Chapter 6

Floating Point

6.1 Floating Point Representation

6.1.1 Non-integral binary numbers

When number systems were discussed in the first chapter, only integer values were discussed. Obviously, it must be possible to represent non-integral numbers in other bases as well as decimal. In decimal, digits to the right of the decimal point have associated negative powers of ten:

$$0.123 = 1 \times 10^{-1} + 2 \times 10^{-2} + 3 \times 10^{-3}$$

Not surprisingly, binary numbers work similarly:

$$0.101_2 = 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = 0.625$$

This idea can be combined with the integer methods of Chapter 1 to convert a general number:

$$110.011_2 = 4 + 2 + 0.25 + 0.125 = 6.375$$

Converting from decimal to binary is not very difficult either. In general, divide the decimal number into two parts: integer and fraction. Convert the integer part to binary using the methods from Chapter 1. The fractional part is converted using the method described below.

Consider a binary fraction with the bits labeled $a, b, c, \ldots$ The number in binary then looks like:

$$0.abcdef\ldots$$

Multiply the number by two. The binary representation of the new number will be:

$$a.bcdef\ldots$$

$$
\begin{aligned}
0.5625 \times 2 &= 1.125 & \text{first bit} &= 1\\
0.125 \times 2 &= 0.25 & \text{second bit} &= 0\\
0.25 \times 2 &= 0.5 & \text{third bit} &= 0\\
0.5 \times 2 &= 1.0 & \text{fourth bit} &= 1
\end{aligned}
$$

Figure 6.1: Converting 0.5625 to binary

$$
\begin{aligned}
0.85 \times 2 &= 1.7\\
0.7 \times 2 &= 1.4\\
0.4 \times 2 &= 0.8\\
0.8 \times 2 &= 1.6\\
0.6 \times 2 &= 1.2\\
0.2 \times 2 &= 0.4\\
0.4 \times 2 &= 0.8\\
0.8 \times 2 &= 1.6
\end{aligned}
$$

Figure 6.2: Converting 0.85 to binary

Note that the first bit is now in the one's place. Replace the a with 0 to get:

$$0.bcdef\ldots$$

and multiply by two again to get:

$$b.cdef\ldots$$

Now the second bit (b) is in the one's position. This procedure can be repeated until as many bits needed are found. Figure 6.1 shows a real example that converts 0.5625 to binary. The method stops when a fractional part of zero is reached.

As another example, consider converting 23.85 to binary. It is easy to convert the integral part ($23 = 10111_2$), but what about the fractional part (0.85)? Figure 6.2 shows the beginning of this calculation. If one looks at the numbers carefully, an infinite loop is found! This means that 0.85 is a repeating binary (as opposed to a repeating decimal in base 10)[1]. There is a pattern to the numbers in the calculation. Looking at the pattern, one can see that $0.85 = 0.11\overline{0110}_2$. Thus, $23.85 = 10111.11\overline{0110}_2$.

One important consequence of the above calculation is that 23.85 can not be represented *exactly* in binary using a finite number of bits. (Just as $\frac{1}{3}$ can not be represented in decimal with a finite number of digits.) As this chapter shows, `float` and `double` variables in C are stored in binary. Thus, values like 23.85 can not be stored exactly into these variables. Only an approximation of 23.85 can be stored.

To simplify the hardware, floating point numbers are stored in a consistent format. This format uses scientific notation (but in binary, using powers of two, not ten). For example, 23.85 or $10111.11011001100110\ldots_2$ would be stored as:

$$1.011111011001100110\ldots \times 2^{100}$$

(where the exponent (100) is in binary). A *normalized* floating point number has the form:

$$1.sssssssssssssss \times 2^{eeeeeee}$$

where $1.sssssssssssss$ is the *significand* and *eeeeeeee* is the *exponent*.

6.1.2 IEEE floating point representation

The IEEE (Institute of Electrical and Electronic Engineers) is an international organization that has designed specific binary formats for storing floating point numbers. This format is used on most (but not all!) computers made today. Often it is supported by the hardware of the computer itself. For example, Intel's numeric (or math) coprocessors (which are built into all its CPU's since the Pentium) use it. The IEEE defines two different formats with different precisions: single and double precision. Single precision is used by `float` variables in C and double precision is used by `double` variables.

Intel's math coprocessor also uses a third, higher precision called *extended precision*. In fact, all data in the coprocessor itself is in this precision. When it is stored in memory from the coprocessor it is converted to either single or double precision automatically.[2] Extended precision uses a slightly different general format than the IEEE float and double formats and so will not be discussed here.

---

[1] It should not be so surprising that a number might repeat in one base, but not another. Think about $\frac{1}{3}$, it repeats in decimal, but in ternary (base 3) it would be 0.1_3.

[2] Some compiler's (such as Borland) `long double` type uses this extended precision. However, other compilers use double precision for both `double` and `long double`. (This is allowed by ANSI C.)

