Chapter 10: Floating-Point Arithmetic

Many 80x86 microprocessor systems-including all Pentium systems, systems with a 486DX, and other systems equipped with a floating-point coprocessor-also have the capability to manipulate numbers stored in floating-point format.

The MASM assembler has directives that accept decimal operands and initialize storage using the IEEE format.

There are two ways to do floating-point arithmetic with a PC.

- ✓ If you have a microprocessor with a floating-point unit built in or a floating-point coprocessor, then you can simply use the floating-point instructions.
- ✓ Otherwise, you can employ a collection of procedures that implement arithmetic operations such as addition and multiplication.

9.1 Two-Pass and One-Pass Assembly

- ✓ Section 10.1 describes the 80x86 floating-point architecture.
- ✓ Section 10.2 describes how to convert floating-point values to and from other formats, including ASCII.
- ✓ Section 10.3 shows floating-point emulation routines of addition, subtraction, multiplication, division, negation, and comparison operations-these routines are useful for floating-point operations on an 80x86 system without built-in floating-point instructions.
- ✓ Section 10.4 gives a brief introduction into using in-line assembly code in C++ code, with C++ for input/output operations, and assembly language for floating-point operations. In-line assembly code is not restricted to floating-point instructions, however.

Chapter10: Floating-Point Arithmetic
10.1 80x86 Floating-Point Architecture

The floating-point unit (FPU) of the chip is almost independent of the rest of the chip.

It has its own internal registers, completely separate from the familiar 80x86 registers.

It executes instructions to do floating-point arithmetic operations, including commonplace operations such as addition or multiplication, and more complicated operations such as evaluation of some transcendental functions.

Not only can it transfer floating-point operands to or from memory, it can also transfer integer or BCD operands to or from the coprocessor.

Nonfloating formats are always converted to floating point when moved to a floating-point register; a number in internal floating-point format can be converted to integer or BCD format as it is moved to memory.

Chapter10: Floating-Point Arithmetic
10.1 80x86 Floating-Point Architecture

The FPU has eight data registers, each 80 bits long.

A ten-byte floating-point format (also specified by IEEE standards) is used for values stored in these registers.

The registers are basically organized as a stack;

for example, when the fld (floating load) instruction is used to transfer a value from memory to the floating point unit, the value is loaded into the register at the top of the stack, and data stored in the stack top and other registers are pushed down one register.

However, some instructions can access any of the eight registers, so that the organization is not a "pure" stack.

Chapter10: Floating-Point Arithmetic
10.1 80x86 Floating-Point Architecture

The names of the eight floating-point registers are

- ✓ **ST, the stack top, also called ST(0),**
- ✓ **ST(1), the register just below the stack top,**
- ✓ **ST(2), the register just below ST(1),**
- ✓ **ST(3), ST(4), ST(5), ST(6), and**
- ✓ **ST(7), the register at the bottom of the stack.**

In addition to the eight data registers, the floating-point unit has several 16-bit control registers. Some of the status word bits are assigned values by floating-point comparison instructions, and these bits must be examined in order for the 80x86 to execute conditional jump instructions based on floating-point comparison. Bits in the FPU control word must sometimes be set to ensure certain modes of rounding.

Chapter10: Floating-Point Arithmetic

10.1 80x86 Floating-Point Architecture

Before considering the floating-point instructions, a few notes are in order.

Each floating-point mnemonic starts with the letter *F*,

a letter that is not used as the first character of any nonfloating instruction.

Most floating-point instructions act on the stack top ST and one other operand in another floating-point register or in memory.

No floating-point instruction can transfer data between an 80x86 general register and a floating-point register

Transfers must be made using a memory location for intermediate storage.

Chapter10: Floating-Point Arithmetic
10.1 80x86 Floating-Point Architecture

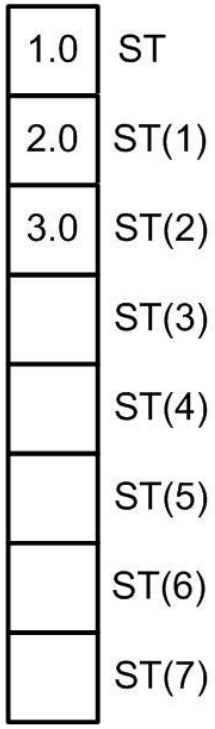
The floating-point instructions will be examined in groups, starting with instructions to push operands onto the stack. Figure 10.1 lists these mnemonics.

