Floating Point Arithmetic Chapter 14

Although integers provide an exact representation for numeric values, they suffer from two major drawbacks: the inability to represent fractional values and a limited dynamic range. Floating point arithmetic solves these two problems at the expense of accuracy and, on some processors, speed. Most programmers are aware of the speed loss associated with floating point arithmetic; however, they are blithely unware of the problems with accuracy.

For many applications, the benefits of floating point outweigh the disadvantages. However, to properly use floating point arithmetic in *any* program, you must learn how floating point arithmetic operates. Intel, understanding the importance of floating point arithmetic in modern programs, provided support for floating point arithmetic in the earliest designs of the 8086 – the 80x87 FPU (floating point unit or math coprocessor). However, on processors eariler than the 80486 (or on the 80486sx), the floating point processor is an optional device; it this device is not present you must simulate it in software.

This chapter contains four main sections. The first section discusses floating point arithmetic from a mathematical point of view. The second section discusses the binary floating point formats commonly used on Intel processors. The third discusses software floating point and the math routines from the UCR Standard Library. The fourth section discusses the 80x87 FPU chips.

14.0 Chapter Overview

This chapter contains four major sections: a description of floating point formats and operations (two sections), a discussion of the floating point support in the UCR Standard Library, and a discussion of the 80x87 FPU (floating point unit). The sections below that have a "•" prefix are essential. Those sections with a "❑" discuss advanced topics that you may want to put off for a while.

- The mathematics of floating point arithmetic.
- IEEE floating point formats.
- The UCR Standard Library floating point routines.
- The 80x87 floating point coprocessors.
- FPU data movement instructions.
- ❑ Conversions.
- Arithmetic instructions.
- Comparison instructions.
- ❑ Constant instructiuons.
- ❑ Transcendental instructions.
- ❑ Miscellaneous instructions.
- ❑ Integer operations.
- ❑ Additional trigonometric functions.

14.1 The Mathematics of Floating Point Arithmetic

A big problem with floating point arithmetic is that it does not follow the standard rules of algebra. Nevertheless, many programmers apply normal algebraic rules when using floating point arithmetic. This is a source of bugs in many programs. One of the primary goals of this section is to describe the limitations of floating point arithmetic so you will understand how to use it properly.

Normal algebraic rules apply only to *infinte precision* arithmetic. Consider the simple statement x:=x+1, x is an integer. On any modern computer this statement follows the normal rules of algebra *as long as overflow does not occur.* That is, this statement is valid only for

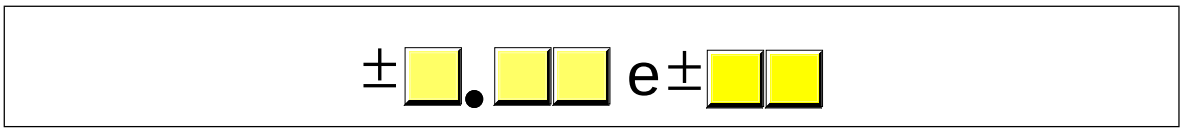

Figure 14.1 Simple Floating Point Format

certain values of x (minint <= x < maxint). Most programmers do not have a problem with this because they are well aware of the fact that integers in a program do not follow the standard algebraic rules (e.g., $5 / 2 \neq 2.5$).

Integers do not follow the standard rules of algebra because the computer represents them with a finite number of bits. You cannot represent any of the (integer) values above the maximum integer or below the minimum integer. Floating point values suffer from this same problem, only worse. After all, the integers are a subset of the real numbers. Therefore, the floating point values must represent the same infinite set of integers. However, there are an infinite number of values between any two real values, so this problem is infinitely worse. Therefore, as well as having to limit your values between a maximum and minimum range, you cannot represent all the values between those two ranges, either.

To represent real numbers, most floating point formats employ scientific notation and use some number of bits to represent a *mantissa* and a smaller number of bits to represent an *exponent*. The end result is that floating point numbers can only represent numbers with a specific number of *significant* digits. This has a big impact on how floating point arithmetic operations. To easily see the impact of limited precision arithmetic, we will adopt a simplified decimal floating point format for our examples. Our floating point format will provide a mantissa with three significant digits and a decimal exponent with two digits. The mantissa and exponents are both signed values (see Figure 14.1).

When adding and subtracting two numbers in scientific notation, you must adjust the two values so that their exponents are the same. For example, when adding 1.23e1 and 4.56e0, you must adjust the values so they have the same exponent. One way to do this is to to convert 4.56e0 to 0.456e1 and then add. This produces 1.686e1. Unfortunately, the result does not fit into three significant digits, so we must either *round* or *truncate* the result to three significant digits. Rounding generally produces the most accurate result, so let's round the result to obtain 1.69e1. As you can see, the lack of *precision* (the number of digits or bits we maintain in a computation) affects the accuracy (the correctness of the computation).

In the previous example, we were able to round the result because we maintained *four* significant digits *during* the calculation. If our floating point calculation is limited to three significant digits *during* computation, we would have had to truncate the last digit of the smaller number, obtaining 1.68e1 which is even less correct. Extra digits available during a computation are known as *guard digits* (or *guard bits* in the case of a binary format). They greatly enhance accuracy during a long chain of computations.

The accuracy loss during a single computation usually isn't enough to worry about unless you are greatly concerned about the accuracy of your computations. However, if you compute a value which is the result of a sequence of floating point operations, the error can *accumulate* and greatly affect the computation itself. For example, suppose we were to add 1.23e3 with 1.00e0. Adjusting the numbers so their exponents are the same before the addition produces 1.23e3 + 0.001e3. The sum of these two values, even after rounding, is 1.23e3. This might seem perfectly reasonable to you; after all, we can only maintain three significant digits, adding in a small value shouldn't affect the result at all. However, suppose we were to add 1.00e0 1.23e3 *ten times*. The first time we add 1.00e0 to 1.23e3 we get 1.23e3. Likewise, we get this same result the second, third, fourth, ..., and tenth time we add 1.00e0 to 1.23e3. On the other hand, had we added 1.00e0 to itself ten times, then added the result (1.00e1) to 1.23e3, we would have gotten a different result, 1.24e3. This is the most important thing to know about limited precision arithmetic:

The order of evaluation can effect the accuracy of the result.

You will get more accurate results if the relative magnitudes (that is, the exponents) are close to one another. If you are performing a chain calculation involving addition and subtraction, you should attempt to group the values appropriately.

Another problem with addition and subtraction is that you can wind up with *false precision*. Consider the computation 1.23e0 - 1.22 e0. This produces 0.01e0. Although this is mathematically equivalent to 1.00e-2, this latter form suggests that the last two digits are exactly zero. Unfortunately, we've only got a single significant digit at this time. Indeed, some FPUs or floating point software packages might actually insert random digits (or bits) into the L.O. positions. This brings up a second important rule concerning limited precision arithmetic:

> *Whenever subtracting two numbers with the same signs or adding two numbers with different signs, the accuracy of the result may be less than the precision available in the floating point format.*

Multiplication and division do not suffer from the same problems as addition and subtraction since you do not have to adjust the exponents before the operation; all you need to do is add the exponents and multiply the mantissas (or subtract the exponents and divide the mantissas). By themselves, multiplication and division do not produce particularly poor results. However, they tend to multiply any error which already exists in a value. For example, if you multiply 1.23e0 by two, when you should be multiplying 1.24e0 by two, the result is even less accurate. This brings up a third important rule when working with limited precision arithmetic:

> *When performing a chain of calculations involving addition, subtraction, multiplication, and division, try to perform the multiplication and division operations first.*

Often, by applying normal algebraic transformations, you can arrange a calculation so the multiply and divide operations occur first. For example, suppose you want to compute x*(y+z). Normally you would add y and z together and multiply their sum by x. However, you will get a little more accuracy if you transform x*(y+z) to get x*y+x*z and compute the result by performing the multiplications first.

Multiplication and division are not without their own problems. When multiplying two very large or very small numbers, it is quite possible for *overflow* or *underflow* to occur. The same situation occurs when dividing a small number by a large number or dividing a large number by a small number. This brings up a fourth rule you should attempt to follow when multiplying or dividing values:

> *When multiplying and dividing sets of numbers, try to arrange the multiplications so that they multiply large and small numbers together; likewise, try to divide numbers that have the same relative magnitudes.*

Comparing floating pointer numbers is very dangerous. Given the inaccuracies present in any computation (including converting an input string to a floating point value), you should *never* compare two floating point values to see if they are equal. In a binary floating point format, different computations which produce the same (mathematical) result may differ in their least significant bits. For example, adding 1.31e0+1.69e0 should produce 3.00e0. Likewise, adding 2.50e0+1.50e0 should produce 3.00e0. However, were you to compare (1.31e0+1.69e0) agains (2.50e0+1.50e0) you might find out that these sums are *not* equal to one another. The test for equality succeeds if and only if all bits (or digits) in the two operands are exactly the same. Since this is not necessarily true after two different floating point computations which should produce the same result, a straight test for equality may not work.

The standard way to test for equality between floating point numbers is to determine how much error (or tolerance) you will allow in a comparison and check to see if one value is within this error range of the other. The straight-forward way to do this is to use a test like the following:

```
if Value1 >= (Value2-error) and Value1 <= (Value2+error) then …
```

Another common way to handle this same comparison is to use a statement of the form:

```
if abs(Value1-Value2) <= error then …
```

Most texts, when discussing floating point comparisons, stop immediately after discussing the problem with floating point equality, assuming that other forms of comparison are perfectly okay with floating point numbers. This isn't true! If we are assuming that x=y if x is within y±error, then a simple bitwise comparison of x and y will claim that x<y if y is greater than x but less than y+error. However, in such a case x should really be treated as equal to y, not less than y. Therefore, we must always compare two floating point numbers using ranges, regardless of the actual comparison we want to perform. Trying to compare two floating point numbers directly can lead to an error. To compare two floating point numbers, x and y, against one another, you should use one of the following forms:

```
=         if abs(x-y) <= error then …
≠         if abs(x-y) > error then …
<         if (x-y) < error then …
≤         if (x-y) <= error then …
>         if (x-y) > error then …
≥         if (x-y) >= error then …
```

You must exercise care when choosing the value for *error*. This should be a value slightly greater than the largest amount of error which will creep into your computations. The exact value will depend upon the particular floating point format you use, but more on that a little later. The final rule we will state in this section is

> *When comparing two floating point numbers, always compare one value to see if it is in the range given by the second value plus or minus some small error value.*

There are many other little problems that can occur when using floating point values. This text can only point out some of the major problems and make you aware of the fact that you cannot treat floating point arithmetic like real arithmetic – the inaccuracies present in limited precision arithmetic can get you into trouble if you are not careful. A good text on numerical analysis or even scientific computing can help fill in the details which are beyond the scope of this text. If you are going to be working with floating point arithmetic, *in any language*, you should take the time to study the effects of limited precision arithmetic on your computations.

14.2 IEEE Floating Point Formats

When Intel planned to introduce a floating point coprocessor for their new 8086 microprocessor, they were smart enough to realize that the electrical engineers and solid-state physicists who design chips were, perhaps, not the best people to do the necessary numerical analysis to pick the best possible binary representation for a floating point format. So Intel went out and hired the best numerical analyst they could find to design a floating point format for their 8087 FPU. That person then hired two other experts in the field and the three of them (Kahn, Coonan, and Stone) designed Intel's floating point format. They did such a good job designing the KCS Floating Point Standard that the IEEE organization adopted this format for the IEEE floating point format[1].

To handle a wide range of performance and accuracy requirements, Intel actually introduced *three* floating point formats: single precision, double precision, and extended precision. The single and double precision formats corresponded to C's float and double types or FORTRAN's real and double precision types. Intel intended to use extended precision for long chains of computations. Extended precision contains 16 extra bits that the

1. There were some minor changes to the way certain degenerate operations were handled, but the bit representation remained essentially unchanged.

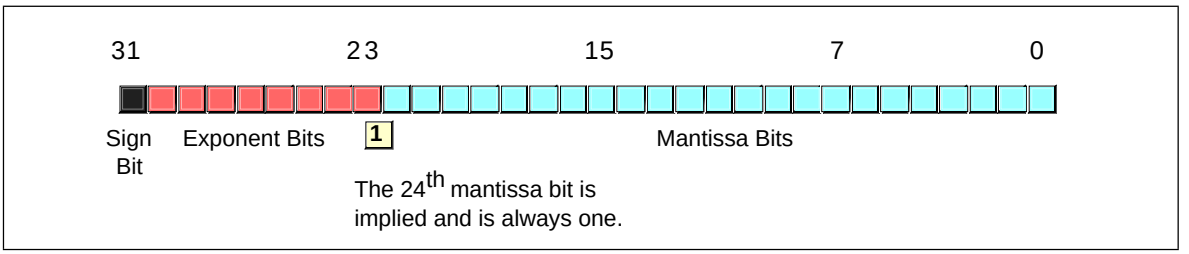

Figure 14.2 32 Bit Single Precision Floating Point Format

calculations could use for guard bits before rounding down to a double precision value when storing the result.

The single precision format uses a one's complement 24 bit mantissa and an eight bit excess-128 exponent. The mantissa usually represents a value between 1.0 to just under 2.0. The H.O. bit of the mantissa is always assumed to be one and represents a value just to the left of the *binary point*[2]. The remaining 23 mantissa bits appear to the right of the binary point. Therefore, the mantissa represents the value:

```
1.mmmmmmm mmmmmmmm mmmmmmmm
```

The "mmmm…" characters represent the 23 bits of the mantissa. Keep in mind that we are working with binary numbers here. Therefore, each position to the right of the binary point represents a value (zero or one) times a successive negative power of two. The implied one bit is always multiplied by 2^0, which is one. This is why the mantissa is always greater than or equal to one. Even if the other mantissa bits are all zero, the implied one bit always gives us the value one[3]. Of course, even if we had an almost infinite number of one bits after the binary point, they still would not add up to two. This is why the mantissa can represent values in the range one to just under two.

Although there are an infinite number of values between one and two, we can only represent eight million of them because we a 23 bit mantissa (the 24^{th} bit is always one). This is the reason for inaccuracy in floating point arithmetic – we are limited to 23 bits of precision in compuations involving single precision floating point values.

The mantissa uses a *one's complement* format rather than two's complement. This means that the 24 bit value of the mantissa is simply an unsigned binary number and the sign bit determines whether that value is positive or negative. One's complement numbers have the unusual property that there are two representations for zero (with the sign bit set or clear). Generally, this is important only to the person designing the floating point software or hardware system. We will assume that the value zero always has the sign bit clear.

To represent values outside the range 1.0 to just under 2.0, the exponent portion of the floating point format comes into play. The floating point format raise two to the power specified by the exponent and then multiplies the mantissa by this value. The exponent is eight bits and is stored in an *excess-127* format. In excess-127 format, the exponent 2^0 is represented by the value 127 (7fh). Therefore, to convert an exponent to excess-127 format simply add 127 to the exponent value. The use of excess-127 format makes it easier to compare floating point values. The single precision floating point format takes the form shown in Figure 14.2.