31	30	23	22	0
s	e		f	

s	sign bit - 0 = positive, 1 = negative
e	biased exponent (8-bits) = true exponent + 7F (127 decimal). The values 00 and FF have special meaning (see text).
f	fraction - the first 23-bits after the 1. in the significand.

Figure 6.3: IEEE single precision

IEEE single precision

Single precision floating point uses 32 bits to encode the number. It is usually accurate to 7 significant decimal digits. Floating point numbers are stored in a much more complicated format than integers. Figure 6.3 shows the basic format of a IEEE single precision number. There are several quirks to the format. Floating point numbers do not use the two's complement representation for negative numbers. They use a signed magnitude representation. Bit 31 determines the sign of the number as shown.

The binary exponent is not stored directly. Instead, the sum of the exponent and 7F is stored from bit 23 to 30. This *biased exponent* is always non-negative.

The fraction part assumes a normalized significand (in the form 1.*sssssssss*). Since the first bit is always an one, the leading one is *not stored!* This allows the storage of an additional bit at the end and so increases the precision slightly. This idea is know as the *hidden one representation*.

One should always keep in mind that the bytes 41 BE CC CD can be interpreted different ways depending on what a program does with them! As as single precision floating point number, they represent 23.850000381, but as a double word integer, they represent 1,103,023,309! The CPU does not know which is the correct interpretation!

How would 23.85 be stored? First, it is positive so the sign bit is 0. Next the true exponent is 4, so the biased exponent is $7F + 4 = 83_{16}$. Finally, the fraction is 01111101100110011001100 (remember the leading one is hidden). Putting this all together (to help clarify the different sections of the floating point format, the sign bit and the faction have been underlined and the bits have been grouped into 4-bit nibbles):

$$\underline{0}\;100\;0001\;1\;\underline{011\;1110\;1100\;1100\;1100\;1100}_2 = 41BECCCC_{16}$$

This is not exactly 23.85 (since it is a repeating binary). If one converts the above back to decimal, one finds that it is approximately 23.849998474. This number is very close to 23.85, but it is not exact. Actually, in C, 23.85 would not be represented exactly as above. Since the left-most bit that was truncated from the exact representation is 1, the last bit is rounded up to 1. So 23.85 would be represented as 41 BE CC CD in hex using single precision. Converting this to decimal results in 23.850000381 which is a slightly better approximation of 23.85.

$e = 0$	and	$f = 0$	denotes the number zero (which can not be normalized) Note that there is a +0 and -0.
$e = 0$	and	$f \neq 0$	denotes a *denormalized number*. These are discussed in the next section.
$e = \text{FF}$	and	$f = 0$	denotes infinity (∞). There are both positive and negative infinities.
$e = \text{FF}$	and	$f \neq 0$	denotes an undefined result, known as *NaN* (Not a Number).

Table 6.1: Special values of f and e

63	62 52	51 0
s	e	f

Figure 6.4: IEEE double precision

How would -23.85 be represented? Just change the sign bit: C1 BE CC CD. Do *not* take the two's complement!

Certain combinations of e and f have special meanings for IEEE floats. Table 6.1 describes these special values. An infinity is produced by an overflow or by division by zero. An undefined result is produced by an invalid operation such as trying to find the square root of a negative number, adding two infinities, *etc.*

Normalized single precision numbers can range in magnitude from 1.0×2^{-126} ($\approx 1.1755 \times 10^{-35}$) to $1.11111\ldots \times 2^{127}$ ($\approx 3.4028 \times 10^{35}$).

Denormalized numbers

Denormalized numbers can be used to represent numbers with magnitudes too small to normalize (*i.e.* below 1.0×2^{-126}). For example, consider the number $1.001_2 \times 2^{-129}$ ($\approx 1.6530 \times 10^{-39}$). In the given normalized form, the exponent is too small. However, it can be represented in the unnormalized form: $0.01001_2 \times 2^{-127}$. To store this number, the biased exponent is set to 0 (see Table 6.1) and the fraction is the complete significand of the number written as a product with 2^{-127} (*i.e.* all bits are stored including the one to the left of the decimal point). The representation of 1.001×2^{-129} is then:

$$\underline{0}\ 000\ 0000\ 0\ \underline{001\ 0010\ 0000\ 0000\ 0000\ 0000}$$

IEEE double precision

IEEE double precision uses 64 bits to represent numbers and is usually accurate to about 15 significant decimal digits. As Figure 6.4 shows, the basic format is very similar to single precision. More bits are used for the biased exponent (11) and the fraction (52) than for single precision.

The larger range for the biased exponent has two consequences. The first is that it is calculated as the sum of the true exponent and 3FF (1023) (not 7F as for single precision). Secondly, a large range of true exponents (and thus a larger range of magnitudes) is allowed. Double precision magnitudes can range from approximately 10^{-308} to 10^{308}.

It is the larger field of the fraction that is responsible for the increase in the number of significant digits for double values.

