
        mov         cx, word ptr increment          ;make this a negative
                                                    ;power of two

get_num_points:
        shl         ax, 1                           ;2*pi radians
        rcl         dx, 1
        loop        get_num_points
        mov         count, dx                       ;divide by l0000h

set_point:
        mov         ax, word ptr x
        mov         dx, word ptr x[2]
        add         ax, 8000h                       ;add .5 to round up
        jnc         store_x                         ;to integers
```

```
        adc         dx, 0h
store_x:
        mov         x_point, dx

        mov         ax, word ptr y
        mov         dx, word ptr y[2]
        add         ax, 8000h                ;add.5
        jnc         store_y
        adc         dx, 0h
store_y:
        mov         y_point, dx


                                             ;your routine for writing
                                             ;to the screen goes here
                                             ;and uses x_point and
                                             ;y_point as screen coordi
                                             ;nates

        mov         ax, word ptr y
        mov         dx, word ptr y [2]
        mov         cx, word ptr increment
update_x:
        sar         dx, 1
        rcr         ax, 1                    ;please note the arithmetic
                                             ;shifts
        loop        update_x                 ;to preserve the correct
                                             ;quadrant
        sub         word ptr x, ax           ;new x equals x - y *
                                             ;increment
        sbb         word ptr x [2], dx

        mov         ax, word ptr x
        mov         dx, word ptr x [2]
        mov         cx, word ptr increment
update_y:
        sar         dx, 1
        rcr         ax, 1
        loop        update_y
        add         word ptr y, ax           ;new y equals y + x *
                                             ;increment
        adc         word ptry [2], dx
        dec         count
        jnz         set_point
```

```
        ret
circle  endp

; ******

line    proc        uses bx cx dx si di, xstart:word, ystart:word, xend:word,
                    yend:word

        local       x:word, y:word, decision:word, x_dif:word, y_dif:word,
                    xstep_diag:word,
                    ystep_diag:word, xstep:word, ystep:word, diag_incr:word,
                    incr:word

        mov         ax, word ptr xstart
        mov         word ptr x, ax                  ;initialize local variables
        mov         ax, word ptr ystart
        mov         word ptry, ax

direction:
        mov         ax, word ptr xend
        sub         ax, word ptr xstart             ;total x distance
        jns         large_x                         ;which direction are we
                                                    ;drawing?
        neg         ax                              ;went  negative
        mov         word ptr xstep_diag, -1
        jmp         short store_xdif
large_x:
        mov         word ptr xstep_diag, 1
store_xdif:
        mov         x_dif, ax

        mov         ax, word ptr yend               ;y  distance
        sub         ax, word ptr ystart
        jns         large_y                         ;which  direction?
        neg         ax
        mov         word ptr ystep_diag, -1
```

```
      jmp       short store_ydif
large_y:
      mov       word ptr ystep_diag, 1
store_ydif:
      mov       word ptr y_dif, ax          ;direction is determinedby
signs

octant:
      mov       ax, word ptr x_dif          ;the axis with greater
                                            ;difference
      mov       bx, word ptr y_dif          ;becomes our reference
      cmp       ax, bx
      jg        bigger_x

      mov       y_dif, ax                   ;we have a bigger y move
                                            ;than x
      mov       x_dif, bx                   ;x won't change on
                                            ;nondiagonal steps,
      sub       ax, ax                      ;y changes every step
      mov       word ptr xstep, ax
      mov       ax, word ptr ystep_diag
      mov       word ptr ystep, ax
      jmp       setup_inc
bigger_x:
      mov       ax, word ptr xstep_diag     ;x changes every step, y
                                            ;changes only
      mov       word ptr xstep, ax          ;on diagonal steps
      sub       ax, ax
      mov       word ptr ystep, ax

setup_inc:
      mov       ax, word ptr y_dif          ;calculate decision
                                            ;variable
      shl       ax, 1
      mov       word ptr incr, ax
      sub       ax, word ptr x_dif
      mov       word ptr decision, ax
      sub       ax, word ptr x-dif
      mov       word ptr diag_incr, ax

      mov       ax, word ptr decision       ;we will do it all in the
                                            ;registers
      mov       bx, word ptr x
      mov       cx, word ptr x_dif
```

```
        mov         dx, word ptr y

line_loop:

;Put your routine for turning pixels on here. Be sure to push ax, cx, dx, and bx
;before destroying them, they are used here. The value for the x coordinate is in
;bx and the value for they coordinate is in dx.

        or          ax, ax
        jns         dpositive
                                                    ;calculate new position and
        add         bx, word ptr xstep              ;update the decision
                                                    ;variable
        add         dx, word ptr ystep
        add         ax, incr
        jmp         short chk_loop
dpositive:
        add         bx, word ptr xstep_diag
        add         dx, word ptr ystep_diag
        add         ax, word ptr diag_incr

chk_loop:
        loop        line_loop
        ret
line    endp
;


;
; ******
;
;smul64- signed mul64


smul64 proc uses bx cx dx di si, operand0:qword, operand1:qword, result:word

        local       sign:byte

        sub         ax, ax
        mov         byte ptr sign, al

        mov         ax, word ptr operand0[6]
        or          ax, ax
        jns         chk_second
        not         byte ptr sign
```

```
        not         word ptr operand0[6]
        not         word ptr operand0[4]
        not         word ptr operand0[2]
        neg         word ptr operand0[0]
        jc          chk_second
        add         word ptr operand0[2], 1
        adc         word ptr operand0[4], 0
        adc         word ptr operand0[6], 0

chk_second:
        mov         ax, word ptr operand1[6]
        or          ax, ax
        jns         multiply_already
        not         byte ptr sign
        not         word ptr operand1[6]
        not         word ptr operand1[4]
        not         word ptr operand1[2]
        neg         word ptr operand1[0]
        jc          chk_second
        add         word ptr operand1[2]
        adc         word ptr operand1[4],0
        adc         word ptr operand1[6],0

multiply_already
        invoke      mu164, operand0, operand1, result

        test        byte ptr sign, -1
        je          leave-already
        mov         di, word ptr result
        not         word ptr [di][14]
        not         word ptr [di][12]
        not         word ptr [di][10]
        not         word ptr [di][8]
        not         word ptr [di][6]
        not         word ptr [di][4]
        not         word ptr [di][2]
        neg         word ptr [di][0]
        jc          leave_already
        add         word ptr [di][2], 1
        adc         word ptr [di][4], 0
        adc         word ptr [di][6], 0
        adc         word ptr [di][8], 0
```

```
        adc         word ptr [di][10], 0
        adc         word ptr [di][12], 0
        adc         word ptr [di][14], 0

leave_already:
        ret
smul64  endp


;
; ******
;
;divmul- division by iterative multiplication
;Underflow and overflow are determined by shifting. if the dividend shifts
;out on any attempt to normalize then we have 'flowed' in which ever
;direction it shifted out.

divmul procuses bx cx dx di si, dividend:qword, divisor:qword, quotient:word

        local       temp[8]:word, dvdnd:qword, dvsr:qword, delta:qword,
                    divmsb:byte,    lp:byte,      tmp:qword

        cld                                             ;upward


        sub         cx, cx
        mov         byte ptr lp, 6                      ;should only take six
                                                        ;passes
        lea         di, word ptr dvdnd
        mov         ax, word ptr dividend[0]
        mov         dx, word ptr dividend[2]
        or          cx, ax
        or          cx, dx
        mov         word ptr [di][0], ax
        mov         word ptr [di][2], dx
        mov         ax, word ptr dividend[4]
        mov         dx, word ptr dividend[6]
        mOV         word ptr [di][4], ax
        mov         word ptr [di][6], dx
        or          cx, ax
        or          cx, dx
        je          ovrflw                              ;zero dividend

        sub         cx, cx
        lea         di, word ptr dvsr
```

```
        mov     ax, word ptr divisor[0]
        mov     dx, word ptr divisor[2]
        or      cx, ax
        or      cx, dx
        mov     word ptr [di][0], ax
        mov     word ptr [di][2], dx
        mov     ax, word ptr divisor[4]
        mov     dx, word ptr divisor[6]
        mov     word ptr [di][4], ax
        mov     word ptr [di][6], dx
        or      cx, ax
        or      cx, dx
        je      ovrflw                          ;zero divisor

        sub     ax, ax
        mov     bx, 8
find_msb:                                       ;look for MSB of divisor
        dec     bx
        dec     bx
        cmp     word ptr [di][bx], ax           ;di is pointing at dvsr
        je      find_msb

        mov     ax, word ptr [di][bx]           ;get MSW
        sub     cx, cx                          ;save shifts here
        cmp     bx, 2h                          ;see if already normalized
        jb      shift_left
        ja      shift_right
        test    word ptr. [di][bx], 8000h       ;normalized?
        jne     norm_dvsr                       ;its already there

shift_left:
        dec     cx
        shl     ax, 1
        test    ah, 80h
        jne     norm_dvsr
        jmp     shift-left                      ;count the number of shifts
                                                ;to normalize

shift_right:
        inc     cx
        shr     ax, 1
        or      ax, ax
        je      norm_dvsr
        jmp     shift-right                     ;count the number of shifts
```

```
                                                ;to normalize

norm_dvsr:
        test        word ptr [di][6], 8000h
        jne         norm_dvdnd                  ;we want to keep
        shl         word ptr [di][0], 1         ;the divisor
        rcl         word ptr [di][2], 1         ;truly normalized
        rcl         word ptr [di][4], 1         ;for maximum
        rcl         word ptr [di][6], 1         ;precision
        jmp         norm_dvsr                   ;this should normalize dvsr

norm_dvdnd:
        cmp         bl, 4h                      ;bx still contains pointer
                                                ;to dvsr
        jbe         chk_2
        add         cl, 10h                     ;adjust for word
        jmp         ready_dvdnd
chk_2:
        cmp         bl, 2h
        jae         ready_dvdnd
        sub         cl, 10h                     ;adjusting again for size
of shift

ready_dvdnd:
        lea         di, word ptr dvdnd
        or          cl, cl
        je          makedelta                   ;no adjustment necessary
        or          cl, cl
        jns         do_dvdnd_right
        neg         cl
        sub         ch, ch
        jmp         do_dvdnd_left

do_dvdnd_right:
        shr         word ptr [di][6], 1         ;no error on underflow
        rcr         word ptr [di][4], 1         ;unless it becomes zero,
                                                ;there may still be some
                                                ;usable infonnation
        rcr         word ptr [di][2], 1
        rcr         word ptr [di][0], 1
        loop        do_dvdnd_right              ;this should normalize dvsr
        sub         ax, ax
        or          ax, word ptr [di][6]
        or          ax, word ptr [di][4]
```

```
        or          ax, word ptr [di][2]
        or          ax, word ptr [di][0]
        jne         setup
        mov         di, word ptr quotient
        mov         cx, 4
rep     stosw
        jmp         divmul_exit                 ;if it is now a zero, that
                                                ;is the result

do_dvdnd_left
        shl         word ptr [di][0], 1
        rcl         word ptr [di][2], 1
        rcl         word ptr [di][4], 1
        rcl         word ptr [di][6], 1
        jc          ovrflw                      ;significant bits shifted
                                                ;out, data unusable
        loop        do_dvdnd_left               ;this should normalize dvsr

setup:
        mov         si, di
        mov         di, word ptr quotient
        mov         cx, 4
rep     movsw                                   ;put shifted dividend into
                                                ;quotient

makedelta:                                      ;this could be done with
                                                ;a table
        lea         si, word ptr dvsr
        lea         di, word ptr delta
        mov         cx, 4
rep     movsw                                   ;move normalized dvsr
                                                ;into delta

        not         word ptr delta[6]
        not         word ptr delta[4]
        not         word ptr delta[2]
        neg         word ptr delta              ;attempt to develop with
                                                ;2's comp
        jc          mloop
        add         word ptr delta[2], 1
        adc         word ptr delta[4], 0
        adc         word ptr delta[6], 0

mloop:
```

```
        invoke      mul64, delta, dvsr, addr temp

        lea         si, word ptr temp[8]
        lea         di, word ptr tmp
        mov         cx, 4
rep     movsw

        invoke      add64, tmp, dvsr, addr dvsr

        lea         di, word ptr divisor
        mov         si, word ptr quotient
        mov         cx, 4
rep     movsw
        invoke      mul64, delta, divisor, addr temp

        sub         ax, ax
        cmp         word ptr temp[6], 8000h          ;an attempt to round;
                                                     ;please bear with me
        jb          no_round                         ;.5 or above rounds up
        add         word ptr temp[8], 1
        adc         word ptr temp[10], ax
        adc         word ptr temp[12], ax
        adc         word ptr temp[14], ax
no_round:

        lea         si, word ptr temp[8]
        lea         di, word ptr tmp                 ;double duty
        mov         cx, 4
rep     movsw
        invoke      add64, divisor, tmp, quotient

        dec         byte ptr lp
        je          divmul_exit
        jmp         makedelta                        ;six passes for 64 bits

ovrflw:
        sub         ax, ax
        not         ax
        mov         cx, 4
        mov         di, word ptr quotient
rep     stosw                                        ;make infinite answer
        jmp         divmul_exit

divmul_exit:
```

```
        popf
        ret
divmul  endp


; ******
;divnewt- division by raphson-newton zero's approximation



divnewt         proc uses bx cx dx di si, dividend:qword, divisor:qword,
quotient:word

        local   temp[8]:word, proportion:qword, shift:byte, qtnt_adjust:byte,
                lp:byte, tmp:qword, unity:qword

        cld                                      ;upward


        sub     cx, cx
        mov     byte ptr lp, 3                   ;should only take three
                                                 ;passes
        mov     qtnt_adjust, cl
        or      cx, word ptr dividend[0]
        or      cx, word ptr dividend[2]
        or      cx, word ptr dividend[4]
        or      cx, word ptr dividend[6]
        je      ovrflw                           ;zero dividend

        sub     cx, cx
        or      cx, word ptr divisor[0]
        or      cx, word ptr divisor [2]
        or      cx, word ptr divisor[4]
        or      cx, word ptr divisor[6]
        je      ovrflw                           ;zero divisor

        sub     ax, ax
        mov     bx, 8
find_msb:                                        ;look for MSB of divisor
        lea     di, word ptr divisor
        dec     bx
        dec     bx
        cmp     word ptr [di][bx], ax            ;di is pointing at divisor
        je      find_msb
```

```
        mov         byte ptr qtnt_adjust, bl
        mov         ax, word ptr [di][bx]          ;get MSW
        sub         cx, cx                         ;save shifts here
        cmp         bx, 2h                         ;see if already normalized
        jb          shift_left
        ja          shift_right
        test        word ptr [di][bx], 8000h       ;normalized?
        jne         norm_dvsr                      ;it's already there

shift_left:
        dec         cx
        shl         ax, 1
        test        ah, 80h
        jne         save_shift
        jmp         shift_left                     ;count the number of shifts
                                                   ;to normalize

shift_right:
        inc         cx
        shr         ax, 1
        or          ax, ax
        je          save_shift
        jmp         shift_right                    ;count the number of shifts
                                                   ;to normalize

save_shift:
        mov         byte ptr shift, cl
        sub         ax, ax
shift_back:
        cmp         word ptr [di][6], ax           ;we will put radix point at
                                                   ;word three
        je          norm_dvsr
        shr         word ptr [di][6], 1
        rcr         word ptr [di][4], 1
        rcr         word ptr [di][2], 1
        rcr         word ptr [di][0], 1
        jmp         shift_back

norm_dvsr:
        test        word ptr [di][4], 8000h
        jne         make_first
        shl         word ptr [di][0], 1            ;the divisor
        rcl         word ptr [di][2], 1            ;truly normalized
```

```
        rcl     word ptr [di][4], 1         ;for maximum
        jmp     norm_dvsr                   ;this should normalize
                                            ;divisor

make_first:
        mov     dx, 1000h
        sub     ax, ax
        mov     bx, word ptr [di][4]        ;first approximation;
                                            ;could come from a table
        div     bx
        sub     dx, dx                      ;keep only the four
                                            ;least bits
        mov     cx, 4
correct_dvsr:
        shl     ax, 1                       ;don't want to waste time
                                            ;with a big shift when a
                                            ;little one will suffice
        rcl     dx, 1
        loop    correct_dvsr
        mov     word ptr divisor[4], ax
        mov     word ptr divisor[6], dx
        sub     cx, cx
        mov     word ptr divisor[2], cx
        mov     word ptr divisor[0], cx
        shr     dx, 1                       ;don't want to waste time
                                            ;with a big shift when a
                                            ;little one will suffice
        rcr     ax, 1
        mul     bx                          ;reconstruct for first
                                            ;attempt
        shl     ax, 1                       ;don't want to waste time
                                            ;with a big shift when a
                                            ;little one will suffice
        rcl     dx, 1
        mov     word ptr unity[4], dx
        sub     cx, cx
        mov     word ptr unity[6], cx
        mov     word ptr unity[2], cx
        mov     word ptr unity, cx

makeproportion:                             ;this could be done with
                                            ;a table

        mov     word ptr proportion[4], dx
        sub     ax, ax
```

```
        mov         word ptr proportion[6], ax
        mov         word ptr proportion[2], ax
        mov         word ptr proportion, ax

invert_proportion:
        not         word ptr proportion[6]
        not         word ptr proportion[4]
        not         word ptr proportion[2]
        neg         word ptr proportion            ;attempt to develop with
                                                   ;two's  complement
        jc          mloop
        add         word ptr proportion[2], 1
        adc         word ptr proportion[4], 0
        adc         word ptr proportion[6], 0

mloop:
        and         word ptr proportion[6], 1
        invoke      mul64, proportion, divisor, addr temp

        lea         si, word ptr temp[6]
        lea         di, word ptr divisor
        mov         cx, 4
        movsw


        invoke      mul64, proportion, unity, addr temp

        lea         si, word ptr temp[6]
        lea         di, word ptr unity
        mOV         cx, 4
        movsw
        lea         si, word ptr temp[6]
        lea         di, word ptr proportion
        mov                     cx, 4
        movsw

        dec         byte ptr lp
        je          div_newt_shift
        jmp         invert_proportion              ;six passes for 64 bits

ovrflw:
        sub         ax, ax
        not         ax
        mov         cx, 4
```

```
        mov     di, word ptr quotient
rep     stosw                                   ;make infinite answer
        jmp     divnewt_exit

divnewt_shift:
        lea     di, word ptr divisor
        mov     cl, byte ptr shift              ;get shift count
        or      cl, cl
        js      qtnt_left                       ;positive, shift left
qtnt_right:
        mov     ch, 10h
        sub     ch, cl
        mov     cl, ch
        sub     ch, ch
        jmp     qtlft

qtnt_left:
        neg     cl
        sub     ch, ch
        add     cl, 10h                         ;we want to take it to
                                                ;the msb
qtlft:
        shl     word ptr [di][0], 1
        rcl     word ptr [di][2], 1
        rcl     word ptr [di][4], 1
        rcl     word ptr [di][6], 1
        loop    qtlft

divnewt_mult:                                   ;multiply reciprocal
times dividend
        sub     ax, ax                          ;see that temp is clear
        mov     cx, 8
        lea     di, word ptr temp
rep     stosw

        invoke  mul64, dividend, divisor, addrtemp
        mov     bx, 4                           ;adjust for magnitude of
                                                ;result
        add     bl, byte ptr qtnt_adjust
        mOV     di, word ptr quotient
        lea     si, word ptr temp
        add     si, bx
```

```
        cmp     bl, 0ah
        jae     write_zero
        mov     cx, 4
rep     movsw
        jmp     divnewt_exit

write_zero:
        mov     cx, 3
rep     movsw
        sub     ax, ax
        stosw
divnewt_exit:
        popf
        ret
divnewt         endp
;
        end
```

APPENDIX D

FPMATH.ASM

```
.DOSSEG
.MODEL small, c, os_dos

include math.inc
;
       .data
;
       .code



;
; ******
;does a single-precision fabs

;
fp_intrnd proc    uses  si di, fp0:dword, fpl:word

       local      flp0:qword,  result:qword

       pushf
       cld
       xor        ax,ax
       lea        di,word ptr flp0
       mov        cx,4
rep    stosw

       lea        si,word ptr fp0
       lea        di,word ptr flp0[2]
       mov        cx,2
rep    movsw

       invoke     intrnd, flp0, addr result

       mov        ax, word ptr result[2]
```

```
        mov         dx, word ptr result[4]
        mov         di, word ptr fpl
        mov         word ptr [di], ax
        mov         word ptr [di][2], dx

        popf
        ret
fp_intrnd  endp


; ******
;intrnd is useful for the transcendental functions
; it rounds to the nearest integer according to the following logic:
; intrnd(x) = if((x-floor(x)) <.5)  floor(x);
;           else ceil(x);
intrnd     proc uses bx dx di si, fp:qword, rptr:word

        local       temp0:qword,  templ:qword,  sign:byte

        pushf
        cld
        sub         ax, ax
        mov         cx, 4
        lea         di, word ptr temp0
rep     stosw
        mov         cx, 4
        lea         di, word ptr temp1
rep     stosw
        mov         di, word ptr rptr
        mov         cx, 4
rep     stosw


        invoke      flr, fp, addr temp0
        invoke      flsub, fp, temp0, addr temp1
        and         word ptr temp1[4], 7fffh;cheap fabs
        invoke      flcomp,temp1,  one_half

        cmp         ax, 1
        jne         intrnd_exit

do_ceil:
        invoke      flceil,  fp, addr temp0
```

```
intrnd_exit:
        mov         ax, word ptr temp0[2]
        mov         dx, word ptr temp0[4]
        mov         di, word ptr rptr
        mov         word ptr [di][2], ax
        mov         word ptr [di][4], dx
        popf
        ret
intrnd     endp


;
;******
;implements floor function
;by calling flr
;
fp_floor proc      uses  si di, fp0:dword, fpl:word

        local       flp0:qword, result:qword

        pushf
        cld
        xor         ax,ax
        lea         di,word ptr flp0
        mov         cx,4
rep     stosw

        lea         si,word ptr fp0
        lea         di,word ptr flp0[2]
        mov         cx,2
rep     movsw

        invoke      flr, flP0, addr result

        mov         ax, word ptr result[2]
        mov         dx, word ptr result[4]
        mov         di, word ptr fpl
        mov         word ptr [di], ax
        mov         word ptr [di][2], dx


        popf
        ret
```

```
fp_floor  endp



; ******
;implements  ceil  function
;by calling flceil
;
fp_ceil  proc      uses   si di,  fp0:dword,  fp1:word

       local       flp0:qword,  result:qword

       pushf
       cld
       xor         ax,ax
       lea         di,word ptr flp0
       mov         cx,4
rep    stosw

       lea         si,word ptr fp0
       lea         di,word ptr flp0[2]
       mov         cx,2
rep    movsw

       invoke      flceil, flp0, addr result

       mov         ax, word ptr result[2]
       mov         dx, word ptr result[4]
       mov         di, word ptr fp1
       mov         word ptr [di], ax
       mov         word ptr [di][2], dx


       popf
       ret
fp_ceil  endp

;
; ******
;floor greatest integer less than or equal to x
;single precision


flr    proc        uses bx dx di si, fp:qword, rptr:word
```

```
        local       shift:byte

        mov         di, word ptr rptr
        mov         bx, wordptr fp[0]              ;get float with extended
                                                   ;precision
        mov         ax, word ptr fp[2]
        mov         dx, word ptr fp[4]
        mov         cx, dx
        and         cx, 7f80h                      ;get rid of sign and mantissa
                                                   ;portion
        shl         cx, 1
        mov         cl, ch
        sub         ch, ch
        sub         cl, 7eh                        ;subtract bias (-1) from
                                                   ;exponent
        jbe         leave_with_zero
        mov         ch, 40
        sub         ch, cl                         ;is it greater than the
                                                   ;mantissa  portion?
        jb          already-floor                  ;there is no fractional part
        mov         byte ptr shift, ch
        mov         cl, ch
        sub         ch, ch
fix:
        shr         dx, 1                          ;shift the number the amount
                                                   ;of times
                                                   ;indicated in the exponent
        rcr         ax, 1
        rcr         bx, 1
        loop        fix                            ;position as fixed point

        mov         cl, byte ptr shift
re_position:
        shl         bx, 1
        rcl         ax, 1
        rcl         dx, 1
        loop        re_position

already_floor:
        mov         word ptr [di][4], dx
        mov         word ptr [di][2], ax
        mov         word ptr [di][0], bx
        sub         ax, ax
```