With a 24 bit mantissa, you will get approximately $6\text{-}^1/_2$ digits of precision (one half digit of precision means that the first six digits can all be in the range 0..9 but the seventh digit can only be in the range 0..x where x<9 and is generally close to five). With an eight

2. The binary point is the same thing as the decimal point except it appears in binary numbers rather than decimal numbers.

3. Actually, this isn't necessarily true. Thye IEEE floating point format supports *denormalized* values where the H.O. bit is not zero. However, we will ignore denormalized values in our discussion.

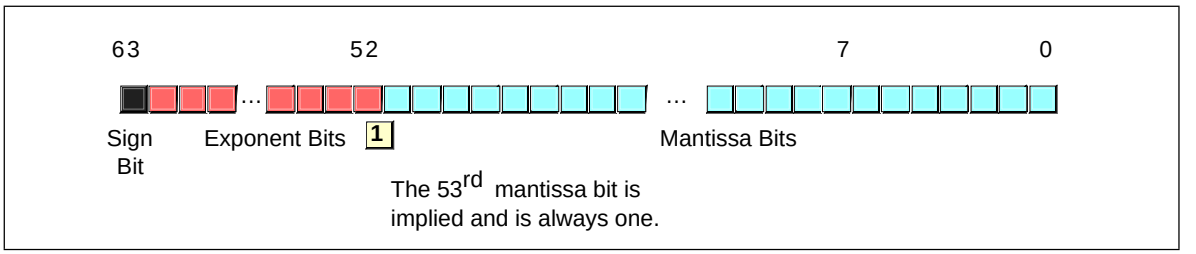

Figure 14.3 64 Bit Double Precision Floating Point Format

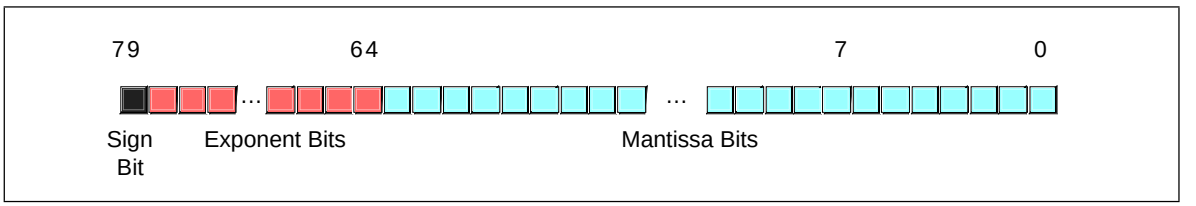

Figure 14.4 80 Bit Extended Precision Floating Point Format

bit excess-128 exponent, the dynamic range of single precision floating point numbers is approximately $2^{\pm 128}$ or about $10^{\pm 38}$.

Although single precision floating point numbers are perfectly suitable for many applications, the dynamic range is somewhat small for many scientific applications and the very limited precision is unsuitable for many financial, scientific, and other applications. Furthermore, in long chains of computations, the limited precision of the single precision format may introduce serious error.

The double precision format helps overcome the problems of single preicision floating point. Using twice the space, the double precision format has an 11-bit excess-1023 exponent and a 53 bit mantissa (with an implied H.O. bit of one) plus a sign bit. This provides a dynamic range of about $10^{\pm 308}$and $14\text{-}^{1}/_{2}$ digits of precision, sufficient for most applications. Double precision floating point values take the form shown in Figure 14.3.

In order to help ensure accuracy during long chains of computations involving double precision floating point numbers, Intel designed the extended precision format. The extended precision format uses 80 bits. Twelve of the additional 16 bits are appended to the mantissa, four of the additional bits are appended to the end of the exponent. Unlike the single and double precision values, the extended precision format does not have an implied H.O. bit which is always one. Therefore, the extended precision format provides a 64 bit mantissa, a 15 bit excess-16383 exponent, and a one bit sign. The format for the extended precision floating point value is shown in Figure 14.4.

On the 80x87 FPUs and the 80486 CPU, all computations are done using the extended precision form. Whenever you load a single or double precision value, the FPU automatically converts it to an extended precision value. Likewise, when you store a single or double precision value to memory, the FPU automatically rounds the value down to the appropriate size before storing it. By always working with the extended precision format, Intel guarantees a large number of guard bits are present to ensure the accuracy of your computations. Some texts erroneously claim that you should never use the extended precision format in your own programs, because Intel only guarantees accurate computations when using the single or double precision formats. This is foolish. By performing all computations using 80 bits, Intel helps ensure (but not guarantee) that you will get full 32 or 64 bit accuracy in your computations. Since the 80x87 FPUs and 80486 CPU do not provide a large number of guard bits in 80 bit computations, some error will inevitably creep into the L.O. bits of an extended precision computation. However, if your computation is correct to 64 bits, the 80 bit computation will always provide *at least* 64 accurate bits. Most of the time you will get even more. While you cannot assume that you get an accurate 80

bit computation, you can usually do better than 64 when using the extended precision format.

To maintain maximum precision during computation, most computations use *normalized* values. A normalized floating point value is one that has a H.O. mantissa bit equal to one. Almost any non-normalized value can be normalized by shifting the mantissa bits to the left and decrementing the exponent by one until a one appears in the H.O. bit of the mantissa. Remember, the exponent is a binary exponent. Each time you increment the exponent, you multiply the floating point value by two. Likewise, whenever you decrement the exponent, you divide the floating point value by two. By the same token, shifting the mantissa to the left one bit position multiplies the floating point value by two; likewise, shifting the mantissa to the right divides the floating point value by two. Therefore, shifting the mantissa to the left one position *and* decrementing the exponent does not change the value of the floating point number at all.

Keeping floating point numbers normalized is beneficial because it maintains the maximum number of bits of precision for a computation. If the H.O. bits of the mantissa are all zero, the mantissa has that many fewer bits of precision available for computation. Therefore, a floating point computation will be more accurate if it involves only normalized values.

There are two important cases where a floating point number cannot be normalized. The value 0.0 is a special case. Obviously it cannot be normalized because the floating point representation for zero has no one bits in the mantissa. This, however, is not a problem since we can exactly represent the value zero with only a single bit.

The second case is when we have some H.O. bits in the mantissa which are zero but the biased exponent is also zero (and we cannot decrement it to normalize the mantissa). Rather than disallow certain small values, whose H.O. mantissa bits and biased exponent are zero (the most negative exponent possible), the IEEE standard allows special *denormalized* values to represent these smaller values[4]. Although the use of denormalized values allows IEEE floating point computations to produce better results than if underflow occurred, keep in mind that denormalized values offer less bits of precision and are inherently less accurate.

Since the 80x87 FPUs and 80486 CPU always convert single and double precision values to extended precision, extended precision arithmetic is actually *faster* than single or double precision. Therefore, the expected performance benefit of using the smaller formats is not present on these chips. However, when designing the Pentium / 586 CPU, Intel redesigned the built-in floating point unit to better compete with RISC chips. Most RISC chips support a native 64 bit double precision format which is faster than Intel's extended precision format. Therefore, Intel provided native 64 bit operations on the Pentium to better compete against the RISC chips. Therefore, the double precision format is the fastest on the Pentium and later chips.

14.3 The UCR Standard Library Floating Point Routines

In most assembly language texts, which bother to cover floating point arithmetic, this section would normally describe how to design your own floating point routines for addition, subtraction, multiplication, and division. This text will not do that for several reasons. First, to design a *good* floating point library requires a solid background in numerical analysis; a prerequisite this text does not assume of its readers. Second, the UCR Standard Library already provides a reasonable set of floating point routines in source code form; why waste space in this text when the sources are readily available elsewhere? Third, floating point units are quickly becoming standard equipment on all modern CPUs or motherboards; it makes no more sense to describe how to manually perform a floating point computation than it does to describe how to manually perform an integer computation. Therefore, this section will describe how to use the UCR Standard Library routines if

4. The alternative would be to underflow the values to zero.

you do not have an FPU available; a later section will describe the use of the floating point unit.

The UCR Standard Library provides a large number of routines to support floating point computation and I/O. This library uses the same memory format for 32, 64, and 80 bit floating point numbers as the 80x87 FPUs. The UCR Standard Library's floating point routines do not exactly follow the IEEE requirements with respect to error conditions and other degenerate cases, and it may produce slightly different results than an 80x87 FPU, but the results will be very close[5]. Since the UCR Standard Library uses the same memory format for 32, 64, and 80 bit numbers as the 80x87 FPUs, you can freely mix computations involving floating point between the FPU and the Standard Library routines.

The UCR Standard Library provides numerous routines to manipulate floating point numbes. The following sections describe each of these routines, by category.

14.3.1 Load and Store Routines

Since 80x86 CPUs without an FPU do not provide any 80-bit registers, the UCR Standard Library must use memory-based variables to hold floating point values during computation. The UCR Standard Library routines use two *pseudo registers*, an accumlator register and an operand register, when performing floating point operations. For example, the floating point addition routine adds the value in the floating point operand register to the floating point accumulator register, leaving the result in the accumulator. The load and store routines allow you to load floating point values into the floating point accumulator and operand registers as well as store the value of the floating point accumulator back to memory. The routines in this category include accop, xaccop, lsfpa, ssfpa, ldfpa, sdfpa, lefpa, sefpa, lefpal, lsfpo, ldfpo, lefpo, and lefpol.

The accop routine copies the value in the floating point accumulator to the floating point operand register. This routine is useful when you want to use the result of one computation as the second operand of a second computation.

The xaccop routine exchanges the values in the floating point accumuator and operand registers. Note that many floating point computations destory the value in the floating point operand register, so you cannot blindly assume that the routines preserve the operand register. Therefore, calling this routine only makes sense after performing some computation which you know does not affect the floating point operand register.

Lsfpa, ldfpa, and lefpa load the floating point accumulator with a single, double, or extended precision floating point value, respectively. The UCR Standard Library uses its own internal format for computations. These routines convert the specified values to the internal format during the load. On entry to each of these routines, es:di must contain the address of the variable you want to load into the floating point accumulator. The following code demonstrates how to call these routines:

```
rVar            real4    1.0
drVar           real8    2.0
xrVar           real10   3.0
                  .
                  .
                  .
                lesi     rVar
                lsfpa
                  .
                  .
                  .
                lesi     drVar
                ldfpa
                  .
                  .
                  .
```

5. Note, by the way, that different floating point chips, especially across different CPU lines, but even within the Intel family, produce slightly different results. So the fact that the UCR Standard Library does not produce the exact same results as a particular FPU is not that important.

```
                lesi    xrVar
                lefpa
```

The lsfpo, ldfpo, and lefpo routines are similar to the lsfpa, ldfpa, and lefpa routines except, of course, they load the floating point operand register rather than the floating point accumulator with the value at address es:di.

Lefpal and lefpol load the floating point accumulator or operand register with a literal 80 bit floating point constant appearing in the code stream. To use these two routines, simply follow the call with a real10 directive and the appropriate constant, e.g.,

```
                lefpal
                real10  1.0
                lefpol
                real10  2.0e5
```

The ssfpa, sdfpa, and sefpa routines store the value in the floating point accumulator into the memory based floating point variable whose address appears in es:di. There are no corresponding ssfpo, sdfpo, or sefpo routines because a result you would want to store should never appear in the floating point operand register. If you happen to get a value in the floating point operand that you want to store into memory, simply use the xaccop routine to swap the accumulator and operand registers, then use the store accumulator routines to save the result. The following code demonstrates the use of these routines:

```
rVar            real4   1.0
drVar           real8   2.0
xrVar           real10  3.0
                  .
                  .
                  .
                lesi    rVar
                ssfpa
                  .
                  .
                  .
                lesi    drVar
                sdfpa
                  .
                  .
                  .
                lesi    xrVar
                sefpa
```

14.3.2 Integer/Floating Point Conversion

The UCR Standard Library includes several routines to convert between binary integers and floating point values. These routines are itof, utof, ltof, ultof, ftoi, ftou, ftol, and ftoul. The first four routines convert signed and unsigned integers to floating point format, the last four routines truncate floating point values and convert them to an integer value.

Itof converts the signed 16-bit value in ax to a floating point value and leaves the result in the floating point accumulator. This routine does not affect the floating point operand register. Utof converts the unsigned integer in ax in a similar fashion. Ltof and ultof convert the 32 bit signed (ltof) or unsigned (ultof) integer in dx:ax to a floating point value, leaving the value in the floating point accumulator. These routines always succeed.

Ftoi converts the value in the floating point accumulator to a signed integer value, leaving the result in ax. Conversion is by truncation; this routine keeps the integer portion and throws away the fractional part. If an overflow occurs because the resulting integer portion does not fit into 16 bits, ftoi returns the carry flag set. If the conversion occurs without error, ftoi return the carry flag clear. Ftou works in a similar fashion, except it converts the floating point value to an unsigned integer in ax; it returns the carry set if the floating point value was negative.

Ftol and ftoul converts the value in the floating point accumulator to a 32 bit integer leaving the result in dx:ax. Ftol works on signed values, ftoul works with unsigned values. As with ftoi and ftou, these routines return the carry flag set if a conversion error occurs.

14.3.3 Floating Point Arithmetic

Floating point arithmetic is handled by the fpadd, fp sub, fpcmp, fpmul, and fpdiv routines. Fpadd adds the value in the floating point accumulator to the floating point accumulator. Fpsub subtracts the value in the floating point operand from the floating point accumulator. Fpmul multiplies the value in the floating accumulator by the floating point operand. Fpdiv divides the value in the floating point accumulator by the value in the floating point operand register. Fpcmp compares the value in the floating point accumulator against the floating point operand.

The UCR Standard Library arithmetic routines do very little error checking. For example, if arithmetic overflow occurs during addition, subtraction, multiplication, or division, the Standard Library simply sets the result to the largest legal value and returns. This is one of the major deviations from the IEEE floating point standard. Likewise, when underflow occurs the routines simply set the result to zero and return. If you divide any value by zero, the Standard Library routines simply set the result to the largest possible value and return. You may need to modify the standard library routines if you need to check for overflow, underflow, or division by zero in your programs.

The floating point comparison routine (fpcmp) compares the floating point accumulator against the floating point operand and returns -1, 0, or 1 in the ax register if the accumulator is less than, equal, or greater than the floating point operand. It also compares ax with zero immediately before returning so it sets the flags so you can use the jg, jge, jl, jle, je, and jne instructions immediately after calling fpcmp. Unlike fpadd, fpsub, fpmul, and fpdiv, fpcmp does not destroy the value in the floating point accumulator or the floating point operand register. Keep in mind the problems associated with comparing floating point numbers!

14.3.4 Float/Text Conversion and Printff

The UCR Standard Library provides three routines, ftoa, etoa, and atof, that let you convert floating point numbers to ASCII strings and vice versa; it also provides a special version of printf, printff, that includes the ability to print floating point values as well as other data types.