Mnemonic	Operand	Action
fld	memory(real)	real value from memory pushed onto stack
fild	memory(integer)	integer value from memory converted to floating point and pushed onto stack
fbld	memory(BCD)	BCD value from memory converted to floating point and pushed onto stack
fld	st(num)	contents of floating-point register pushed onto stack
fldl	(none)	1.0 pushed onto stack
fldz	(none)	0.0 pushed onto stack
fldpi	(none)	π(pi) pushed onto stack
fldl2e	(none)	log2(e) pushed onto stack
fldl2t	(none)	log2(10) pushed onto stack
fldlg2	(none)	log10(2) pushed onto stack
fldln2	(none)	loge(2) pushed onto stack

Figure 10.1 Floatin-point load instructions

Chapter10: Floating-Point Arithmetic
10.1 80x86 Floating-Point Architecture

Some examples illustrate how these instructions work. Suppose that the floating-point register stack contains



with values shown in decimal rather than in IEEE floating-point format. If the data segment contains

```
fpValue     REAL4   10.0
intValue    DWORD   20
bcdValue    TBYTE   30
```

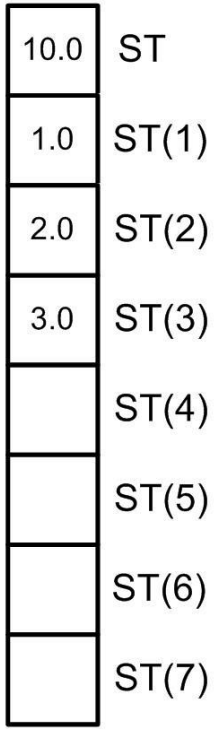
then the values assembled will be 41200000 for *fpValue*, 00000014 for *intValue*, and 00000000000000000030 for *bcdValue*.

Chapter10: Floating-Point Arithmetic
10.1 80x86 Floating-Point Architecture

If the instruction

```
fld   fpValue
```

is executed, the register stack will contain



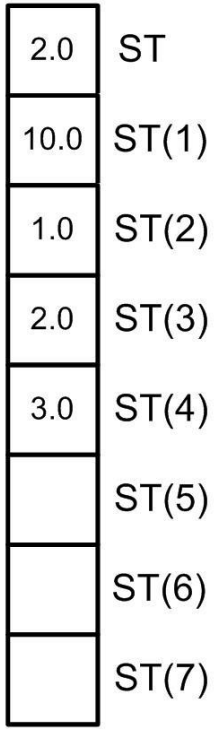
The original values have all been pushed down one register position on the stack. Starting with these values, if the instruction fld st(2) is executed, the register stack will contain

Chapter10: Floating-Point Arithmetic
10.1 80x86 Floating-Point Architecture

Starting with previous values, if the instruction

```
fld   st(2)
```

is executed, the register stack will contain



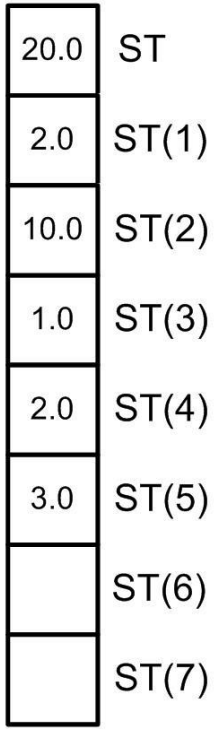
Notice that the value 2.0 from ST(2) has been pushed onto the top of the stack, but not removed from the stack. Starting with these values, assume that the instruction fild intValue is executed. The new contents of the register stack will be