```
        mov          word ptr [di][6], ax

fir_exit:
        ret
leave_with_one:
        lea          si, word ptr one
        mov          di, word ptr rptr
        mov          cx, 4
rep     movsw
        jmp          fir_exit

leave_with_zero:
        sub          ax, ax
        mov          cx, 4
        mov          di, word ptr rptr
rep     stosw
        jmp          short fir_exit

flr     endp

;
; ******
;
;flceil least integer greater than or equal to x
;single precision
;
;
flceil     proc uses bx dx di si, fp:qword, rptr:word
        local        shift:byte

        mov          di, word ptr rptr
        mov          bx, word ptr fp[0]              ;get float with extended
                                                     ;precision
        mOV          ax, word ptr fp[2]
        mov          dx, word ptr fp[4]
        sub          cx, cx
        or           cx, bx
        or           cx, ax
        or           cx, dx
        je           leave_with_zero;this is a zero
        mov          cx, dx
        and          cx, 7f80hq                      ;get rid of sign and mantissa
                                                     ;portion
        shl          cx, 1
        mov          cl, ch
```

```
        sub         ch, ch
        sub         cl, 7eh                         ;subtract bias (-1) from
                                                    ;exponent
        jbe         leave_with_one
        mov         ch, 40
        sub         ch, cl                          ;is it greater than the
                                                    ;mantissa  portion?
        jb          already_ceil                    ;there is no fractional part
        mov         byte ptr shift, ch
        mov         cl, ch
        sub         ch, ch
fix:
        shr         dx, 1                           ;shift the number the amount
                                                    ;of times indicated in the
                                                    ;exponent
        rcr         ax, 1
        rcr         bx, 1
        rcr         word ptr [di][6], 1             ;put guard digits in MSW of
                                                    ;data  type
        loop        fix                             ;position  as  fixedpoint

        cmp         word  ptr  [di][6],0h
        je          not_quite_enough
        add         bx, 1                           ;roundup
        adc         ax, 0
        adc         dx, 0

not_quite_enough:
        mov         cl, byte ptr  shift
re_position:
        shl         bx, 1
        rcl         ax, 1
        rcl         dx, 1
        loop        re_position

already_ceil:
        mov         word  ptr  [di][4],  dx
        mov         word  ptr  [di][2],  ax
        mov         word  ptr  [di][0],  bx
        sub         ax, ax
        mov         word  ptr  [di][6],  ax

ceil-exit:
        ret
```

```
        ret

leave_with_one:
        lea         si, word ptr one
        mov         di, word ptr rptr
        mov         cx, 4
rep     movsw
        jmp         ceil_exit

leave_with_zero:
        sub         ax, ax
        mov         cx, 4
        mov         di, word ptr rptr
rep     stosw
        jmp         short ceil_exit

; ******


round proc          uses bx dx di, fp:qword, rptr:word

        mov         ax,word ptr fp[0]
        mov         bx,word ptr fp[2]
        mov         dx,word ptr fp[4]
        cmp         ax,8000h
        jb          round_ex                        ;less than half
        jne         needs_rounding
        test        bx,l
        je          round_ex
        jmp         short  needs_rounding

        xor         bx,l                            ;round to even if odd
                                                    ;and odd if even
;       or          bx,l                            ;round down if odd and up if
                                                    ;even
        jmp         round_ex
needs_rounding:
        and         dx,7fh
        add         bx,lh
        adc         dx,0
        test        dx,80h                          ;if this is a one, there will
                                                    ;be an
        je          renorm                          ;overflow
        mov         ax, word ptr fp[4]
```

```
        and         ax,0ff80h                   ;get exponent and sign
        add         ax,80h                      ;kick it up one
        jo          over_flow
        or          dx,ax
        jmp         short round_ex
renorm:
        mov         ax,word ptr fp[4]
        and         ax,0ff80h                   ;get exponent and sign
        or          dx,ax
round_ex:
        sub         ax, ax
round_exl:
        mov         di,word ptr rptr
        mov         word ptr [di][0],ax
        mov         word ptr [di][2],bx
        mov         word ptr [di][4],dx
        sub         ax, ax
        mov         word ptr [di][6], ax
        ret

over_flow:
        xor         ax,ax
        mov         bx,ax                       ;return a quiet NAN if
                                                ;overflow
        not         ax
        mov         dx,ax
        xor         dx, 7fH
        jmp         short round_exl
round endp

;
; ******
;
;does a single-precision fabs
;

fp_abs    proc uses  si di,  fp0:dword,  fpl:word

        local       flp0:qword,  result:qword

        xor         ax,ax
        lea         di,word ptr flp0
        mov         cx,4
rep     stosw
```

```
      lea        si,word ptr fp0
      lea        di,word ptr flp0[2]
      mov        cx,2
rep   movsw

      invoke       flabs, flp0, addr result

      mov        ax, word ptr result[2]
      mov        dx, word ptr result[4]
      mov        di, word ptr fp1
      mov        word ptr [di], ax
      mov        word ptr [di][2], dx


      ret
fp_absendp


; ******
; extended-precision absolute value (fabs)
;

flabs proc uses bx cx dx si di, fp0:qword, result:word

      mov        di, word ptr result
      mov        ax, word ptr fp0
      mov        word ptr [di], ax
      mov        ax, word ptr fp0[2]
      mov        word ptr [di][2], ax
      mov        ax, word ptr fp0[4]
      and        ax, 7fffh       ;strip sign, make positive
      mov        word ptr [di] [4] , ax
      ret
flabs endp

;
; ******
;
;does a floating-point compare
;returns with answer in ax
fp_comp          proc uses  si di,
                                 fp0:dword, fpl:dword

      local       flp0:qword, flp1:qword
```

```
        xor       ax,ax
        lea       di,word ptr flp0
        mov       cx,4
  rep   stosw

        lea       di,word ptr flpl
        mov       cx,4
  rep   stosw

        lea       si,word ptr fp0
        lea       di,word ptr flp0[2]
        mov       cx,2
  rep   movsw

        lea       si,word ptr fp1
        lea       di,word ptr flp1[2]
        mov       cx,2
  rep   movsw

        invoke     flcomp, flp0, flp1

        ret
fp_camp           endp

;
;***
;
;internal routine for comparison of floating-point values

;
flcomp    proc uses  cx si di,
                          fp0:qword,  fpl:qword

        pushf
        std

        lea       si,word ptr fp0[4]
        lea       di,word ptr fp1[4]
        test      word  ptr  fp0[4],8000h        ;is the first positive.
        je        plus_l                         ;yes
        test      word  ptr  fpl[4],8000h
        je        second_gtr                     ;second not negative, there
                                                 ;fore  greater
        xchg      di,si
```

```
compare:
        mov         cx,3
repe    cmpsw
        ja          first_gtr
        jb          second_gtr
        jmp         short both-same
;
plus_l:
        test        word ptr fp1[4],8000h
        je          compare
        jmp         first_gtr
;
second_gtr:
        mov         ax,-1
        jmp         short fpcmp_ex
first_gtr:
        mov         ax,1
        jmp         short fpcmp_ex
both-same:
        sub         ax,ax
fpcmp_ex:
        popf
        ret
flcompendp

;
; ******
;
fp_sub      proc uses si di,
                                    fp0:dword, fp1:dword, rptr:word

        local       flp0:qword, flp1:qword, result:qword

        pushf
        cld
        xor         ax,ax
        lea         di,word ptr result
        mov         cx,4
rep     stosw

        lea         di,word ptr flp0
        mov         cx,4
rep     stosw
```

```
        lea         di,word ptr flp1
        mov         cx,4
rep     stosw

        lea         si,word ptr fp0
        lea         di,word ptr flp0[2]
        mov         cx,2
rep     movsw

        lea         si,word ptr fp1
        lea         di,wordptr flp1[2]
        mov         cx,2
rep     movsw

        invoke      flsub, flp0, flp1, addr result
                                                    ;pass pointer to called
                                                    ;routine
        invoke      round, result, addr result

        lea         si,word ptr result[2]
        mov         di,rptr
        mov         cx,2
        movsw
        popf
        ret
fp_sub      endp


;
; ***
;internal
;
;
flsub   proc        uses bx cx dx si di,
                              fp0:qword, fp1:qword, rptr:word

        xor         word ptr fp1[4],8000h           ;complement sign bit

        invoke      fladd, fp0, fp1, rptr           ;pass pointer to called
                                                    ;routine
        ret
flsub   endp
```

```
;
;******
;

fp_add      proc uses bx cx dx si di,
                    fp0:dword,  fpl:dword,  rptr:word

       local        flp0:qword,  flpl:qword,  result:qword

       pushf
       cld
       xor          ax,ax
       lea          di,word ptr result
       mov          cx,4
  rep  stosw

       lea          di,word ptr flp0
       mov          cx,4
  rep  stosw

       lea          di,word ptr flp1
       mov          cx,4
  rep  stosw

       lea          si,word ptr fp0
       lea          di,word ptr flp0[2]
       mov          cx,2
  rep  movsw

       lea          si,word ptr fp1
       lea          di,word ptr flp1[2]
       mov          cx,2
  rep  movsw

       invoke       fladd, flp0, flp1, addr result

       invoke       round,  result,  addr result

       lea          si,word ptr result [2]
       mov          di,rptr
       mov          cx,2
       movsw
       popf
       ret
fp_addendp
```

```
;***
;internal

fladd  proc       uses bx cx dx si di,
                  fp0:qword,  fp1:qword,  rptr:word

       local      opa:qword,  opb:qword,  signa:byte,
                  signb:byte,  exponent:byte,  sign:byte,
                  flag:byte, diff:byte, sign0:byte, sign1:byte,
                  exp0:byte,  exp1:byte

       pushf
       std
;decrement

       xor        ax,ax
;clear  appropriate  variables
       lea        di,word ptr opa[6]              ;larger  operand
       mov        cx, 4
rep    stosw      word ptr [di]
       lea        di,word ptr opb[6]              ;smaller  operand
       mov        cx,4
rep    stosw      word ptr [di]
       mov        byte ptr sign0, al
       mov        byte ptr sign1, al
       mov        byte ptr flag,al
       mov        byte ptr sign,al                ;clear  sign

chk_fp0:
       sub        bx, bx                          ;check  for  zero
       mov        ax, word ptr fp0[4]
       and        ax, 7fffh
       cmp        ax, bx
       jne        chk_fpl
       mov        ax, word ptr fp0[2}
       cmp        ax, bx
       jne        chk_fpl
       mov        ax, word ptr fp0
       cmp        ax, bx
       jne        chk_fpl
```

```
        lea         si,word ptr fp1[6]          ;return other addend
        jmp         short leave_with_other

chk_fp1:
        mov         ax, word ptr fp1[4]         ;check for zero
        and         ax, 7fffh
        cmp         ax, bx
        jne         do_add
        mov         ax, word ptr fp1[2]
        cmp         ax, bx
        jne         do_add
        mov         ax, word ptr fp1
        cmp         ax, bx
        jne         do_add
        lea         si,word ptr fp0[6]          ;return other addend

;*******************
leave_with_other:
        mov         di,word ptr rptr;one of the operands was zero
        add         di,6                        ;the other operand is the
                                                ;only
        mov         cx,4                        ;answer
rep     movsw
        jmp         fp_addex
;*******************

do_add:
        lea         si,word ptr fp0
        lea         bx,word ptr fp1

        mov         ax,word ptr [si][4]         ;fp0
        shl         ax,:                        ;dump the sign
        rcl         byte ptr sign0, 1           ;collect the sign
        mov         byte ptr exp0, ah           ;get the exponent

        mov         dx,word ptr [bx][4]         ;fPl
        shl         dx,1                        ;get sign
        rcl         byte ptr sign1, 1
        mov         byte ptr exp1, dh           ;and the exponent
        sub         ah, dh

        mov         byte ptr diff, ah           ;and now the difference

restore-missing-bit:                            ;set up operands
```

```
        and         word ptr fp0[4], 7fh
        or          word ptr fp0[4], 80h

        mov         ax, word ptr fp1
        mov         bx, word ptr fp1[2]
        mov         dx, word ptr fp1[4]
        and         dx,7fh
        or          dx,80h
        mov         word ptr fp1[4], dx

find_largest:
        cmp         byte ptr diff,0
        je          cmp_rest
        test        byte ptr diff,80h            ;test fornegative
        je          numa_bigger
        jmp         short  numb_bigger

cmp_rest:
        cmp         dx, word ptr fp0[4]
        ja          numb_bigger
        jb          numa_bigger

        cmp         bx, word ptr fp0[2]
        ja          numb_bigger
        jb          numa_bigger

        cmp         ax, word ptr fp0[0]
        jb          numa_bigger

numb_bigger:
        sub         ax, ax
          mov       al,byte ptr diff
        neg         al
        mov         byte ptr diff,al             ;save  difference
        cmp         al,40                        ;do range test
        jna         in_range

;******************                              ;this is an exit!!!!!
        lea         si, word ptr fp1[6]          ;this is a range error
leave-with-largest:                              ;operands will not line up
        mov         di, word ptr rptr            ;for a valid addition
        add         di,6                         ;leave with largest operand
        mov         cx,4                         ;that is where the signifi
```

```
                                                      ;cance
rep     movsw                                         ;is anyway
        jmp          fp_addex
range_errora:
        lea          si,word ptr fp0[6]
        jmp          short  leave_with_largest
;******************

in_range:
        mov          al,byte ptr exp1
        mov          byte ptr exponent,al             ;save exponent of largest
                                                      ;value
        mov          al, byte ptr sign0
        mov          byte ptr signb, al
        mov          al, byte ptr sign1
        mov          signa, al

        lea          si, word ptr fp1[6]              ;load opa with largest
                                                      ;operand
        lea          di, word, ptr opa [6]
        mov          cx, 4
rep     movsw

signb_positive:
        lea          si, word ptr fp0[4]              ;set to load opb
        jmp          shift_into_position


numa_bigger:
        sub          ax, ax
        mov          al,byte ptr diff
        cmp          al,40
        jae          range_errora                     ;do range test

        mov          al,byte ptr exp0
        mOV          byte ptr exponent,al             ;save exponent of largest
                                                      ;value
        mOV          al, byte ptr sign1
        mov          byte ptr signb, al
        mov          al, byte ptr sign0
        mov          byte ptr signa, al

        lea          si, word ptr fp0[6]              ;1oad opa with largest
```

```
                                                ;operand
        lea         di, word ptr opa[6]
        mov         cx,4
rep     movsw

        lea         si, word ptr fp1[4]         ;set to load opb


shift_into_position:                            ;align  operands
        xor         ax,ax
        mov         bx,4
        mov         cl,3
        mov         ah,byte ptr diff
        shr         ax,cl                       ;ah contains # of bytes, al #
                                                ;of bits
        mov         cx,5h
        shr         al,cl

        sub         bl,ah                       ;reset pointer below initial
                                                ;zeros
        lea         di,byte ptr opb
        add         di,bx

        mov         cx,bx
        inc         cx
load_operand:
        movsb
        loop        load_operand

        mov         cl,al
        xor         ch,ch
        or          cx,cx
        je          end_shift
shift_operand:
        shr         word ptr opb[6],1
        rcr         word ptr opb[4],1
        rcr         word ptr opb[2],1
        rcr         word ptr opb[0],l
        loop        shift_operand

end_shift:
        mov         al, byte ptr signa
        cmp         al, byte ptr signb
        je          just_add
```

```
                                                        ;signs alike

opb_negative:
                                                        ;signs disagree
        not         word ptr opb[6];do2's complement
        not         word ptr opb[4]
        not         word ptr opb[2]
        neg         word ptr opb[0]
        jc          just_add
        add         word ptr opb[2],1
        adc         word ptr opb[4],0
        adc         word ptr opb[6],0
        jmp         just_add



just_add:
        invoke      add64, opa, opb, rptr

handle_sign:
        mov         si, word ptr rptr
        mov         dx, word ptr [si][4]
        mov         bx, word ptr [si][2]
        mov         ax, word ptr [si][0]

norm:
        sub         cx, cx
        cmp         ax,cx
        jne         not_zero
        cmp         bx,cx
        jne         not-zero
        cmp         dx,cx
        jne         not_zero
        jmp         write_result                        ;exit with a zero
not_zero:
        mov         cx,64
        cmp         dx,0h
        je          rotate_result_left
        cmp         dh,00h
        jne         rotate_result_right
        test        dl,80h
        je          rotate_result_left
        jmp         short done_rotate
rotate_result_right:
```

```
        shr         dX,l
        rcr         bx,l
        rcr         ax,1
        inc         byte ptr exponent             ;decrement exponent with each
                                                  ;shift
        test        dx,0ff00h
        je          done_rotate
        loop        rotate_result_right
rotate_result_left:
        shl         ax,1
        rcl         bx,l
        rcl         dx,l
        dec         byte ptr exponent             ;decrement exponent with each
                                                  ;shift
        test        dx,80h
        jne         done_rotate
        loop        rotate_result_left
done_rotate:
        and         dx,7fh
        shl         dx, 1
        or          dh, byte ptr exponent         ;insert  exponent
        shr         dx, 1
        mov         cl, byte ptr sign             ;sign of the result of the
                                                  ;operation
        or          cl, cl
        je          fix_sign
        or          dx,8000h
fix_sign:
        mov         cl,byte ptr signa             ;sign of the larger operand
        or          cl, cl
        je          write-result
        or          dx,8000h                      ;negative
write_result:
        mov         di,word ptr rptr
        mov         word ptr [di],ax
        mov         word ptr [di][2],bx
        mov         word ptr [di][4],dx
        sub         ax,ax
        mov         word ptr [di][6],ax
fp_addex:
        popf
        ret
fladd endp
```

```
;******
;
fp_div    proc c  uses si di,
                              fp0:dword, fp1:dword, rptr:word

      local      flp0:qword,  flp1:qword,  result:qword

      pushf
      cld
      xor        ax,ax
      lea        di,word ptr result
      mov        cx,4
  rep stosw

      lea        di,word ptr flp0
      mov        cx,4
  rep stosw

      lea        di,word ptr flp1
      mov        cx,4
  rep stosw

      lea        si,word ptr fp0
      lea        di,word ptr flp0[2]
      mov        cx,2
  rep movsw

      lea        si,word ptr fp1
      lea        di,word ptr flp1[2]
      mov        cx,2
  rep movsw

      invoke     fldiv, flp0, flp1, addr result     ;pass pointer to called
                                                    ;routine

      invoke     round,  result,  addr result

      lea        si,word ptr result[2]
      mov        di,rptr
      mov        cx,2
      movsw
```

```
      popf
      ret
fp_div    endp



; ***


;
fldiv proc        C  uses bx cx dx si di,
                          fp0:qword,  fp1:qword,  rptr:word

      local       qtnt:qword,  sign:byte,  exponent:byte,  rmndr:qword

      pushf
      std
      xor         ax,ax

      mov         byte ptr sign, al              ;begin error and situation
                                                 ;checking
      lea         si,word ptr fp0                ;name a pointer to each fp
      lea         bx,word ptr fp1

      mov         ax,word ptr [si][4]
      shl         ax,1
      and         ax,0ff00h                      ;check for zero
      jne         chk_b
      jmp         return_infinite;infinity
chk_b:
      mov         dx,word ptr [bx][4]
      sh1         dx,l
      and         dx,0ff00h
      jne         b_notz
      jmp         divide_b_zero                  ;infinity, divide by zero is
                                                 ;undefined
b_notz:
      cmp         dx,0ff00h
      jne         check_identity
      jmp         make_zero                      ;divisor is infinite
check-identity:
      mov         di,bx
      add         di,4                           ;will decrement selves
```

```
        add         si,4
        mov         cx,3
repe    cmpsw
        jne         not-same                    ;these guys are the same
        mov         ax,word ptr dgt[8];return a one
        mov         bx,word ptr dgt[10]
        mov         dx,word ptr dgt[12]
        mov         di,word ptr rptr
        mov         word ptr [di],ax
        mov         word ptr [di][2],bx
        mov         word ptr [di][4],dx
        sub         ax,ax
        mov         word ptr [di][6],ax
        jmp         fldivex
not_same:                                       ;get exponents
        lea         si,word ptr fp0             ;reset pointers
        lea         bx,word ptr fp1

        sub         ah,dh                       ;add exponents
        add         ah,77h                      ;subtract bias minus two
                                                ;digits
        mov         byte ptr exponent,ah        ;save exponent
        mov         dx, word ptr [si][4]        ;check sign
        or          dx, dx
        jns         a_plus
        not         byte ptr sign
a_plus:
        mov         dx,word ptr [bx][4]
        or          dx, dx
        jns         restore_missing_bit
        not         byte ptr sign
restore-missing-bit:                            ;line up operands for divi
                                                ;sion
        and         word ptr fp0[4], 7fh
        or          word ptr fp0[4],80h

        mov         dx, word ptr fp1[4]
        and         dx, 7fh
        or          dx, 80h
        cmp         dx, word ptr fp0[4]         ;see if divisor is greater
                                                ;than
        ja          store_dvsr
        inc         byte ptr exponent
        shr         word ptr fp0[4], 1
```

```
        rcr         word ptr fp0[2], 1
        rcr         word ptr fp0[0], 1
store_dvsr:
        mov         word ptr fp1[4], dx


divide:
        invoke      div64, fp0, fp1, addr fp0
        mov         dx, word ptr fp0[2]
        mov         bx, word ptr fp0[0]
        sub         ax, ax

        sub         cx,cx
        cmp         ax,cx
        jne         not_zero
        cmp         bx,cx
        jne         not-zero
        cmp         dx,cx
        jne         not_zero
        jmp         fix_sign                        ;exit with a zero
not_zero:
        mov         cx,64
        cmp         dx,0h
        je          rotate_result_left
        cmp         dh,00h
        jne         rotate_result_right
        test        dl,80h
        je          rotate_result_left
        jmp         short  done_rotate
rotate_result_right:
        shr         dx,l
        rcr         bx,l
        rcr         ax,1
        test        dx,0ff00h
        je          done_rotate
        inc         byte ptr exponent               ;decrement exponent with each
                                                    ;shift
        loop        rotate_result_right
rotate_result_left:
        shl         word ptr qtnt,1
        rcl         ax,1
        rcl         bx,l
        rcl         dx,l
        test        dx,80h
```

```
        jne         done_rotate
        dec         byte ptr exponent             ;decrement exponent with each
                                                  ;shift
        loop        rotate_result_left
done_rotate:
        and         dx,7fh
        shl         dx,1
        or          dh, byte ptr exponent         ;insert exponent
        shr         dx, 1
        mov         cl,byte ptr sign
        or          cl,cl
        je          fix_sign
        or          dx,8000h
fix-sign:
        mov         di,word ptr rptr
        mov         word ptr [di],ax
        mov         word ptr [di][2],bx
        mov         word ptr [di][4],dx
        sub         ax,ax
        mov         word ptr [di][6],ax
fldivex:
        popf
        ret

return_infinite:
        sub         ax, ax
        mov         bx, ax
        not         ax
        mov         dx, ax
        and         dx, 0f80h                     ;infinity
        jmp         short fix_sign
divide_by_zero:
        sub         ax,ax
        not         ax
        jmp         short finish-error

make_zero:
        xor         ax,ax
                                                  ;positive zero
finish-error:
        mov         di,word ptr rptr
        add         di,6
        mov         cx,4
rep     stos        word ptr [di]
```