Ftoa converts a floating point number to an ASCII string which is a decimal representation of that floating point number. On entry, the floating point accumulator contains the number you want to convert to a string. The es:di register pair points at a buffer in memory where ftoa will store the string. The al register contains the field width (number of print positions). The ah register contains the number of positions to display to the right of the decimal point. If ftoa cannot display the number using the print format specified by al and ah, it will create a string of "#" characters, ah characters long. Es:di must point at a byte array containing at least al+1 characters and al should contain at least five. The field width and decimal length values in the al and ah registers are similar to the values appearing after floating point numbers in the Pascal write statement, e.g.,

```
write(floatVal:al:ah);
```

Etoa outputs the floating point number in exponential form. As with ftoa, es:di points at the buffer where etoa will store the result. The al register must contain at least eight and is the field width for the number. If al contains less than eight, etoa will output a string of "#" characters. The string that es:di points at must contain at least al+1 characters. This conversion routine is similar to Pascal's write procedure when writing real values with a single field width specification:

```
write(realvar:al);
```

The Standard Library printff routine provides all the facilities of the standard printf routine plus the ability to handle floating point output. The printff routine includes sev-

eral new format specifications to print floating point numbers in decimal form or using scientific notation. The specifications are

- %x.yF Prints a 32 bit floating point number in decimal form.
- %x.yGF Prints a 64 bit floating point number in decimal form.
- %x.yLF Prints an 80 bit floating point number in decimal form.
- %zE Prints a 32 bit floating point number using scientific notation.
- %zGE Prints a 64 bit floating point number using scientific notation.
- %zLE Prints an 80 bit floating point value using scientific notation.

In the format strings above, *x* and *z* are integer constants that denote the field width of the number to print. The *y* item is also an integer constant that specifies the number of positions to print after the decimal point. The *x.y* values are comparable to the values passed to ftoa in al and ah. The *z* value is comparable to the value etoa expects in the al register.

Other than the addition of these six new formats, the printff routine is identical to the printf routine. If you use the printff routine in your assembly language programs, you should *not* use the printf routine as well. Printff duplicates all the facilities of printf and using both would only waste memory.

14.4 The 80x87 Floating Point Coprocessors

When the 8086 CPU first appeared in the late 1970's, semiconductor technology was not to the point where Intel could put floating point instrutions directly on the 8086 CPU. Therefore, they devised a scheme whereby they could use a second chip to perform the floating point calculations – the floating point unit (or FPU)[6]. They released their original floating point chip, the 8087, in 1980. This particular FPU worked with the 8086, 8088, 80186, and 80188 CPUs. When Intel introduced the 80286 CPU, they released a redesigned 80287 FPU chip to accompany it. Although the 80287 was compatible with the 80386 CPU, Intel designed a better FPU, the 80387, for use in 80386 systems. The 80486 CPU was the first Intel CPU to include an on-chip floating point unit. Shortly after the release of the 80486, Intel introduced the 80486sx CPU that was an 80486 without the built-in FPU. To get floating point capabilities on this chip, you had to add an 80487 chip, although the 80487 was really nothing more than a full-blown 80486 which took over for the "sx" chip in the system. Intel's Pentium/586 chips provide a high-performance floating point unit directly on the CPU. There is no floating point coprocessor available for the Pentium chip.

Collectively, we will refer to all these chips as the 80x87 FPU. Given the obsolesence of the 8086, 80286, 8087, and 80287 chips, this text will concentrate on the 80387 and later chips. There are some differences between the 80387/80486/Pentium floating point units and the earlier FPUs. If you need to write code that will execute on those earlier machines, you should consult the appropriate Intel documentation for those devices.

14.4.1 FPU Registers

The 80x87 FPUs add 13 registers to the 80386 and later processors: eight floating point data registers, a control register, a status register, a tag register, an instruction pointer, and a data pointer. The data registers are similar to the 80x86's general purpose register set insofar as all floating point calculations take place in these registers. The control register contains bits that let you decide how the 80x87 handles certain degenerate cases like rounding of inaccurate computations, control precision, and so on. The status register is similar to the 80x86's flags register; it contains the condition code bits and several other floating point flags that describe the state of the 80x87 chip. The tag register contains several groups of bits that determine the state of the value in each of the eight general purpose registers. The instruction and data pointer registers contain certain state information

6. Intel has also refered to this device as the Numeric Data Processor (NDP), Numeric Processor Extension (NPX), and math coprocessor.

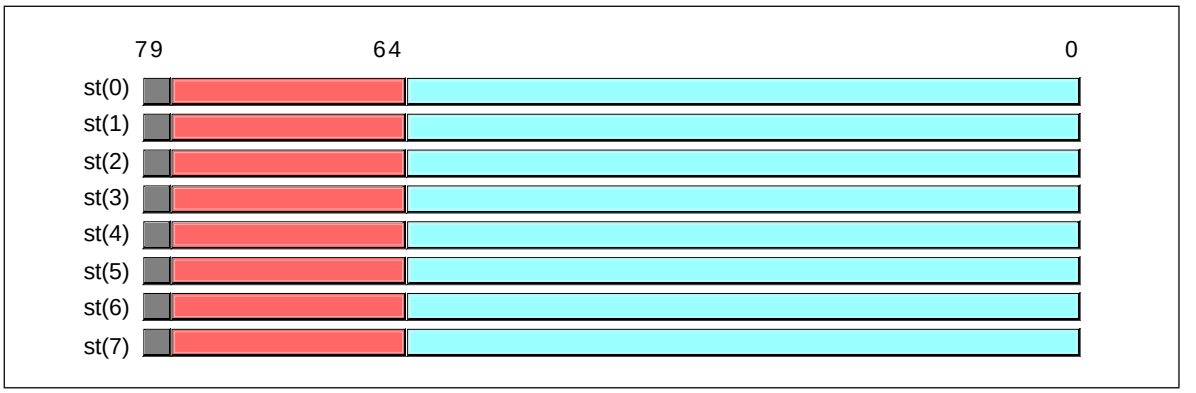

Figure 14.5 80x87 Floating Point Register Stack

about the last floating point instruction executed. We will not consider the last three registers in this text, see the Intel documentation for more details.

14.4.1.1 The FPU Data Registers

The 80x87 FPUs provide eight 80 bit data registers organized as a stack. This is a significant departure from the organization of the general purpose registers on the 80x86 CPU that comprise a standard general-purpose register set. Intel refers to these registers as ST(0), ST(1), …, ST(7). Most assemblers will accept ST as an abbreviation for ST(0).

The biggest difference between the FPU register set and the 80x86 register set is the stack organization. On the 80x86 CPU, the ax register is always the ax register, no matter what happens. On the 80x87, however, the register set is an eight element stack of 80 bit floating point values (see Figure 14.5). ST(0) refers to the item on the top of the stack, ST(1) refers to the next item on the stack, and so on. Many floating point instructions push and pop items on the stack; therefore, ST(1) will refer to the previous contents of ST(0) after you push something onto the stack. It will take some thought and practice to get used to the fact that the registers are changing under you, but this is an easy problem to overcome.

14.4.1.2 The FPU Control Register

When Intel designed the 80x87 (and, essentially, the IEEE floating point standard), there were no standards in floating point hardware. Different (mainframe and mini) computer manufacturers all had different and incompatible floating point formats. Unfortunately, much application software had been written taking into account the idiosyncrasies of these different floating point formats. Intel wanted to designed an FPU that could work with the majority of the software out there (keep in mind, the IBM PC was three to four years away when Intel began designing the 8087, they couldn't rely on that "mountain" of software available for the PC to make their chip popular). Unfortunately, many of the features found in these older floating point formats were mutually exclusive. For example, in some floating point systems rounding would occur when there was insufficient precision; in others, truncation would occur. Some applications would work with one floating point system but not with the other. Intel wanted as many applications as possible to work with as few changes as possible on their 80x87 FPUs, so they added a special register, the FPU *control register*, that lets the user choose one of several possible operating modes for the 80x87.

The 80x87 control register contains 16 bits organized as shown in Figure 14.6.

Bit 12 of the control register is only present on the 8087 and 80287 chips. It controls how the 80x87 responds to infinity. The 80387 and later chips always use a form of infinitly known and *affine closure* because this is the only form supported by the IEEE

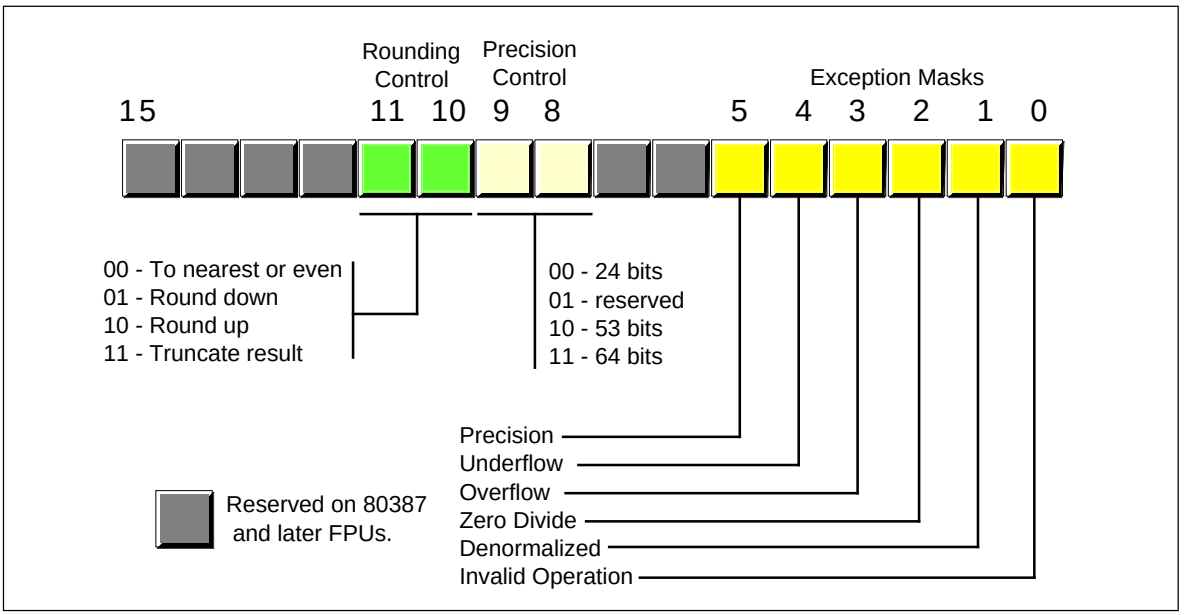

Figure 14.6 80x87 Control Register

754/854 standards. As such, we will ignore any further use of this bit and assume that it is always programmed with a one.

Bits 10 and 11 provide rounding control according to the following values:

Table 58: Rounding Control

Bits 10 & 11	Function
00	To nearest or even
01	Round down
10	Round up
11	Truncate

The “00” setting is the default. The 80x87 rounds values above one-half of the least significant bit up. It rounds values below one-half of the least significant bit down. If the value below the least significant bit is exactly one-half the least significant bit, the 80x87 rounds the value towards the value whose least significant bit is zero. For long strings of computations, this provides a reasonable, automatic, way to maintain maximum precision.

The round up and round down options are present for those computations where it is important to keep track of the accuracy during a computation. By setting the rounding control to round down and performing the operation, the repeating the operation with the rounding control set to round up, you can determine the minimum and maximum ranges between which the true result will fall.

The truncate option forces all computations to truncate any excess bits during the computation. You will rarely use this option if accuracy is important to you. However, if you are porting older software to the 80x87, you might use this option to help when porting the software.

Bits eight and nine of the control register control the precision during computation. This capability is provided mainly to allow compatbility with older software as required by the IEEE 754 standard. The precision control bits use the following values:

Table 59: Mantissa Precision Control Bits

Bits 8 & 9	Precision Control
00	24 bits
01	Reserved
10	53 bits
11	64 bits

For modern applications, the precision control bits should always be set to “11” to obtain 64 bits of precision. This will produce the most accurate results during numerical computation.

Bits zero through five are the *exception masks*. These are similar to the interrupt enable bit in the 80x86’s flags register. If these bits contain a one, the corresponding condition is ignored by the 80x87 FPU. However, if any bit contains zero, and the corresponding condition occurs, then the FPU immediately generates an interrupt so the program can handle the degenerate condition.

Bit zero corresponds to an invalid operation error. This generally occurs as the result of a programming error. Problem which raise the invalid operation exception include pushing more than eight items onto the stack or attempting to pop an item off an empty stack, taking the square root of a negative number, or loading a non-empty register.

Bit one masks the *denormalized* interrupt which occurs whenever you try to manipulate denormalized values. Denormalized values generally occur when you load arbitrary extended precision values into the FPU or work with very small numbers just beyond the range of the FPU’s capabilities. Normally, you would probably *not* enable this exception.

Bit two masks the *zero divide* exception. If this bit contains zero, the FPU will generate an interrupt if you attempt to divide a nonzero value by zero. If you do not enable the zero division exception, the FPU will produce NaN (not a number) whenever you perform a zero division.

Bit three masks the overflow exception. The FPU will raise the overflow exception if a calculation overflows or if you attempt to store a value which is too large to fit into a destination operand (e.g., storing a large extended precision value into a single precision variable).

Bit four, if set, masks the *underflow* exception. Underflow occurs when the result is too *small* to fit in the desintation operand. Like overflow, this exception can occur whenever you store a small extended precision value into a smaller variable (single or double precision) or when the result of a computation is too small for extended precision.

Bit five controls whether the *precision* exception can occur. A precision exception occurs whenever the FPU produces an imprecise result, generally the result of an internal rounding operation. Although many operations will produce an exact result, many more will not. For example, dividing one by ten will produce an inexact result. Therefore, this bit is usually one since inexact results are very common.

Bits six and thirteen through fifteen in the control register are currently undefined and reserved for future use. Bit seven is the interrupt enable mask, but it is only active on the 8087 FPU; a zero in this bit enables 8087 interrupts and a one disables FPU interrupts.

The 80x87 provides two instructions, FLDCW (load control word) and FSTCW (store control word), that let you load and store the contents of the control register. The single operand to these instructions must be a 16 bit memory location. The FLDCW instruction loads the control register from the specified memory location, FSTCW stores the control register into the specified memory location.

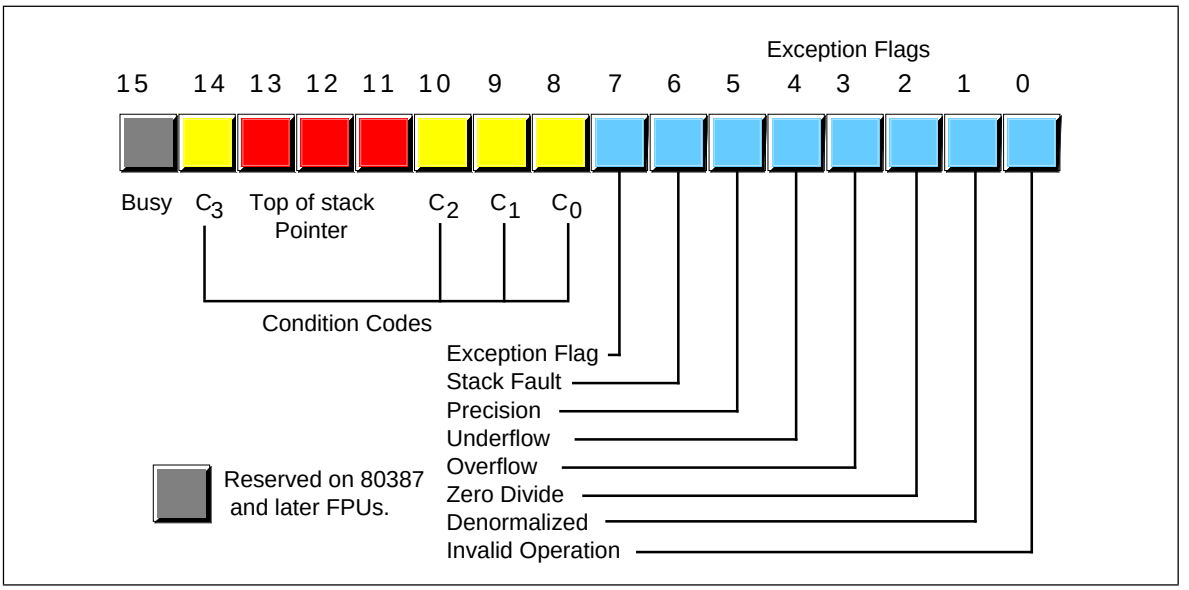

Figure 14.7 FPU Status Register

14.4.1.3 The FPU Status Register

The FPU status register provides the status of the coprocessor at the instant you read it. The FSTSW instruction stores the16 bit floating point status register into the mod/reg/rm operand. The status register s a 16 bit register, its layout appears in Figure 14.7.