Chapter10: Floating-Point Arithmetic
10.1 80x86 Floating-Point Architecture

Starting with previous values, assume that the instruction

```
fild   intValue
```

is executed. The new contents of the register stack will be



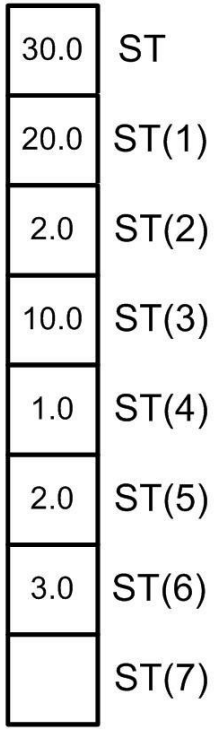
What is not obvious here is that the 32-bit value 00000014 is converted to an 80-bit floating-point value. An integer operand must be word length, doubleword length, or quadword length—byte length integer operands are allowed.

Chapter10: Floating-Point Arithmetic
10.1 80x86 Floating-Point Architecture

If the instruction

```
fbld   bcdValue
```

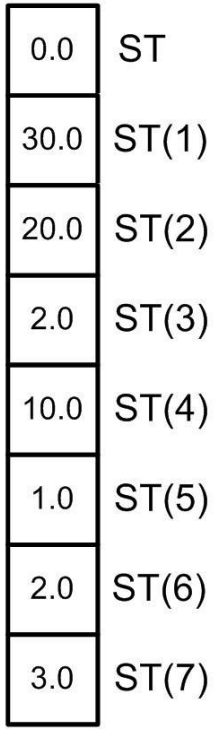
is now executed, the stack values will become



where the 80 bit BCD value is converted to the very different 80 bit floating-point format.

Chapter10: Floating-Point Arithmetic
10.1 80x86 Floating-Point Architecture

Finally, if the instruction fldz is executed, the register stack will contain



The stack is now full. No further value can be pushed onto the stack unless some value is popped from the stack, or the stack is cleared.

The instruction `finit` initializes the floating-point unit and clears the contents of all eight registers. Often a program that uses the floating-point unit will include the statement

```
finit       ;initialize the math processor
```

near the beginning of the code.

Chapter10: Floating-Point Arithmetic
10.1 80x86 Floating-Point Architecture

You can trace floating-point operations using Windbg. Figure 10.2 shows a screen dump following execution of the code on the left pane. A floating-point window is shown in the right pane.