As an example, consider 23.85 again. The biased exponent will be $4 + 3\text{FF} = 403$ in hex. Thus, the double representation would be:

$\underline{0}$ 100 0000 0011 $\underline{\text{0111 1101 1001 1001 1001 1001 1001 1001 1001 1001 1001 1001 1010}}$

or 40 37 D9 99 99 99 99 9A in hex. If one converts this back to decimal, one finds 23.8500000000000014 (there are 12 zeros!) which is a much better approximation of 23.85.

The double precision has the same special values as single precision[3]. Denormalized numbers are also very similar. The only main difference is that double denormalized numbers use 2^{-1023} instead of 2^{-127}.

6.2 Floating Point Arithmetic

Floating point arithmetic on a computer is different than in continuous mathematics. In mathematics, all numbers can be considered exact. As shown in the previous section, on a computer many numbers can not be represented exactly with a finite number of bits. All calculations are performed with limited precision. In the examples of this section, numbers with an 8-bit significand will be used for simplicity.

6.2.1 Addition

To add two floating point numbers, the exponents must be equal. If they are not already equal, then they must be made equal by shifting the significand of the number with the smaller exponent. For example, consider $10.375 + 6.34375 = 16.71875$ or in binary:

$$\begin{array}{rr} & 1.0100110 \times 2^3 \\ + & 1.1001011 \times 2^2 \\ \hline \end{array}$$

[3] The only difference is that for the infinity and undefined values, the biased exponent is 7FF not FF.

These two numbers do not have the same exponent so shift the significand to make the exponents the same and then add:

$$\begin{array}{rr} & 1.0100110 \times 2^3 \\ + & 0.1100110 \times 2^3 \\ \hline & 10.0001100 \times 2^3 \end{array}$$

Note that the shifting of 1.1001011×2^2 drops off the trailing one and after rounding results in 0.1100110×2^3. The result of the addition, 10.0001100×2^3 (or 1.00001100×2^4) is equal to 10000.110_2 or 16.75. This is *not* equal to the exact answer (16.71875)! It is only an approximation due to the round off errors of the addition process.

It is important to realize that floating point arithmetic on a computer (or calculator) is always an approximation. The laws of mathematics do not always work with floating point numbers on a computer. Mathematics assumes infinite precision which no computer can match. For example, mathematics teaches that $(a + b) - b = a$; however, this may not hold true exactly on a computer!

6.2.2 Subtraction

Subtraction works very similarly and has the same problems as addition. As an example, consider $16.75 - 15.9375 = 0.8125$:

$$\begin{array}{rr} & 1.0000110 \times 2^4 \\ - & 1.1111111 \times 2^3 \\ \hline \end{array}$$

Shifting 1.1111111×2^3 gives (rounding up) 1.0000000×2^4

$$\begin{array}{rr} & 1.0000110 \times 2^4 \\ - & 1.0000000 \times 2^4 \\ \hline & 0.0000110 \times 2^4 \end{array}$$

$0.0000110 \times 2^4 = 0.11_2 = 0.75$ which is not exactly correct.

6.2.3 Multiplication and division

For multiplication, the significands are multiplied and the exponents are added. Consider $10.375 \times 2.5 = 25.9375$:

$$\begin{array}{rl} & 1.0100110 \times 2^3 \\ \times & 1.0100000 \times 2^1 \\ \hline & \quad 10100110 \\ + & 10100110 \\ \hline & 1.10011111000000 \times 2^4 \end{array}$$

Of course, the real result would be rounded to 8-bits to give:

$$1.1010000 \times 2^4 = 11010.000_2 = 26$$

Division is more complicated, but has similar problems with round off errors.

6.2.4 Ramifications for programming

The main point of this section is that floating point calculations are not exact. The programmer needs to be aware of this. A common mistake that programmers make with floating point numbers is to compare them assuming that a calculation is exact. For example, consider a function named f(x) that makes a complex calculation and a program is trying to find the function's roots[4]. One might be tempted to use the following statement to check to see if x is a root:

```
if ( f(x) == 0.0 )
```

But, what if f(x) returns 1×10^{-30}? This very likely means that x is a *very* good approximation of a true root; however, the equality will be false. There may not be any IEEE floating point value of x that returns exactly zero, due to round off errors in f(x).