```
        jmp          short fldivex
fldiv endp

;
; ******
;
;
fp_mul     proc c   uses si di,
                              fp0:dword,  fp1:dword,  rptr:word

        local        flp0:qword,  flp1:qword,  result:qword

        pushf
        cld
        xor          ax,ax
        lea          di,word ptr result
        mov          cx,4
rep     stosw

        lea          di,word ptr flp0
        mov          cx, 4
rep     stosw

        lea          di,word ptr flp1
        mov          cx, 4
rep     stosw

        lea          si,word ptr fp0
        lea          di,word ptr flp0[2]
        mov          cx, 2
rep     movsw

        lea          si,word ptr fp1
        lea          di,word ptr flp1[2]
        mov          cx,2
rep     movsw

        invoke       flmul, flp0, flp1, addr result   ;pass pointer to called
                                                      ;routine

        invoke       round,  result,  addr  result

        lea          si,word ptr result [2]
        mov          di,rptr
```

```
        mov        cx,2
rep     movsw
        popf
        ret
fp_mul     endp
;
;***
;
;

flmul   proc       C  uses bx cx dx si di,
                            fp0:gword,  fp1:gword,  rptr:word

        local      result[6]:word,sign:byte,   exponent:byte

        pushf
        std
        sub        ax,ax
        mov        byte ptr  sign,al
        lea        di,word ptr  result[10]
        mov        cx,6
rep     stosw


        lea        si,word ptr fp0               ;name a pointer to each fp
        lea        bx,word ptr fp1
        mov        ax,word ptr  [si][4]
        shl        ax,1
        and        ax,0ff00h                     ;check for zero
        jne        is_a_inf
        jmp        make_zero                     ;zero exponent
is_a_inf:
        cmp        ax,0ff00h
        jne        is_b_zero
        jmp        return_infinite               ;multiplicand is infinite
is_b_zero:
        mov        dx,word ptr  [bx][4]
        shl        dx,l
        and        dx,0ff00h                     ;check for zero
        jnz        is_b_inf
        jmp        make_zero                     ;zero exponent
is_b_inf:
        cmp        dx,0ff00h
        jne        get_exp
```

```
        jmp         return-infinite                 ;multiplicand is infinite
;
get_exp:
        sub         ah, 77h
        add         ah, dh                          ;add exponents
        mov         byte ptr exponent,ah            ;save exponent

        mov         dx,word ptr [si][4]
        or          dx, dx
        jns         a_plus
        not         byte ptr sign
a_plus:
        mov         dx,word ptr [bx][4]
        or          dx, dx
        jns         restore_missing_bit
        not         byte ptr sign

restore_missing_bit:
        and         word ptr fp0[4], 7fh            ;remove the sign and exponent
        or          word ptr fp0[4], 80h            ;and restore the hidden bit
        and         word ptr fp1[4], 7fh
        or          word ptr fp1[4], 80h

        invoke      mu164a, fp0, fp1, addr result ;multiply

        mov         dx,word ptr result [10]
        mov         bx,word ptr result[8]
        mov         ax,word ptr result[6]

        sub         cx,cx
        cmp         word ptr result[4], cx
        jne         not_zero
        cmp         ax,cx
        jne         not_zero
        cmp         bx,cx
        jne         not_zero
        cne         dx,cx
        jne         not_zero
        jmp         fix_sign                        ;exit with a zero
not_zero:
        mov         cx,64
        cmp         dx,0h
        je          rotate_result_left
        cmp         dh,00h
```

```
        jne         rotate_result_right
        test        dl,80h
        je          rotate_result_left
        jmp         short done_rotate
rotate_result_right:
        shr         dx,1
        rcr         bx,1
        rcr         ax,1
        test        dx,0ff00h
        je          done_rotate
        inc         byte ptr exponent          ;decrement exponent with each
                                               ;shift
        loop        rotate_result_right
rotate_result_left:
        shl         word ptr result[2], 1
        rcl         word ptr result[4], 1
        rcl         ax,1
        rcl         bx,1
        rcl         dx,1
        test        dx,80h
        jne         done_rotate
        dec         byte ptr exponent          ;decrement exponent with each
                                               ;shift
        loop        rotate_result_left
done_rotate:
        and         dx,7fh
        shl         dx, 1
        or          dh, byte ptr exponent      ;insert exponent
        shr         dx, 1
        mov         cl,byte ptr sign
        or          cl,cl
        je          fix_sign
        or          dx,8000h
fix_sign:
        mov         di,word ptr rptr
        mov         word ptr [di], ax
        mov         word ptr [di][2], bx
        mov         word ptr [di][4], dx
        sub         ax, ax
        mov         word ptr [di][6], ax
fp_mulex:
        popf
        ret
 :
```

```
return_infinite:
        sub         ax, ax
        mov         bx, ax
        not         ax
        mov         dx, ax
        and         fix,0f80h                        ;infinity
        jmp         short fix_sign

make_zero:
        xor         ax,ax
finish_error:
        mov         di, word ptr rptr
        add         di, 6
        mov         cx, 4
rep     stos        word ptr [di]
        jmp         short  fp_mulex
flmul   endp


;******
; cylinder- finds the volume of a cylinder using the floatingpoint rou-
                                                     ;tines in this module.
;                     volume = pi * r * r h
        .data
pi      qword       404956c10000H
        .code
;
cylinder    proc uses bx cx dx si di,
                                       radius:dword, height:dword, area:word

        local       result:qword,  r:qword,  h:qword

        sub         ax, ax                           ;clear space for intermediate
                                                     ;variables
        mov         cx, 4
        lea         di,word ptr r
rep     stosw

        mov         cx, 4
        lea         di, word ptr h
rep     stosw

        mov         ax, word ptr radius[0]           ;move IEEE format to extended
                                                     ;format
```

```
        mov         dx, word ptr radius[2]
        mov         word ptr r[2], ax
        mov         word ptr r[4], dx
        mov         ax, word ptr height[0]
        mov         dx, word ptr height[2]
        mov         word ptr h[2], ax
        mov         word ptr h[4], dx

        invoke      flmul, r, r, addr result
                                                    ;do r squared
        invoke      flmul, pi, result, addr result
                                                    ;multiply result by pi
        invoke      flmul, h, result, addr result
                                                    ;multiply by height
        invoke      round, result, addr result
                                                    ;round the result

        mov         di, word ptr area
        mov         ax, word ptr result[2]          ;move result back to IEEE
                                                    ;format
        mov         dx, word ptr result[4]
        mov         word ptr [di],ax
        mov         word ptr [di][2],dx

        ret
cylinder  endp

; ******
;       fixed-point support for floating-point routines

; ******
;Multiplies operands by ten, returning result in multiplicand
;and overflow byte in ax. Used for binary-to-decimal conversions
;multiplicand is a pointer to a double.

multen    proc uses bx cx dx di si, multiplicand:word

        mov         di,word ptr multiplicand
        mov         dx,word ptr [di]
        mov         cx,word ptr [di][2]
        sub         ax,ax

        shl         dx,1                            ;multiply by two
        rcl         cx, 1
```

```
        rcl         ax, 1

        mov         word ptr [di],dx                ;save result
        mov         word ptr [di][2],cx
        mov         word ptr [di][4],ax


        shl         dx,l                            ;multiply by four
        rcl         cx, 1
        rcl         ax,1


        shl         dx,l                            ;now make it eight
        rcl         cx, 1
        rcl         ax,1

        add         dx,word ptr [di]                ;add back the two to make ten
        adc         cx,word ptr [di][2]
        adc         ax,word ptr [di][4]

        mov         word ptr [di],dx;go home
        mov         word ptr [di][2],cx
        ret
multen    endp


; ******
;div64
;will divide a quadword operand by adivisor using linear interpolation.
;dividend occupies upper three words of a 6-word array
;divisor occupies lower three words of a 6-word array
;used by floating-point division only
div64   proc uses es ds,
                    dvdnd:qword, dvsr:qword, qtnt:word

        local       result:tbyte, tmp0:qword,
                                tmp1:qword, opa:qword, opb:qword

        pushf
        cld

        sub         ax, ax
        lea         di, word ptr result
        mov         cx, 4
```

```
rep     stosw
        lea         di, word ptr tmp0;quotient
        mov         cx, 4
rep     stosw

setup:
        mov         bx, word ptr dvsr[3]
continue_setup:
        lea         si, word ptr dvdnd              ;divisor no higher than
        lea         di, word ptr tmpo               ;receives stuff for quotient
        sub         dx, dx

        mov         ax, word ptr [si][3]
        div         bx
        mov         word ptr [di][4], ax            ;result goes into quotient
        mov         ax, word ptr [si][l]
        div         bx
        mov         word ptr [di][2], ax            ;result goes into quotient
        sub         ax, ax
        mov         ah, byte ptr [si]
        div         bx
        mov         word ptr [di][0], ax            ;result goes into quotient

chk_estimate:
        invoke      mu164a, tmp0, dvsr, addr result

        lea         di, word ptr tmp0
        mov         ax, word ptr result[7]
        cmp         ax, word ptr dvdnd[3]
        jle         div_exit

        sub         ax, ax
        sub         word ptr [di], 1
        sbb         word ptr [di][2],ax
        sbb         word ptr [di][4],ax
        mov         word ptr [di][6],ax          ;don't need a remainder for
                                                 ;this divide

div_exit:
        mov         si, di
        mov         di, word ptr qtnt
        inc         di
        inc         di
        mov         cx, 4
```

```
rep     movsw
        popf
        ret
div64  endp

; ******
;*Mu164a -Multiplies two unsigned 5-byte integers. The
;* procedure allows for a product of twice the length of the multipliers,
;*  thus  preventing  overflows.
mu164a proc uses ax dx,
                        multiplicand:qword,  multiplier:qword,  result:word


        mov        di,word ptr result
        sub        cx, cx

;
        mov        ax, word ptr multiplicand[4]   ;multiply multiplicand MSW
        mul        word ptr multiplier[4]         ;by multiplier high word
        mov        word ptr [dil[8], ax

        mov        ax, word ptr multiplicand[4]   ;multiply multiplicand MSW
        mul        word ptr multiplier[2]         ;by second MSW
        mov        word ptr [di][6], ax           ;of  multiplier
        add        word ptr [di][8], dx

        mov        ax, word ptr multiplicand[4]   ;multiply multiplicand high
                                                  ;word
        mul        word ptr multiplier[0]         ;by third MSW
        mov        word ptr  [di][4], ax          ;of multiplier
        add        word ptr  [di][6], dx
        adc        word ptr  [di][8], cx          ;propagate  carry
;

        mov        ax, word ptr multiplicand[2]   ;multiply second MSW
        mul        word ptr multiplier[4]         ;of multiplicand by MSW
        add        word ptr  [di][6], ax          ;of multiplier
        adc        word ptr  [di][8], dx

        mov        ax, word ptr multiplicand[2] ;multiply second MSW of
        mul        word ptr multiplier[2]         ;multiplicand by second MSW
        add        word ptr  [di][4], ax          ;of multiplier
        adc        word ptr  [di][6], dx
```

```
        adc     word ptr [di][8], cx            ;add any remnant carry

        mov     ax, word ptr multiplicand[2]    ;multiply second MSW
        mul     word ptr multiplier[0]          ;of multiplicand by LSW
        mov     word ptr [di][2], ax            ;of multiplier
        add     word ptr [di][4], dx
        adc     word ptr [di][6], cx
        adc     word ptr [di][8], cx            ;add any remnant carry

        mov     ax, word ptr multiplicand[0]    ;multiply multiplicand LSW
        mul     word ptr multiplier[4]          ;by MSW of multiplier
        add     word ptr [di][4], ax
        adc     word ptr [di][6], dx
        adc     word ptr [di][8], cx            ;add any remnant carry

        mov     ax, word ptr multiplicand[0]    ;multiply multiplicand LSW
        mul     word ptr multiplier[2]          ;by second MSW
        add     word ptr [di][2], ax            ;of multiplier
        adc     word ptr [di][4], dx
        adc     word ptr [di][6], cx            ;add any remnant carry
        adc     word ptr [di][8], cx            ;add any remnant carry

        mov     ax, word ptr multiplicand[0]    ;multiply multiplicand LSW
        mul     word ptr multiplier[0]          ;by multiplier low word
        mov     word ptr [di][0], ax
        add     word ptr [di][2], dx
        adc     word ptr [di][4], cx            ;add any remnant carry
        adc     word ptr [di][6], cx            ;add any remnant carry
        adc     word ptr [di][8], cx            ;add any remnant carry

        ret
mul64a  endp
```

APPENDIX E

IO.ASM

```
.dosseg
.model small, c, os_dos
include math.inc


        .data
;
inf     byte        "infinite", 0
zro     byte        "0.0",0
hundred             byte        64h
iten    word        0ah
powers  equ         one
maxchar             equ         8

        .code
; ******
;dectohex
;pointer to a packed BCD is used to convert to binary
;
dectohex proc uses ax bx cx si di, dntgr:word

        local       double:dword

        xor         ax,ax
        mov         si,word ptr dntgr
        mov         cx,4

cnvt_int:

        mov         al,byte ptr [si]
        aam                                         ;expand to unpacked form
```

```
        push        ax
        xchg        ah,al                               ;get high nibble
        sub         ah,ah
        add         bx,ax

        call        near ptr mten                       ;multiply by ten

        pop         ax
        sub         ah,ah
        add         bx,ax

        call        near ptr mten                       ;multiply by ten

        loop        cnvt_int
        ret

mten:
        shl         bx,1
        rcl         dx,1
        mov         word ptr double,bx
        mov         word ptr double[2],dx
        shl         bx,l
        rcl         dx,l
        shl         bx,l
        rcl         dx,l
        add         bx,word ptr double
        adc         dx,word ptr double[2]
        retn
dectohex            endp
; ******
;converts single-precision floating point to an ASCII string
; caller is responsible for array bounds of ASCII string
; callable from C
ftoasc proc uses si di, fp:dword, rptr:word

        local       flp:qword

        cld
        xor         ax,ax
```

```
        lea         di,word ptr flp
        mov         cx,4
rep     stosw

        lea         si,word ptr fp
        lea         di,word ptr flp[2]
        mov         cx,2
rep     movsw

        invoke      fta, flp, rptr

        ret
ftoasc  endp

; ***
; conversion of floating point to ASCII

;
fta     proc uses bx cx dx si di, fp:qword, sptr:word
        local       sinptr:byte, fixptr:qword, exponent:byte,
                                leading_zeros:byte, ndg:byte

        pushf
        std

        xor         ax,ax
        lea         di,wordptr  fixptr[6]
        mOV         cx,4
rep     stosw
        mov         byte ptr [sinptr],al                    ;clear the sign
        mov         byteptr [leading_zeros],al
        mOV         byte ptr [ndg],al
        mov         byte ptr [exponent],al

ck_neg:
        test        word ptr fp[4],8000h                    :get the sign
        je          gtr_0
        xor         word ptr fp[4],8000h                    ;make positive
        not         byte ptr [sinptr]                       ;it is negative
```

```
;***
gtr_0:
        invoke      flcomp, fp, one            ;another kind of normalization
        cmp         ax,1h
        je          less_than_ten
        dec         byte ptr [ndg]
        cmp         byte ptr [ndg],-37
        jl          zero_result
        invoke      flmul, fp, ten, addr fp
        jmp         short gtr_0

less_than_ten
        invoke      flcomp, fp, ten
        cmp         ax, -1
        je          norm_fix
        inc         byte ptr [ndg]
        cmp         byte ptr [ndg],37
        53          infinite_result
        invoke      fldiv, fp, ten, addr fp
        jmp         short less_than_ten

Rnd:
        invoke      round, fp, addr fp         ;fixup for translation

norm_fix:                                      ;this is for ASCII conversion
        mov         ax,word ptr fp[0]          ;dump the sign bit
        mov         bx,word ptr fp[2]
        mov         dx,word ptr fp[4]
        shl         dx,l

get_exp:
        mov         byte ptr exponent, dh
        sub         byte ptr exponent, 7fh     ;remove bias

        mov         cx,8h
        sub         cl,byte ptr exponent
        js          infinite_result            ;could come out zero
                                               ;but this is as far as I
        lea         di,word ptr fixptr
do_shift:                                      ;can go
```

```
        stc                                     ;restore hidden bit
        rcr         dl,1
        sub         cx,1
        je          put_upper
shift_fraction:
        shr         dl,1                        ;shift significand into
        rcr         bx,1                        ;fractional part
        rcr         ax, 1
        loop        shift_fraction

put-upper:
        mov         word ptr [di], ax
        mov         word ptr [di][2],bx
        mov         al,dl
        mov         byte ptr fixptr[4],dl       ;write integer portion
        xchg        ah,al

        sub         dx,dx
        mov         di,word ptr sptr
        cld                                     ;reverse direction of write
        inc         dx
        mov         al,' '
        cmp         byte ptr sinptr,0ffh        ;is it a minus?
        jne         put_sign
        mov         al,'-'
put_sign:
        stosb

        lea         si, byte ptr fixptr[3]

write_integer:
        xchg        ah,al                       ;al contains integer
                                                ;portion

        aam
        xchg        al,ah
        or          al,'0'
        call        near ptr str_wrt
        xchg        al,ah
        or          al,'0'
        call        near ptr str_wrt
```

```
        inc         dx                              ;max char count
        dec         si

do_decimal:
        mov         al,'.'
        stosb
do_decimal1:
        invoke      multen, addr fixptr             ;convert binary fraction
        or          al,'0'                          ;to decimal fraction
        call        nearptr str_wrt
        inc         dx
        cmp         dx,maxchar
        je          do_em
        jmp         short do_decimal1

do_exp:
        sub         ax,ax
        cmp         al,byte ptr ndg
        jne         write_exponent
        jmp         short last_byte

write_exponent:
        mov         al,'e'
        stosb
        mov         al,byte ptr ndg
        or          al,al
        jns         finish_exponent
        xchg        al,ah
        mov         al,'-'
        stosb
        neg         ah
        xchg        al,ah
        sub         ah,ah
finish_exponent:
        cbw
        aam                                         ;cheap conversion
        xchg        ah,al
        or          al,'0'
        stosb
        xchg        ah,al
```

```
        or          al,'0'
        stosb
last_byte:
        sub         al,al
        stosb
        popf
fta_ex:
        ret

infinite_result:
        mov         di,word ptr sptr
        mov         si,offset inf
        mov         cx, 9
rep     movsb
        mov         ax, -1
        jmp         short fta_ex

zero_result:
        mov         di,word ptr sptr
        mov         si,offset zro
        mov         cx, 9
rep     movsb
        mov         ax,-1
        jmp         short fta_ex

strwrt:
        cmp         al,'0'
        jne         putt
        test        byte ptr leading_zeros,-1
        je          nope
putt:
        test        byte ptr leading_zeros,-1
        jne         pmt
        not         leading_zeros
pmt:
        stosb
nope:
        retn
fta     endp
;
```

```
;
; ******
;
;Unsigned conversion from floating-point notation to integer (long).
;This is in fixed-point format; the upper two words are the integer
;and the lower two are the fraction.
;
ftofx   proc uses si di, fp:dword, fixptr:word

        local       flp:qword

        cld
        xor         ax,ax
        lea         di,word ptr flp
        mov         cx,4
rep     stosw

        lea         si,word ptr fp
        lea         di,word ptr flp[2]
        mov         cx,2
rep     movsw

        invoke      ftfx, flp, fixptr

        ret
ftofx endp

;******


;unsigned conversion from ascii string to short real
atf     proc uses si di, string:word, rptr:word        ;one word for near pointer

        local exponent:byte, fp:qword, numsin:byte, expsin:byte,
                    dp_flag:byte, digits:byte, dp:byte

        pushf
        std
        xor         ax,ax
        lea         di,word ptr fp[6]                   ;clear the floating
                                                        ;variable
```

```
        mov         cx,8
rep     stosw       word ptr [di]

        mov         si,string

do_numbers:
        mov         byte ptr [exponent],al
        mov         byte ptr dp_flag,al
        mov         byte ptr numsin,al
        mov         byte ptr expsin,al
        mov         byte ptr dp,al
        mov         byte ptr digits,8h          ;count of total digits;
                                                ;rounding digit is eight

;
;begin by checking for a sign or a number or a '.'

do_num:
        mov         bl, [si]
        cmp         bl,'-'
        jne         not_minus                   ;it is a negative number
        not         [numsin]
        inc         si
        mov         bl,es:[si]
        jmp         not_sign

not_minus:
        cmp         bl,'+'
        jne         not_sign
        inc         si
        mov         al, [si]

not_sign:
        cmp         bl,'.'                      ;check for decimal point
        jne         not_dot
        test        byte ptr [dp],80h
        jne         end_o_cnvt
        not         dp_flag
        inc         si
        mov         bl,[si]
```

```
not_dot:
        cmp       bl,'0'                          ;get legitimate number
        jb        not_a_num
        cmp       bl,'9'
        ja        not_a_num
        invoke    flmul, fp, ten, addr fp         ;multiply floating point
        mov       bl, [si]                        ;accumulator by l0.0
        sub       bl,30h
        sub       bh,bh                           ;clear upper byte
        shl       bx,l                            ;multiply index for proper
                                                  ;offset
        shl       bx,1
        shl       bx,1
        invoke    fladd, fp, dgt[bx], addr fp
        test      byte ptr [dp_flag],0ffh         ;have we encountered a
                                                  ;decimal point yet?
        je        no_dot_yet
        dec       [dp]
no_dot_yet:
        inc       si
        dec       byte ptr digits
        jc        not_a_num
        mov       bl,es:[si]
        jmp       not-sign

not_a_num:
        mov       bl, [si]
        or        bl,lower_case
        cmp       bl,'e'                          ;check for decimal point
        je        chk_exp                         ;looks like we may have an
                                                  ;exponent
        jmp       end_o_cnvt


chk_exp:
        inc       si
        mov       bl, [si]
        cmp       bl,'-'
        jne       chk_plus
```

```
        not         [expsin]
        jmp         short  chk_expl
chk_plus:
        cmp         bl,'+'
        jne         short  chk_exp2
chk_exp1:
        inc         si
        mov         bl, [si]
chk_exp2:
        cmp         bl,'0'
        jb          end_o_cnvt
        cmp         bl,'9'
        ja          end_o_cnvt
        sub         ax,ax
        mov         al, byte ptr [exponent]
        mul         iten
        mov         byte ptr [exponent],al
        mov         bl, [si]
        sub         bl,30h
        or          byte ptr [exponent],bl
        jmp         short chk_exp1

end_o_cnvt:
        sub     cx,cx
        mov     al,byte ptr [expsin]
        mov     cl,byte ptr [dp]
        or          al,al
        jns         pos_exp
        sub     cl,byte ptr [exponent]
        jmp     short chk_numsin
pos_exp:
        add     cl,byte ptr [exponent]                  ;exponent

chk_numsin:
        cmp     word ptr numsin,0ffh
        jne     chk_expsin
        or      word ptr fp[4],8000h                    ;if exponent negative,
chk_expsin:                                             ;so is number
        xor         ax,ax
        or          cl,cl
```

```
        jns         do_pospow                           ;make exponent positive
        neg         cl

do_negpow:
        or          cl,cl                               ;is exponent zero yet?
        je          atf_ex
        inc         ax
        test        cx,1h                               ;check for one in lsb
        je          do_negpowa
        mov         bx,ax
        push        ax
        shl         bx,1
        shl         bx,1
        shl         bx,1
        invoke      fldiv, fp, powers[bx], addr fp      ;divide by power of two
        pop         ax
do_negpowa:
        shr         cx,1
        jmp         short do_negpow

do_pospow:
        or          cl,cl                               ;is exponent zero yet?
        je          atf_ex
        inc         ax
        test        cx,lh                               ;check for one in lsb
        je          do_pospowa
        mov         bx,ax
        push        ax
        shl         bx,1
        shl         bx,1
        invoke      flmul, fp, powers[bx], addr fp      ;multiply by power of two
        pop         ax
do_pospowa:
        shr         cx,1
        jmp         shortdo_pospow
atf_ex:
        invoke      round, fp, addr fp

        mov         di,word ptr rptr
        mov         ax,word ptr fp
```

```
        mov         bx,word ptr fp[2]
        mov         dx,word ptr fp[4]
        mov         word ptr [di],bx
        mov         word ptr [di][2],dx
        popf
        ret
atf     endp

; ******



;Unsigned conversion from quadword fixed-point to short real.
;The intention is to accommodate long and int conversions as well.
;Binary is passed on the stack and rptr is a pointer
;to the result.

ftf     proc uses si di, binary:qword, rptr:word    ;one word for near
                                                    ;pointer

        local  exponent:byte,  numsin:byte

        pushf
        xor         ax, ax
;
        mov    di, word ptr rptr                    ;point at future float
        add         di, 6
        lea         si, byte ptr binary[0]          ;point to quadword
        mov         bx, 7                           ;index
;
do_numbers:
        mOV         byte ptr [exponent], al
        mov         byte ptr numsin, al
        mov         dx, ax
;
do_num:
        mov         al, byte ptr [si][bx]
        or          al, al
;record sign
        jns         find_top
        not         byte ptr numsin                 ;this one is negative
        not         word ptr binary[6]
```