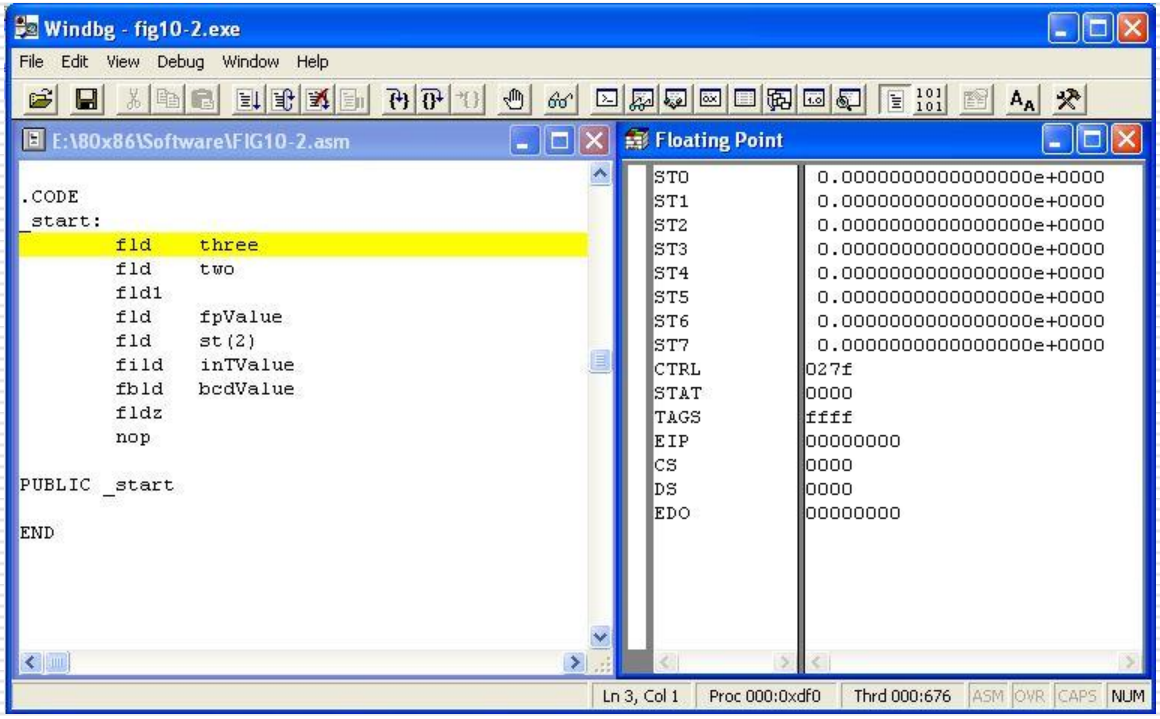


Figure 10.2: Windbg view of floating point execution

Chapter10: Floating-Point Arithmetic
10.1 80x86 Floating-Point Architecture

Figure 10.3 lists the floating-point instructions that are used to copy data from the stack top to memory or to another floating-point register.

Mnemonic	Operand	Action
fst	st(num)	replaces contents of ST(num) by copy of value from ST; only ST(num) is affected
fstp	st(num)	replaces contents of ST(num) by copy of value from ST; ST popped off the stack
fst	memory(real)	copy of ST stored as real value in memory; the stack is not affected
fstp	memory(real)	copy of ST stored as real value in memory; ST popped off the stack
fist	memory(integer)	copy of ST converted to integer and stored in memory
fistp	memory(integer)	copy of ST converted to integer and stored in memory; ST popped off the stack
fbstp	memory(BCD)	copy of ST converted to BCD and stored in memory; ST popped off the stack

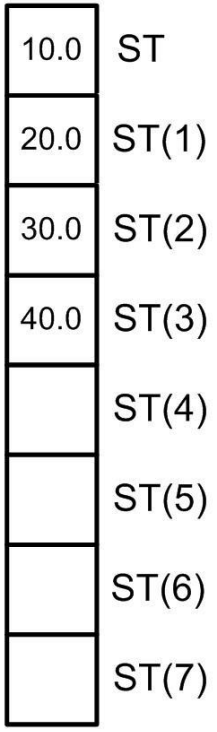
Figure 10.3: Floating-point data store instructions

Chapter10: Floating-Point Arithmetic
10.1 80x86 Floating-Point Architecture

A few examples illustrate the actions of and the differences between these instructions. Assume that the directive

```
intValue    DWORD   ?
```

is coded in the data segment. Suppose that the floating-point register stack contains



Chapter10: Floating-Point Arithmetic
10.1 80x86 Floating-Point Architecture

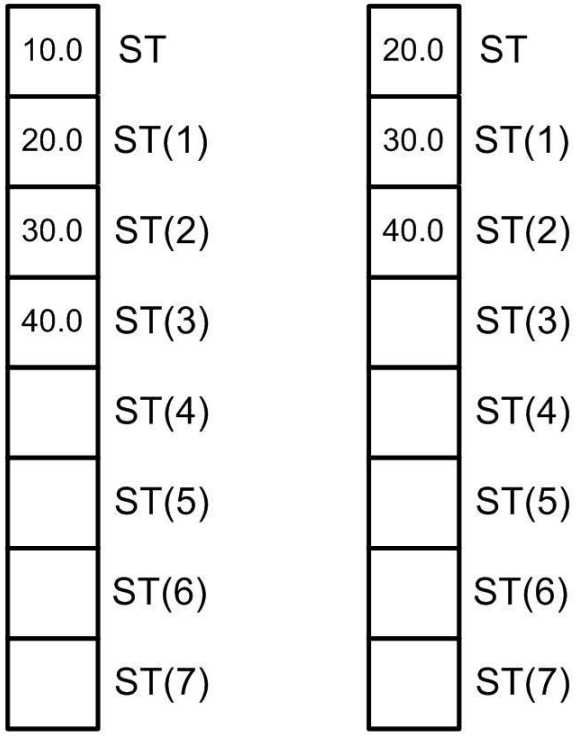
The left diagram shows the resulting stack if

```
fist   intValue
```