A much better method would be to use:

```
if ( fabs(f(x)) < EPS )
```

where EPS is a macro defined to be a very small positive value (like 1×10^{-10}). This is true whenever f(x) is very close to zero. In general, to compare a floating point value (say x) to another (y) use:

```
if ( fabs(x − y)/fabs(y) < EPS )
```

6.3 The Numeric Coprocessor

6.3.1 Hardware

The earliest Intel processors had no hardware support for floating point operations. This does not mean that they could not perform float operations. It just means that they had to be performed by procedures composed of many non-floating point instructions. For these early systems, Intel did provide an additional chip called a *math coprocessor*. A math coprocessor has machine instructions that perform many floating point operations much faster than using a software procedure (on early processors, at least 10 times

[4] A root of a function is a value x such that $f(x) = 0$

faster!). The coprocessor for the 8086/8088 was called the 8087. For the 80286, there was a 80287 and for the 80386, a 80387. The 80486DX processor integrated the math coprocessor into the 80486 itself.[5] Since the Pentium, all generations of 80x86 processors have a builtin math coprocessor; however, it is still programmed as if it was a separate unit. Even earlier systems without a coprocessor can install software that emulates a math coprocessor. These emulator packages are automatically activated when a program executes a coprocessor instruction and run a software procedure that produces the same result as the coprocessor would have (though much slower, of course).

The numeric coprocessor has eight floating point registers. Each register holds 80 bits of data. Floating point numbers are *always* stored as 80-bit extended precision numbers in these registers. The registers are named `ST0`, `ST1`, `ST2`, ... `ST7`. The floating point registers are used differently than the integer registers of the main CPU. The floating point registers are organized as a *stack*. Recall that a stack is a *Last-In First-Out* (LIFO) list. `ST0` always refers to the value at the top of the stack. All new numbers are added to the top of the stack. Existing numbers are pushed down on the stack to make room for the new number.

There is also a status register in the numeric coprocessor. It has several flags. Only the 4 flags used for comparisons will be covered: C_0, C_1, C_2 and C_3. The use of these is discussed later.

6.3.2 Instructions

To make it easy to distinguish the normal CPU instructions from coprocessor ones, all the coprocessor mnemonics start with an `F`.

Loading and storing

There are several instructions that load data onto the top of the coprocessor register stack:

`FLD` *`source`*	loads a floating point number from memory onto the top of the stack. The *source* may be a single, double or extended precision number or a coprocessor register.
`FILD` *`source`*	reads an *integer* from memory, converts it to floating point and stores the result on top of the stack. The *source* may be either a word, double word or quad word.
`FLD1`	stores a one on the top of the stack.
`FLDZ`	stores a zero on the top of the stack.

There are also several instructions that store data from the stack into memory. Some of these instructions also *pop* (*i.e.* remove) the number from

[5]However, the 80486SX did *not* have have an integrated coprocessor. There was a separate 80487SX chip for these machines.

the stack as it stores it.

`FST` *dest*	stores the top of the stack (`ST0`) into memory. The *destination* may either be a single or double precision number or a coprocessor register.
`FSTP` *dest*	stores the top of the stack into memory just as `FST`; however, after the number is stored, its value is popped from the stack. The *destination* may either a single, double or extended precision number or a coprocessor register.
`FIST` *dest*	stores the value of the top of the stack converted to an integer into memory. The *destination* may either a word or a double word. The stack itself is unchanged. How the floating point number is converted to an integer depends on some bits in the coprocessor's *control word*. This is a special (non-floating point) word register that controls how the coprocessor works. By default, the control word is initialized so that it rounds to the nearest integer when it converts to integer. However, the `FSTCW` (Store Control Word) and `FLDCW` (Load Control Word) instructions can be used to change this behavior.
`FISTP` *dest*	Same as `FIST` except for two things. The top of the stack is popped and the *destination* may also be a quad word.

There are two other instructions that can move or remove data on the stack itself.

`FXCH ST`*n*	exchanges the values in `ST0` and `ST`*n* on the stack (where n is register number from 1 to 7).
`FFREE ST`*n*	frees up a register on the stack by marking the register as unused or empty.

Addition and subtraction

Each of the addition instructions compute the sum of `ST0` and another operand. The result is always stored in a coprocessor register.

`FADD` *src*	`ST0 +=` *src*. The *src* may be any coprocessor register or a single or double precision number in memory.
`FADD` *dest*`, ST0`	*dest* `+= ST0`. The *dest* may be any coprocessor register.
`FADDP` *dest* or `FADDP` *dest*`, ST0`	*dest* `+= ST0` then pop stack. The *dest* may be any coprocessor register.
`FIADD` *src*	`ST0 += (float)` *src*. Adds an integer to `ST0`. The *src* must be a word or double word in memory.

There are twice as many subtraction instructions than addition because the order of the operands is important for subtraction (*i.e.* $a + b = b + a$, but $a - b \neq b - a$!). For each instruction, there is an alternate one that subtracts in the reverse order. These reverse instructions all end in either

```
1  segment .bss
2  array        resq SIZE
3  sum          resq 1
4
5  segment .text
6        mov    ecx, SIZE
7        mov    esi, array
8        fldz                  ; ST0 = 0
9  lp:
10       fadd   qword [esi]    ; ST0 += *(esi)
11       add    esi, 8         ; move to next double
12       loop   lp
13       fstp   qword sum      ; store result into sum
```

Figure 6.5: Array sum example

`R` or `RP`. Figure 6.5 shows a short code snippet that adds up the elements of an array of doubles. On lines 10 and 13, one must specify the size of the memory operand. Otherwise the assembler would not know whether the memory operand was a float (dword) or a double (qword).