```
        not         word ptr binary[4]
        not         word ptr binary[2]
        neg         word ptr binary[0]
        jc          find_top
        add         word ptr binary[2], 1
        adc         word ptr binary[4], 0
        adc         word ptr binary[6], 0


find_top:
        cmp         bl, dl
        je          make_zero                   ;we traversed the entire
                                                ;number
        mov         al, byte ptr [si][bx]
        or          al,al
        jne         found_it
        dec         bx                          ;move index
        jmp         short find_top

found_it:
        mov         dl, 80h                     ;test for MSB
        cmp         bl, 4
        cmp         shift_right
        je          final_right

shift_left
        std
        mov         cx, 4                       ;points to MSB
        sub         cx, bx                      ;target
        shl         cx, 1
        shl         cx, 1
        shl         cx, 1                       ;times 8
        neg         cx
        mov         byte ptr [exponent], cl

        lea         di, byte ptr binary[4]
        lea         si, byte ptr binary
        add         si, bx
        mov         cx, bx
        inc         cx
```

```
rep     movsb

        mov         cx, 4
        sub         cx, bx
        sub         ax, ax
rep     stosb
        jmp         short final_right

shift_right:
        cld
        mov         cx, bx                          ;points to MSB
        sub         cx, 4                           ;target
        lea         si, byte ptr binary[4]
        mov         di, si
        sub         di, cx

        shl         cl, 1
        shl         cl, 1
        shl         cl, 1
                                                    ;times 8
        mov         byte ptr [exponent], cl

        mov         cx, bx
        sub         cx, 4
        inc         cx
rep     movsb
        sub         bx, 4
        mov         cx, 4
        sub         cx, bx
        sub         ax, ax
        lea         di, word ptr binary
rep     stosb

final_right:
        lea         si, byte ptr binary[4]
final_right1:
        mov         al, byte ptr [si]
        test        al, dl
        jne         aligned
        dec         byte ptr exponent
```

```
        shl         word ptr binary[0], 1
        rcl         word ptr binary[2], 1
        rcl         word ptr binary[4], 1
        jmp         short final_right1


aligned:
        shl         al, 1
                                                     ;clearbit
        mov         ah, 86h
        add         ah, byte ptr exponent
        cmp         numsin,dh
        je          positive
        stc
        jmp         short get_ready_to_go
positive:
        clc

get-ready_to_go:
        rcr         ax, 1                            ;put it all back the way it
                                                     ;should be
        mov         word ptr binary[4], ax

ftf_ex:
        invoke      round, binary, rptr

exit:
        popf
        ret

make_zero:
        std
        sub         ax, ax                           ;zero it all out
        mov         cx, 4
rep     stosw
        jmp         short exit
ftf     endp

;
; ***
```

```
;Conversion of floating point to fixed point
;float enters as quadword
;pointer, sptr, points to result
;This could use an external routine as well. When the float
;enters here, it is in extended format

ftfx   proc uses bx cx dx si di, fp:qword, sptr:word

       local        sinptr:byte, exponent:byte

       pushf
       std

       xor          ax,ax
       mov          byte ptr [sinptr],al             ;clear the sign
       mov          byte ptr [exponent],al
       mov          di,word ptr sptr                 ;point to result
;
; ***
;
;
do_rnd:
       invoke       round, fp, addr fp               ;fixup for translation
;
set_sign:
       mov          ax,word ptr fp[0]
       mov          bx,word ptr fp[2]
       mov          dx,word ptr fp[4]
       or           dx,dx
       jns          get_exponent
       not          byte ptr [sinptr]                ;it is negative

get_exponent:
       sub          cx,cx
       shl          dx,l
       sub          dh,86h                           ;remove bias from exponent
       mov          byte ptr exponent, dh
       mov          cl,dh
       and          dx,0ffh                          ;save number portion
       stc
       rcr          dl,1                             ;restore hidden bit
```

```
;
which_way:
        or              cl,cl
        jns             shift_left
        neg             cl

shift_right:
        cmp             cl,28h
        ja              make_zero                        ;no significance, too small
make_fraction:
        shr             dx,1
        rcr             bx,1
        rcr             ax,1
        loop            make_fraction
        mov             word ptr [di][0],ax
        mOV             word ptr [di][2],bx
        mov             word ptr [di][4],dx
        jmp             short print_result

shift_left:
        cmp             cl,18h
        ja              make_max                         ;failed significance, too
big
make_integer:
        shl             bx,1
        rcl             dx,1
        rcl             ax,1
        loop            make_integer
        mov             word ptr [di][6],ax
        mov             word ptr [di][4],dx
        mov             word ptr [di][2],bx

print_result:
        test            byte ptr [sinptr], 0ffh
        je              exit
        not             word ptr [di][6]                 ;two's complement
        not             word ptr [di][4]
        not             word ptr [di][2]
        neg             word ptr [di][0]
        jc              exit
```

```
        add         word ptr [di][2],1
        adc         word ptr [di][4],0
        adc         word ptr [di][6],0

exit:
        popf
        ret

make_zero:
        sub         ax,ax
        mov         cx,4
rep     stosw
        jmp         short exit

make_max:
        sub         ax,ax
        mov         cx,2
rep     stosw
        not         ax
        stosw
        and         word ptr [di][4], 7f80h
        not         ax
        stosw
        jmp         short exit


ftfx    endp
;
;*****************************************************
; dnt_bn - decimal integer to binary conversion routine
;unsigned
;It is expected that decptr points at a string of ASCII decimal digits.
;Each digit is taken in turn and converted until eight have been converted
;or until a nondecimal number is encountered.
;This might be used to pull a number from a communications buffer.
;Returns with no carry if successful and carry set if not.

dnt_bn  proc        uses bx cx dx si di, decptr:word, binary:word
```

```
        mov         si,word ptr decptr          ;get pointer to the MSB of the
                                                ;decimal
                                                ;value
        sub         ax,ax
        mov         bx,ax
        mov         dx,bx
        mov         cx, 9

decimal_conversion:
        mov         al,byte ptr [si]
        cmp         al,'0'                      ;check for decimal digit
        jb          work_done
        cmp         al,'9'
        ja          work_done                   ;if it gets past here, it must
                                                ;be OK
        call        near ptr times-ten
        xor         al,'0'                      ;convert to number
        add         bx,ax
        adc         dx,0                        ;propagate any carries
        inc         si
        loop        decimal_conversion
oops:
        stc                                     ;more than eight digits or
                                                ;something
        ret

work-done:
        mov         di, word ptr binary
        mov         word ptr [di],bx
        mov         word ptr [di][2],dx         ;store result
        clc
        ret

times_ten:
        push        ax
        push        cx
        shl         bx,l
        rcl         dx,l

        mov         ax,bx
```

```
        mov         cx,dx

        shl         bx,1
        rcl         dx,1

        shl         bx,1
        rcl         dx,1

        add         bx,ax
        adc         dx,cx                               ;multiply by ten

        pop         cx
        pop         ax
        retn
dnt_bn  endp

;*****************************************************
;bn-dnt - a conversion routine that converts binary data to decimal
;A double word is converted. Up to eight decimal digits are
;placed in the array pointed at by decptr. If more are required to adequately
;convert this number, the attempt is aborted and an error flagged.

bn_dnt  proc        uses bx cx dx si di, binary:dword, decptr:word

        lea         si,word ptr binary                  ;get pointer to the MSBb of
                                                        ;the decimal
                                                        ;value
        mov         di,word ptr decptr                  ;string of decimal ASCII
                                                        ;digits
        mov         cx, 9
        add         di,cx                               ;point to the end of the
                                                        ;string
                                                        ;this is for correct
                                                        ordering
        sub         bx,bx
        mov         dx,bx
        mov         byte ptr [di],bl                    ;see that string is
                                                        ;zero-terminated
        dec         di
```

```
binary_conversion:
        sub         dx,dx
        mov         ax,word ptr [si][2]
        or          ax,ax
        je          chk_empty
        div         iten                            ;divide by ten
        mov         word ptr [si][2],ax
        or          dx,dx
        je          chk_empty

divide_lower:
        mov         ax, word ptr [si]
        or          ax,ax
        jne         not_zero
        or          dx, ax
        je          put_zero
not_zero:
        div         iten
put_zero:
        mov         word ptr [si],ax
        or          dl,'0'
        mov         byte ptr [di],dl
        dec         di
        loop        binary_conversion

oops:
        mov         ax,-1
        stc
        ret

chk_empty:
        or          dx,dx
        je          still_nothing
        jmp         short divide_lower
still_nothing:
        mov         ax,word ptr [si]
        or          ax,ax
        je          empty
        jmp         short not_zero
```

```
empty:
        inc       di
        mov       si,di
        mov       di, word ptr decptr
        mov       cx,9
rep     movsw

finished:
        sub       ax,ax
        clc
        ret
bn_dnt  endp


;*****************************************************
;bfc_dc -A conversion routine that converts a binary fraction (doubleword)
;To decimal ASCII representation pointed to by the string pointer, decptr.
;Set for eight digits; it could be longer.

bfc_dc  proc      uses bx cx dx si di bp, fraction:dword, decptr:word

        local     sva:word, svb:word, svd:word


        mov       di,word ptr decptr              ;point to ASCII output
                                                  ;string
        mov       bx,word ptr fraction
        mov       dx,word ptr fraction[2]         ;get fractional part

        mov       cx,8                            ;digit counter
        sub       ax,ax

        mov       byte ptr [di],'.'               ;to begin the ASCII
                                                  ;fraction
        inc       di

decimal_conversion:
        or        ax,dx                           ;check for zero operand
        or        ax,bx                           ;check for zero operand
        jz        work_done
```

```
        sub         ax,ax

        shl         bx,1                        ;multiply fraction by ten
        rcl         dx,1
        rcl         ax,1

                                                ;times 2 multiple
        mov         word ptr svb,bx
        mov         word ptr svd,dx
        mov         word ptr sva,ax

        shl         bx,1
        rcl         dx,1
        rcl         ax,1

        shl         bx,1
        rcl         dx,1
        rcl         ax,1

        add         bx,word ptr svb
        adc         dx,word ptr svd             ;multiply by ten
        adc         ax,word ptr sva

        or          al,'0'
        mov         byte ptr [di],al
        inc         di
        sub         ax,ax
        loop        decimal_conversion

work_done:
        mov         byte ptr [di],al            ;end string with a null
        clc
        ret
bfc_dc  endp
;
;
;*****************************************************
;dfc_bn - A conversion routine that converts an ASCII decimal fraction
;to binary representation. Decptr points to the decimal string to be converted.
;The conversion will produce a double word result.
```

```
;The fraction is expected to be padded to the right if it does not
;fill eight digits.
;
dfcbn proc        uses bx cx dx si di, decptr:word, fraction:word

        pushf
        cld

        mov         di, word ptr decptr
        sub         ax,ax
        mov         cx, 9
repne   scasb
        dec         di
        dec         di                          ;point to least
                                                ;significant byte
        mov         si,di

        mov         di, word ptr fraction
        mov         word ptr [di],ax
        mov         word ptr [di][2], ax

        mov         cx,8

        sub         dx,dx


binary_conversion:
        mov         ax, word ptr [di][2]        ;get high word of result
                                                ;variable
        mov         dl, byte ptr [si]
        cmp         dl, '0'                     ;check for decimal digit
        jb          oops
        cmp         dl, '9'
        ja          oops                        ;if it gets past here, it
                                                ;must be o.k.
        xor         dl, '0'                     ;deASCIIize

        dec         si

        sub
```

```
        or          bx,dx
        or          bx,ax
        jz          no_div0                         ;prevent a divide by zero
        div         iten                            ;divide by ten
no_div0:
        mov         word ptr [di][2],ax

        mov         ax,word ptr [di]
        sub         bx,bx
        or          bx,dx
        or          bx,ax
        jz          no_div1                         ;prevent a divide by zero
        div         iten
no_divl:
        mov         word ptr [di],ax

        sub         dx,dx
        loop        binary_conversion

work_done:
        popf
        sub         ax,ax
        clc
        ret


oops:
        popf
        mov         ax,-1
        stc
        ret
dfc_bn  endp
:
;
; ******
;table conversion routines
```

```
        .data

int_tab         dword       3b9aca00h, 05f5e100h, 00989680h, 000f4240h,
                            000186a0h, 00002710h, 000003e8h, 00000064h,
                            0000000ah, 00000001h
frac_tab        dword       1999999ah, 028f5c29h, 00418937h, 00068db9h,
                            0000a7c5h, 000010c6h, 000001adh, 0000002ah,
                            00000004h
tab_end         dword       00000000h

;
       .code

;convert ASCII decimal to fixed-point binary
;
tb_dcbn              proc   uses bx cx dx si di,
                            sptr:word, fxptr:word

       local        sign:byte

       mov          di, word ptr sptr            ;point to result
       mOV          si, word ptr fxptr           ;point to ASCII string
       lea          bx, word ptr frac_tab        ;point into table

       mov          cx,4                         ;clear the target variable
       sub          ax,ax
       sub          dx,dx
rep    stosw

       mov          di, word ptr sptr            ;point to result

       mov          cl,al                        ;to count integers
       mov          ch,9h                        ;max int digits
       mov          byte ptr sign, al            ;assume positive

       mov          al, byte ptr [si]            ;get character
       cmp          al,'-'                       ;check for sign
       je           negative
       cmp          al,'+'
       je           positive
```

```
count:
        cmp     al,'.'
        je      fnd_dot
chk_frac:
        cmp     al,0                            ;end of string?
        je      gotnumber
        cmp     al,'0'                          ;is it a number then?
        jb      not_a_number
        cmp     al,'9'
        ja      not_a_number
cntnu:
        inc     cl                              ;count
        cmp     cl,ch                           ;check size
        ja      too_big
        inc     si                              ;next character
        mov     al, byte ptr [si]               ;get character
        or      dh,dh                           ;int or frac
        jne     chk_frac
        jmp     short count                     ;count characters in int

fnd_dot:
        mov     dh,cl
        inc     dh                              ;can't be zero
        mov     dl,13h                          ;includes decimal point
        xchg    ch,dl
        jmp     short cntnu
negative:
        not     sign
positive:
        inc     si
        mov     word ptr fxptr,si
        mov     al, byte ptr [si]
        jmp     short count

gotnumber:
        sub     ch,ch
        xchg    cl,dh                           ;get int count
        dec     cl
        shl     word ptr cx,1                   ;multiply by four
```

```
        shl     word ptr cx,l
        sub     bx,cx                           ;index into able
        sub     cx,cx
        mov     si,word ptr fxptr               ;point at string again
cnvrt_int:
        mov     cl,byte ptr [si]                ;get first character
        cmp     cl,'.'
        je      handle_fraction                 ;go do fraction, if any
        cmp     cl,0
        je      do_sign                         ;end of string
        sub     cl,'O'
        mov     ax,word ptr [bx][2]
        mul     cx
        add     word ptr [di][4l,ax
        adc     word ptr [di][6],dx
        mov     ax,word ptr [bx]
        mul     cx
        add     word ptr [di][4],ax
        adc     word ptr [di][6],dx
        add     bx,4                            ;drop table pointer
        inc     si
        jmp     short cnvrt_int

handle_fraction:
        inc     si                              ;skip decimal point
cnvrt_frac:
        mov     cl,byte ptr [si]                ;get first character
        cmp     cl,0
        je      do_sign                         ;end of string
        sub     cl,'0'
        mov     ax,word ptr [bx][2]             ;this can never result in
                                                ;a carry
        mul     cx
        add     word ptr [di][2],ax
        mov     ax,word ptr [bx]
        mul     cx
        add     word ptr [di][0],ax
        adc     word ptr [di][2],dx
        add     bx,4                            ;drop table pointer
        inc     si
```

```
        jmp             short cnvrt_frac

do_sign:
        mov             al,byte ptr sign
        or              al,al
        je              exit                                    ;it is positive
        not             word ptr [di][6]
        not             word ptr [dil[4]
        not             word ptr [di][2]
        neg             word ptr [di]
        jc              exit
        add             word ptr [di][2],1
        adc             word ptr [di][4],0
        adc             word ptr [di][6],0
exit:
        ret

not_a_number:
        sub             ax,ax
        not             ax
too_big:
        stc
        jmp             short exit
tb_dcbn                 endp


;converts binary to ASCII decimal

tb_bndc                 proc   uses bx cx dx si di,
                               sptr:word,  fxptr:word

        local           leading_zeros:byte

        mov             si, word ptr fxptr              ;point to input fix point
        mov             di, word ptr sptr               ;point to ascii string
        lea             bx, word ptr int_tab            ;point into table

        sub             ax,ax
        mov             byte ptr leading_zeros, al      ;assume positive
```

```
        mov     ax, word ptr [si][6]
        or      ax,ax
        jns     positive
        mov     byte ptr [di],'-'
        inc     di
        not     word ptr [si][6]            ;complement
        not     word ptr [si][4]
        not     word ptr [si][2]
        neg     word ptr [si][0]
        jc      positive
        add     word ptr [si][2],1
        adc     word ptr [si][4],0
        adc     word ptr [si][6],0

positive:
        mov     dx, word ptr [si][6]
        mov     ax, word ptr [si][4]        ;get integerportion
        sub     cx,cx
walk_tab:
        cmp     dx, word ptr [bx][2]
        ja      gotnumber
        jb      pushptr
        cmp     ax, word ptr [bx]
        jae     gotnumber
pushptr:
        cmp     byte ptr cl, leading_zeros
        je      skip_zero
        mov     word ptr [di],'0'

cntnu:
        inc     di
skip_zero:
        inc     bx
        inc     bx
        inc     bx
        inc     bx
        cmp     bx, offset word ptr frac_tab
        jae     handle_fraction
        jmp     shortwalk_tab
```

```
gotnumber:
        sub         cx,cx
        inc         leading_zeros
cnvrt_int:
        call        near ptr index
        jmp         short cntnu

handle_fraction:
        cmp         byte ptr leading_zeros,0
        jne         do_frac
        mov         byte ptr [di], '0'
        inc         di
do_frac:
        mov         word ptr [di],'.'                   ;put decimal point
        inc         di
get_frac:
        mov         dx, word ptr [si][2]
        mov         ax, word ptr [si][0]
        sub         cx,cx
walk_tabl:
        cmp         dx, word ptr [bx][2]
        ja          small_enuf
        jb          pushptr1
        cmp         ax, word ptr [bx]
        jae         small_enuf
pushptr1:
        mov         byte ptr [di],'0'
skip_zero1:
        inc         di
        inc         bx
        inc         bx
        inc         bx
        inc         bx
        cmp         bx, offset word ptr tab_end
        jae         exit
        jmp         short walk_tab1

small_enuf:
        sub         cx,cx
small_enuf1:
```

```
        call        near ptr index
        jmp         short skip_zero1

exit:
        inc         di
        sub         cl,cl
        mov         byte ptr [si],cl                ;end of string
        ret

index:
        inc         cx
        sub         ax, word ptr [bx]
        sbb         dx, word ptr [bx][2]
        jnc         index                           ;subtract until a carry
        dec         cx
        add         ax, word ptr [bx]               ;put it back
        adc         dx, word ptr [bx][2]
        or          cl,'0'                          ;make it ASCII
        mov         byte ptr [di],cl
        retn
tb_bndc             endp


; ******
;hex to ascii conversion using xlat
;simple and common table driven routine to convert from hexidecimal
;notation to ascii
;quadword  argument is passed on the stack, with the result returned
;in a string pointed to by sptr

        .data

hextab  byte        '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a',
                    'b', 'c', 'd', 'e', 'f'

        .code

hexasc  proc        uses bx cx dx si di, hexval:qword, sptr:word
```

```
        lea         si, byte ptr hexval[7]
        mov         di, word ptr sptr
        mov         bx, offset byte ptr hextab
        mov         cx,8                          ;number of bytes to be
                                                  ;converted

make_ascii:
        mov         al, byte ptr [si]
        mov         ah, al
        shr         ah,1
        shr         ah,1
        shr         ah,1
        shr         ah,1
        and         al, 0fh                       ;unpack byte
        xchg        al,ah                         ;high nibble first

        xlat
        mov         byte ptr[di],al
        inc         di
        xchg        al,ah                         ;now the lower nibble
        xlat
        mov         byte ptr [di],al
        inc         di

        dec         si
        loop        make_ascii

        sub         al, al
        mov         byte ptr [di],al

        ret

hexasc  endp
;
        end
```

APPENDIX F

TRANS.ASM and TABLE.ASM

TRANS.ASM

```
    .model small, c, os_dos

        include math.inc

;
       .data
inf              byte          "infinite", 0
zro              byte          "0.0",0


zero             qword      000000000000h
one_over_pi      qword      3ea2f9836e4eh
two_over_pi      qword      3f22f9836e4eh
half_pi          qword      3fc90fdaa221h
one_over_ln2     qword      3fb8aa3b295ch
ln2              qword      3f317217f7d1h
sqrt_half        qword      3f3504f30000h
expeps           qword      338000000001h
eps              qword      39fffff70000h
ymax             qword      45c90fdb0000h
big_x            qword      42a000000000h
littlex          qword      0c2a000000000h
y0a              qword      3ed5a9a80000h
y0b              qword      3f1714ba0000h
quarter          qword      3e8000000000h
circulark        qword      9b74eda7h
hyperk           qword      1351e8755h
```

```
plus                qword     3f800000000h
minus               qword     0bf800000000h


hundred             byte    64h
iten                word    0ah
maxchar             equ     8
;
        .code

;
;*****
;taylorsin - derives a sin by using a infinite series. this is in radians.
;expects argument in quadword format, expects to return the same
;input must be x^2<1
;
taylorsin           proc uses bx cx dx di si, argument:qword,  sine:word



       invoke polyeval, argument, sine, addr polysin, 10

       ret
taylorsin endp


; ******
;polyeval- evaluates polynomials according to Horner's rule
;expects to be passed a pointer to a table of coefficients,
;a number to evaluate, and the degree of the polynomial
;the argument conforms to the quadword fixedpoint format

polyeval            proc uses bx cx dx di si, argument:qword, output :word,
                    coeff:word, n:byte

       local        cf:qword,  result[8]:word

       pushf
       cld
       sub          ax, ax
```

```
        mov       cx, 4
        lea       di, word ptr cf
rep     stosw                                  ;clear the accumulator
        lea       di, word ptr result
        mov       cx,8
rep     stosw

eval:
        mov       si, word ptr coeff           ;point at table
        sub       bx, bx
        mov       bl, byte ptr n               ;point at coefficient of n-
                                               ;degree
                                               ;this is the beginning of our
                                               ;approximation
        shl       bx, 1
        shl       bx, 1
        shl       bx, 1                        ;multiply by eight for the
                                               ;quadword
        add       si, bx
        mov       ax, word ptr [si]
        mov       bx, word ptr [si][2]
        mov       cx, word ptr [si][4]
        mov       dx, word ptr [si][6]
        lea       di, word ptr cf
        add       word ptr [di], ax
        adc       word ptr [di][2], bx         ;add new coefficient to
                                               ;accumulator
        adc       word ptr [di][4], cx
        adc       word ptr [di][6], dx

        invoke    smul64, argument, cf, addr result

        lea       si, word ptr result [4]
        lea       di, word ptr cf
        mov       cx,4
rep     movsw

        dec       byte ptr n                   ;decrement pointer
        jns       eval
```

```
polyeval_exit:
        mov         di, word ptr output
        lea         si, word ptr cf
        mov         cx,4
rep     movsw                                       ;write to the output
        popf
        ret
polyeval endp

;
;log using a table and linear interpolation
;logarithms of negative numbers require imaginary numbers
;natural logs can be derived by multiplying result by 2.3025
;
lgl0    proc uses bx cx si di, argument:word, logptr:word

        local       powers_of_two:byte

        pushf
        std                                         ;increment down for zero check
                                                    ;to come

        sub         ax,ax
        mov         cx, 4
        mov         di, word ptr logptr             ;clear log output
        add         di, 6
rep     stosw

        mov         si, word ptr logptr             ;point at output which is zero
        add         si, 6                           ;most significant word
        mov         di, word ptr argument           ;point at input
        add         di, 6                           ;most significant word
        mov         ax, word ptr [di]
        or          ax, ax
        js          exit                            ;we don't do negatives

        sub         ax, ax
        mov         cx, 4
repe    cmpsw                                       ;find the first nonzero, or
                                                    ;return
```