Bits zero through five are the exception flags. These bits are appear in the same order as the exception masks in the control register. If the corresponding condition exists, then the bit is set. These bits are independent of the exception masks in the control register. The 80x87 sets and clears these bits regardless of the corresponding mask setting.

Bit six (active only on 80386 and later processors) indicates a *stack fault*. A stack fault occurs whenever there is a stack overflow or underflow. When this bit is set, the C_1 condition code bit determines whether there was a stack overflow (C_1=1) or stack underflow (C_1=0) condition.

Bit seven of the status register is set if *any* error condition bit is set. It is the logical OR of bits zero through five. A program can test this bit to quickly determine if an error condition exists.

Bits eight, nine, ten, and fourteen are the coprocessor condition code bits. Various instructions set the condition code bits as shown in the following table:

Table 60: FPU Condition Code Bits

Instruction	Condition Code Bits				Condition
	C3	C2	C1	C0	
fcom, fcomp, fcompp, ficom, ficomp	0	0	X	0	ST > source
	0	0	X	1	ST < source
	1	0	X	0	ST = source
	1	1	X	1	ST or source undefined
	X = Don't care				

Table 60: FPU Condition Code Bits

Instruction	Condition Code Bits				Condition
	C3	C2	C1	C0	
ftst	0	0	X	0	ST is positive
	0	0	X	1	ST is negative
	1	0	X	0	ST is zero (+ or -)
	1	1	X	1	ST is uncomparable
fxam	0	0	0	0	+ Unnormalized
	0	0	1	0	-Unnormalized
	0	1	0	0	+Normalized
	0	1	1	0	-Normalized
	1	0	0	0	+0
	1	0	1	0	-0
	1	1	0	0	+Denormalized
	1	1	1	0	-Denormalized
	0	0	0	1	+NaN
	0	0	1	1	-NaN
	0	1	0	1	+Infinity
	0	1	1	1	-Infinity
	1	X	X	1	Empty register
fucom, fucomp, fucompp	0	0	X	0	ST > source
	0	0	X	1	ST < source
	1	0	X	0	ST = source
	1	1	X	1	Unorder
	X = Don’t care				

Table 61: Condition Code Interpretation

Insruction(s)	C_0	C_3	C_2	C_1
fcom, fcomp, fcmpp, ftst, fucom, fucomp, fucompp, ficom, ficomp	Result of comparison. See table above.	Result of comparison. See table above.	Operand is not comparable.	Result of comparison (see table above) or stack overflow / underflow (if stack exception bit is set).
fxam	See previous table.	See previous table.	See previous table.	Sign of result, or stack overflow / underflow (if stack exception bit is set).
fprem, fprem1	Bit 2 of remainder	Bit 0 of remainder	0- reduction done. 1- reduction incomplete.	Bit 1 of remainder or stack overflow / underflow (if stack exception bit is set).
fist, fbstp, frndint, fst, fstp, fadd, fmul, fdiv, fdivr, fsub, fsubr, fscale, fsqrt, fpatan, f2xm1, fyl2x, fyl2xp1	Undefined	Undefined	Undefined	Round up occurred or stack overflow / underflow (if stack exception bit is set).
fptan, fsin, fcos, fsincos	Undefined	Undefined	0- reduction done. 1- reduction incomplete.	Round up occurred or stack overflow / underflow (if stack exception bit is set).
fchs, fabs, fxch, fincstp, fdecstp, *constant loads* , fxtract, fld, fild, fbld, fstp (80 bit)	Undefined	Undefined	Undefined	Zero result or stack overflow / underflow (if stack exception bit is set).
fldenv, fstor	Restored from memory operand.	Restored from memory operand.	Restored from memory operand.	Restored from memory operand.
fldcw, fstenv, fstcw, fstsw, fclex	Undefined	Undefined	Undefined	Undefined
finit, fsave	Cleared to zero.	Cleared to zero.	Cleared to zero.	Cleared to zero.

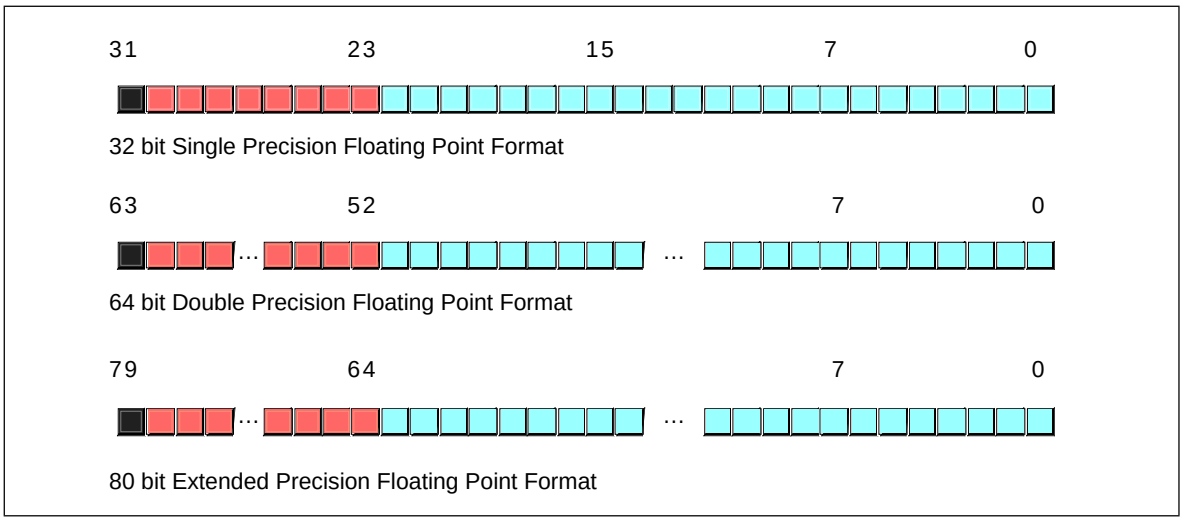

Figure 14.8 80x87 Floating Point Formats

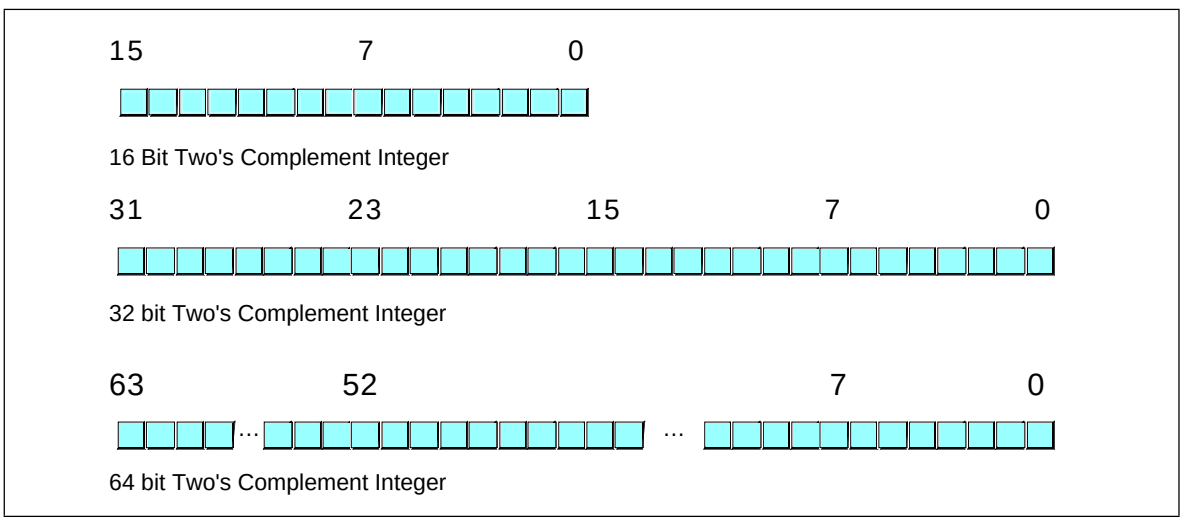

Figure 14.9 80x87 Integer Formats

Bits 11-13 of the FPU status register provide the register number of the top of stack. During computations, the 80x87 adds (modulo eight) the *logical* register numbers supplied by the programmer to these three bits to determine the *physical* register number at run time.

Bit 15 of the status register is the *busy* bit. It is set whenever the FPU is busy. Most programs will have little reason to access this bit.

14.4.2 FPU Data Types

The 80x87 FPU supports seven different data types: three integer types, a packed decimal type, and three floating point types. Since the 80x86 CPUs already support integer data types, these are few reasons why you would want to use the 80x87 integer types. The packed decimal type provides a 17 digit signed decimal (BCD) integer. However, we are avoiding BCD arithmetic in this text, so we will ignore this data type in the 80x87 FPU. The remaining three data types are the 32 bit, 64 bit, and 80 bit floating point data types we've looked at so far. The 80x87 data types appear in Figure 14.8, Figure 14.9, and Figure 14.10.

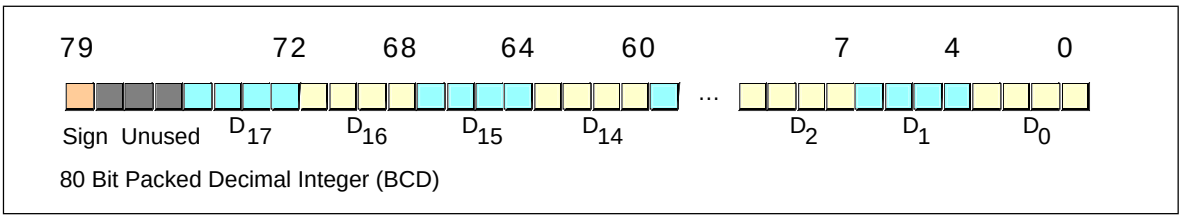

Figure 14.10 80x87 Packed Decimal Formats

The 80x87 FPU generally stores values in a *normalized* format. When a floating point number is normalized, the H.O. bit is always one. In the 32 and 64 bit floating point formats, the 80x87 does not actually store this bit, the 80x87 always assumes that it is one. Therefore, 32 and 64 bit floating point numbers are always normalized. In the extended precision 80 bit floating point format, the 80x87 does *not* assume that the H.O. bit of the mantissa is one, the H.O. bit of the number appears as part of the string of bits.

Normalized values provide the greatest precision for a given number of bits. However, there are a large number of non-normalized values which we *can* represent with the 80 bit format. These values are very close to zero and represent the set of values whose mantissa H.O. bit is not zero. The 80x87 FPUs support a special form of 80 bit known as *denormalized* values. Denormalized values allow the 80x87 to encode very small values it cannot encode using normalized values, but at a price. Denormalized values offer less bits of precision than normalized values. Therefore, using denormalized values in a computation may introduce some slight inaccuracy into a computation. Of course, this is always better than underflowing the denormalized value to zero (which could make the computation even less accurate), but you must keep in mind that if you work with very small values you may lose some accuracy in your computations. Note that the 80x87 status register contains a bit you can use to detect when the FPU uses a denormalized value in a computation.

14.4.3 The FPU Instruction Set

The 80387 (and later) FPU adds over 80 new instructions to the 80x86 instruction set. We can classify these instructions as *data movement instructions, conversions, arithmetic instructions, comparisons, constant instructions, transcendental instructions,* and *miscellaneous instructions.* The following sections describe each of the instructions in these categories.

14.4.4 FPU Data Movement Instructions

The data movement instructions transfer data between the internal FPU registers and memory. The instructions in this category are fld, fst, fstp, and fxch. The fld instructions always pushes its operand onto the floating point stack. The fstp instruction always pops the top of stack after storing the top of stack (tos) into its operation. The remaining instructions do not affect the number of items on the stack.

14.4.4.1 The FLD Instruction

The fld instruction loads a 32 bit, 64 bit, or 80 bit floating point value onto the stack. This instruction converts 32 and 64 bit operand to an 80 bit extended precision value before pushing the value onto the floating point stack.

The fld instruction first decrements the tos pointer (bits 11-13 of the status register) and then stores the 80 bit value in the physical register specified by the new tos pointer. If the source operand of the fld instruction is a floating point data register, ST(*i*), then the actual

register the 80x87 uses for the load operation is the register number *before* decrementing the tos pointer. Therefore, fld st or fld st(0) duplicates the value on the top of the stack.

The fld instruction sets the stack fault bit if stack overflow occurs. It sets the the denormalized exception bit if you load an 80 bit denormalized value. It sets the invalid operation bit if you attempt to load an empty floating point register onto the stop of stack (or perform some other invalid operation).

Examples:

```
fld     st(1)
fld     mem_32
fld     MyRealVar
fld     mem_64[bx]
```

14.4.4.2 The FST and FSTP Instructions

The fst and fstp instructions copy the value on the top of the floating point register stack to another floating point register or to a 32, 64, or 80 bit memory variable. When copying data to a 32 or 64 bit memory variable, the 80 bit extended precision value on the top of stack is rounded to the smaller format as specified by the rounding control bits in the FPU control register.

The fstp instruction pops the value off the top of stack when moving it to the destination location. It does this by incrementing the top of stack pointer in the status register after accessing the data in st(0). If the destination operand is a floating point register, the FPU stores the value at the specified register number *before* popping the data off the top of the stack.

Executing an fstp st(0) instruction effectively pops the data off the top of stack with no data transfer. Examples:

```
fst     mem_32
fstp    mem_64
fstp    mem_64[ebx*8]
fst     mem_80
fst     st(2)
fstp    st(1)
```

The last example above effectively pops st(1) while leaving st(0) on the top of the stack.

The fst and fstp instructions will set the stack exception bit if a stack underflow occurs (attempting to store a value from an empty register stack). They will set the precision bit if there is a loss of precision during the store operation (this will occur, for example, when storing an 80 bit extended precision value into a 32 or 64 bit memory variable and there are some bits lost during conversion). They will set the underflow exception bit when storing an 80 bit value value into a 32 or 64 bit memory variable, but the value is too small to fit into the destination operand. Likewise, these instructions will set the overflow exception bit if the value on the top of stack is too big to fit into a 32 or 64 bit memory variable. The fst and fstp instructions set the denormalized flag when you try to store a denormalized value into an 80 bit register or variable[7]. They set the invalid operation flag if an invalid operation (such as storing into an empty register) occurs. Finally, these instructions set the C_1 condition bit if rounding occurs during the store operation (this only occurs when storing into a 32 or 64 bit memory variable and you have to round the mantissa to fit into the destination).