`FSUB` *`src`*	`ST0 -=` *`src`*. The *src* may be any coprocessor register or a single or double precision number in memory.
`FSUBR` *`src`*	`ST0 =` *`src`* `- ST0`. The *src* may be any coprocessor register or a single or double precision number in memory.
`FSUB` *`dest`*`, ST0`	*`dest`* `-= ST0`. The *dest* may be any coprocessor register.
`FSUBR` *`dest`*`, ST0`	*`dest`* `= ST0 -` *`dest`*. The *dest* may be any coprocessor register.
`FSUBP` *`dest`* or `FSUBP` *`dest`*`, ST0`	*`dest`* `-= ST0` then pop stack. The *dest* may be any coprocessor register.
`FSUBRP` *`dest`* or `FSUBRP` *`dest`*`, ST0`	*`dest`* `= ST0 -` *`dest`* then pop stack. The *dest* may be any coprocessor register.
`FISUB` *`src`*	`ST0 -= (float)` *`src`*. Subtracts an integer from `ST0`. The *src* must be a word or double word in memory.
`FISUBR` *`src`*	`ST0 = (float)` *`src`* `- ST0`. Subtracts `ST0` from an integer. The *src* must be a word or double word in memory.

Multiplication and division

The multiplication instructions are completely analogous to the addition instructions.

`FMUL` *`src`*	`ST0 *=` *`src`*. The *src* may be any coprocessor register or a single or double precision number in memory.
`FMUL` *`dest`*`, ST0`	*`dest`* `*= ST0`. The *dest* may be any coprocessor register.
`FMULP` *`dest`* or `FMULP` *`dest`*`, ST0`	*`dest`* `*= ST0` then pop stack. The *dest* may be any coprocessor register.
`FIMUL` *`src`*	`ST0 *= (float)` *`src`*. Multiplies an integer to `ST0`. The *src* must be a word or double word in memory.

Not surprisingly, the division instructions are analogous to the subtraction instructions. Division by zero results in an infinity.

`FDIV` *`src`*	`ST0 /=` *`src`*. The *src* may be any coprocessor register or a single or double precision number in memory.
`FDIVR` *`src`*	`ST0 =` *`src`* `/ ST0`. The *src* may be any coprocessor register or a single or double precision number in memory.
`FDIV` *`dest`*`, ST0`	*`dest`* `/= ST0`. The *dest* may be any coprocessor register.
`FDIVR` *`dest`*`, ST0`	*`dest`* `= ST0 /` *`dest`*. The *dest* may be any coprocessor register.
`FDIVP` *`dest`* or `FDIVP` *`dest`*`, ST0`	*`dest`* `/= ST0` then pop stack. The *dest* may be any coprocessor register.
`FDIVRP` *`dest`* or `FDIVRP` *`dest`*`, ST0`	*`dest`* `= ST0 /` *`dest`* then pop stack. The *dest* may be any coprocessor register.
`FIDIV` *`src`*	`ST0 /= (float)` *`src`*. Divides `ST0` by an integer. The *src* must be a word or double word in memory.
`FIDIVR` *`src`*	`ST0 = (float)` *`src`* `/ ST0`. Divides an integer by `ST0`. The *src* must be a word or double word in memory.

Comparisons

The coprocessor also performs comparisons of floating point numbers. The `FCOM` family of instructions does this operation.

```
1  ;     if ( x > y )
2  ;
3        fld    qword [x]       ; ST0 = x
4        fcomp  qword [y]       ; compare ST0 and y
5        fstsw  ax              ; move C bits into FLAGS
6        sahf
7        jna    else_part       ; if x not above y, goto else_part
8  then_part:
9        ; code for then part
10       jmp    end_if
11 else_part:
12       ; code for else part
13 end_if:
```

Figure 6.6: Comparison example

`FCOM` *src*	compares `ST0` and *src*. The *src* can be a coprocessor register or a float or double in memory.
`FCOMP` *src*	compares `ST0` and *src*, then pops stack. The *src* can be a coprocessor register or a float or double in memory.
`FCOMPP`	compares `ST0` and `ST1`, then pops stack twice.
`FICOM` *src*	compares `ST0` and `(float)` *src*. The *src* can be a word or dword integer in memory.
`FICOMP` *src*	compares `ST0` and `(float)`*src*, then pops stack. The *src* can be a word or dword integer in memory.
`FTST`	compares `ST0` and 0.

These instructions change the C_0, C_1, C_2 and C_3 bits of the coprocessor status register. Unfortunately, it is not possible for the CPU to access these bits directly. The conditional branch instructions use the FLAGS register, not the coprocessor status register. However, it is relatively simple to transfer the bits of the status word into the corresponding bits of the FLAGS register using some new instructions:

`FSTSW` *dest*	Stores the coprocessor status word into either a word in memory or the AX register.
`SAHF`	Stores the AH register into the FLAGS register.
`LAHF`	Loads the AH register with the bits of the FLAGS register.