```
                                                        ;zero
        je          exit

reposition_argument:                                    ;shift so msb is a one
        mov         si, word ptr argument               ;point at input
        add         si, 6                               ;most significant word
        mov         di, si                              ;shift the one eight times
        inc         cx                                  ;make this a one
        mov         ax, 4
        sub         ax, cx                              ;determine number of emptywords
        shl         ax, 1                               ;words to bytes
        sub         si, ax                              ;point to first nonzero word
        shl         ax, 1
        shl         ax, 1
        shl         ax, 1                               ;multiply by eight
        mov         bl, al
rep     movsw                                           ;shift

        mov         si, word ptr argument
        mov         ax, word ptr [si][6]
keep_shifting:
        or          ax, ax                              ;shift until msb is a one
        js          done_with_shift
        shl         word ptr [si][0], 1
        rcl         word ptr [si][2], 1
        rcl         word ptr [si][4], 1
        rcl         ax, 1
        inc         bl                                  ;count shifts as powers of two
        jmp         short keep-shifting                 ;normalize

done_with_shift
        mov         word ptr [si][6],ax                 ;ax will be a pointer
        mov         byte ptr powers_of_two, bl
        sub         bx, bx
        mov         bl, ah                              ;will point into 127 entry table
        shl         bl, 1                               ;get rid of top bit to form
                                                        ;actual pointer
        add         bx, offset word ptr logl0_tbl
                                                        ;linear interpolation
        mov         ax, word ptr [bx]                   ;get first approximation (floor)
```

```
        inc      bx
        inc      bx
        mov      bx, word ptr [bx]        ;and following approximation
                                          ;(ceil)
        sub      bx, ax                   ;find  difference
        xchg     ax, bx

        mul      byte ptr [si][6]         ;multiply by fraction bits
        mov      al, ah                   ;drop fractional places
        sub      ah, ah
        add      ax, bx                   ;add interpolated value to
                                          ;original

get_power:
        mov      bl, 31                   ;need to correct for power
                                          ;of two
        sub      bl, byte ptr powers_of_two
        sub      bh, bh
        shl      bx,1
        shl      bx,1                     ;point into this table
        lea      si, word ptr log10_power
        add      si, bx
        sub      dx, dx
        add      ax, word ptr [si]        ;add log of power
        adc      dx, word ptr [si][2]
        mov      di, word ptr logptr
        mov      word ptr [di][2],ax      ;write result to qword fixed
                                          ;point
        mov      word ptr [di][4],dx
        sub      cx,cx
        mov      word ptr [di],cx
        mov      word ptr [di][6],cx
exit:
        popf
        ret

lg10    endp


;
```

```
;sqrt using a table and linear interpolation
;this method has real problems as the powers increase
sqrtt proc uses bx cx si di, argument:word, sqrptr:word

        local       powers-of_two:byte

        pushf
        std                                     ;increment up

        sub         ax, ax
        mov         cx, 4
        mov         di, word ptr sqrptr         ;clear sqrt output
        add         di, 6
rep     stosw

        mov         si, word ptr sqrptr         ;clear sqrt output
        add         si, 6
        mov         di, word ptr argument       ;pointer to input
        add         di, 6
        mov         ax, word ptr [di]
        or          ax, ax
        js          exit                        ;we don't do negatives

        sub         ax, ax
        mov         cx, 4
repe    cmpsw                                   ;find the first nonzero, or
                                                ;return
                                                ;zero
        je          exit

reposition_argument:
        mov         si, word ptr argument
        add         si, 6
        mov         di, si                      ;shift the one eight times
        inc         cx                          ;this was a zero
        mov         ax, 4
        sub         ax, cx                      ;determine number of emptywords
        shl         ax, 1                       ;bytes to words
        sub         si, ax                      ;point to first nonzero word
        shl         ax, 1
```

```
        shl         ax, 1
        shl         ax, 1                       ;multiply by eight
        mov         bl, al
rep     movsw                                   ;shift
        mov         si, word ptr argument
        mov         ax, word ptr [si][6]
keep_shifting:
        or          ax, ax
        js          done_with_shift
        shl         word ptr [si][0], 1
        rcl         word ptr [si][2], 1
        rcl         word ptr [si][4], 1
        rcl         ax, 1
        inc         bl
        jmp         short keep_shifting         ;normalize

done_with_shift
        mov         word ptr [si][6],ax
        mov         byte ptr powers_of_two, bl
        sub         bx, bx
        mov         bl, ah
        shl         bl, 1
        add         bx, offset word ptr sqr_tbl
                                                 linear interpolation
        mov         ax, word ptr [bx]
        inc         bx
        inc         bx
        mov         bx, word ptr [bx]
        sub         bx, ax
        xchg        ax, bx

        mul         byte ptr [si][6]            ;multiply by fraction bits
        mov         al, ah                      ;factor out fractional places
        sub         ah, ah
        add         ax, bx                      ;add interpolated value to
                                                ;original

        mov         bl, byte ptr powers_of_two
        sub         bh, bh
        shl         bx,1
```

```
        lea          si, word ptr sqr_power
        add          si, bx
        sub          dx, dx
        mul          word ptr [si]              ;multiply by inverse of root
        mov          di, word ptr sqrptr
        mov          word ptr [di][2],ax
        mov          word ptr [di][4],dx
        sub          cx,cx
        mov          word ptr [di],cx
        mov          word ptr [di][6],cx
exit:
        popf
        ret

sqrtt  endp


;
;sines and cosines using a table and linear interpolation
;(degrees)

dcsin  proc  uses  bx  cx  si  di,  argument:word,  cs_ptr:word,  cs_flag:byte

        local        powers_of_two:byte,  sign:byte

        pushf
        std                                     ;increment  down

        sub          ax, ax
        mov          byte ptr sign, al          ;clear  sign  flag
        mov          cx,4
        mov          di, word ptr cs_ptr        ;clear sin/cos output
        add          di,6
rep     stosw
                                                ;first check arguments for zero
        add          di, 8                      ;reset  pointer
        mov          si, di
        mov          di, word ptr argument
        add          di, 6
        mov          cx, 4
```

```
repe    cmpsw                                   ;find the first nonzero, or
                                                ;return
        je          zero-exit
        jmp         prepare-arguments

zero_exit:
        cmp         byte ptr cs_flag, al        ;ax is zero
        jne         cos_0                       ;sin(0) = 0
        jmp         exit
cos_0:
        inc         ax
        inc         ax                          ;point di at base of output
        add         si,ax                       ;make ax a one
        dec         ax                          ;cos(0) = 1
        mov         word ptr [si][4],ax         ;one
        jmp         exit

prepare_arguments:
        mov         si, word ptr argument
        mov         ax, word ptr [si][4]        ;get integer portion of angle
        sub         dx, dx
        mov         cx, 360
        idiv        cx                          ;modular arithmetic to reduce
                                                ;angle
        or          dx, dx                      ;we want the remainder
        jns         quadrant
        add         dx, 360                     ;angle has gotta be positive for
                                                ;this
                                                ;to work
quadrant:
        mov         bx, dx                      ;we will use this to compute the
                                                ;value of the function
        mov         ax, dx                      ;put angle in ax
        sub         dx, dx
        mov         cx, 90
        div         cx                          ;and this to compute the sign
                                                ;ax holds an index to the
                                                ;quadrant
switch:
        cmp         byte ptr cs_flag, 0         ;what do we want
```

```
        je          do-sin


cos_range:
        cmp         ax, 0
        jg          cchk_180
        jmp         walk_up                     ;use incrementing method

cchk_180:
        cmp         ax, 1
        jg          cchk_270
        not         byte ptr sign               ;set sign flag
        neg         bx
        add         bx, 180
        jmp         walk_back                   ;use decrementing method

cchk_270:
        cmp         ax, 2
        jg          clast_90
        not         byte ptr sign               ;set sign flag
        sub         bx, 180
        jmp         walk-up

clast_90:
        neg         bx
        add         bx, 360
        jmp         walk_back



do_sin:                                         ;find the range of the argument
        cmp         ax, 0
        jg          schk_180
        neg         bx
        add         bx, 90
        jmp         walk_back                   ;use decrementing method

schk_180:
        cmp         ax, 1
        jg          schk_270
```

```
        sub     bx, 90
        jmp     walk_up                 ;use incrementing method

schk_270:
        cmp     ax, 2
        jg      slast_90
        not     byte ptr sign           ;set sign flag
        neg     bx
        add     bx, 270
        jmp     walk_back

slast_90:
        not     byte ptr sign           ;set sign flag
        sub     bx, 270
        jmp     walk_up
;
;
;
walk_up:
        shl     bx, 1                   ;use angle to point into the
                                        ;table
        add     bx, offset word ptr sine_tbl
        mov     dx, word ptr [bx]       ;get cos/sine of angle
        mov     ax, word ptr [si][2]    ;get fraction bits
        or      ax, ax
        je      write_result
                                        ;linear interpolation
        inc     bx                      ;get next approximation
        inc     bx
        mov     cx, dx
        mov     ax, word ptr [bx]
        sub     ax, dx                  ;find difference
        jnc     pos_res0
        neg     ax
        mul     word ptr [si][2]        ;multiply by fraction bits
        not     dx
        neg     ax
        jc      leave_walk_up
        inc     dx
```

```
        jmp         leave-walk-up
pos_res0:
        mul         word ptr [si][2]
leave_walk_up:
        add         dx, cx                      ;by fraction bits and addin
                                                ;angle
        jmp         write_result

walk_back:
        shl         bx, 1                       ;point into table
        add         bx, offset word ptr sine_tbl
        mov         dx, word ptr [bx]           ;get cos/sine of angle
        mov         ax, word ptr [si][2]        ;get fraction bits
        or          ax, ax
        je          write_result

        dec         bx
        dec         bx
        mov         cx, dx
        mov         ax, word ptr [bx]           ;get next incremental cos/sine
        sub         ax, dx                      ;get difference
        jnc         pos_res1
        neg         ax
        mul         word ptr [si][2]            ;multiply by fraction bits
        not         dx
        neg         ax
        jc          leave-walk-back
        inc         dx
        jmp         leave-walk-back
pos_res1:
        mul         word ptr [si][2]            ;multiply by fraction bits
leave_walk_back:
        add         dx, cx                      ;by fraction bits and add in
                                                ;angle

write_result:
        mov         di, word ptr cs_ptr
        mov         word ptr [di], ax           ;stuff result into variable
        mov         word ptr [di][2], dx        ;setup output for qword fixed
                                                ;point
```

```
        sub       ax, ax                    ;radix point between the double
                                            ;words
        mov       word ptr [di][4], ax
        mov       word ptr [di][6], ax
        cmp       byte ptr sign, al
        je        exit
        not       word ptr [di][6]
        not       word ptr [di][4]
        not       word ptr [di][2]
        neg       word ptr [di][0]
        jc        exit
        add       word ptr [di][2],1
        adc       word ptr [di][4],ax
        adc       word ptr [di][6],ax
exit:
        popf
        ret

dcsin endp



;
; ******
;
;gets exponent of floating point word
;
fr_xp proc        uses si di, fp0:dword, fp1:word, exptr:word

        local     flp0:qword, flp1:qword

        pushf
        cld

        xor       ax,ax
        lea       di,word ptr flp0
        mov       cx,4
rep     stosw

        lea       si,word ptr fp0
```

```
        lea        di,word ptr flp0[2]
        mov        cx,2
rep     movsw

        invoke     frxp, flp0, addr flp1, exptr

        lea        si,word ptr flp1[2]
        mov        di,word ptr fp1
        mov        cx,2
rep     movsw

        popf
        ret
fr_xp endp


;frxp performs an operation similar to the c function frexp. used
;for floating point math routines.
;returns the exponent -bias of a floating point number.
;it does not convert to floating point first, but expects a single
;precision number on the stack.



frxp    proc       uses di, float:qword, fraction:word, exptr:word

        pushf
        cld
        mov        di, word ptr exptr        ;assign pointer to exponent
        mov        ax, word ptr float[4]     ;get upper word of float
        mov        dx, word ptr float[2]
        sub        cx, cx
        or         cx, ax
        or         cx, dx
        je         make_it_zero              ;it is a zero
        shl        ax, 1
        rcl        cl, 1                     ;save the sign
        sub        ah, 7eh                   ;subtract bias to place float
                                             ;.5<=x<1
        mov        byte ptr [di],ah
```

```
        mov         ah, 7eh
        shr         cl, 1                   ;replace the sign
        rcr         ax, 1
        mov         word ptr float[4], ax
        mov         di, word ptr fraction
        lea         si, word ptr float
        mov         cx, 4
rep     movsw
frxp_exit:
        popf
        ret
make_it_zero:
        sub         ax, ax
        mov         byte ptr [di], al
        mov         di, word ptr fraction
rep     stosw
        jmp         frxp_exit
frxp    endp


; ******
;creates float from fraction and exponent
;
ld_xp proc          uses si di, fp0:dword, power:word, exp:byte

        local       flp0:qword,  result:qword

        pushf
        cld

        xor         ax,ax

        lea         di,word ptr flp0
        mov         cx,4
rep     stosw

        lea         si,word ptr fp0
        lea         di,word ptr flp0[2]
        mov         cx,2
rep     movsw
```

```
        invoke      ldxp, flp0, addr result, exp

        lea         si,word ptr result[2]
        mov         di,word ptr power
        mov         cx,2
rep     movsw

        popf
        ret
ld_xp   endp


;ldxp is similar to ldexp in c, it is used for math functions
;takes from the stack, an input float(extended and returns a pointer to
;a value to
;the power of two
;passed with it.


ldxp proc       uses di, float:qword, power:word, exp:byte

        mov         ax, word ptr float[4]       ;get upper word of float
        mov         dx, word ptr float[2]       ;extended bits are not checked
        sub         cx, cx
        or          cx, ax
        or          cx, dx
        je          return_zero
        shl         ax, 1                       ;save the sign
        rcl         cl, 1
        mov         ah, 7eh
        add         ah, byte ptr exp
        jc          ld_overflow

        shr         cl, 1                       ;return the sign
        rcr         word ptr ax, 1              ;position exponent
        mov         word ptr float[4], ax

ldxp_exit:
```

```
        mov         cx, 4
        mov         di, word ptr power
        lea         si, word ptr float
rep     movsw
        ret

        ret

ld_overflow:
        mov         word ptr float[4], 7f80h
        sub         ax, ax
        mov         word ptr float[2], ax
        mov         word ptr float[0], ax
        jmp         ldxp_exit

return_zero:
        sub         ax, ax
        mov         di, word ptr power
        mov         cx, 4
rep     stosw
        jmp         ldxp_exit
ldxp    endp


;
; ******
;
; FX_SQR
;accepts integers.
;Remember that the powers follow the powers of two, i.e., the root of a double
word
;is a word, the root of a word is a byte, the root of a byte is a nibble, etc.
;new_estimate = (radicand/last_estimate+last_estimate)/2,last_estimate=
new_estimate.

fx_sqr proc uses bx cx dx di si, radicand:dword, root:word

        local       estimate:word, cntr:byte

                    byte ptr cntr, 16
                    bx, bx                          ;to test radicand
```

```
        mov         ax, word ptr radicand
        mov         dx, word ptr radicand[2]
        or          dx, dx
        js          sign_exit
        je          zero_exit
        jmp         find_root                   ;not zero
zero_exit:
        or          ax, ax                      ;no negatives or zeros
        jne         find_root
sign_exit:                                      ;indicate error in the operation
        stc
        sub         ax, ax
        mov         dx, ax
        jmp         root_exit

find_root:
        sub         byte ptr cntr, 1
        jc          root-exit                   ;will exit with carry set and an
                                                ;approximate  root
find_root1:
        or          dx, dx                      ;must be zero
        je          fits                        ;some kind of estimate
        shr         dx, 1
        rcr         ax, 1
        jmp         find_root1                  ;cannot have a root greater
                                                ;than 16 bits foe
                                                ;a 32 bit radicand!
fits:
        mov         word ptr estimate, ax       ;store first estimate of root
        sub         dx, dx
        mov         ax, word ptr radicand[2]
        div         word ptr estimate
        mov         bx, ax                      ;save quotient from division of
                                                ;upperword
        mov         ax, word ptr radicand
        div         word ptr estimate           ;divide lower word
        mov         dx, bx                      ;concatenate quotients

        add         ax, word ptr estimate       ;(radicand/estimate+estimate)/
                                                ;2
```

```
        adc         dx, 0
        shr         dx, 1
        rcr         ax, 1

        or          dx, dx                   ;to prevent any modular aliasing
        jne         find_root
        cmp         ax, word ptr estimate    ;is the estimate still changing?
        jne         find_root
        clc                                  ;clear the carry to indicate
                                             ;success

root_exit:
        mov         di, word ptr root
        mov         word ptr [di], ax
        mov         word ptr [di][2], dx
        ret
fx_sqr  endp



;
; ******
  school_sqr
;accepts integers
school_sqr          proc uses bx cx dx di si, radicand:dword, root:word

        local       estimate:qword, bits:byte

        sub         bx, bx
        mov         ax, word ptr radicand
        mov         dx, word ptr radicand[2]
        or          dx, dx
        js          sign_exit
        je          zero_exit
        jmp         setup                    ;not zero
zero_exit:
        or          ax, ax                   ;no negatives or zeros
        jne         setup
sign_exit:                                   ;indicate error in the operation
        sub         ax, ax                   ;can't do negatives
```

```
        mov         dx, ax                          ;zero for fail
        stc
        jmp         root_exit

setup:
        mov         byte ptr bits, 16
        mov         word ptr estimate, ax
        mov         word ptr estimate[2], dx
        sub         ax, ax
        mov         word ptr estimate[4], ax
        mov         word ptr estimate[6], ax
        mov         bx, ax                          ;root
        mov         cx, ax
        mov         dx, ax                          ;intermediate

findroot:
        shl         word ptr estimate, 1
        rcl         word ptr estimate[2], 1
        rcl         word ptr estimate[4], 1
        rcl         word ptr estimate[6], 1

        shl         word ptr estimate, 1
        rcl         word ptr estimate[2], 1
        rcl         word ptr estimate[4], 1
        rcl         word ptr estimate[6], 1         ;double shift radicand

        shl         ax, 1
        rcl         bx, 1                           ;shift root

        mov         cx, ax
        mov         dx, bx
        shl         cx, 1
        rcl         dx, 1                           ;root*2
        add         cx, 1
        adc         dx, 0                           ;+l

subtract_root:
        sub         word ptr estimate[4], cx        ;accumulator-2*root+l
        sbb         word ptr estimate[6], dx
        jnc         r_plus_one
```

```
	add	word ptr estimate[4], cx
	adc	word ptr estimate[6], dx
	jmp	continue_loop
r-plus-one:
	add	ax, 1
	adc	bx, 0					;r+=l
continue_loop:
	dec	byte ptr bits
	jne	findroot
	clc

root_exit:
	mov	di, word ptr root
	mov	word ptr [di], ax
	mov	word ptr [di][2], bx

	ret
school_sqr	endp



; ******
;fp-cos


fp_cos proc uses si di, fp0:dword, fp1:word

	local	flp0:qword, result:qword, sign:byte

	pushf
	cld
	xor	ax,ax
	lea	di,word ptr flp0
	mov	cx,4
rep stosw

	lea	si,word ptr fp0
	lea	di,word ptr flp0[2]
	mov	cx,2
rep movsw
```

```
    sub         al, al
    mov         byte ptr sign, al

    invoke      fladd, flp0, half_pi, addr flp0
    mov         ax, word ptr flp0[4]
    or          ax, ax
    jns         positive
    not         byte ptr sign                   ;is it less than zero?
                                                ;positive:


    invoke      flsin, flp0, addr result, sign

    mov         ax, word ptr result[2]
    mov         dx, word ptr result[4]
    mov         di, word ptr fp1
    mov         word ptr [di], ax
    mov         word ptr [di][2], dx

    popf
    ret
fp_cos  endp

;
;******
;fp_sin
;
;
fp_sin proc uses si di, fp0:dword, fp1:word

    local       flp0:qword, result:qword, sign:byte

    pushf
    cld
    xor         ax,ax
    lea         di,word ptr flp0
    mov         cx,4
rep stosw
```

```
        lea          si,word ptr fp0
        lea          di,word ptr flp0[2]
        mov          cx,2
    rep movsw

        sub          al, al
        mov          byte ptr sign, al
        mov          ax, word ptr flp0[4]
        or           ax, ax
        jns          positive
        not          byte ptr sign                  ;is it less than zero?
                                                    ;positive:

        invoke        flsin, flp0, addr result, sign

        invoke       round, result, addr result

        mov          ax, word ptr result[2]
        mov          dx, word ptr result[4]
        mov          di, word ptr fp1
        mov          word ptr [di], ax
        mov          word ptr [di][2], dx

        popf
        ret
fp_sinendp


;
;******
;flsin
;

flsin proc           uses bx cx dx si di, fp0:qword, fp1:word, sign:byte

        local        result:qword, temp0:qword, temp1:qword,
                     y:qword, u:qword

        pushf
        cld
```

```
        invoke      flcomp, fp0, ymax               ;error, entry value too
                                                    ;large
        cmp         ax, 1
        jl          absx

error_exit:
        lea         di, word ptr result
        sub         ax, ax
        mov         cx, 4
rep     stosw
        jmp         writeout

absx:
        mov         ax, word ptr fp0[4]             ;make  absolute
        or          ax, ax
        jns         deconstruct_exponent
        and         ax, 7fffh
        mov         word ptr fp0[4], ax

deconstruct_exponent:
        invoke      flmul, fp0, one_over_pi, addr result
                                                    ;x/pi

        invoke      intrnd, result, addr temp0
                                                    ;intrnd(x/pi)

        mov         ax, word ptr temp0[2]           ;determine  if  integerhas
                                                    ;odd or even
        mov         dx,word ptr temp0[4]            ;number  of  bits
        mov         cx, dx
        and         cx, 7f80h                       ;get rid of sign and
                                                    ;mantissa portion
        shl         cx, 1
        mov         cl, ch
        sub         ch, ch
        sub         cl, 7fh                         ;subtract bias (-1) from
                                                    ;exponent
        js          not-odd
```

```
        inc         cl
        or          cl, cl
        je          xpi
extract_int:
        shl         ax, 1
        rcl         dx, 1
        rcl         word ptr bx, 1
        loop        extract_int                 ;position as fixedpoint
        test        dh, 1
        je          xpi
        not         byte ptr sign
not_odd:


xpi:
                                                ;extended precision multiply
                                                ;by pi
        invoke      flmul, sincos[8*0], temp0, addr result
                                                ;intrnd(x/pi)*c1

        invoke      flsub, fp0, result, addr result
                                                ;|x|-intrnd(x/pi)

        invoke      flmul, temp0, sincos[8*1], addr temp1
                                                ;intrnd(x/pi)*c2

        invoke      flsub, result, temp1, addry
                                                ;y

chk_eps:
        invoke      flabs, y, addr temp0        ;is the argument less than eps?
        invoke      flcomp, temp0, eps
        or          ax, ax
        jns         r_g
        lea         di, word ptr result
        sub         ax, ax
        mov         cx, 4
rep     stosw
        jmp         writeout
```