14.4.4.3 The FXCH Instruction

The fxch instruction exchanges the value on the top of stack with one of the other FPU registers. This instruction takes two forms: one with a single FPU register as an operand,

7. Storing a denormalized value into a 32 or 64 bit memory variable will always set the underflow exception bit.

the second without any operands. The first form exchanges the top of stack with the specified register. The second form of fxch swaps the top of stack with st(1).

Many FPU instructions, e.g., fsqrt, operate only on the top of the register stack. If you want to perform such an operation on a value that is not on the top of stack, you can use the fxch instruction to swap that register with tos, perform the desired operation, and then use the fxch to swap the tos with the original register. The following example takes the square root of st(2):

```
fxch    st(2)
fsqrt
fxch    st(2)
```

The fxch instruction sets the stack exception bit if the stack is empty. It sets the invalid operation bit if you specify an empty register as the operand. This instruction always clears the C_1 condition code bit.

14.4.5 Conversions

The 80x87 chip performs all arithmetic operations on 80 bit real quantities. In a sense, the fld and fst/fstp instructions are conversion instructions as well as data movement instructions because they automatically convert between the internal 80 bit real format and the 32 and 64 bit memory formats. Nonetheless, we'll simply classify them as data movement operations, rather than conversions, because they are moving real values to and from memory. The 80x87 FPU provides five routines which convert to or from integer or binary coded decimal (BCD) format when moving data. These instructions are fild, fist, fistp, fbld, and fbstp.

14.4.5.1 The FILD Instruction

The fild (integer load) instruction converts a 16, 32, or 64 bit two's complement integer to the 80 bit extended precision format and pushes the result onto the stack. This instruction always expects a single operand. This operand must be the address of a word, double word, or quad word integer variable. Although the instruction format for fild uses the familiar mod/rm fields, the operand must be a memory variable, even for 16 and 32 bit integers. You cannot specify one of the 80386's 16 or 32 bit general purpose registers. If you want to push an 80x86 general purpose register onto the FPU stack, you must first store it into a memory variable and then use fild to push that value of that memory variable.

The fild instruction sets the stack exception bit and C_1 (accordingly) if stack overflow occurs while pushing the converted value. Examples:

```
fild    mem_16
fild    mem_32[ecx*4]
fild    mem_64[ebx+ecx*8]
```

14.4.5.2 The FIST and FISTP Instructions

The fist and fistp instructions convert the 80 bit extended precision variable on the top of stack to a 16, 32, or 64 bit integer and store the result away into the memory variable specified by the single operand. These instructions convert the value on tos to an integer according to the rounding setting in the FPU control register (bits 10 and 11). As for the fild instruction, the fist and fistp instructions will not let you specify one of the 80x86's general purpose 16 or 32 bit registers as the destination operand.

The fist instruction converts the value on the top of stack to an integer and then stores the result; it does not otherwise affect the floating point register stack. The fistp instruction pops the value off the floating point register stack after storing the converted value.

These instructions set the stack exception bit if the floating point register stack is empty (this will also clear C_1). They set the precision (imprecise operation) and C_1 bits if rounding occurs (that is, if there is any fractional component to the value in st(0)). These instructions set the underflow exception bit if the result is too small (i.e., less than one but greater than zero or less than zero but greater than -1). Examples:

```
fist    mem_16[bx]
fist    mem_64
fistp   mem_32
```

Don't forget that these instructions use the rounding control settings to determine how they will convert the floating point data to an integer during the store operation. Be default, the rouding control is usually set to "round" mode; yet most programmers expect fist/fistp to truncate the decimal portion during conversion. If you want fist/fistp to truncate floating point values when converting them to an integer, you will need to set the rounding control bits appropriately in the floating point control register.

14.4.5.3 The FBLD and FBSTP Instructions

The fbld and fbstp instructions load and store 80 bit BCD values. The fbld instruction converts a BCD value to its 80 bit extended precision equivalent and pushes the result onto the stack. The fbstp instruction pops the extended precision real value on tos, converts it to an 80 bit BCD value (rounding according to the bits in the floating point control register), and stores the converted result at the address specified by the destination memory operand. Note that there is no fbst instruction which stores the value on tos without popping it.

The fbld instruction sets the stack exception bit and C_1 if stack overflow occurs. It sets the invalid operation bit if you attempt to load an invalid BCD value. The fbstp instruction sets the stack exception bit and clears C_1 if stack underflow occurs (the stack is empty). It sets the underflow flag under the same conditions as fist and fistp. Examples:

```
; Assuming fewer than eight items on the stack, the following
; code sequence is equivalent to an fbst instruction:

                fld     st(0)       ;Duplicate value on TOS.
                fbstp   mem_80

; The following example easily converts an 80 bit BCD value to
; a 64 bit integer:

                fbld    bcd_80      ;Get BCD value to convert.
                fist    mem_64      ;Store as an integer.
```

14.4.6 Arithmetic Instructions

The arithmetic instructions make up a small, but important, subset of the 80x87's instruction set. These instructions fall into two general categories – those which operate on real values and those which operate on a real and an integer value.

14.4.6.1 The FADD and FADDP Instructions

These two instructions take the following forms:

```
fadd
faddp
fadd    st(i), st(0)
fadd    st(0), st(i)
faddp   st(i), st(0)
fadd    mem
```

The first two forms are equivalent. They pop the two values on the top of stack, add them, and push their sum back onto the stack.

The next two forms of the fadd instruction, those with two FPU register operands, behave like the 80x86's add instruction. They add the value in the second register operand to the value in the first register operand. Note that one of the register operands must be st(0)[8].

The faddp instruction with two operands adds st(0) (which must always be the second operand) to the destination (first) operand and then pops st(0). The destination operand must be one of the other FPU registers.

The last form above, fadd with a memory operand, adds a 32 or 64 bit floating point variable to the value in st(0). This instruction will convert the 32 or 64 bit operands to an 80 bit extended precision value before performing the addition. Note that this instruction does *not* allow an 80 bit memory operand.

These instructions can raise the stack, precision, underflow, overflow, denormalized, and illegal operation exceptions, as appropriate. If a stack fault exception occurs, C_1 denotes stack overflow or underflow.

14.4.6.2 The FSUB, FSUBP, FSUBR, and FSUBRP Instructions

These four instructions take the following forms:

```
fsub
fsubp
fsubr
fsubrp

fsub     st(i). st(0)
fsub     st(0), st(i)
fsubp    st(i), st(0)
fsub     mem

fsubr    st(i). st(0)
fsubr    st(0), st(i)
fsubrp   st(i), st(0)
fsubr    mem
```

With no operands, the fsub and fsubp instructions operate identically. They pop st(0) and st(1) from the register stack, compute st(0)-st(1), and the push the difference back onto the stack. The fsubr and fsubrp instructions (reverse subtraction) operate in an almost identical fashion except they compute st(1)-st(0) and push that difference.

With two register operands (*destination, source*) the fsub instruction computes *destination := destination - source*. One of the two registers must be st(0). With two registers as operands, the fsubp also computes *destination := destination - source* and then it pops st(0) off the stack after computing the difference. For the fsubp instruction, the source operand must be st(0).

With two register operands, the fsubr and fsubrp instruction work in a similar fashion to fsub and fsubp, except they compute *destination := source - destination.*

The fsub mem and fsubr mem instructions accept a 32 or 64 bit memory operand. They convert the memory operand to an 80 bit extended precision value and subtract this from st(0) (fsub) or subtract st(0) from this value (fsubr) and store the result back into st(0).

These instructions can raise the stack, precision, underflow, overflow, denormalized, and illegal operation exceptions, as appropriate. If a stack fault exception occurs, C_1 denotes stack overflow or underflow.

8. Because you will use st(0) quite a bit when programming the 80x87, MASM allows you to use the abbreviation st for st(0). However, this text will explicitly state st(0) so there will be no confusion.

14.4.6.3 The FMUL and FMULP Instructions

The fmul and fmulp instructions multiply two floating point values. These instructions allow the following forms:

```
fmul
fmulp

fmul     st(0), st(i)
fmul     st(i), st(0)
fmul     mem

fmulp    st(i), st(0)
```

With no operands, fmul and fmulp both do the same thing – they pop st(0) and st(1), multiply these values, and push their product back onto the stack. The fmul instructions with two register operands compute *destination := destination * source*. One of the registers (source or destination) must be st(0).

The fmulp st(i), st(0) instruction computes st(i) := st(i) * st(0) and then pops st(0). This instruction uses the value for *i* before popping st(0). The fmul mem instruction requires a 32 or 64 bit memory operand. It converts the specified memory variable to an 80 bit extended precision value and the multiplies st(0) by this value.

These instructions can raise the stack, precision, underflow, overflow, denormalized, and illegal operation exceptions, as appropriate. If rounding occurs during the computation, these instructions set the C_1 condition code bit. If a stack fault exception occurs, C_1 denotes stack overflow or underflow.

14.4.6.4 The FDIV, FDIVP, FDIVR, and FDIVRP Instructions

These four instructions allow the following forms:

```
fdiv
fdivp
fdivr
fdivrp

fdiv     st(0), st(i)
fdiv     st(i), st(0)
fdivp    st(i), st(0)

fdivr    st(0), st(i)
fdivr    st(i), st(0)
fdivrp   st(i), st(0)

fdiv     mem
fdivr    mem
```

With zero operands, the fdiv and fdivp instructions pop st(0) and st(1), compute st(0)/st(1), and push the result back onto the stack. The fdivr and fdivrp instructions also pop st(0) and st(1) but compute st(1)/st(0) before pushing the quotient onto the stack.

With two register operands, these instructions compute the following quotients:

```
fdiv     st(0), st(i)          ;st(0) := st(0)/st(i)
fdiv     st(i), st(0)          ;st(i) := st(i)/st(0)
fdivp    st(i), st(0)          ;st(i) := st(i)/st(0)
fdivr    st(i), st(i)          ;st(0) := st(0)/st(i)
fdivrp   st(i), st(0)          ;st(i) := st(0)/st(i)
```

The fdivp and fdivrp instructions also pop st(0) after performing the division operation. The value for *i* in this two instructions is computed before popping st(0).

These instructions can raise the stack, precision, underflow, overflow, denormalized, zero divide, and illegal operation exceptions, as appropriate. If rounding occurs during the computation, these instructions set the C_1 condition code bit. If a stack fault exception occurs, C_1 denotes stack overflow or underflow.

14.4.6.5 The FSQRT Instruction

The fsqrt routine does not allow any operands. It computes the square root of the value on tos and replaces st(0) with this result. The value on tos must be zero or positive, otherwise fsqrt will generate an invalid operation exception.

This instruction can raise the stack, precision, denormalized, and invalid operation exceptions, as appropriate. If rounding occurs during the computation, fsqrt sets the C_1 condition code bit. If a stack fault exception occurs, C_1 denotes stack overflow or underflow.

Example:

```
; Compute Z := sqrt(x**2 + y**2);

                fld     x           ;Load X.
                fld     st(0)       ;Duplicate X on TOS.
                fmul                ;Compute X**2.

                fld     y           ;Load Y.
                fld     st(0)       ;Duplicate Y on TOS.
                fmul                ;Compute Y**2.

                fadd                ;Compute X**2 + Y**2.
                fsqrt               ;Compute sqrt(x**2 + y**2).
                fst     Z           ;Store away result in Z.
```

14.4.6.6 The FSCALE Instruction

The fscale instruction pops two values off the stack. It multiplies st(0) by $2^{st(1)}$ and pushes the result back onto the stack. If the value in st(1) is not an integer, fscale truncates it towards zero before performing the operation.

This instruction raises the stack exception if there are not two items currently on the stack (this will also clear C_1 since stack underflow occurs). It raises the precision exception if there is a loss of precision due to this operation (this occurs when st(1) contains a large, negative, value). Likewise, this instruction sets the underflow or overflow exception bits if you multiply st(0) by a very large positive or negative power of two. If the result of the multiplication is very small, fscale could set the denormalized bit. Also, this instruction could set the invalid operation bit if you attempt to fscale illegal values. Fscale sets C_1 if rounding occurs in an otherwise correct computation. Example:

```
                fild    Sixteen     ;Push sixteen onto the stack.
                fld     x           ;Compute x * (2**16).
                fscale
                  .
                  .
                  .
Sixteen         word    16
```

14.4.6.7 The FPREM and FPREM1 Instructions

The fprem and fprem1 instructions compute a *partial remainder*. Intel designed the fprem instruction before the IEEE finalized their floating point standard. In the final draft of the IEEE floating point standard, the definition of fprem was a little different than Intel's original design. Unfortunately, Intel needed to maintain compatibility with the existing software that used the fprem instruction, so they designed a new version to handle the IEEE partial remainder operation, fprem1. You should always use fprem1 in new software you write, therefore we will only discuss fprem1 here, although you use fprem in an identical fashion.

Fprem1 computes the *partial* remainder of st(0)/st(1). If the difference between the exponents of st(0) and st(1) is less than 64, fprem1 can compute the exact remainder in one

operation. Otherwise you will have to execute the fprem1 two or more times to get the correct remainder value. The C_2 condition code bit determines when the computation is complete. Note that fprem1 does *not* pop the two operands off the stack; it leaves the partial remainder in st(0) and the original divisor in st(1) in case you need to compute another partial product to complete the result.

The fprem1 instruction sets the stack exception flag if there aren't two values on the top of stack. It sets the underflow and denormal exception bits if the result is too small. It sets the invalid operation bit if the values on tos are inappropriate for this operation. It sets the C_2 condition code bit if the partial remainder operation is not complete. Finally, it loads C_3, C_1, and C_0 with bits zero, one, and two of the quotient, respectively.

Example:

```
; Compute Z := X mod Y

                fld     y
                fld     x
PartialLp:      fprem1
                fstsw   ax          ;Get condition bits in AX.
                test    ah, 100b    ;See if C2 is set.
                jnz     PartialLp   ;Repeat if not done yet.
                fstp    Z           ;Store remainder away.
                fstp    st(0)       ;Pop old y value.
```

14.4.6.8 The FRNDINT Instruction

The frndint instruction rounds the value on tos to the nearest integer using the rounding algorithm specified in the control register.

This instruction sets the stack exception flag if there is no value on the tos (it will also clear C_1 in this case). It sets the precision and denormal exception bits if there was a loss of precision. It sets the invalid operation flag if the value on the tos is not a valid number.

14.4.6.9 The FXTRACT Instruction

The fxtract instruction is the complement to the fscale instruction. It pops the value off the top of the stack and pushes a value which is the integer equivalent of the exponent (in 80 bit real form), and then pushes the mantissa with an exponent of zero (3fffh in biased form).