Figure 6.6 shows a short example code snippet. Lines 5 and 6 transfer the C_0, C_1, C_2 and C_3 bits of the coprocessor status word into the FLAGS register. The bits are transfered so that they are analogous to the result of a comparison of two *unsigned* integers. This is why line 7 uses a `JNA` instruction.

The Pentium Pro (and later processors (Pentium II and III)) support two new comparison operators that directly modify the CPU's FLAGS register.

`FCOMI` *src*	compares `ST0` and *src*. The *src* must be a coprocessor register.
`FCOMIP` *src*	compares `ST0` and *src*, then pops stack. The *src* must be a coprocessor register.

Figure 6.7 shows an example subroutine that finds the maximum of two doubles using the `FCOMIP` instruction. Do not confuse these instructions with the integer comparison functions (`FICOM` and `FICOMP`).

Miscellaneous instructions

This section covers some other miscellaneous instructions that the coprocessor provides.

`FCHS`	`ST0 = - ST0` Changes the sign of `ST0`
`FABS`	$\texttt{ST0} = \lvert\texttt{ST0}\rvert$ Takes the absolute value of `ST0`
`FSQRT`	$\texttt{ST0} = \sqrt{\texttt{ST0}}$ Takes the square root of `ST0`
`FSCALE`	$\texttt{ST0} = \texttt{ST0} \times 2^{\lfloor \texttt{ST1} \rfloor}$ multiples `ST0` by a power of 2 quickly. `ST1` is not removed from the coprocessor stack. Figure 6.8 shows an example of how to use this instruction.

6.3.3 Examples

6.3.4 Quadratic formula

The first example shows how the quadratic formula can be encoded in assembly. Recall that the quadratic formula computes the solutions to the quadratic equation:

$$ax^2 + bx + c = 0$$

The formula itself gives two solutions for x: x_1 and x_2.

$$x_1, x_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The expression inside the square root ($b^2 - 4ac$) is called the *discriminant*. Its value is useful in determining which of the following three possibilities are true for the solutions.

1. There is only one real degenerate solution. $b^2 - 4ac = 0$
2. There are two real solutions. $b^2 - 4ac > 0$
3. There are two complex solutions. $b^2 - 4ac < 0$

Here is a small C program that uses the assembly subroutine:

quadt.c

```
#include <stdio.h>

int quadratic( double, double, double, double *, double *);

int main()
{
  double a,b,c, root1, root2;

  printf("Enter a, b, c: ");
  scanf("%lf %lf %lf", &a, &b, &c);
  if (quadratic( a, b, c, &root1, &root2) )
    printf("roots: %.10g %.10g\n", root1, root2);
  else
    printf("No real roots\n");
  return 0;
}
```

quadt.c

Here is the assembly routine:

quad.asm

```
; function quadratic
; finds solutions to the quadratic equation:
;       a*x^2 + b*x + c = 0
; C prototype:
;   int quadratic( double a, double b, double c,
;                  double * root1, double *root2 )
; Parameters:
;   a, b, c - coefficients of powers of quadratic equation (see above)
;   root1   - pointer to double to store first root in
;   root2   - pointer to double to store second root in
; Return value:
;   returns 1 if real roots found, else 0

%define a               qword [ebp+8]
%define b               qword [ebp+16]
%define c               qword [ebp+24]
%define root1           dword [ebp+32]
%define root2           dword [ebp+36]
%define disc            qword [ebp-8]
```

```
%define one_over_2a     qword [ebp-16]

segment .data
MinusFour       dw      -4

segment .text
        global  _quadratic
_quadratic:
        push    ebp
        mov     ebp, esp
        sub     esp, 16         ; allocate 2 doubles (disc & one_over_2a)
        push    ebx             ; must save original ebx

        fild    word [MinusFour]; stack -4
        fld     a               ; stack: a, -4
        fld     c               ; stack: c, a, -4
        fmulp   st1             ; stack: a*c, -4
        fmulp   st1             ; stack: -4*a*c
        fld     b
        fld     b               ; stack: b, b, -4*a*c
        fmulp   st1             ; stack: b*b, -4*a*c
        faddp   st1             ; stack: b*b - 4*a*c
        ftst                    ; test with 0
        fstsw   ax
        sahf
        jb      no_real_solutions ; if disc < 0, no real solutions
        fsqrt                   ; stack: sqrt(b*b - 4*a*c)
        fstp    disc            ; store and pop stack
        fld1                    ; stack: 1.0
        fld     a               ; stack: a, 1.0
        fscale                  ; stack: a * 2^(1.0) = 2*a, 1
        fdivp   st1             ; stack: 1/(2*a)
        fst     one_over_2a     ; stack: 1/(2*a)
        fld     b               ; stack: b, 1/(2*a)
        fld     disc            ; stack: disc, b, 1/(2*a)
        fsubrp  st1             ; stack: disc - b, 1/(2*a)
        fmulp   st1             ; stack: (-b + disc)/(2*a)
        mov     ebx, root1
        fstp    qword [ebx]     ; store in *root1
        fld     b               ; stack: b
        fld     disc            ; stack: disc, b
        fchs                    ; stack: -disc, b
```

```
        fsubrp  st1             ; stack: -disc - b
        fmul    one_over_2a     ; stack: (-b - disc)/(2*a)
        mov     ebx, root2
        fstp    qword [ebx]     ; store in *root2
        mov     eax, 1          ; return value is 1
        jmp     short quit

no_real_solutions:
        mov     eax, 0          ; return value is 0

quit:
        pop     ebx
        mov     esp, ebp
        pop     ebp
        ret
```