```
r_g:

       invoke       flmul, y, y, addr u
                                                          ;evaluater(g)
                                                          ;((r4*g+r3)*g+r2)*g+rl)*g
       invoke       flmul, u, sincos[8*5], addr result
       invoke       fladd, sincos[8*4], result, addr result

       invoke       flmul, u, result, addr result
       invoke       fladd, sincos[8*3], result, addr result

       invoke       flmul, u, result, addr result
       invoke       fladd, sincos[8*2], result, addr result

       invoke       flmul, u, result, addr result
                                                          ;result == z

fxr:
       invoke       flmul, result, y, addr result

       invoke       fladd, result, y, addr result
                                                          ;r*r+f

handle_sign:
       cmp          byte ptr sign, -1
       jne          writeout
       xor          word ptr result[4], 8000h
                                                          ;result * sign
writeout:
       mov          di, word ptr fp1
       lea          si, word ptr result
       mov          cx, 4
rep    movsw


flsin_exit:


       popf
       ret
flsin  endp
```

```
;
; ******
;
; fp_tan
;
;
fp_tan  proc        uses si di, fp0:dword, fp1:word

        local       flp0:qword,  result:qword

        pushf
        cld
        xor         ax,ax
        lea         di,word ptr flp0
        mov         cx,4
rep     stosw

        lea         si,word ptr fp0
        lea         di,word ptr flp0[2]
        mov         cx,2
rep     movsw


        invoke      fltancot, flp0, addr result

        mov         ax, word ptr result[2]
        mov         dx, word ptr result[4]
        mov         di, word ptr fp1
        mov         word ptr [di], ax
        mov         word ptr [di][2], dx

        popf
        ret
fp_tanendp
;
;******
;
;fltancot
```

```
fltancot  proc       uses bx cx dx si di, fp0:qword, fp1:word

        local        flp0:qword, result:qword, temp0:qword, temp1:qword,
                     sign:byte, xnum:qword, xden:qword, xn:qword, f:qword,
                     g:qword, fxpg:qword, qg:qword

        pushf
        cld

        sub          ax, ax
        mov          byte ptr sign, al            ;clear the sign flag

        lea          di, word ptr g
        mov          cx, 4
rep     stosw                                     ;place input argument in
                                                  ;variable

        lea          di, word ptr f
        mov          cx, 4
rep     stosw                                     ;place input argument in
                                                  ;variable

        shl          word ptr fp0[4], 1
        rcl          byte ptr sign, 1
        shr          word ptr fp0[4], 1           ;absolute value for comparison

        invoke       flcomp, fp0, ymax            ;error,entry value too large
        cmp          ax, 1
        jl           continue
        lea          di, word ptr result
        sub          ax, ax
        mov          cx, 4
rep     stosw
        jmp          fltancot_exit

continue:
        shl          word ptr fp0[4], 1
        shr          byte ptr sign, 1
        rcr          word ptr fp0[4], 1           ;restore sign
```

```
        invoke      flmul, fp0, two-over-pi, addr result
                                                ;x*2/pi

        invoke      intrnd, result, addr xn
                                                ;intmd(x*2/pi)

        mov         ax, word ptr xn[2]          ;determine if integer has odd
                                                ;or even
        mov         dx, word ptr xn[4]          ;number of bits
        mov         cx, dx
        and         cx, 7f80h                   ;get rid of sign and
                                                ;mantissa portion
        shl         cx, 1
        mov         cl, ch
        sub         ch, ch
        or          cl, cl
        je          not-odd
        sub         cl, 7fh                     ;subtract bias (-1) from
                                                ;exponent
        js          not-odd
        inc         cl
        or          cl, cl
        je          not-odd
        and         dx, 7fh
        or          dx, 80h                     ;restore hidden bit
extract_int:
        shl         ax, 1
        rcl         dx, 1
        rcl         word ptr bx, 1
        loop        extract_int                 ;position as fixedpoint
        test        dh, 1
        je          not_odd
        mov         byte ptr sign, -1
not_odd:

        invoke      flmul, xn, tancot[8*0], addr temp0
        invoke      flsub, fp0, temp0, addr temp0
                                                ;(x-xn*c1)

        invoke      flmul, xn, tancot[8*1], addr temp1
```

```
                                                ;xn*c2

        invoke      flsub, temp0, temp1, addr f
                                                ;(x-xn*c1)-xn*c2


        invoke      flabs, f, addr temp1        ;|f|<eps?
        invoke      flcomp, temp1, eps
        or          ax, ax
        jns         compute
        lea         si, word ptr f              ;f->xnum
        lea         di, word ptr xnum
        mov         cx, 4
rep     movsw
        lea         si, word ptr one            ;1.0->xden
        lea         di, word ptr xden
        mov         cx, 4
rep     movsw
        jmp         compute-result

compute:
        invoke      flmul, f, f, addr g         ;f*f->g

        invoke      flmul, g, tancot[8*3], addr temp0
        invoke      flmul, f, temp0, addr temp0
        invoke      fladd, temp0, f, addr fxpg
                                                ;fxpg=(p2*g+pl)*g*f
                                                ;+ f

        invoke      flmul, g, tancot[8*6], addr temp0
        invoke      fladd, temp0, tancot[8*5], addr temp0
        invoke      flmul, g, temp0, addr temp0
        invoke      fladd,temp0,tancot[8*4], addr qg
                                                ;qg = (q2 * g + q1) * g +q0

        lea         si, word ptr fxpg
        lea         di, word ptr xnum
        mov         cx, 4
rep     movsw
        lea         si, word ptr qg
```

```
        lea         di, word ptr xden
        mov         cx, 4
rep     movsw

compute_result:
        mov         al, byte ptr sign           ;even or odd
        or          al, al
        je          xnum_xden
        xor         word ptr xnum[4],8000h      ;make it negative
        jmp         short xden_xnum

xden_xnum:
        invoke      fldiv, xden, xnum, fp1
        jmp         fltancot_exit

xnum_xden:
        invoke      fldiv, xnum, xden, fpl

fltancot_exit:
        popf
        ret
fltancot endp




;
;******
;fp_sqr


fp_sqr  proc        uses si di, fp0:dword, fp1:word

        local       flp0:qword, result:qword

        pushf
        cld
        xor         ax,ax
        lea         di,word ptr flp0
```

```
        mov         cx,4
        stosw

        lea         si,word ptr fp0
        lea         di,word ptr flp0[2]
        mov         cx,2
        movsw


        invoke      flsqr, flp0, addr result

        invoke      round, result, addr result

        mov         ax, word ptr result[2]
        mov         dx, word ptr result[4]
        mov         di, word ptr fp1
        mov         word ptr [di], ax
        mov         word ptr [di][2], dx

        popf
        ret
fp_sqr  endp



;
; ******
; flsqr


flsqr  proc         uses bx cx dx si di, fp0:qword, fp1:word

        local       result:qword, temp0:qword, temp1:qword, exp:byte,
                    xn:qword, f:qword, y0:qword, m:byte

        pushf
        cld


        lea         di, word ptr xn
```

```
        sub       ax, ax
        mov       cx, 4
rep     stosw

        invoke    flcomp, fp0, zero         ;error, entry value too large
        cmp       ax, 1
        je        ok
        cmp       ax, 0
        je        got-result
        mov       di, word ptr fp1
        sub       ax, ax
        mov       cx, 4
rep     stosw
        not       ax
        and       ax, 7f80h
        mov       word ptr result[4],ax     ;make it plus infinity
        jmp       flsqr_exit

got_result:
        mov       di, word ptr fp1
        sub       ax, ax
        mov       cx, 4
rep     stosw
        jmp       flsqr_exit

ok:
        invoke    frxp, fp0, addr f, addr exp ;get exponent


        invoke    flmul, f, y0b, addr temp0
        invoke    fladd, temp0, y0a, addry


heron:                                      ;two passes through
        invoke    fldiv, f, y0, addr temp0  ;(x/r+r)/2 is all we need
        invoke    fladd, y0, temp0, addr temp0
        mov       ax, word ptr temp0[4]
        shl       ax, 1
        sub       ah, 1                     ;should always be safe
        shr       ax, 1
        mov       word ptr temp0[4], ax     ;subtracts one half by
```

```
                                                ;decrementing the exponent
                                                ;one
        invoke     fldiv, f, temp0, addr temp1
        invoke     fladd, temp0, temp1, addr temp0
        mov        ax, word ptr temp0[4]
        shl        ax, 1
        sub        ah, 1                        ;should always be safe
        shr        ax, 1
        mov        word ptr y0[4], ax           ;subtracts one half by
        mov        ax, word ptr temp0[2]        ;decrementing the exponent
                                                ;one
        mov        word ptr y0[2], ax
        mov        ax, word ptr temp0
        mov        word ptr y0, ax
        sub        ax, ax
        mov        word ptr y0[6], ax

chk_n:
        mov        al, byte ptr exp
        mov        cl, al
        sar        al, 1                        ;arithmetic  shift
        jnc        evn
odd:
        invoke     flmul, y0, sqrt_half, addr y0     ;adjustment for uneven
                                                ;exponent

        mov        al, cl
        inc        al
        sar        al, 1
evn:
        mov        cl, al                       ;n/2->m

power:
        mov        ax, word ptr y0[4]
        shl        ax, 1
        add        ah, cl

write_result:
        shr        ax, 1
        mov        word ptr y0[4], ax
```

```
        lea        si, word ptr y0
        mov        di, word ptr fp1
        mov        cx, 4
rep     movsw

flsqr_exit:
        popf
        ret

flsqr  endp

;
;******
;
;lgb - log to base 2
;input argument must be be l<= x < 2
;multiply the result by .301029995664 (4d104d42h) to convert to base 10
;higher powers of 2 can be derivedby counting the number of shifts required
;to bring the number between 1 and 2, calculating that lgb then adding, as the
;integer portion, the number of shifts as that is the power of the number.

lgb     proc       uses bx cx dx di si, argument:qword, result:word


        local      k:byte, z:qword

        mov        di,word ptr result
        sub        ax,ax
        mov        cx, 4
rep     stosw                                  ;make y zero
        inc        al
        mov        byte ptr k, al ;make k == 1

        mov        ax, word ptr argument
        mov        bx, word ptr argument[2]
        mov        dx, word ptr argument[4]

        shr        dx, 1                       ;z=argument/2
        rcr        bx, 1
        rcr        ax, 1                       ;scale argument for z
```

```
        lea         di, word ptr z
        mov         word ptr [di], ax
        mov         word ptr [di][2], bx
        mov         word ptr [di][4], dx

  xl:
        mov         ax, word ptr argument
        mov         bx, word ptr argument[2]
        mov         dx, word ptr argument [4]     ;argument between 1.0 and 2.0

        sub         cx, cx
        cmp         ax, cx                        ;test for 1.0
        jne         not_done_yet
        cmp         bx, ax
        jne         not_done_yet
        inc         cx
        cmp         dx, ax
        jne         not_done_yet

        jmp         logb_exit

not_done_yet:
        sub         ax, word ptr z                ;x-z<l?
        sbb         bx, word ptr z[2]
        jc          shift

reduce:
        mov         word ptr argument, ax         :x<-x-z
        mov         word ptr argument[2], bx
        mov         word ptr argument[4], dx

        sub         cx, cx
        mov         cl, byte ptr k

shiftk:
        shr         dx, 1
        rcr         bx, 1
        rcr         ax, 1
        loop        shiftk
        mov         word ptr z, ax                ;z<-argument<<k
```

```
        mov         word ptr z[2], bx
        mov         word ptr z[4], dx

        sub         bx, bx
        mov         bl, byte ptr k
        cmp         bl, 20
        ja          logb_exit

        dec         bl
        shl         bx, 1
        shl         bx, 1
        shl         bx, 1                   ;point into table of qwords

        lea         si, word ptr log2
        mov         ax, word ptr [si][bx]
        mov         cx, word ptr [si][bx][2]
        mov         dx, word ptr [si][bx][4]   ;get log of power
        mov         di, word ptr result
        add         word ptr [di], ax
        adc         word ptr [di][2], cx
        adc         word ptr [di][4], dx
        jmp         xl

shift:
        shr         word ptr z[4], 1
        rcr         word ptr z[2], 1
        rcr         word ptr z, 1

        inc         byte ptr k

        jmp         xl


logb_exit:
        ret
lgb     endp


;
; ******
;
;pwrb - base 10    to power
```

```
;input argument must be be l<= x <2
pwrb    proc        uses bx cx dx di si, argument:qword, result:word


        local       k:byte, z:qword

        mov         di, word ptr result         ;y
        sub         ax, ax
        mov         cx, 2
rep     stosw                                   ;make y one
        inc         ax
        stosw
        dec         ax
        stosw
        mov         byte ptr k, al              ;make k = 0
x0:
        mov         ax, word ptr argument
        mov         cx, word ptr argument[2]
        mov         dx, word ptr argument[4]    ;argument 0<= x < 1

        sub         bx, bx
        cmp         ax, bx                      ;testfor 0.0
        jne         not_done_yet
        cmp         cx, bx
        jne         not_done_yet
        cmp         dx, bx
        jne         not_done_yet

        jmp         pwrb_exit

not_done_yet
        sub         bx, bx
        mov         bl, byte ptr k
        cmp         bl, 20h
        ja          pwrb_exit

        shl         bx, 1
        shl         bx, 1
        shl         bx, 1                       ;point into table of qwords

        lea         si, word ptr power10
```

```
        cmp         dx, word ptr [si][bx][4]
        jb          increase
        ja          reduce
        cmp         cx, word ptr [si][bx][2]
        jb          increase
        ja          reduce
        cmp         ax, word ptr [si][bx]
        jb          increase

reduce:
        sub         ax, word ptr [si][bx]
        sbb         cx, word ptr [si][bx][2]
        sbb         dx, word ptr [si][bx][4]
        mov         word ptr argument, ax           ;x<-x-z
        mov         word ptr argument[2], cx
        mov         word ptr argument[4], dx

        sub         cx, cx
        mov         cl, byte ptr k

        mov         si, word ptr result
        mov         ax, word ptr [si]
        mov         bx, word ptr [si][2]
        mov         dx, word ptr [si][4]
        cmp         cl, 0
        je          no_shiftk
shiftk:
        shr         dx, 1
        rcr         bx, 1
        rcr         ax, 1
        loop        shiftk
no_shiftk:
        add         word ptr [si], ax               ;z<-argument<<k
        adc         word ptr [si][2], bx
        adc         word ptr [si][4], dx

        jmp         x0

increase:
```

```
        inc         byte ptr k
        jmp         x0


pwrb_exit:
        ret
pwb     endp



;******
;circular- implementation of the circular routine, a subset of the CORDIC devices
;

;
circular            proc uses bx cx dx di si, x:word, y:word, z:word

        local       smallx:qword, smally:qword, smallz:qword, i:byte,
                    shifter:word

        lea         di, word ptr smallx
        mov         si, word ptr x
        mov         cx, 4
rep     movsw

        lea         di, word ptr smally
        mov         si, word ptry
        mov         cx, 4
rep     movsw

        lea         di, word ptr smallz
        mov         si, word ptr z
        mov         cx, 4
rep     movsw


        sub         ax, ax
        mov         byte ptr i, al              ;i=0
```

```
        mov         bx, ax
        mov         cx, ax

twist:
        sub         ax, ax
        mov         al, i
        mov         word ptr shifter, ax

        mov         si, word ptr x                ;multiply by 2ˆ-i
        mov         ax, word ptr [si]
        mov         bx, word ptr [si][2]
        mov         cx, word ptr [si][4]
        mov         dx, word ptr [si][6]

        cmp         word ptr shifter, 0
        je          load_smallx
shiftx:
        sar         dx, 1                         ;note the arithmetic shift
        rcr         cx, 1                         ;for sign extension
        rcr         bx, 1
        rcr         ax, 1
        dec         word ptr shifter
        jnz         shiftx
load_smallx:
        mov         word ptr smallx, ax
        mov         word ptr smallx[2], bx
        mov         word ptr smallx[4], cx
        mov         word ptr smallx[6], dx        ;x=x>>i

        sub         ax, ax
        mov         al, i
        mov         word ptr shifter, ax

        mov         si, word ptr y
        mov         ax, word ptr [si]
        mov         bx, word ptr [si][2]
        mov         cx, word ptr [si][4]
        mov         dx, word ptr [si][6]

        cmp         word ptr shifter, 0           ;multiply by 2^-i
```

```
        je          load_smally
shifty:
        sar         dx, 1                       ;note the arithmetic shift
        rcr         cx, 1                       ;for sign extension
        rcr         bx, 1
        rcr         ax, 1
        dec         word ptr shifter
        jnz         shifty
load_smally:
        mov         word ptr smally, ax
        mov         word ptr smally[2], bx
        mov         word ptr smally[4], cx
        mov         word ptr smally[6], dx      ;y=Y>>i

get_atan:
        sub         bx, bx
        mov         bl, i
        shl         bx, 1
        shl         bx, 1                       ;got to point into a dword table
        lea         si, word ptr atan_array
        mov         ax, word ptr [si][bx]
        mov         dx, word ptr [si][bx] [2]

        mov         word ptr smallz, ax
        mov         word ptr smallz[2], dx      ;z=atan[i]
        sub         ax, ax
        mov         word ptr smallz[4], ax
        mov         word ptr smallz[6], ax

test_Z:
        mov         si, word ptr z
        mov         ax, word ptr [si][6]
        or          ax, ax
        jns         positive

negative:
        mov         ax, word ptr smally
        mov         bx, word ptr smally[2]
        mov         cx, word ptr smally[4]
        mov         dx, word ptr smally[6]
```

```
        mov     di, word ptr x
        add     word ptr [di], ax
        adc     word ptr [di][2], bx
        adc     word ptr [di][4], cx
        adc     word ptr [di][6], dx            ;x += y

        mov     ax, word ptr smallx
        mov     bx, word ptr smallx[2]
        mov     cx, word ptr smallx[4]
        mov     dx, word ptr smallx[6]

        mov     di, word ptr y
        sub     word ptr [di], ax
        sbb     word ptr [di][2], bx
        sbb     word ptr [di][4], cx
        sbb     word ptr [di][6], dx            ;Y -= x

        mov     ax, word ptr smallz
        mov     bx, word ptr smallz[2]
        mov     cx, word ptr smallz[4]
        mov     dx, word ptr smallz[6]

        mov     di, word ptr z
        add     word ptr [di], ax
        adc     word ptr [di][2], bx
        adc     word ptr [di][4], cx
        adc     word ptr [di][6], dx            ;x += y


        jmp     for_next

positive:
        mov     ax, word ptr smally
        mov     bx, word ptr smally [2]
        mov     cx, word ptr smally[4]
        mov     dx, word ptr smally[6]
        mov     di, word ptr x
        sub     word ptr [di], ax
        sbb     word ptr [di][2], bx
```

```
        sbb         word ptr [di][4], cx
        sbb         word ptr [di][6], dx            ;x -= y

        mov         ax, word ptr smallx
        mov         bx, word ptr smallx[2]
        mov         cx, word ptr smallx[4]
        mov         dx, word ptr smallx[6]
        mov         di, word ptr y
        add         word ptr [di], ax
        adc         word ptr [di][2], bx
        adc         word ptr [di][4], cx
        adc         word ptr [dil[6], dx            ;Y += x

        mov         ax, word ptr smallz
        mov         bx, word ptr smallz[2]
        mov         cx, word ptr smallz[4]
        mov         dx, word ptr smallz[6]
        mov         di, word ptr z
        sub         word ptr [di], ax
        sbb         word ptr [di][2], bx
        sbb         word ptr [di][4], cx
        sbb         word ptr [di][6], dx            ;x -= y

for_next:
        inc         byte ptr i                      ;bump  exponent
        cmp         byte ptr i, 32
        ja          circular-exit
        jmp         twist

circular-exit


        ret
circular        endp

;
;******
;icirc- implementation of the inverse circular routine, a subset of the cordic
;devices
;
```

```
;
;
icirc proc uses bx cx dx di si, x:word, y:word, z:word

        local       smallx:qword, smally:qword, smallz:qword, i:byte,
                    shifter:word

        lea         di, word ptr smallx
        mov         si, word ptr x
        mov         cx, 4
rep     movsw

        lea         di, word ptr smally
        mov         si, word ptry
        mov         cx, 4
rep     movsw

        lea         di, word ptr smallz
        mov         si, word ptr z
        mov         cx, 4
rep     movsw


        sub         ax, ax
        mov         byte ptr i, al ;i=0
        mov         bx, ax
        mov         cx, ax

twist:
        sub         ax, ax
        mov         al, i
        mov         word ptr shifter, ax

        mov         si, word ptr x                      ;multiply by2^-i
        mov         ax, word ptr [si]
        mov         bx, word ptr [si][2]
        mov         cx, word ptr [si][4]
        mov         dx, word ptr [si][6]

        cmp         word ptr shifter, 0
```

```
        je          load_smallx
shiftx:
        sar         dx, 1
        rcr         cx, 1
        rcr         bx, 1
        rcr         ax, 1
        dec         word ptr shifter
        jnz         shiftx
load_smallx:
        mov         word ptr smallx, ax
        mov         word ptr smallx[2], bx
        mov         word ptr smallx[4], cx
        mov         word ptr smallx[6], dx           ;x=X>>i

        sub         ax, ax
        mov         al, i
        mov         word ptr shifter, ax

        mov         si, word ptry
        mov         ax, word ptr [si]
        mov         bx, word ptr [si][2]
        mov         cx, word ptr [si][4]
        mov         dx, word ptr [si][6]

        cmp         word ptr shifter, 0
        je          load_smally
shifty:
        sar         dx, 1
        rcr         cx, 1
        rcr         bx, 1
        rcr         ax, 1
        dec         word ptr shifter
        jnz         shifty
load_smally:
        mov         word ptr smally, ax
        mov         word ptr smally[2], bx
        mov         word ptr smally[4], cx
        mov         word ptr smally[6], dx           ;y=Y>>i

get_atan:
```

```
        sub         bx, bx
        mov         bl, i
        shl         bx, 1
        shl         bx, 1                       ;got to point into a dword table
        lea         si, word ptr atan_array
        mov         ax, word ptr [si][bx]
        mov         dx, word ptr [si][bx][2]

        mov         word ptr smallz, ax
        mov         word ptr smallz[2], dx      ;z=atan[i]
        sub         ax, ax
        mov         word ptr smallz[4], ax
        mov         word ptr smallz[6], ax

test_Y:
        mov         si, word ptr y
        mov         ax, word ptr [si][6]
        or          ax, ax
        js          positive

negative:
        mov         ax, word ptr smally
        mov         bx, word ptr smally[2]
        mov         cx, word ptr smally[4]
        mov         dx, word ptr smally[6]

        mov         di, word ptr x
        add         word ptr [di], ax
        adc         word ptr [di][2], bx
        adc         word ptr [di][4], cx
        adc         word ptr [di][6], dx        ;x += y

        mov         ax, word ptr smallx
        mov         bx, word ptr smallx[2]
        mov         cx, word ptr smallx[4]
        mov         dx, word ptr smallx[6]

        mov         di, word ptr y
        sub         word ptr [di], ax
        sbb         word ptr [di][2], bx
```

```
        sbb     word ptr [di][4], cx
        sbb     word ptr [di][6], dx        ;Y -= x

        mov     ax, word ptr smallz
        mov     bx, word ptr smallz[2]
        mov     cx, word ptr smallz[4]
        mov     dx, word ptr smallz[6]

        mov     di, word ptr z
        add     word ptr [di], ax
        adc     word ptr [di][2], bx
        adc     word ptr [di][4], cx
        adc     word ptr [di][6], dx        -x += y


        jmp     for_next

positive:
        mov     ax, word ptr smally
        mov     bx, word ptr smally[2]
        mov     cx, word ptr smally[4]
        mov     dx, word ptr smally[6]
        mov     di, word ptr x
        sub     word ptr [di], ax
        sbb     word ptr [dil[2], bx
        sbb     word ptr [di][4], cx
        sbb     word ptr [di][6], dx        ;x -= y

        mov     ax, word ptr smallx
        mov     bx, word ptr smallx[2]
        mov     cx, word ptr smallx[4]
        mov     dx, word ptr smallx[6]
        mov     di, word ptr y
        add     word ptr [di], ax
        adc     word ptr [di][2], bx
        adc     word ptr [di][4], cx
        adc     word ptr [di][6], dx        ;Y += x

        mov     ax, word ptr smallz
        mov     bx, word ptr smallz[2]
```

```
        mov         cx, word ptr smallz[4]
        mov         dx, word ptr smallz[6]
        mov         di, word ptr z
        sub         word ptr [di], ax
        sbb         word ptr [di][2], bx
        sbb         word ptr [di][4], cx
        sbb         word ptr [di][6], dx        ;x -= y

for_next:
        inc         byte ptr i
        cmp         byte ptr i, 32
        ja          icircular_exit
        jmp         twist

icircular_exit:
        ret
icirc endp




; ******
;hyper- implementation of the hyperbolic routine, a subset of the cordic devices


;
hyper proc uses bx cx dx di si, x:word, y:word, z:word

        local       smallx:qword, smally:qword, smallz:qword, i:byte,
                    shifter:word

        lea         di, word ptr smallx
        mov         si, word ptr x
        mov         cx, 4
rep     movsw

        lea         di, word ptr smally
        mov         si, word ptr y
        mov         cx, 4
```