This instruction raises the stack exception if there is a stack underflow when popping the original value or a stack overflow when pushing the two results (C_1 determines whether stack overflow or underflow occurs). If the original top of stack was zero, fxtract sets the zero division exception flag. The denormalized flag is set if the result warrants it; and the invalid operation flag is set if there are illegal input values when you execute fxtract.

Example:

```
; The following example extracts the binary exponent of X and
; stores this into the 16 bit integer variable Xponent.

                fld     x
                fxtract
                fstp    st(0)
                fistp   Xponent
```

14.4.6.10 The FABS Instruction

Fabs computes the absolute value of st(0) by clearing the sign bit of st(0). It sets the stack exception bit and invalid operation bits if the stack is empty.

Example:

```
; Compute X := sqrt(abs(x));

                fld     x
                fabs
                fsqrt
                fstp    x
```

14.4.6.11 The FCHS Instruction

Fchs changes the sign of st(0)'s value by inverting its sign bit. It sets the stack exception bit and invalid operation bits if the stack is empty. Example:

```
; Compute X := -X if X is positive, X := X if X is negative.

                fld     x
                fabs
                fchs
                fstp    x
```

14.4.7 Comparison Instructions

The 80x87 provides several instructions for comparing real values. The fcom, fcomp, fcompp, fucom, fucomp, and fucompp instructions compare the two values on the top of stack and set the condition codes appropriately. The ftst instruction compares the value on the top of stack with zero. The fxam instrution checks the value on tos and reports sign, normalization, and tag information.

Generally, most programs test the condition code bits immediately after a comparison. Unfortunately, there are no conditional jump instructions that branch based on the FPU condition codes. Instead, you can use the fstsw instruction to copy the floating point status register (see "The FPU Status Register" on page 785) into the ax register; then you can use the sahf instruction to copy the ah register into the 80x86's condition code bits. After doing this, you can can use the conditional jump instructions to test some condition. This technique copies C_0 into the carry flag, C_2 into the parity flag, and C_3 into the zero flag. The sahf instruction does not copy C_1 into any of the 80x86's flag bits.

Since the sahf instruction does not copy any 80x87 processor status bits into the sign or overflow flags, you cannot use the jg, jl, jge, or jle instructions. Instead, use the ja, jae, jb, jbe, je, and jz instructions when testing the results of a floating point comparison. *Yes, these conditional jumps normally test unsigned values and floating point numbers are signed values.* However, use the unsigned conditional branches anyway; the fstsw and sahf instructions set the 80x86 flags register to use the unsigned jumps.

14.4.7.1 The FCOM, FCOMP, and FCOMPP Instructions

The fcom, fcomp, and fcompp instructions compare st(0) to the specified operand and set the corresponding 80x87 condition code bits based on the result of the comparison. The legal forms for these instructions are

```
fcom
fcomp
fcompp

fcom    st(i)
fcomp   st(i)

fcom    mem
fcomp   mem
```

With no operands, fcom, fcomp, and fcompp compare st(0) against st(1) and set the processor flags accordingly. In addition, fcomp pops st(0) off the stack and fcompp pops both st(0) and st(1) off the stack.

With a single register operand, fcom and fcomp compare st(0) against the specified register. Fcomp also pops st(0) after the comparison.

With a 32 or 64 bit memory operand, the fcom and fcomp instructions convert the memory variable to an 80 bit extended precision value and then compare st(0) against this value, setting the condition code bits accordingly. Fcomp also pops st(0) after the comparison.

These instructions set C_2 (which winds up in the parity flag) if the two operands are not comparable (e.g., NaN). If it is possible for an illegal floating point value to wind up in a comparison, you should check the parity flag for an error before checking the desired condition.

These instructions set the stack fault bit if there aren't two items on the top of the register stack. They set the denormalized exception bit if either or both operands are denormalized. They set the invalid operation flag if either or both operands are quite NaNs. These instructions always clear the C_1 condition code.

14.4.7.2 The FUCOM, FUCOMP, and FUCOMPP Instructions

These instructions are similar to the fcom, fcomp, and fcompp instructions, although they only allow the following forms:

```
fucom
fucomp
fucompp
fucom    st(i)
fucomp   st(i)
```

The difference between fcom/fcomp/fcompp and fucom/fucomp/fucompp is relatively minor. The fcom/fcomp/fcompp instructions set the invalid operation exception bit if you compare two NaNs. The fucom/fucomp/fucompp instructions do not. In all other cases, these two sets of instructions behave identically.

14.4.7.3 The FTST Instruction

The ftst instruction compares the value in st(0) against 0.0. It behaves just like the fcom instruction would if st(1) contained 0.0. Note that this instruction does not differentiate -0.0 from +0.0. If the value in st(0) is either of these values, ftst will set C_3 to denote equality. If you need to differentiate -0.0 from +0.0, use the fxam instruction. Note that this instruction does *not* pop st(0) off the stack.

14.4.7.4 The FXAM Instruction

The fxam instruction examines the value in st(0) and reports the results in the condition code bits (see "The FPU Status Register" on page 785 for details on how fxam sets these bits). This instruction does not pop st(0) off the stack.

14.4.8 Constant Instructions

The 80x87 FPU provides several instructions that let you load commonly used constants onto the FPU's register stack. These instructions set the stack fault, invalid opera-

tion, and C_1 flags if a stack overflow occurs; they do not otherwise affect the FPU flags. The specific instructions in this category include:

```
fldz                ;Pushes +0.0.
fld1                ;Pushes +1.0.
fldpi               ;Pushes π.
fldl2t              ;Pushes log2(10).
fldl2e              ;Pushes log2(e).
fldlg2              ;Pushes log10(2).
fldln2              ;Pushes ln(2).
```

14.4.9 Transcendental Instructions

The 80387 and later FPUs provide eight transcendental (log and trigonometric) instructions to compute a partial tangent, partial arctangent, 2^x-1, y * $\log_2(x)$, and y * $\log_2(x+1)$. Using various algebraic identities, it is easy to compute most of the other common transcendental functions using these instructions.

14.4.9.1 The F2XM1 Instruction

F2xm1 computes $2^{st(0)}$-1. The value in st(0) must be in the range -1.0 ≤ st(0) ≤ +1.0. If st(0) is out of range f2xm1 generates an undefined result but raises no exceptions. The computed value replaces the value in st(0). Example:

```
; Compute 10^x using the identity: 10^x = 2^(x*lg(10)) (lg = log2).

                fld     x
                fldl2t
                fmul
                f2xm1
                fld1
                fadd
```

Note that f2xm1 computes 2^x-1, which is why the code above adds 1.0 to the result at the end of the computation.

14.4.9.2 The FSIN, FCOS, and FSINCOS Instructions

These instructions pop the value off the top of the register stack and compute the sine, cosine, or both, and push the result(s) back onto the stack. The fsincos pushes the sine followed by the cosine of the original operand, hence it leaves cos(st(0)) in st(0) and sin(st(0)) in st(1).

These instructions assume st(0) specifies an angle in radians and this angle must be in the range -2^{63} < st(0) < $+2^{63}$. If the original operand is out of range, these instructions set the C_2 flag and leave st(0) unchanged. You can use the fprem1 instruction, with a divisor of 2π, to reduce the operand to a reasonable range.

These instructions set the stack fault/C_1, precision, underflow, denormalized, and invalid operation flags according to the result of the computation.

14.4.9.3 The FPTAN Instruction

Fptan computes the tangent of st(0) and pushes this value and then it pushes 1.0 onto the stack. Like the fsin and fcos instructions, the value of st(0) is assumed to be in radians and must be in the range -2^{63}<st(0)<$+2^{63}$. If the value is outside this range, fptan sets C_2 to indicate that the conversion did not take place. As with the fsin, fcos, and fsincos instructions, you can use the fprem1 instruction to reduce this operand to a reasonable range using a divisor of 2π.

If the argument is invalid (i.e., zero or π radians, which causes a division by zero) the result is undefined and this instruction raises no exceptions. Fptan will set the stack fault, precision, underflow, denormal, invalid operation, C_2, and C_1 bits as required by the operation.

14.4.9.4 The FPATAN Instruction

This instruction expects two values on the top of stack. It pops them and computes the following:

$st(0) = \tan^{-1}(st(1) / st(0))$

The resulting value is the arctangent of the ratio on the stack expressed in radians. If you have a value you wish to compute the tangent of, use fld1 to create the appropriate ratio and then execute the fpatan instruction.

This instruction affects the stack fault/C_1, precision, underflow, denormal, and invalid operation bits if an problem occurs during the computation. It sets the C_1 condition code bit if it has to round the result.

14.4.9.5 The FYL2X and FYL2XP1 Instructions

The fyl2x and fyl2xp1 instructions compute st(1) * $\log_2$(st(0)) and st(1) * $\log_2$(st(0)+1), respectively. Fyl2x requires that st(0) be greater than zero, fyl2xp1 requires st(0) to be in the range:

$$\left(-1-\left(\frac{\sqrt{2}}{2}\right)\right) < st(0) < \left(1-\left(\frac{\sqrt{2}}{2}\right)\right)$$

Fyl2x is useful for computing logs to bases other than two; fyl2xp1 is useful for computing compound interest, maintaining the maximum precision during computation.

Fyl2x can affect *all* the exception flags. C_1 denotes rounding if there is not other error, stack overflow / underflow if the stack fault bit is set.

The fyl2xp1 instruction does not affect the overflow or zero divide exception flags. These exceptions occur when st(0) is very small or zero. Since fyl2xp1 adds one to st(0) before computing the function, this condition never holds. Fyl2xp1 affects the other flags in a manner identical to fyl2x.

14.4.10 Miscellaneous instructions

The 80x87 FPU includes several additional instructions which control the FPU, synchronize operations, and let you test or set various status bits. These instructions include finit/fninit, fdisi/fndisi, feni/fneni, fldcw, fstcw/fnstcw, fclex/fnclex, fsave/fnsave, frstor, frstpm, fstsw/fnstsw, fstenv/fnstenv, fldenv, fincstp, fdecstp, fwait, fnop, and ffree. The fdisi/fndisi, feni/fneni, and frstpm are active only on FPUs earlier than the 80387, so we will not consider them here.

Many of these instructions have two forms. The first form is Fxxxx and the second form is FNxxxx. The version without the "N" emits an fwait instruction prior to opcode (which is standard for most coprocessor instructions). The version with the "N" does not emit the fwait opcode ("N" stands for *no wait*).

14.4.10.1 The FINIT and FNINIT Instructions

The finit instruction intializes the FPU for proper operation. Your applications should execute this instruction before executing any other FPU instructions. This instruction ini-

tializes the control register to 37Fh (see “The FPU Control Register” on page 782), the status register to zero (see “The FPU Status Register” on page 785) and the tag word to 0FFFFh. The other registers are unaffected.

14.4.10.2 The FWAIT Instruction

The fwait instruction pauses the system until any currently executing FPU instruction completes. This is required because the FPU on the 80486sx and earlier CPU/FPU combinations can execute instructions in parallel with the CPU. Therefore, any FPU instruction which reads or writes memory could suffer from a data hazard if the main CPU accesses that same memory location before the FPU reads or writes that location. The fwait instruction lets you synchronize the operation of the FPU by waiting until the completion of the current FPU instruction. This resolves the data hazard by, effectively, inserting an explict “stall” into the execution stream.

14.4.10.3 The FLDCW and FSTCW Instructions

The fldcw and fstcw instructions require a single 16 bit memory operand:

```
fldcw     mem_16
fstcw     mem_16
```

These two instructions load the control register (see “The FPU Control Register” on page 782) from a memory location (fldcw) or store the control word to a 16 bit memory location (fstcw).

When using the fldcw instruction to turn on one of the exceptions, if the corresponding exception flag is set when you enable that exception, the FPU will generate an immediate interrupt before the CPU executes the next instruction. Therefore, you should use the fclex instruction to clear any pending interrupts before changing the FPU exception enable bits.

14.4.10.4 The FCLEX and FNCLEX Instructions

The fclex and fnclex instructions clear all exception bits the stack fault bit, and the busy flag in the FPU status register (see “The FPU Status Register” on page 785).

14.4.10.5 The FLDENV, FSTENV, and FNSTENV Instructions

```
fstenv    mem_14b
fnstenv   mem_14b
fldenv    mem_14b
```

The fstenv/fnstenv instructions store a 14-byte FPU environment record to the memory operand specified. When operating in real mode (the only mode this text considers), the environment record takes the form appearing in Figure 14.11.

You must execute the fstenv and fnstenv instructions with the CPU interrupts disabled. Furthermore, you should always ensure that the FPU is not busy before executing this instruction. This is easily accomplished by using the following code:

```
pushf                ;Preserve I flag.
cli                  ;Disable interrupts.
fstenv    mem_14b    ;Implicit wait for not busy.
fwait                ;Wait for operation to finish.
popf                 ;Restore I flag.
```

The fldenv instruction loads the FPU environment from the specified memory operand. Note that this instruction lets you load the the status word. There is no explicit instruction like fldcw to accomplish this.

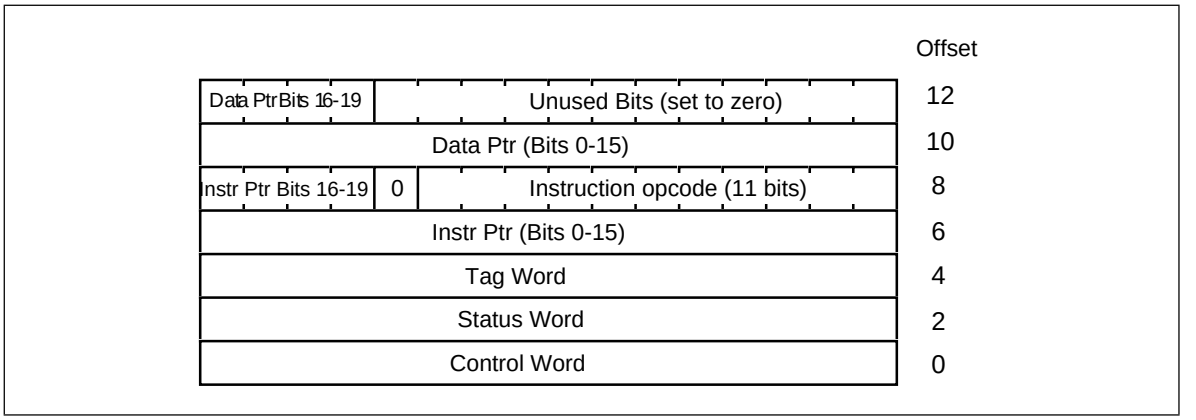

Figure 14.11 FPU Environment Record (16 Bit Real Mode)

14.4.10.6 The FSAVE, FNSAVE, and FRSTOR Instructions

```
fsave     mem_94b
fnsave    mem_94b
frstor    mem_94b
```

These instructions save and restore the state of the FPU. This includes saving all the internal control, status, and data registers. The destination location for fsave/fnsave (source location for frstor) must be 94 bytes long. The first 14 bytes correspond to the environment record the fldenv and fstenv instructions use; the remaining 80 bytes hold the data from the FPU register stack written out as st(0) through st(7). Frstor reloads the environment record and floating point registers from the specified memory operand.

The fsave/fnsave and frstor instructions are mainly intended for task switching. You can also use fsave/fnsave and frstor as a "push all" and "pop all" sequence to preserve the state of the FPU.