quad.asm

6.3.5 Reading array from file

In this example, an assembly routine reads doubles from a file. Here is a short C test program:

readt.c

```c
/*
 * This program tests the 32-bit read_doubles() assembly procedure.
 * It reads the doubles from stdin. (Use redirection to read from file.)
 */
#include <stdio.h>
extern int read_doubles( FILE *, double *, int );
#define MAX 100

int main()
{
  int i,n;
  double a[MAX];

  n = read_doubles(stdin, a, MAX);

  for( i=0; i < n; i++ )
     printf("%3d %g\n", i, a[i]);
  return 0;
}
```

readt.c

Here is the assembly routine

read.asm

```
segment .data
format  db      "%lf", 0        ; format for fscanf()

segment .text
        global  _read_doubles
        extern  _fscanf

%define SIZEOF_DOUBLE   8
%define FP              dword [ebp + 8]
%define ARRAYP          dword [ebp + 12]
%define ARRAY_SIZE      dword [ebp + 16]
%define TEMP_DOUBLE     [ebp - 8]

;
; function _read_doubles
; C prototype:
;   int read_doubles( FILE * fp, double * arrayp, int array_size );
; This function reads doubles from a text file into an array, until
; EOF or array is full.
; Parameters:
;   fp         - FILE pointer to read from (must be open for input)
;   arrayp     - pointer to double array to read into
;   array_size - number of elements in array
; Return value:
;   number of doubles stored into array (in EAX)

_read_doubles:
        push    ebp
        mov     ebp,esp
        sub     esp, SIZEOF_DOUBLE      ; define one double on stack

        push    esi                     ; save esi
        mov     esi, ARRAYP             ; esi = ARRAYP
        xor     edx, edx                ; edx = array index (initially 0)

while_loop:
        cmp     edx, ARRAY_SIZE         ; is edx < ARRAY_SIZE?
        jnl     short quit              ; if not, quit loop
```

```
;
; call fscanf() to read a double into TEMP_DOUBLE
; fscanf() might change edx so save it
;
        push    edx                     ; save edx
        lea     eax, TEMP_DOUBLE
        push    eax                     ; push &TEMP_DOUBLE
        push    dword format            ; push &format
        push    FP                      ; push file pointer
        call    _fscanf
        add     esp, 12
        pop     edx                     ; restore edx
        cmp     eax, 1                  ; did fscanf return 1?
        jne     short quit              ; if not, quit loop

;
; copy TEMP_DOUBLE into ARRAYP[edx]
; (The 8-bytes of the double are copied by two 4-byte copies)
;
        mov     eax, [ebp - 8]
        mov     [esi + 8*edx], eax      ; first copy lowest 4 bytes
        mov     eax, [ebp - 4]
        mov     [esi + 8*edx + 4], eax  ; next copy highest 4 bytes

        inc     edx
        jmp     while_loop

quit:
        pop     esi                     ; restore esi

        mov     eax, edx                ; store return value into eax

        mov     esp, ebp
        pop     ebp
        ret
```

read.asm

6.3.6 Finding primes

This final example looks at finding prime numbers again. This implementation is more efficient than the previous one. It stores the primes it has found in an array and only divides by the previous primes it has found instead of every odd number to find new primes.

One other difference is that it computes the square root of the guess for the next prime to determine at what point it can stop searching for factors. It alters the coprocessor control word so that when it stores the square root as an integer, it truncates instead of rounding. This is controlled by bits 10 and 11 of the control word. These bits are called the RC (Rounding Control) bits. If they are both 0 (the default), the coprocessor rounds when converting to integer. If they are both 1, the coprocessor truncates integer conversions. Notice that the routine is careful to save the original control word and restore it before it returns.