```
rep     movsw

        lea         di, word ptr smallz
        mov         si, word ptr z
        mov         cx, 4
rep     movsw


        sub         al, al
        inc         al
        mov         byte ptr i, al ;i=1

twister:
        call        near ptr twist

for_next:
        cmp         byte ptr i, 4
        jne         chk_13
        call        near ptr twist
chk_13:
        cmp         byte ptr i, 13
        jne         chk_max                         ;add in repeating term
        call        near ptr twist
chk_max:
        inc         byte ptr i
        cmp         byte ptr i, 32
        ja          hyper_exit
        jmp         twister

hyper_exit:
        ret

twist:
        sub         ax, ax
        mov         al, i
        mov         word ptr shifter, ax

        mov         si, word ptr x
        mov         ax, word ptr [si]
        mov         bx, word ptr [si][2]
```

```
        mov         cx, word ptr [si][4]
        mov         dx, word ptr [si][6]

shiftx:
        sar         dx, 1
        rcr         cx, 1
        rcr         bx, 1
        rcr         ax, 1
        dec         word ptr shifter
        jnz         shiftx
load_smallx:
        mov         word ptr smallx, ax
        mov         word ptr smallx[2], bx
        mov         word ptr smallx[4], cx
        mov         word ptr smallx[6], dx      ;x=X>>i

        sub         ax, ax
        mov         al, i
        mov         word ptr shifter, ax

        mov         si, word ptry
        mov         ax, word ptr [si]
        mov         bx, word ptr [si][2]
        mov         cx, word ptr [si][4]
        mov         dx, word ptr [si][6]

shifty:
        sar         dx, 1
        rcr         cx, 1
        rcr         bx, 1
        rcr         ax, 1
        dec         word ptr shifter
        jnz         shifty
load_smally:
        mov         word ptr smally, ax
        mov         word ptr smally[2], bx
        mov         word ptr smally[4], cx
        mov         word ptr smally[6], dx      ;y=Y>>i

get_atan:
```

```
        sub         bx, bx
        mov         bl, i
        shl         bx, 1
        shl         bx, 1                      ;got to point into a dword table
        lea         si, word ptr atanh_array
        mov         ax, word ptr [si][bx]
        mov         dx, word ptr [si][bx][2]

        mov         word ptr smallz, ax
        mov         word ptr smallz[2], dx     ;z=atanh[i]
        sub         ax, ax
        mov         word ptr smallz[4], ax
        mov         word ptr smallz[6], ax

test_Z:
        mov         si, word ptr z
        mov         ax, word ptr [si][6]
        or          ax, ax
        jns         positive

negative:
        mov         ax, word ptr smally
        mov         bx, word ptr smally[2]
        mov         cx, word ptr smally[4]
        mov         dx, word ptr smally[6]

        mov         di, word ptr x
        sub         word ptr [di], ax
        sbb         word ptr [di][2], bx
        sbb         word ptr [di][4], cx
        sbb         word ptr [di][6], dx       ;x -= y

        mov         ax, word ptr smallx
        mov         bx, word ptr smallx[2]
        mov         cx, word ptr smallx[4]
        mov         dx, word ptr smallx[6]

        mov         di, word ptr y
        sub         word ptr [di], ax
        sbb         word ptr [di][2], bx
        sbb         word ptr [di][4], cx
```

```
        sbb         word ptr [di][6], dx           ;Y -= x

        mov         ax, word ptr smallz
        mov         bx, word ptr smallz[2]
        mov         cx, word ptr smallz[4]
        mov         dx, word ptr smallz[6]

        mov         di, word ptr z
        add         word ptr [di], ax
        adc         word ptr [di][2], bx
        adc         word ptr [di][4], cx
        adc         word ptr [di][6], dx           ;x += y
        jmp         twist_exit

positive:
        mov         ax, word ptr smally
        mov         bx, word ptr smally[2]
        mov         cx, word ptr smally[4]
        mov         dx, word ptr smally[6]
        mov         di, word ptr x
        add         word ptr [di], ax
        adc         word ptr [di][2], bx
        adc         word ptr [di][4], cx
        adc         word ptr [di][6], dx           ;x += y

        mov         ax, word ptr smallx
        mov         bx, word ptr smallx [2]
        mov         cx, word ptr smallx [4]
        mov         dx, word ptr smallx [6]
        mov         di, word ptry
        add         word ptr [di], ax
        adc         word ptr [di][2], bx
        adc         word ptr [di][4], cx
        adc         word ptr [di][6], dx           ;Y += x

        mov         ax, word ptr smallz
        mov         bx, word ptr smallz[2]
        mov         cx, word ptr smallz[4]
        mov         dx, word ptr smallz[6]
        mov         di, wordptr z
```

```
        sub          word ptr [di], ax
        sbb          word ptr [di][2], bx
        sbb          word ptr [di][4], cx
        sbb          word ptr [di][6], dx           ;x -= y
twist_exit:
        retn

hyper  endp


;
;******
;
;ihyper- implementation of the inverse hyperbolic routine, a subset of the
;CORDIC devices.

;

ihyper proc uses bx cx dx di si, x:word, y:word, z:word

        local        smallx:qword, smally:qword, smallz:qword, i:byte,
                     shifter:word

        lea          di, word ptr smallx
        mov          si, word ptr x
        mov          cx, 4
rep     movsw

        lea          di, word ptr smally
        mov          si, word ptr y
        mov          cx, 4
rep     movsw

        lea          di, word ptr smallz
        mov          si, word ptr z
        mov          cx, 4
rep     movsw


        sub          al, al
        inc          al
```

```
        mov         byte ptr i, al ;i=0

twister:
        call        near ptr twist

for_next:
        cmp         byte ptr i, 4
        jne         chk_13
        call        near ptr twist
chk_13:
        cmp         byte ptr i, 13
        jne         chk_max                     ;add in repeating term
        call        near ptr twist
chk_max:
        inc         byte ptr i
        cmp         byte ptr i, 32
        ja          ihyper_exit
        jmp         twister

ihyper_exit:
        ret
;
twist:
        sub         ax, ax
        mov         al, i
        mov         word ptr shifter, ax

        mov         si, word ptr x
        mov         ax, word ptr [si]
        mov         bx, word ptr [si][2]
        mov         cx, word ptr [si][4]
        mov         dx, word ptr [si][6]

shiftx:
        sar         dx, 1
        rcr         cx, 1
        rcr         bx, 1
        rcr         ax, 1
        dec         word ptr shifter
        jnz         shiftx
```

```
load_smallx:
        mov         word ptr smallx, ax
        mov         word ptr smallx[2], bx
        mov         word ptr smallx[4], cx
        mov         word ptr smallx[6], dx      ;x=X>>i

        sub         ax, ax
        mov         al, i
        mov         word ptr shifter, ax

        mov         si, word ptr y
        mov         ax, word ptr [si]
        mov         bx, word ptr [si][2]
        mov         cx, word ptr [si][4]
        mov         dx, word ptr [si][6]

shifty:
        sar         dx, 1
        rcr         cx, 1
        rcr         bx, 1
        rcr         ax, 1
        dec         word ptr shifter
        jnz         shifty
load_smally:
        mov         word ptr smally, ax
        mov         word ptr smally[2], bx
        mov         word ptr smally[4], cx
        mov         word ptr smally[6], dx      ;y=Y>>i

get_atan:
        sub         bx, bx
        mov         bl, i
        shl         bx, 1
        shl         bx, 1                       ;got to point into a dword table
        lea         si, word ptr atanh_array
        mov         ax, word ptr [si][bx]
        mov         dx, word ptr [si][bx][2]

        mov         word ptr smallz, ax
        mov         word ptr smallz[2], dx      ;z=atanh[i]
```

```
        sub         ax, ax
        mov         word ptr smallz[4], ax
        mov         word ptr smallz[6], ax

test_Y:
        mov         si, word ptr y
        mov         ax, word ptr [si][6]
        or          ax, ax
        js          positive

negative:
        mov         ax, word ptr smally
        mov         bx, word ptr smally[2]
        mov         cx, word ptr smally[4]
        mov         dx, word ptr smally[6]

        mov         di, word ptr x
        sub         word ptr [di], ax
        sbb         word ptr [di][2], bx
        sbb         word ptr [di][4], cx
        sbb         word ptr [di][6], dx          ;x -= y

        mov         ax, word ptr smallx
        mov         bx, word ptr smallx[2]
        mov         cx, word ptr smallx[4]
        mov         dx, word ptr smallx[6]

        mov         di, word ptr y
        sub         word ptr [di], ax
        sbb         word ptr [di][2], bx
        sbb         word ptr [di][4], cx
        sbb         word ptr [di][6], dx          ;Y -= x

        mov         ax, word ptr smallz
        mov         bx, word ptr smallz[2]
        mov         cx, word ptr smallz[4]
        mov         dx, word ptr smallz[6]

        mov         di, word ptr z
        add         word ptr [di], ax
```

```
        adc         word ptr [di][2], bx
        adc         word ptr [di][4], cx
        adc         word ptr [di][6], dx          ;z += z
        jmp         twist-exit

positive:
        mov         ax, word ptr smally
        mov         bx, word ptr smally[2]
        mov         cx, word ptr smally[4]
        mov         dx, word ptr smally[6]
        mov         di, word ptr x
        add         word ptr [di], ax
        adc         word ptr [di][2], bx
        adc         word ptr [di][4], cx
        adc         word ptr [di][6], dx          ;x += y

        mov         ax, word ptr smallx
        mov         bx, word ptr smallx[2]
        mov         cx, word ptr smallx[4]
        mov         dx, word ptr smallx[6]
        mov         di, word ptr y
        add         word ptr [di], ax
        adc         word ptr [di][2], bx
        adc         word ptr [di][4], cx
        adc         word ptr [di][6], dx          ;Y += x

        mov         ax, word ptr smallz
        mov         bx, word ptr smallz[2]
        mov         cx, word ptr smallz[4]
        mov         dx, word ptr smallz[6]
        mov         di, word ptr z
        sub         word ptr [di], ax
        sbb         word ptr [di][2], bx
        sbb         word ptr [di][4], cx
        sbb         word ptr [di][6], dx          ;z -= z

twist_exit:
        retn
ihyper  endp
```

```
; ******
;rinit - initializes random number generator based upon input seed
;
;
;
        .data
a       dword           69069
IMAX    equ             32767
rantop  word            IMAX
ran1    dword           256 dup (0)
xsubi   dword           lh                          ;global iterative seed for
                                                    ;random number generator, change
                                                    ;this value to change default
init    byte            0h                          ;global variable signalling
                                                    ;whether the generator has be
                                                    ;initialized or not
        .code

rinit   proc uses bx cx dx si di, seed:dword


        lea             di, word ptr ran1

        mov             ax, word ptr seed[2]
        mov             word ptr xsubi[2], ax       ;put in seed variable
        mov             ax, word ptr seed           ;get seed
        mov             word ptr xsubi, ax

        mov             cx, 256

fill-array:
        invoke          congruent
        mov             word ptr [di], ax
        mov             word ptr [di][2], dx
        add             di, 4
        loop            fill-array

rinit_exit:
```

```
        sub          ax, ax
        not          ax
        mov          byte ptr init, al
        ret

rinit   endp

;
; ******
;
;congruent -performs simple congruential algorithm

;

;
congruent proc uses bx cx
        mov          ax, word ptr xsubi          ;a*seed (mod2^32)
        mul          word ptr a
        mov          bx, ax                      ;lower word of result
        mov          cx, dx                      ;upper word
        mov          ax, word ptr xsubi[2]
        mul          word ptr a
        add          ax, cx
        adc          dx, 0

        add          ax, word ptr xsubi          ;a multiplication by one is just
                                                 ;an add, right?
        adc          dx, word ptr xsubi[2]
        mov          dx, bx
        mov          word ptr xsubi, bx
        mov          word ptr xsubi[2], ax
        ret
congruent endp


;******
;
;irandom- generates random floats using the linear congruential method


irandom            proc uses bx cx dx si di
```

```
        lea         si, word ptr ran1
        mov         al, byte ptr init          ;check for initialization
        or          al, al
        jne         already-initialized
        invoke      rinit,  xsubi              ;default to 1
already_initialized:
        invoke      congruent                  ;get a random number
        and         ax, 0ffh                   ;every fourth byte, right?
        shl         ax, 1
        shl         ax, 1                      ;multiply by four
        add         si, ax                     ;point to number in array
        mov         di, si                     ;so we can put one there too
        invoke      congruent
        mov         bx, word ptr [si]
        mov         cx, word ptr [si][2]       ;get number from array
        mov         word ptr [di], ax
        mov         word ptr [di][2], dx       ;replace it with another
        mov         word ptr xsubi, bx
        mov         word ptr xsubi[2], cx      ;seed for next random

        mov         ax, bx
        mul         word ptr rantop            ;scale output by rantop, the
                                               ;maximum size of the random
        mov         ax, dx                     ;number if rantop were made
                                               ;0ffffH, the value could be used
                                               ;directly as a fraction

        ret
irandom             endp

;
        end
```

TABLE.ASM

```
        .dosseg
        .model small, c, os_dos

        include math.inc


;
        .data
;
;sines(degrees)
sine_tbl word       0ffffh, 0fff6h, 0ffd8h, 0ffa6h, 0ff60h, 0ff06h,
                    0fe98h, 0fe17h, 0fd82h, 0fcdgh, 0fc1ch, 0fb4bh,
                    0fa67h, 0f970h, 0f865h, 0f746h, 0f615h, 0f4dOh,
                    0f378h, 0f20dh, 0f08fh, 0eeffh, 0ed5bh, 0eba6h,
                    0egdeh, 0e803h, 0e617h, 0e419h, 0e208h, 0dfe7h,
                    0ddb3h, 0db6fh, 0d919h, 0d6b3h, 0d43bh, 0d1b3h,
                    0cf1bh, 0cc73h, 0cgbbh, 0c6f3h, 0c41bh, 0c134h,
                    0be3eh, 0bb39h, 0b826h, 0b504h, 0b1d5h, 0ae73h

         word       0ab4ch, 0a7f3h, 0a48dh, 0a1lbh, 09d9bh, 09a10h,
                    09679h, 092d5h, 08f27h, 08b6dh, 087a8h, 083d9h,
                    08000h, 07c1ch, 0782fh, 07438h, 07039h, 06c30h,
                    0681fh, 06406h, 05fe6h, 05bbeh, 0578eh, 05358h,
                    04f1bh, 04ad8h, 04690h, 04241h, 03deeh, 03996h,
                    03539h, 030d8h, 02c74h, 0280ch, 023aOh, 01f32h,
                    01acah, 0164fh, 011dbh, 00d65h, 008efh, 00477h,
                    Oh


;
;log(x/128)
log10_tbl word      00000h, 000ddh, 001b9h, 00293h,
                    0036bh, 00442h, 00517h, 005ebh, 006bdh, 0078eh,
                    0085dh, 0092ah, 009f6h, 00ac1h, 00b8ah, 00c51h,
                    00d18h, 00dddh, 00ea0h, 00f63h, 01024h, 010e3h,
                    011a2h, 0125fh, 0131bh, 013d5h, 0148fh, 01547h,
                    015feh, 016b4h, 01769h, 0181ch, 018cfh, 01980h

        word        01a30h, 01adfh, 01b8dh, 01c3ah, 01ce6h, 01dg1h,
```

```
                    01e3bh, 01ee4h, 01f8ch, 02033h, 020d9h, 0217eh,
                    02222h, 022c5h, 02367h, 02409h, 024a9h, 02548h,
                    025e7h, 02685h, 02721h, 027bdh, 02858h, 028f3h

        word        0298ch, 02a25h, 02abdh, 02b54h, 02beah, 02c7fh,
                    02d14h, 02da8h, 02e3bh, 02ecdh, 02f5fh, 02ff0h,
                    03080h, 0310fh, 0319eh, 0322ch, 032b9h, 03345h,
                    033d1h, 0345ch, 034e7h, 03571h, 035fah, 03682h,
                    0370ah, 03792h, 03818h, 0389eh, 03923h, 039a8h

        word        03a2ch, 03ab0h, 03b32h, 03bb5h, 03c36h, 03cb7h,
                    03d38h, 03db8h, 03e37h, 03eb6h, 03f34h, 03fb2h,
                    0402fh, 040ach, 04128h, 041a3h, 0421eh, 04298h,
                    04312h, 0438ch, 04405h, 0447dh, 044f5h, 0456ch,
                    045e3h, 04659h, 046cfh, 04744h, 047b9h, 0482eh

         word       048a2h, 04915h, 04988h, 049fbh, 04a6dh, 04adeh,
                    04b50h, 04bc0h, 04c31h, 04ca0h, 04d10h

;log(2**x)
log10_power dword 000000h, 004d10h, 009a20h, 00e730h, 013441h, 018151h,
                    01ce61h, 021b72h, 026882h, 02b592h, 0302a3h, 034fb3h,
                    039cc3h, 03e9d3h, 0436e4h, 0483f4h, 04d104h, 051e15h,
                    056b25h, 05b835h, 060546h, 065256h, 069f66h, 06ec76h,
                    073987h, 078697h, 07d3a7h, 0820b8h, 086dc8h, 08bad8h,
                    0907e9h, 0954f9h

;——————————————————————————————
;sqrt(x+128)*2**24
;these are terribly rough, perhaps combined with Euclid's method
;they would produce high quality numbers

sqr_tbl             word0b504h, 0b5b9h, 0b66dh, 0b720h, 0b7d3h, 0b885h,
                    0b936h, 0b9e7h, 0ba97h, 0bb46h, 0bbfSh, 0bca3h,
                    0bd50h, 0bdfdh, 0beagh, 0bf55h, 0c000h, 0c0aah,
                    0c154h, 0c1fdh, 0c2a5h, 0c34eh, 0c3f5h, 0c49ch,
                    0c542h, 0c5e8h, 0c68eh, 0c732h, 0c7d7h, 0c87ah

        word        0cg1dh, 0c9c0h, 0ca62h, 0cb04h, 0cba5h, 0cc46h,
                    0cce6h, 0cd86h, 0ce25h, 0cec3h, 0cf62h, 0d000h,
```

```
                0d09dh, 0d13ah. 0d1d6h, 0d272h, 0d30dh, 0d3a8h,
                0d443h, 0d4ddh, 0d577h, 0d610h, 0d6a9h, 0d742h,
                0d7dah, 0d871h, 0d908h, 0d99fh, 0da35h, 0dacbh

       word     0dbG1h, 0dbf6h, 0dc8bh, 0dd1fh, 0ddb3h, 0de47h,
                0dedah, 0df6dh, 0e000h, 0e092h, 0e123h, 0e1b5h,
                0e246h, 0e2d6h, 0e367h, 0e3f7h, 0e486h, 0e515h,
                0e5a4h, 0e633h, 0e6c1h, 0e74fh, 0e7dch, 0e869h,
                0e8f6h, 0e983h, 0ea0fh, 0eagbh, 0eb26h, 0ebb1h

       word     0ec3ch, 0ecc7h, 0ed51h, 0eddbh, 0ee65h, 0eeeeh,
                0ef77h, 0f000h, 0f088h, 0f110h, 0f198h, 0f21fh,
                0f2a6h, 0f32dh, 0f3b4h, 0f43ah, 0f4c0h, 0f546h,
                0f5cbh, 0f651h, 0f6d6h, 0f75ah, 0f7deh, 0f863h,
                0f8e6h, 0f96ah, 0fgedh, 0fa70h, 0faf3h, 0fb75h

       word     0fbf7h, 0fc79h, 0fcfbh, 0fd7ch, 0fdfdh, 0fe7eh,
                0feffh, 0ff7fh, 00000h


;sqrt(2**x)
sqr_power word  00ffffh, 00b504h, 008000h, 005a82h, 004000h, 002d41h,
                002000h, 0016a0h, 001000h, 000b50h, 000800h, 0005a8h,
                000400h, 0002d4h, 000200h, 00016ah, 000100h, 0000b5h,
                000080h, 00005ah, 000040h, 00002dh, 000020h, 000016h,
                000010h, 00000bh, 000008h, 000006h, 000004h, 000002h,
                000002h, 000001h, 000001h
```

```
atanh_array     dword   0h, 8c9f53d5h, 4162bbeah, 202b1239h, 1005588ah,
                        800aac4h, 4001556h, 20002aah, 1000055h,
                        80000ah, 400001h, 200000h, 100000h, 80000h, 40000h,
                        20000h, 10000h, 8000h, 3fffh, 1fffh,
                        0fffh, 7ffh, 3ffh, 1ffh, 0ffh, 7fh, 3fh, 1fh, 0fh, 7h,
                        3h, 1h, 0h
```

```
atan_array    dword    0c90fdaa2h, 76b19c16h, 3eb6ebf2h, 1fd5ba9bh, 0ffaaddch,
                       7ff556fh, 3ffeaabh, 1fffd55h,
                       0ffffabh, 7ffff5h, 3fffffh, 200000h, 100000h, 80000h,
                       40000h, 20000h, 10000h, 8000h,
                       4000h, 2000h, 1000h, 800h, 400h, 200h, 100h, 80h, 40h,
                       20h, 10h, 8h, 4h, 2h, 1h

power2        qword    100000000h, 95c01a3ah, 5269e12fh, 2b803473h, 1663f6fah,
                       0b5d69bah, 5b9e5a1h, 2dfca16h, 1709c46h,
                       5c60aah, 2e2d71h, 171600h, 0b8adlh, 5c55dh, 2e2abh,
                       17155h, 0b8aah, 5c55h, 2e2ah, 1715h, 0b8ah,
                       5c5h, 2e2h, 171h, 0b8h, 5ch, 2eh, 17h, 0bh, 5h, 2h, 1h

log2          qword    100000000h, 6a3fe5c6h, 31513015h, 17d60496h, 0bb9ca64h,
                       5d0fba1h, 2e58f74h, 1720d9ch, 0b8d875h,
                       5c60aah, 2e2d71h, 171600h, 0b8ad1h, 5c55dh, 2e2abh,
                       17155h, 0b8aah, 5c55h, 2e2ah, 1715h, 0b8ah,
                       5c5h, 2e2h, 171h, 0b8h, 5ch, 2eh, 17h, 0bh, 5h, 2h, 1h


power10       qword    4d104d42h, 2d145116h, 18cf1838h, 0d1854ebh, 6bd7e4bh,
                       36bd211h,  1b9476ah,
                       0dd7ea4h, 6ef67ah, 378915h, 1bc802h, 0de4dfh, 6f2a7h,
                       37961h,  1bcb4h,  0de5bh,
                       6f2eh, 3797h, 1bcbh, 0de6h, 6f3h, 379h, 1bdh, 0deh, 6fh,
                       38h, 1ch, 0eh, 7h, 3h, 2h, 1h

alg           qword    3f3180000000h, 0b95e8082e308h, 3ede5bd8a937h,
                       0beee08307e16h, 3c5ed689e495h, 0c0b286223e39h,
                       3f8000000000h

xp            qword    3f3000000000h, 3bb90bfbe8efh, 3e8000000000h,
                       3b885307cc09h, 3f0000000000h,
                       3d4cbf5b2122h

sincos        qword    404900000000h, 3a7daa20968bh, 0be2aaaa8fdbeh,
                       3c088739cb85h, 0b94fb2227f1ah,
                       362e9c5a91d8h

tancot        qword    3fc900000000h, 39fdaa22168ch, 3f8000000000h,
```

```
                        0bdc433b8376bh, 3f8000000000h, 0bedbb7af3f84h,
                        3c1f33753551h

polytan      qword      100000000h, 0, 0aaaaaaabh, 0, 33333333h, 0, 0db6db6dbh,
                        0, 1c71c71ch, 0, 0e8ba2e8ch, 0, 13b13b14h, 0,
                        0eeeeeeefh, 0, 0f0f0f0fh, 0, 0f286bca2h, 0, 0c30c30ch,
                        0, 0f4de9bd3h, 0, 0a3d70a4h, 0, 0f684bda1h, 0, 8d3dcb1h,
                        0, 0f7bdef7ch, 0

polysin      qword      100000000h, 0, 0ffffffffd5555555h, 0, 222222221, 0,
                        0fffffffffff2ff30h, 0, 2e3ch, 0, 0ffffffffffffff94h

dgt          qword      000000000000h, 3f8000000000h, 400000000000h,
                        404000000000h, 408000000000h, 40a000000000h,
                        40c000000000h, 40e000000000h, 410000000000h,
                        411000000000h
one          qword      3f8000000000h
ten          qword      412000000000h, 42c800000000h, 461c40000000h,
                        4cbebc200000h, 5a0e1bc9bf00h, 749dc5ada82bh
one-half     qword      3f0000000000h
;
     end
```

APPENDIX G

Math.C