Like the fstenv and fldenv instructions, interrupts should be disabled while saving or restoring the FPU state. Otherwise another interrupt service routine could manipulate the FPU registers and invalidate the operation of the fsave/fnsave or frestore operation. The following code properly protects the environment data while saving and restore the FPU status:

```
; Preserve the FPU state, assume di points at the environment
; record in memory.

                pushf
                cli
                fsave     [si]
                fwait
                popf
                  .
                  .
                  .
                pushf
                cli
                frstor    [si]
                fwait
                popf
```

14.4.10.7 The FSTSW and FNSTSW Instructions

```
fstsw    ax
fnstsw   ax
fstsw    mem_16
fnstsw   mem_16
```

These instructions store the FPU status register (see “The FPU Status Register” on page 785) into a 16 bit memory location or the ax register. These instructions are unusual in the sense that they can copy an FPU value into one of the 80x86 general purpose registers. Of course, the whole purpose behind allowing the transfer of the status register into ax is to allow the CPU to easily test the condition code register with the sahf instruction.

14.4.10.8 The FINCSTP and FDECSTP Instructions

The fincstp and fdecstp instructions do not take any operands. They simply increment and decrement the stack pointer bits (mod 8) in the FPU status register. These two instructions clear the C_1 flag, but do not otherwise affect the condition code bits in the FPU status register.

14.4.10.9 The FNOP Instruction

The fnop instruction is simply an alias for fst st, st(0). It performs no other operation on the FPU.

14.4.10.10The FFREE Instruction

```
ffree    st(i)
```

This instruction modifies the tag bits for register *i* in the tags register to mark the specified register as emtpy. The value is unaffected by this instruction, but the FPU will no longer be able to access that data (without resetting the appropriate tag bits).

14.4.11 Integer Operations

The 80x87 FPUs provide special instructions that combine integer to extended precision conversion along with various arithmetic and comparison operations. These instructions are the following:

```
fiadd    int
fisub    int
fisubr   int
fimul    int
fidiv    int
fidivr   int

ficom    int
ficomp   int
```

These instructions convert their 16 or 32 bit integer operands to an 80 bit extended precision floating point value and then use this value as the source operand for the specified operation. These instructions use st(0) as the destination operand.

14.5 Sample Program: Additional Trigonometric Functions

This section provides various examples of 80x87 FPU programming. This group of routines provides several trigonometric, inverse trigonometric, logarithmic, and exponential functions using various algebraic identities. All these functions assume that the input values are on the stack are are within valid ranges for the given functions. The trigonometric routines expect angles expressed in radians and the inverse trig routines produce angles measured in radians.

This program (transcnd.asm) appears on the companion CD-ROM.

```
                .xlist
                include     stdlib.a
                includelib  stdlib.lib
                .list

                .386
                .387
                option    segment:use16

dseg            segment   para public 'data'

result          real8     ?

; Some variables we use to test the routines in this package:

cotvar          real8     3.0
cotRes          real8     ?
acotRes         real8     ?

cscvar          real8     1.5
cscRes          real8     ?
acscRes         real8     ?

secvar          real8     0.5
secRes          real8     ?
asecRes         real8     ?

sinvar          real8     0.75
sinRes          real8     ?
asinRes         real8     ?

cosvar          real8     0.25
cosRes          real8     ?
acosRes         real8     ?

Two2xvar        real8     -2.5
Two2xRes        real8     ?
lgxRes          real8     ?

Ten2xVar        real8     3.75
Ten2xRes        real8     ?
logRes          real8     ?

expVar          real8     3.25
expRes          real8     ?
lnRes           real8     ?

Y2Xx            real8     3.0
Y2Xy            real8     3.0
Y2XRes          real8     ?

dseg            ends

cseg            segment   para public 'code'
                assume    cs:cseg, ds:dseg
```

```
; COT(x) - Computes the cotangent of st(0) and leaves result in st(0).
;                st(0) contains x (in radians) and must be between
;                -2**63 and +2**63
;
;       There must be at least one free register on the stack for
;       this routine to operate properly.
;
;      cot(x) = 1/tan(x)

cot             proc     near
                fsincos
                fdivr
                ret
cot             endp

; CSC(x) - computes the cosecant of st(0) and leaves result in st(0).
;                st(0) contains x (in radians) and must be between
;                -2**63 and +2**63.
;       The cosecant of x is undefined for any value of sin(x) that
;                produces zero (e.g., zero or pi radians).
;
;       There must be at least one free register on the stack for
;       this routine to operate properly.
;
;      csc(x) = 1/sin(x)

csc             proc     near
                fsin
                fld1
                fdivr
                ret
csc             endp

; SEC(x) - computes the secant of st(0) and leaves result in st(0).
;                st(0) contains x (in radians) and must be between
;                -2**63 and +2**63.
;
;      The secant of x is undefined for any value of cos(x) that
;      produces zero (e.g., pi/2 radians).
;
;      There must be at least one free register on the stack for
;      this routine to operate properly.
;
;      sec(x) = 1/cos(x)

sec             proc     near
                fcos
                fld1
                fdivr
                ret
sec             endp

; ASIN(x)- Computes the arcsine of st(0) and leaves the result in st(0).
;                Allowable range: -1<=x<=+1
;                There must be at least two free registers for this
;                function to operate properly.
;
;      asin(x) = atan(sqrt(x*x/(1-x*x)))

asin            proc     near
                fld      st(0)        ;Duplicate X on tos.
                fmul                  ;Compute X**2.
                fld      st(0)        ;Duplicate X**2 on tos.
                fld1                  ;Compute 1-X**2.
                fsubr
                fdiv                  ;Compute X**2/(1-X**2).
                fsqrt                 ;Compute sqrt(x**2/(1-X**2)).
                fld1                  ;To compute full arctangent.
                fpatan                ;Compute atan of the above.
                ret
```

```
asin            endp

; ACOS(x)- Computes the arccosine of st(0) and leaves the
;               result in st(0).
;               Allowable range: -1<=x<=+1
;               There must be at least two free registers for
;               this function to operate properly.
;
;       acos(x) = atan(sqrt((1-x*x)/(x*x)))

acos            proc    near
                fld     st(0)       ;Duplicate X on tos.
                fmul                ;Compute X**2.
                fld     st(0)       ;Duplicate X**2 on tos.
                fld1                ;Compute 1-X**2.
                fsubr
                fdiv                ;Compute (1-x**2)/X**2.
                fsqrt               ;Compute sqrt((1-X**2)/X**2).
                fld1                ;To compute full arctangent.
                fpatan              ;Compute atan of the above.
                ret
acos            endp

; ACOT(x)- Computes the arccotangent of st(0) and leaves the
;               result in st(0).
;               X cannot equal zero.
;               There must be at least one free register for
;               this function to operate properly.
;
;       acot(x) = atan(1/x)

acot            proc    near
                fld1                ;fpatan computes
                fxch                ; atan(st(1)/st(0)).
                fpatan              ; we want atan(st(0)/st(1)).
                ret
acot            endp

; ACSC(x)- Computes the arccosecant of st(0) and leaves the
;               result in st(0).
;               abs(X) must be greater than one.
;               There must be at least two free registers for
;               this function to operate properly.
;
;       acsc(x) = atan(sqrt(1/(x*x-1)))

acsc            proc    near
                fld     st(0)       ;Compute x*x
                fmul
                fld1                ;Compute x*x-1
                fsub
                fld1                ;Compute 1/(x*x-1)
                fdivr
                fsqrt               ;Compute sqrt(1/(x*x-1))
                fld1
                fpatan              ;Compute atan of above.
                ret
acsc            endp

; ASEC(x)- Computes the arcsecant of st(0) and leaves the
;               result in st(0).
;               abs(X) must be greater than one.
;               There must be at least two free registers for
;               this function to operate properly.
;
;       asec(x) = atan(sqrt(x*x-1))

asec            proc    near
                fld     st(0)       ;Compute x*x
                fmul
```

```
                fld1                    ;Compute x*x-1
                fsub
                fsqrt                   ;Compute sqrt(x*x-1)
                fld1
                fpatan                  ;Compute atan of above.
                ret
asec            endp

; TwoToX(x)- Computes 2**x.
;               It does this by using the algebraic identity:
;
;               2**x = 2**int(x) * 2**frac(x).
;               We can easily compute 2**int(x) with fscale and
;               2**frac(x) using f2xm1.
;
;               This routine requires three free registers.

SaveCW          word    ?
MaskedCW        word    ?

TwoToX          proc    near
                fstcw   cseg:SaveCW

; Modify the control word to truncate when rounding.

                fstcw   cseg:MaskedCW
                or      byte ptr cseg:MaskedCW+1, 1100b
                fldcw   cseg:MaskedCW

                fld     st(0)           ;Duplicate tos.
                fld     st(0)
                frndint                 ;Compute integer portion.

                fxch                    ;Swap whole and int values.
                fsub    st(0), st(1) ;Compute fractional part.

                f2xm1                   ;Compute 2**frac(x)-1.
                fld1
                fadd                    ;Compute 2**frac(x).

                fxch                    ;Get integer portion.
                fld1                    ;Compute 1*2**int(x).
                fscale
                fstp    st(1)           ;Remove st(1) (which is 1).

                fmul                    ;Compute 2**int(x) * 2**frac(x).

                fldcw   cseg:SaveCW  ;Restore rounding mode.
                ret
TwoToX          endp


; TenToX(x)- Computes 10**x.
;
;               This routine requires three free registers.
;
;       TenToX(x) = 2**(x * lg(10))

TenToX          proc    near
                fldl2t                  ;Put lg(10) onto the stack
                fmul                    ;Compute x*lg(10)
                call    TwoToX          ;Compute 2**(x * lg(10)).
                ret
TenToX          endp

; exp(x)- Computes e**x.
;
;               This routine requires three free registers.
;
;       exp(x) = 2**(x * lg(e))
```

```
exp             proc    near
                fldl2e                  ;Put lg(e) onto the stack.
                fmul                    ;Compute x*lg(e).
                call    TwoToX          ;Compute 2**(x * lg(e))
                ret
exp             endp

; YtoX(y,x)- Computes y**x (y=st(1), x=st(0)).
;
;               This routine requires three free registers.
;
;               Y must be greater than zero.
;
;       YtoX(y,x) = 2 ** (x * lg(y))

YtoX            proc    near
                fxch                    ;Compute lg(y).
                fld1
                fxch
                fyl2x

                fmul                    ;Compute x*lg(y).
                call    TwoToX          ;Compute 2**(x*lg(y)).
                ret
YtoX            endp

; LOG(x)- Computes the base 10 logarithm of x.
;
;               Usual range for x (>0).
;
;       LOG(x) = lg(x)/lg(10).

log             proc    near
                fld1
                fxch
                fyl2x                   ;Compute 1*lg(x).
                fldl2t                  ;Load lg(10).
                fdiv                    ;Compute lg(x)/lg(10).
                ret
log             endp


; LN(x)- Computes the base e logarithm of x.
;
;               X must be greater than zero.
;
;       ln(x) = lg(x)/lg(e).

ln              proc    near
                fld1
                fxch
                fyl2x                   ;Compute 1*lg(x).
                fldl2e                  ;Load lg(e).
                fdiv                    ;Compute lg(x)/lg(10).
                ret
ln              endp

; This main program tests the various functions in this package.

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax
                meminit

                finit

; Check to see if cot and acot are working properly.
```

```
                fld     cotVar
                call    cot
                fst     cotRes
                call    acot
                fstp    acotRes

                printff
                byte    "x=%8.5gf, cot(x)=%8.5gf, acot(cot(x)) = %8.5gf\n",0
                dword   cotVar, cotRes, acotRes

; Check to see if csc and acsc are working properly.

                fld     cscVar
                call    csc
                fst     cscRes
                call    acsc
                fstp    acscRes

                printff
                byte    "x=%8.5gf, csc(x)=%8.5gf, acsc(csc(x)) = %8.5gf\n",0
                dword   cscVar, cscRes, acscRes

; Check to see if sec and asec are working properly.

                fld     secVar
                call    sec
                fst     secRes
                call    asec
                fstp    asecRes

                printff
                byte    "x=%8.5gf, sec(x)=%8.5gf, asec(sec(x)) = %8.5gf\n",0
                dword   secVar, secRes, asecRes

; Check to see if sin and asin are working properly.

                fld     sinVar
                fsin
                fst     sinRes
                call    asin
                fstp    asinRes

                printff
                byte    "x=%8.5gf, sin(x)=%8.5gf, asin(sin(x)) = %8.5gf\n",0
                dword   sinVar, sinRes, asinRes

; Check to see if cos and acos are working properly.

                fld     cosVar
                fcos
                fst     cosRes
                call    acos
                fstp    acosRes

                printff
                byte    "x=%8.5gf, cos(x)=%8.5gf, acos(cos(x)) = %8.5gf\n",0
                dword   cosVar, cosRes, acosRes

; Check to see if 2**x and lg(x) are working properly.

                fld     Two2xVar
                call    TwoToX
                fst     Two2xRes
                fld1
                fxch
                fyl2x
                fstp    lgxRes

                printff
                byte    "x=%8.5gf, 2**x =%8.5gf, lg(2**x) = %8.5gf\n",0
```

```
                    dword    Two2xVar, Two2xRes, lgxRes

; Check to see if 10**x and l0g(x) are working properly.

                    fld      Ten2xVar
                    call     TenToX
                    fst      Ten2xRes
                    call     LOG
                    fstp     logRes

                    printff
                    byte     "x=%8.5gf, 10**x =%8.2gf, log(10**x) = %8.5gf\n",0
                    dword    Ten2xVar, Ten2xRes, logRes

; Check to see if exp(x) and ln(x) are working properly.

                    fld      expVar
                    call     exp
                    fst      expRes
                    call     ln
                    fstp     lnRes

                    printff
                    byte     "x=%8.5gf, e**x =%8.2gf, ln(e**x) = %8.5gf\n",0
                    dword    expVar, expRes, lnRes

; Check to see if y**x is working properly.

                    fld      Y2Xy
                    fld      Y2Xx
                    call     YtoX
                    fstp     Y2XRes

                    printff
                    byte     "x=%8.5gf, y =%8.5gf, y**x = %8.4gf\n",0
                    dword    Y2Xx, Y2Xy, Y2XRes

Quit:               ExitPgm
Main                endp
cseg                ends

sseg                segment  para stack 'stack'
stk                 byte     1024 dup ("stack ")
sseg                ends

zzzzzzseg           segment  para public 'zzzzzz'
LastBytes           byte     16 dup (?)
zzzzzzseg           ends
                    end      Main

Sample program output:
x= 3.00000, cot(x)=-7.01525, acot(cot(x)) = 3.00000
x= 1.50000, csc(x)= 1.00251, acsc(csc(x)) = 1.50000
x= 0.50000, sec(x)= 1.13949, asec(sec(x)) = 0.50000
x= 0.75000, sin(x)= 0.68163, asin(sin(x)) = 0.75000
x= 0.25000, cos(x)= 0.96891, acos(cos(x)) = 0.25000
x=-2.50000, 2**x  = 0.17677, lg(2**x)     = -2.50000
x= 3.75000, 10**x = 5623.41, log(10**x)   = 3.75000
x= 3.25000, e**x  = 25.79,   ln(e**x)     = 3.25000
x= 3.00000, y     = 3.00000, y**x         = 27.0000
```

14.6 Laboratory Exercises

14.6.1 FPU vs StdLib Accuracy

In this laboratory exercise you will will run two programs that perform 20,000,000 floating point additions. These programs do the first 10,000,000 additions using the 80x87 FPU, they do the second 10,000,000 additions using the Standard Library's floating point routines. This exercise demonstrates the relative accuracy of the two floating point mechanisms.