Here is the C driver program:

fprime.c

```c
#include <stdio.h>
#include <stdlib.h>
/*
 * function find_primes
 * finds the indicated number of primes
 * Parameters:
 *   a - array to hold primes
 *   n - how many primes to find
 */
extern void find_primes( int * a, unsigned n );

int main()
{
  int status;
  unsigned i;
  unsigned max;
  int * a;

  printf("How many primes do you wish to find? ");
  scanf("%u", &max);

  a = calloc( sizeof(int ), max);

  if ( a ) {

    find_primes(a,max);

    /* print out the last 20 primes found */
    for( i= ( max > 20 ) ? max - 20 : 0; i < max; i++ )
      printf("%3d %d\n", i+1, a[i]);
```

```c

    free(a);
    status = 0;
  }
  else {
    fprintf(stderr, "Can not create array of %u ints\n", max);
    status = 1;
  }

  return status;
}
```

fprime.c

Here is the assembly routine:

prime2.asm

```
segment .text
        global  _find_primes
;
; function find_primes
; finds the indicated number of primes
; Parameters:
;   array  - array to hold primes
;   n_find - how many primes to find
; C Prototype:
;extern void find_primes( int * array, unsigned n_find )
;
%define array         ebp + 8
%define n_find        ebp + 12
%define n             ebp - 4           ; number of primes found so far
%define isqrt         ebp - 8           ; floor of sqrt of guess
%define orig_cntl_wd  ebp - 10          ; original control word
%define new_cntl_wd   ebp - 12          ; new control word

_find_primes:
        enter   12,0                    ; make room for local variables

        push    ebx                     ; save possible register variables
        push    esi

        fstcw   word [orig_cntl_wd]     ; get current control word
        mov     ax, [orig_cntl_wd]
```

```
        or      ax, 0C00h               ; set rounding bits to 11 (truncate)
        mov     [new_cntl_wd], ax
        fldcw   word [new_cntl_wd]

        mov     esi, [array]            ; esi points to array
        mov     dword [esi], 2          ; array[0] = 2
        mov     dword [esi + 4], 3      ; array[1] = 3
        mov     ebx, 5                  ; ebx = guess = 5
        mov     dword [n], 2            ; n = 2
;
; This outer loop finds a new prime each iteration, which it adds to the
; end of the array. Unlike the earlier prime finding program, this function
; does not determine primeness by dividing by all odd numbers. It only
; divides by the prime numbers that it has already found. (That's why they
; are stored in the array.)
;
while_limit:
        mov     eax, [n]
        cmp     eax, [n_find]           ; while ( n < n_find )
        jnb     short quit_limit

        mov     ecx, 1                  ; ecx is used as array index
        push    ebx                     ; store guess on stack
        fild    dword [esp]             ; load guess onto coprocessor stack
        pop     ebx                     ; get guess off stack
        fsqrt                           ; find sqrt(guess)
        fistp   dword [isqrt]           ; isqrt = floor(sqrt(quess))
;
; This inner loop divides guess (ebx) by earlier computed prime numbers
; until it finds a prime factor of guess (which means guess is not prime)
; or until the prime number to divide is greater than floor(sqrt(guess))
;
while_factor:
        mov     eax, dword [esi + 4*ecx]        ; eax = array[ecx]
        cmp     eax, [isqrt]                    ; while ( isqrt < array[ecx]
        jnbe    short quit_factor_prime
        mov     eax, ebx
        xor     edx, edx
        div     dword [esi + 4*ecx]
        or      edx, edx                        ; && guess % array[ecx] != 0 )
        jz      short quit_factor_not_prime
        inc     ecx                             ; try next prime
```

```
        jmp     short while_factor

;
; found a new prime !
;
quit_factor_prime:
        mov     eax, [n]
        mov     dword [esi + 4*eax], ebx        ; add guess to end of array
        inc     eax
        mov     [n], eax                        ; inc n

quit_factor_not_prime:
        add     ebx, 2                          ; try next odd number
        jmp     short while_limit

quit_limit:

        fldcw   word [orig_cntl_wd]             ; restore control word
        pop     esi                             ; restore register variables
        pop     ebx

        leave
        ret
```

prime2.asm

```
global _dmax

segment .text
; function _dmax
; returns the larger of its two double arguments
; C prototype
; double dmax( double d1, double d2 )
; Parameters:
;   d1   - first double
;   d2   - second double
; Return value:
;   larger of d1 and d2 (in ST0)
%define d1   ebp+8
%define d2   ebp+16
_dmax:
        enter   0, 0

        fld     qword [d2]
        fld     qword [d1]          ; ST0 = d1, ST1 = d2
        fcomip  st1                 ; ST0 = d2
        jna     short d2_bigger
        fcomp   st0                 ; pop d2 from stack
        fld     qword [d1]          ; ST0 = d1
        jmp     short exit
d2_bigger:                          ; if d2 is max, nothing to do
exit:
        leave
        ret
```

Figure 6.7: FCOMIP example

```
segment .data
x           dq  2.75          ; converted to double format
five        dw  5

segment .text
      fild   dword [five]      ; ST0 = 5
      fld    qword [x]         ; ST0 = 2.75, ST1 = 5
      fscale                   ; ST0 = 2.75 * 32, ST1 = 5
```

Figure 6.8: `FSCALE` example