```
#include<io.h>
#include<conio.h>
#include<stdio.h>
#include <fcntl.h>        /* O_constant definitions */
#include<sys\types.h>
#include <sys\stat.h>     /* S_constant definitions */
#include<malloc.h>
#include<errno.h>
#include<math.h>
#include<float.h>
#include<stdlib.h>
#include<time.h>
#include<string.h>

#define TRUE 1
#define FALSE 0


union{
       float realsmall;
       double realbig;
       int smallint;
       long bigint;
       char bytes[16];
       int words[8];
       long dwords[4];
}operand0;

union{
       float realsmall;
       double realbig;
       int smallint;
```

```
        long bigint;
        char bytes[16];
        int words[8];
        long dwords[4];
}operand1;

union{
        float realsmall;
        double realbig;
        int smallint;
        long bigint;
        char bytes[16];
        int words[8];
        long dwords[4];
}operand2;

union{
        float realsmall;
        double realbig;
        int smallint;
        long bigint;
        char bytes[16];
        int words[8];
        long dwords[4];
}answer0;

union{
        float realsmall;
        double realbig;
        int smallint;
        long bigint;
        char bytes[16];
        int words[8];
        long dwords[4];
}answer1;

/*doubles are used to indicate to C to push a quadword parameter, please see*/
/*the unions above for more information on how to manipulate these parameters*/
extern void lgb(union answr, double *);
extern void pwrb(double, double *);
```

```
extern int irandom(void);
extern void rinit(int);
extern void divnewt(double, double, double*);
extern void divmul(double, double, double*);
extern void ftf(double, double*);
extern void ftfx(double, double*);
extern void taylorsin(double, double*);
extern void ihyper(double *, double *, double *);
extern void hyper(double *, double *, double *);
extern void icirc(double *, double *, double *);
extern void circular(double *, double *, double *);
extern void fp_sqr(float, float*);
extern void fp_tan(float, float*);
extern void fp_cos(float, float*);
extern void fp_sin(float, float*);

extern void fp_mul(float, float, float *);
extern void fp_div(float, float, float *);
extern void fp_add(float, float, float *);
extern void fp_sub(float, float, float *);
extern void fp_abs(float, float*);
extern void lg10(double *, double *);
extern void sqrtt(double *, double *);
extern void dcsin(double *, double *, unsigned char);
extern atf(char*string, float *asm_val);
extern ftofx(float, long*);
extern ftoasc(float, char*);
extern fr_xp(float, float *, char *);
extern ld_xp(float, float*, char);
extern fx_sqr(long, long*);
extern school_sqr(long, long*);

extern dnt_bn(char *, int *);
extern dfc_bn(char *, int *);
extern bn_dnt(unsigned long int, char *);
extern bfc_dc(unsigned long int, char *);
extern fp_intrnd(float, float*);
extern fp_ceil(float, float*);
extern fp_floor(float, float*);
```

```
int  binary_integer;
char decimal_string0[20];
char decimal_string1[20];
char string0[25], string1[25];
long   radicand;
long   root;
char  exponent;
float temp;
float value;
float  mantissa;
float asm_val0, asm_val1;
float floor_test;
float ceil_test;
float  intrnd_test;
float  asm_mul;
float  tst_asm_mul;
float  asm_div;
float  asm_add;
float  asm_sub;
float  mul_tst;
float  asm_mul_tst;
float  div_tst;
float add_tst;
float sub_tst;
float  fpsin;
float  fpsqr;
float  fplog;
float  fplog10;

/*this routine scales a random number to a maximum without using a modular
operation*/
int get random(int max)
{
                    unsigned  long  a,  b;

                                    a = irandom();
                                    b = max*a;
                                    return(b/32768);
}
```

```
main()

        float  fp_numa;
        float  fp_numb;
        float  fp_numc;
        float  fp_numd;
        long numa, numb, numc, numd;
        double  dwrd;
        double  test;
        float nt;
        char *buf;
        int ad_buf, ch, j;

        double  error;
        unsigned long temporary;
        unsigned count = 0x1000, cnt = 0, errcnt,
           passes, maxpass, cycle_cnt;

        dwrd=4294967296.0;          /*2^32*/
        nt = 65536;                             /*2^16*/

        ad_buf = open( "tstdata", O_TEXT | O_WRONLY | O_CREAT |
           O_TRUNC, S_IREAD | S_IWRITE );

        if( ad_buf == -1 ) {
               perror("\nopen failed");
               exit(-1);
               }

        /* allocate a file buffer.*/
        If( (buf = (char*)malloc( (size_t)count )) ==NULL) {
           perror("\nnot enuf memory");
        exit(-1);


        cycle_cnt = 0;


        do{
        rinit((unsigned  int)time(NULL)  );
```

```
maxpass=1000;
error= 0.00001;  /*a zero error can result in errors of +0.0 or -0.0
                 reported*/
errcnt = 0;      /*smaller errors sometimes exceedthe
                 precisionof a single real*/
passes = 0;
do{

getrandom(irandom());

while((numa=getrandom(irandom())) == 0);
if((irandom() * .001) >15) fp_numa = (float)numa * -1.0;
else fp_numa = (float)numa;
while((numb = getrandom(irandom())) == 0);
if((irandom0 * .001) >15) fp_numb = (float)numb * -1.0;
else fp_numb = (float)numb;

while((numc = irandom()) == 0);
fp_sqr((float)numc, &fp_numc);
fp_numa *= fp_numc;
while((numd = irandom()) == 0);
fp_sqr((float)numd, &fp_numd);
fp_numb *= fp_numd;


sprintf(buf,"\ntwo random floats are fp_numa %f and
    fp_numb %f", fp_numa, fp_numb);
if(count = write( ad_buf, buf, strlen(buf) ) == - 1)
    perror("couldn't write");
test=(double)fp_numa;
gcvt((double)fp_numa, 8, string0); /*needed to test asm
                                     conversions*/
gcvt((double)fp_numb, 8, string1);
sprintf(buf,"\nstring0 (fp_numa): %s, string1 (fp_numb): %s",
    string0, string1);
if(count = write( ad_buf, buf, strlen(buf) ) == - 1)
    perror("couldn't write");

atf(string0, &asm_va10); /*convert string to float*/
```

```
atf(string1, &asm_val1);
sprintf(buf,"\nasm_val0(string0):%fandasm-val1(stringl):    %f",
    fp_numa,  fp_numb);
if(count = write( ad_buf, buf, strlen(buf) ) == - 1)
   perror("couldn't write");


mul_tst=fp_numa*fp_numb;
asm_mul_tst = asm_val0*asm_val1;
div_tst = fp_numa/fp_numb;
add_tst = fp_numa+fp_numb;
sub_tst = fp_numa-fp_numb;

fp_mul(asm_val0, asm_vail, &asm_mul);
fp_mul(fp3uma,  fp_numb,  &tst_asm_mul);
fp_div(asm_val0, asm_val1, &asm_div);
fp_add(asm_val0, asm_val1, &asm_add);
fp_sub(asm_val0, asm_vall, &asm_sub);

sprintf(buf,"\nfp_numa*fp_numb, msc = %f, asm = %f,
   difference = %f", mul_tst, asm_mul, mul_tst-asm_mul);
if(count = write( ad_buf, buf, strlen(buf) ) == - 1)
   perror("couldn't write");
sprintf(buf,"\nfp_numa/fp_numb, msc = %f, asm = %f,
   difference = %f", div_tst, asm_div, div_tst-asm_div);
if(count = write( ad_buf, buf, strlen(buf) ) == - 1)
   perror("couldn't   write");
sprintf(buf,"\nfp_numa+fp_numb, msc = %f, asm = %f,
   difference = %f", add_tst, asm_add, add_tst-asm_add);
if(count = write( ad_buf, buf, strlen(buf) ) == - 1)
   perror("couldn't write");
sprintf(buf,"\nfp_numa-fp_numb, msc = %f, asm = %f,
   difference = %f", sub_tst,  asm_sub,  sub_tst-asm_sub);
if(count = write( ad_buf, buf, strlen(buf) ) == - 1)
   perror("couldn't write");

temp = (float)getrandom(100);
fp_sqr(temp,  &fpsqr);
sprintf(buf,"\nsqrt(%f),msc = %f, asm = %f", temp,
    (float)sqrt((double)temp),fpsqr)   ;
```

```
        if(count=write( ad_buf, buf, strlen(buf)) == - 1)
           perror("couldn't write");

        fp_sin(temp,  &fpsin);
        sprintf(buf,"\nfp_sin(%f), msc = %f, asm = %f", temp,
            (float)sin((double)temp),   fpsin);
        if(count = write( ad_buf, buf, strlen(buf) 1 == - 1)
           perror("couldn't write");



/*error reporting*/

        sprintf(buf,"\niteration: %x", cnt++);
        if(count = write( ad_buf, buf, strlen(buf) 1 == - 1)
           perror("couldn't write");

        sprintf(buf,"\nfp-numais %f and fp_numb is %f", fp_numa, fp_numb);
        if(count = write( ad_buf, buf, strlen(buf) 1 == - 1)
           perror("couldn't write");

        sprintf(buf,"\nstring0 is %s and string1 is %s", string0, string1);
        if(count = write( ad_buf, buf, strlen(buf)) == - 1)
           perror("couldn't write");


        if((fabs((double)mul_tst-(double)asm_mul)) >error) {
           errcnt++;
        sprintf(buf,"\nmsc multiplication says %f, I say %f, error= %f",
           mul_tst, asm_mul, mul_tst-asm_mull;
        if(count = write( ad_buf, buf, strlen(buf)) == - 1)
           perror("couldn't write");


        if((fabs((double)div_tst-(double)asm_div)) >error) {
           errcnt++;
        sprintf(buf,"\nmsc division says %f, I say %f, error= %f",
           div_tst, asm_div, div_tst-asm_div);
        if(count = write( ad_buf, buf, strlen(buf)) == - 1)
           perror("couldn't write");
```

```
if((fabs((double)sub_tst-(double)asm_sub)) >error) {
   errcnt++;
 sprintf(buf,"\nmsc subtraction says %f, I say %f, error= %f",
   sub_tst, asm_sub, sub_tst-asm_sub);
 if(count = write( ad_buf, buf, strlen(buf) ) == - 1)
   perror("couldn't write");


if((fabs((double)add_tst-(double)asm_add)) >error) {
   errcnt++;
   sprintf(buf,"\nmsc addition says %f, I say %f, error= %f",
   add_tst, asm_add, add_tst-asm_add);
   if(count = write( ad_buf, buf, strlen(buf) ) == - 1
   perror("couldn't write");


printf(".");
sprintf(buf,"\n");
if(count = write( ad_buf, buf, strlen(buf) ) == - 1
   perror("couldn't write");

passes++;

    }while(!kbhit() && ! (passes == maxpass));
    cycle_cnt++;
    }while(!errcnt && !kbhit());

printf("\nerrors: %d cycles: %d pass: %d", errcnt, cycle_cnt,
   passes);
close( ad_buf );
free( buf );
```

Glossary

abscissa
On the Cartesian Axes, it is the distance from a point to the y axis.

accumulator
A general purpose register on many microprocessors. It may be the target or destination operand for an instruction, and will often have specific instructions that affect it only.

accuracy
The degree of correctness of a quantity or expression.

add-with-carry
To add a value to a destination variable *with* the current state of the carry flag.

addend
A number or quantity added to another.

addition
The process of incrementing by a value, or joining one set with another.

additional numbering systems
Numbering systems in which the symbols combine to form the next higher group. An example of this is the Roman system. See Chapter 1.

algorithm
A set of guidelines or rules for solving a problem in a finite number of steps.

align
To arrange in memory or a register to produce a proper relationship.

arithmetic
Operations involving addition, subtraction, multiplication, division, powers and roots.

ASCII
The American Standard Code for Information Interchange. A seven bit code used for the interpretation of a byte of data as a character.

associative law
An arithmetic law which states that the order of combination or operation of the operands has no influence on the result. The associative law of multiplication is (a*b)*c=a*(b*c).

atan
Arctangent. This is the *angle* for which we have the tangent.

atanh
The Inverse Hyperbolic Tangent. This

is the *angle* for which we have the hyperbolic tangent.

augend
A number or quantity to which another is added.

base
A grouping of counting units that is raised to various powers to produce the principal counting units of a numbering system.

binary
A system of numeration using base 2.

bit—Binary digIT.

Boolean
A form of algebra proposed by George Boole in 1847. This is a combinatorial system allowing the processing of operands with operators such as AND, OR, NOT, IF, THEN, and EXCEPT.

byte
A grouping of bits the computer or CPU operates upon as a unit. Generally, a byte comprises 8 bits.

cardinal
A counting number, or natural number indicating quantity but not order.

carry flag
A bit in the status register of many microprocessors and micro controllers indicating whether the result of an operation was to large for the destination data type. An overflow from an unsigned addition or a borrow from an unsigned subtraction might cause a carry.

ceil
The least integer greater than or equal to a value.

coefficient
A numerical factor, such as 5 in $5x$.

complement- An inversion or a kind of negation. A one's complement results in each zero of an operand becoming a one and each one becoming a zero. To perform a two's complement, first one's complement the operand, then increment by one.

commutative law
An arithmetic law which states that the order of the operands has no influence on the result of the operation. The commutative law of addtition is

$a + b = b + a$.

congruence
Two numbers or quantities are congruent, if, after division by the same value, their remainders are equal.

coordinates
A set of two or more numbers determining the position of a point in a space of a given dimension.

CORDIC
COrdinate Rotation Digital Computer. The CORDIC functions are a group of algorithms that are capable of computing high quality approximations of the

transcendental functions and require very little in the way of arithmetic power from the processor.

cosine

In the triangle, the ratio x/r is a function of the angle *q* known as the cosine.

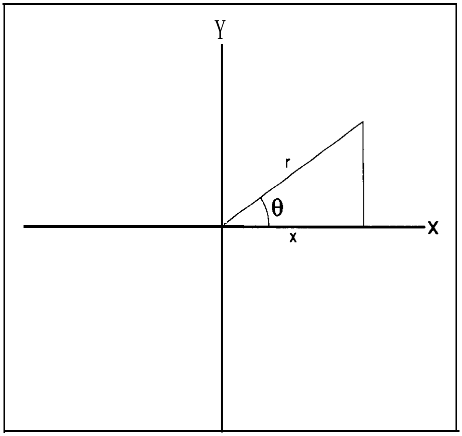


Figure 1. A Right Triangle.

decimal

having to do with base 10.

decimal-point

Radix point for base 10.

denominator

The divisor in a fraction.

denormal

A fraction with a minimum exponent and leading bit of the significand zero.

derivative

The instantaneous rate of change of a function with respect to a variable.

distributive law

An arithmetic law that describes a connection between operations. This distributive law is as follows: $a*(b+c)=a*b+a*c$. Note that the multiplication is distributed *over the addition.*

dividend

The number to be divided.

division

Iterative subtraction of one operand from another.

divisor

The number used to divide another, such as the dividend.

double-precision

For IEEE floating point numbers, it is twice the single precision format length or 64 bits.

doubleword (dword)

Twice the number of bits in a word. On the 8086, it is 32 bits.

exception

In IEEE floating point specification, an exception is a special case that may require attention. There are five exceptions and each has a trap that may be enabled or disabled. The exceptions are:

- Invalid operation, including addition or subtraction with ∞ as an operand, multiplication using ∞ as an operand, ∞/∞ or 0/0, division with invalid operands, a remainder operation where the divisor is zero or unnormalized or the dividend is infinite.

- Division by zero.
- Overflow. The rounded result produced a legal number but an exponent too large for the floating point format.
- Underflow. The result is too small for the floating point format.

Inexact result without an invalid operation exception. The rounded result is not exact.

far
A function or pointer is defined as *far* if it employs more than a word to identify it. This usually means that it is not within the same 64K segment with the function or routine referencing it.

fixed-point
A form of arithmetic in which the radix point is always assumed to be in the same place.

floating-point
A method of numerical expression, in which the number is represented by a fraction, a scaling factor (exponent), and a sign.

floor
The greatest integer less than or equal to a value.

fraction
The symbolic (or otherwise) quotient of two quantities.

guard digits
Digits to the right of the significand or significant bits to provide added precision to the results of arithmetic computations.

hidden bit
The most significant bit of the floating point significand. It exists, but is not represented, just to the left of the radix point and is always a one (except in the case of the denormal).

integer (int)
A whole number. A word on a personal computer, 16 bits.

interpolate
To determine a value between two known values.

irrational number
A number that can not be represented *exactly* in a particular base.

K-space
K-spaces are multi-dimensional or k-dimensional where K is an integer.

linear congruential
A method of producing pseudo-random numbers using modular arithmetic.

linear interpolation
The process of approximating f(x) by fitting a straight line to a function at the

desired point and using proportion to estimate theposition of the unknown on that line. See Chapter 6.

logarithm (log)

In any base, x, where $x^n = b$, n is the logarithm of b to the base x. Another notation is $n = log_x b$.

long

A double word. On a personal computer, 32 bits.

long real

The long real is defined by IEEE 754 as a double precision floating-point number.

LSB

Least Significant Bit.

LSW

Least Significant Word.

mantissa

The fractional part of a floating point number.

minimax

A mathematical technique that produces a polynomial approximation optimized for the least maximum error.

minuend

The number you are subtracting from.

modulus

The range of values of a particular system. This is the basis of modular arithmetic, such as used in telling time. For example, 4 A.M. plus 16 hours is 8 P.M. ((4 + 16) mod 12 = 8).

MPU

Micro-Processor-Unit.

MSB

Most Significant Bit.

MSW

Most significant Word.

multiplicand

The number you are multiplying.

multiplication

Iterative addition of one operand with another.

multiplier

The number you are multiplying by.

multiprecision

Methods of performing arithmetic that use a greater number of bits that provided in the word size of the computer.

NAN

These can be either Signaling or Quiet according to the IEEE 754 specification. A NAN (Not A Number) is the result of an operation that has not mathematical interpretation, such as $0 \div 0$.

natural numbers

All positive integers beginning with zero.

near

A function or pointer is defined as *near* if it is within a 64K segment with the function or routine referencing it. Thus, it requires only a single 16 bit word to identify it.

negative

A negative quantity, minus. Beginning at zero, the number line stretches in two directions. In one direction, there are the natural numbers, which are positive integers. In the other direction, there are the negative numbers. The opposite of a positive number.

nibble

Half a byte, typically four bits.

normalization

The process of producing a number whose left most significant digit is a one.

number ray

An illustration of the basic concepts associated with natural numbers. Any two natural numbers may have only one of the following relationships: $n_1 < n_2, n_1 = n_2, n_1 > n_2$ See Chapter 1.

numeration

System for counting or numbering.

numerator

- The dividend in a fraction.
- Octal
- Base 8.
- One's-complement
- A bit by bit inversion of a number. All ones are made zeros and zeros are made ones.

operand

A number or value with which or upon which an operation is performed.

ordinal

A number that indicates position, such as first or second.

ordinate

On the Cartesian Axes, it is the distance from a point to the x axis.

overflow

When a number grows to great through rounding or another arithmetic process for its data type, it overflows.

packed decimal

Method for storage of decimal numbers in which each of the two nibbles in a hexadecimal byte are used to hold decimal digits.

polynomial

An algebraic function of summed terms, where each term consists of a constant multiplier (factor) and at least one variable raised to an integer power. It is of the form:

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

positional numbering systems

A numbering system in which the value of a number is based upon its position, the value of any position is equal to the number multiplied by the base of the system taken to the power of the position. See Chapter 1.

positive
Plus. Those numbers to the right of zero on the number line. The opposite of a negative number.

power
Multiplying a value, x, by itself *n* number of times raises it the the power *n*. The notation is x^n.

precision
Number of digits used to represent a value.

product
The result of a multiplication.

quadword (qword)
Four words. On an 8086, this would be 64 bits.

quotient
The result of a division.

radicand
The quantity under the radical. Three is the radicand in the expression $\sqrt{3}$, which represents the square root of three.

radix
The base of a numbering system.

radix point
The division in a number between its integer portion and fractional portion. In the decimal system, it is the decimal point.

rational number
A number capable of being represented *exactly* in a particular base.

real number
A number possessing a fractional extension.

remainder
The difference between the dividend and the product of the divisor and the quotient.

resolution
The constituent parts of a system. This has to do with the precision the arithmetic uses to represent values, the greater the precision, the more resolution.

restoring division
A form of division in which the divisor is subtracted from the dividend until an underflow occurs. At this point, the divisor is added back into the dividend. The number of times the divisor could be subtracted without underflow is returned as the quotient and the last minuend is returned as the remainder.

root
The *nth* root of a number, x, (written: $a = \sqrt[n]{x}$ is that number when raised to the *nth* power is equal to the original number $(x = a^n)$.

rounding
A specified method of reducing the number of digits in a number while adjusting the remaining digits accordingly.

scaling
A technique that brings a number within certain bounds by multiplication or division by a factor. In a floating point number, the significand is always between 1.0 and 2.0 and the exponent is the scaling factor.

seed
The initial input to the linear congruential psuedo-random number generator.

short real
The short real is defined by IEEE 754 as a single precision floating point number.

sign-extension
The sign of the number-one for negative, zero for positive-fills any unused bits from the MSB of the actual number to the MSB of the data type. For example, -9H, in two's complement notation is f7H expressed in eight bits and fff7H in sixteen. Note that the sign bit fills out the data type to the MSB.

significant digits
The principal digits in a number.

significand
In a floating point number, it is the leading bit (implicit or explicit) to the immediate left of the radix point and the fraction to the right.

sine
In Figure one, it is the ratio y/r.

single-precision
In accordance with the IEEE format, it is a floating point comprising 32 bits, with a 24 bit significand, eight bit exponent, and sign bit.

subtraction
The process opposite to addition. Deduction or taking away.

subtrahend
A number you subtract from another.

sum
The result of an addition.

tangent (tan)
In figure one, the ratio y/x denotes the tangent.

two's complement
A one's complement plus one.

under flow
This occurs when the result of an operation requires a borrow.

whole number
An integer.

word
The basic precision offered by a computer. On an 8086, it is 16 bits.

Index

Symbols

A

B

C

D

Numerical Methods

Numerical Methods brings together in one source, the mathematical techniques professional assembly-language programmers need to write arithmetic routines for real-time embedded systems.

This book presents a broad approach to microprocessor arithmetic, covering everything from data on the positional number system to algorithms for developing elementary functions. Fixed point and floating point routines are developed and thoroughly discussed, teaching you how to customize the routines or write your own, even if you are using a compiler.

Many of the explanations in this book are complemented with interesting techniques and useful 8086 and pseudo-code examples. These include algorithms for drawing circles and lines without resorting to trigonometry or floating point, making the algorithms very fast and efficient, In addition, there are examples highlighting various techniques for performing division on large operands such as linear interpolation, the Newton-Raphson iteration, and iterative multiplication.

The companion disk (in MS/PC-DOS format) contains the routines presented in the book plus a simple C shell that can be used to exercise them.

Other topics include:

- Positional number theory, bases, and signed arithmetic.
- Algorithms for performing integer arithmetic.
- Fixed point and floating point mathematical techniques without a coprocessor.
- Taylor expansions, Homers Method, and pseudo-random numbers.
- Input, Output, and Conversion methods.
- Elementary functions including fixed-point algorithms, computing with tables, cordic algorithms, and polynomial evaluations.

Don Morgan is a professional programmer and consultant with more than 20 years of programming experience. He is also a contributor to *Dr. Dobb's Journal* and resides in Simi Valley, CA.

Why this book is for you—See page 1

DISK INCLUDED

M&T BOOKS

M&T Books
411 Borel Avenue
San Mate., CA 94402

LEVEL	ADVANCED
TOPIC	PROGRAMMING/NUMERICS
SOFTWARE	ASSEMBLY LANGUAGE
HARDWARE	PC

53695

9 781558 512320

ISBN 1-55851-232-2
>$36.95