For your lab report: assemble and run the EX14_1.asm program (it's on the companion CD-ROM). This program adds together 10,000,000 64-bit floating point values and prints their sum. Describe the results in your lab report. Time these operations and report the time difference in your lab report. Note that the *exact* sum these operations should produce is 1.00000010000e+0000.

After running Ex14_1.asm, repeat this process for the Ex14_2.asm file. Ex14_2 differs from Ex14_1 insofar as Ex14_2 lets the Standard Library routines operate on 80-bit memory operands (the FPU cannot operate on 80-bit memory operands, so this part remains unchanged). Time the execution of Ex14_2's two components. Compare these times against the running time of Ex14_1 and explain any differences.

```
; EX14_1.asm
;
; This program runs some tests to determine how well the floating point
; arithmetic in the Standard Library compares with the floating point
; arithmetic on the 80x87.  It does this performing various operations
; using both methods and comparing the result.
;
; Of course, you must have an 80x87 FPU (or 80486 or later processor)
; in order to run this code.

                .386
                option      segment:use16

                include     stdlib.a
                includelib  stdlib.lib

dseg            segment  para public 'data'

; Since this is an accuracy test, this code uses REAL8 values for
; all operations

slValue1        real8    1.0
slSmallVal      real8    1.0e-14

Value1          real8    1.0
SmallVal        real8    1.0e-14

Buffer          byte     20 dup (0)

dseg            ends


cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

Main            proc
                mov      ax, dseg
                mov      ds, ax
                mov      es, ax
                meminit

                finit                ;Initialize the FPU

; Do 10,000,000 floating point additions:
```

```
                printff
                byte    "Adding 10,000,000 FP values together with the “
                byte    “FPU",cr,lf,0

                mov     ecx, 10000000
FPLoop:         fld     Value1
                fld     SmallVal
                fadd
                fstp    Value1
                dec     ecx
                jnz     FPLoop

                printff
                byte    "Result = %20GE\n",cr,lf,0
                dword   Value1

; Do 10,000,000 floating point additions with the Standard Library fpadd
; routine:

                printff
                byte    cr,lf
                byte    "Adding 10,000,000 FP values together with the “
                byte    “StdLib", cr,lf
                byte    "Note: this may take a few minutes to run, don't “
                byte    “get too impatient"
                byte    cr,lf,0

                mov     ecx, 10000000
SLLoop:         lesi    slValue1
                ldfpa
                lesi    slSmallVal
                ldfpo
                fpadd
                lesi    slValue1
                sdfpa
                dec     ecx
                jnz     SLLoop

                printff
                byte    "Result = %20GE\n",cr,lf,0
                dword   slValue1

Quit:           ExitPgm             ;DOS macro to quit program.
Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             db      1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       db      16 dup (?)
zzzzzzseg       ends
                end     Main




; EX14_2.asm
;
; This program runs some tests to determine how well the floating point
; arithmetic in the Standard Library compares with the floating point
; arithmetic on the 80x87.  It lets the standard library routines use
; the full 80-bit format since they allow it and the FPU does not.
;
; Of course, you must have an 80x87 FPU (or 80486 or later processor)
; in order to run this code.
```

```
                .386
                option      segment:use16

                include     stdlib.a
                includelib  stdlib.lib

dseg            segment   para public 'data'

slValue1        real10    1.0
slSmallVal      real10    1.0e-14

Value1          real8     1.0
SmallVal        real8     1.0e-14

Buffer          byte      20 dup (0)

dseg            ends

cseg            segment   para public 'code'
                assume    cs:cseg, ds:dseg

Main            proc
                mov       ax, dseg
                mov       ds, ax
                mov       es, ax
                meminit
                finit                 ;Initialize the FPU

; Do 10,000,000 floating point additions:

                printff
                byte      "Adding 10,000,000 FP values together with the “
                byte      “FPU",cr,lf,0

                mov       ecx, 10000000
FPLoop:         fld       Value1
                fld       SmallVal
                fadd
                fstp      Value1
                dec       ecx
                jnz       FPLoop

                printff
                byte      "Result = %20GE\n",cr,lf,0
                dword     Value1

; Do 10,000,000 floating point additions with the Standard Library fpadd
; routine:

                printff
                byte      cr,lf
                byte      "Adding 10,000,000 FP values together with the “
                byte      “StdLib", cr,lf
                byte      "Note: this may take a few minutes to run, don't “
                byte      “get too impatient"
                byte      cr,lf,0

                mov       ecx, 10000000
SLLoop:         lesi      slValue1
                lefpa
                lesi      slSmallVal
                lefpo
                fpadd
                lesi      slValue1
                sefpa
                dec       ecx
                jnz       SLLoop

                printff
```

```
                        byte    "Result = %20LE\n",cr,lf,0
                        dword   slValue1

Quit:                   ExitPgm             ;DOS macro to quit program.
Main                    endp

cseg                    ends

sseg                    segment para stack 'stack'
stk                     db      1024 dup ("stack   ")
sseg                    ends

zzzzzzseg               segment para public 'zzzzzz'
LastBytes               db      16 dup (?)
zzzzzzseg               ends
                        end     Main
```

14.7 Programming Projects

14.8 Summary

For many applications integer arithmetic has two insurmountable drawbacks – it is not easy to represent fractional values with integers and integers have a limited dynamic range. Floating point arithmetic provides an approximation to real arithmetic that overcomes these two limitations.

Floating point arithmetic, however, is not without its own problems. Floating point arithmetic suffers from *limited precision*. As a result, inaccuracies can creep into a calculation. Therefore, floating point arithmetic does not completely follow normal algebraic rules. There are five very important rules to keep in mind when using floating point arithmetic: (1`) The order of evaluation can affect the accuracy of the result; (2) Whenever adding and subtracting numbers, the accuracy of the result may be less than the precision provided by the floating point format; (3) When performing a chain of calculations involving addition, subtraction, multiplication, and division, try to perform the multiplication and division operations first; (4) When multiplying and dividing values, try to multiply large and small numbers together first and try to divide numbers with the same relative magnitue first; (5) When comparing two floating point numbers, always keep in mind that errors can creep into the computations, therefore you should check to see if one value is within a certain range of the other. For more information, see

- “The Mathematics of Floating Point Arithmetic” on page 771

Early on Intel recognized the need for a hardware floating point unit. They hired three mathematicians to design highly accurate floating point formats and algorithms for their 80x87 family of FPUs. These formats, with slight modifications, become the IEEE 754 and IEEE 854 floating point standards. The IEEE standard actually provides for three different formats: a 32 bit standard precision format, a 64 bit double precision format, and an extended precision format. Intel implemented the extended precision format using 80 bits[9]. The 32 bit format uses a 24 bit mantissa (the H.O. bit is an implied one and is not stored in the 32 bits), an eight bit bias 127 exponent, and a one bit sign. The 64 bit format provides a 53 bit mantissa (again, the H.O. bit is always one and is not stored in the 64 b it value), an 11 bit excess 1023 exponent, and a one bit sign. The 80 bit extended precision format uses a 64 bit exponent, a 15 bit excess 16363 exponent, and a single bit sign. For more information, see

- “IEEE Floating Point Formats” on page 774

9. The IEEE standard only requires that the extended precision format contain more bits than the double precision format.

Although 80x87 FPUs and CPUs with built-in FPUs (80486 and Pentium) are becoming very common, it is still possible that you may need to execute code that uses floating point arithmetic on a machine without an FPU. In such cases you will need to supply software routines to execute the floating point arithmetic. Fortunately, the UCR Standard Library provides a set of floating point routines you can call. The Standard Library includes routines to load and store floating point values, convert between integer and floating point formats, add, subtract, multiply, and divide floating point values, convert between ASCII and floating point, and output floating point values. Even if you have an FPU installed, the Standard Library's conversion and output routines are quite useful. For more information, see

- "The UCR Standard Library Floating Point Routines" on page 777

For fast floating point arithmetic, software doesn't stand a chance against hardware. The 80x87 FPUs provide fast and convient floating point operations by extended the 80x86's instruction set to handle floating point arithmetic. In addition to the new instructions, the 80x87 FPUs also provide eight new data registers, a control register, a status register, and several other internal registers. The FPU data registers, unlike the 80x86's general purpose registers, are organized as a stack. Although it is possible to manipulate the registers as though they were a standard register file, most FPU applications use the stack mechanism when computing floating point results. The FPU control register lets you initialize the 80x87 FPU in one of several different modes. The control register lets you set the rounding control, the precision available during computation, and choose which exceptions can cause an interrupt. The 80x87 status register reports the current state of the FPU. This register provides bits that determine if the FPU is currently busy, determine if a previous instruction has generated an exception, determine the physical register number of the top of the register stack, and provide the FPU condition codes. For more information on the 80x87 register set, see

- "The 80x87 Floating Point Coprocessors" on page 781
- "FPU Registers" on page 781
- "The FPU Data Registers" on page 782
- "The FPU Control Register" on page 782
- "The FPU Status Register" on page 785

In addition to the IEEE single, double, and extended preoision data types, the 80x87 FPUs also support various integer and BCD data types. The FPU will automatically convert to and from these data types when loading and storing such values. For more information on these data type formats, see

- "FPU Data Types" on page 788

The 80x87 FPUs provide a wide range of floating point operations by augmenting the 80x86's instruction set. We can classify the FPU instructions into eight categories: data movement instructions, conversions, arithmetic instructions, comparison instructions, constant instructions, transcendental instructions, miscellaneous instructions, and integer instructions. For more information on these instruction types, see

- "The FPU Instruction Set" on page 789
- "FPU Data Movement Instructions" on page 789
- "Conversions" on page 791
- "Arithmetic Instructions" on page 792
- "Comparison Instructions" on page 797
- "Constant Instructions" on page 798
- "Transcendental Instructions" on page 799
- "Miscellaneous instructions" on page 800
- "Integer Operations" on page 803

Although the 80387 and later FPUs provide a rich set of transcendental functions, there are many trigonometric, inverse trigonometric, exponential, and logarithmic functions missing from the instruction set. However, the missing functions are easy to synthesize using algebraic identities. This chapter provides source code for many of these routines as an example of FPU programming. For more information, see

- “Sample Program: Additional Trigonometric Functions” on page 804

14.9 Questions

1) Why don't the normal rules of algebra apply to floating point arithmetic?

2) Give an example of a sequence of operations whose order of evaluation will produce different results with finite precision arithmetic.

3) Explain why limited precision addition and subtraction operations can cause a loss of precision during a calculation.

4) Why should you, if at all possible, perform multiplications and divisions first in a calculation involving multiplication or division as well as addition or subtraction?

5) Explain the difference between a normalized, unnormalized, and denormalized floating point value.

6) Using the UCR Standard Library, convert the following expression to 80x86 assembly code (assume all variables are 64 bit double precision values). Be sure to perform any necessary algebraic manipulations to ensure the maximum accuracy. You can assume all variables fall in the range ±1e-10…±1e+10.

a) Z := X * X + Y * Y

b) Z := (X-Y)*Z

c) Z := X*Y - X / Y

d) Z := (X+Y) / (X-Y)

e) Z := (X*X) / (Y*Y)

f) Z := X*X + Y + 1.0

7) Convert the above statements to 80x87 FPU code.

8) The following problems provide definitions for the *hyperbolic trigonometric* functions. Encode each of these using the 80x87 FPU instructions and the exp(x) and ln(x) routines provided in this chapter.

a) $\sinh x = \frac{e^x - e^{-x}}{2}$

b) $\cosh x = \frac{e^x + e^{-x}}{2}$

c) $\tanh x = \frac{\sinh x}{\cosh x}$

d) $\operatorname{csch} x = \frac{1}{\sinh x}$

e) $\operatorname{sech} x = \frac{1}{\cosh x}$

f) $\coth x = \frac{\cosh x}{\sinh x}$

g) $\operatorname{asinh} x = ln(x + \sqrt{x^2 + 1})$

h) $\operatorname{acosh} x = ln(x + \sqrt{x^2 - 1})$

i) $\operatorname{atanh} x = \frac{ln(\frac{1+x}{1-x})}{2}$

j) $\operatorname{acsch} x = ln(\frac{x \pm \sqrt{1+x^2}}{x})$

k) $\operatorname{asech} x = ln(\frac{x \pm \sqrt{1-x^2}}{x})$

l) $\operatorname{atanh} x = \frac{ln(\frac{x+1}{x-1})}{2}$

9) Create a log(x,y) function which computes $\log_y x$. The algebraic identity for this is

$$\log_y x = \frac{\log_2 x}{\log_2 y}$$

10) Interval arithmetic involves performing a calculation with every result rounded down and then repeating the computation with each result rounded up. At the end of these two computations, you know that the true result must lie between the two computed results. The rounding control bits in the FPU control register let you select round up and round down modes. Repeat question six applying interval arithmetic and compute the two bounds for each of those problems (a-f).

11) The mantissa precision control bits in the FPU control register simply control where the FPU rounds results. Selecting a lower precision does not improve the performance of the FPU. Therefore, any new software you write should set these two bits to ones to get 64 bits of precision when performing calculations. Can you provide one reason why you might want to set the precision to something other than 64 bits?

12) Suppose you have two 64 bit variables, X and Y, that you want to compare to see if they are equal. As you know, you should not compare them directly to see if they are equal, but rather see if they are less than some small value apart. Suppose ε, the error constant, is 1e-300. Provide the code to load ax with zero if X=Y and load ax with one if X≠Y.

13) Repeat problem 12, except test for:

a) $X \leq Y$ b) $X < Y$

c) $X \geq Y$ d) $X > Y$

e) $X \neq Y$

14) What instruction can you use to see if the value in st(0) is denormalized?

15) Assuming no stack underflow or overflow, what is the C_1 condition code bit usually used for?

16) Many texts, when describing the FPU chip, suggest that you can use the FPU to perform integer arithmetic. An argument generally given is that the FPU can support 64 bit integers whereas the CPU can only support 16 or 32 bit integers. What is wrong with this argument? Why would you *not* want to use the FPU to perform integer arithmetic? Why does the FPU even provide integer instructions?

17) Suppose you have a 64 bit double precision floating point value in memory. Describe how you could take the absolute value of this variable without using the FPU (i.e., by using only 80x86 instructions).

18) Explain how to change the sign of the variable in question 17.

19) Why does the TwoToX function (see “Sample Program: Additional Trigonometric Functions” on page 804) have to compute the result using fscale and fyl2x? Why can’t it use fyl2x along?

20) Explain a possible problem with the following code sequence:

```
stp     mem_64
xor     byte ptr mem_64+7, 80h          ;Tweak sign bit
```