is executed and the right diagram shows the resulting stack if

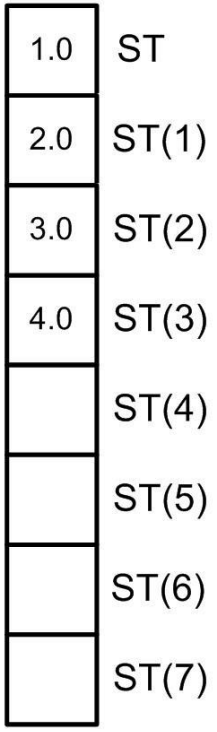
```
fistp   intValue
```

is executed. In both cases, the contents of *intValue* will be 0000000A, the doubleword length 2's complement integer version of the floating-point number 10.0.



Chapter10: Floating-Point Arithmetic
10.1 80x86 Floating-Point Architecture

The situation is a bit more confusing when the destination is one of the floating-point registers. Suppose that at execution time the floating register stack contains



Chapter10: Floating-Point Arithmetic
10.1 80x86 Floating-Point Architecture

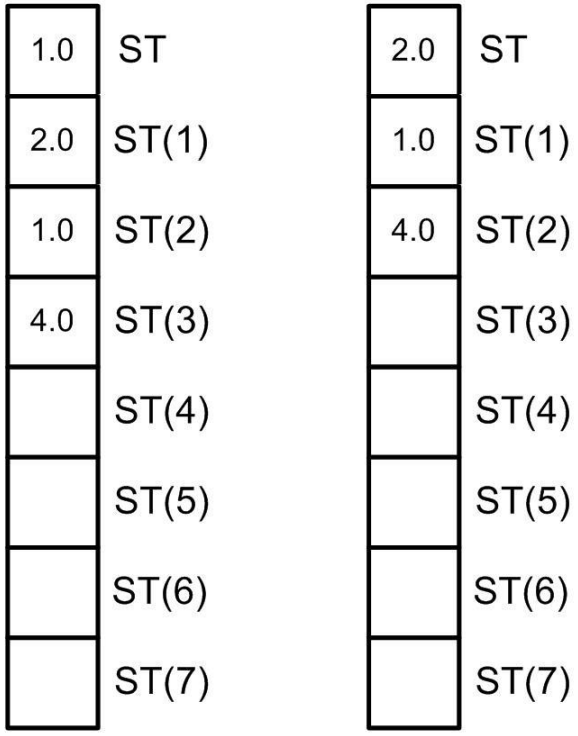
The left diagram shows the resulting stack if

```
fst   st(2)
```

is executed and the right diagram shows the resulting stack if

```
fstp   st(2)
```

is executed. In the first case, a copy of ST has been stored in ST(2). In the second case, the copy has been made, and then the stack has been popped.



Chapter10: Floating-Point Arithmetic
10.1 80x86 Floating-Point Architecture

In addition to the load and store instructions listed above, the floating-point unit has an **fxch** instruction that will exchange the contents of two floating-point registers. With no operand,

```
fxch                    ;exchange ST and ST(1)
```

will exchange the contents of the stack top and ST(1) just below ST on the stack.

With a single operand, for example,

```
fxch    st(3)           ;exchange ST and ST(3)
```

will interchange ST with the specified register.

Chapter10: Floating-Point Arithmetic
10.1 80x86 Floating-Point Architecture

Figure 10.4 shows the floating-point addition instructions.

Mnemonic	Operand	Action
fadd	none	pops both ST and ST(1); adds these values; pushes sum onto the stack
fadd	st(num),st	adds ST(num) and ST; replaces ST(num) by the sum
Fadd	st,st(num)	adds ST and ST(num); replaces ST by the sum
fadd	memory(real)	adds ST and real number from memory; replaces ST by the sum
fiadd	memory(integer)	adds ST and integer from memory; replaces ST by the sum
faddp	st(num),st	adds ST(num) and ST; replaces ST(num) by the sum; pops ST from stack

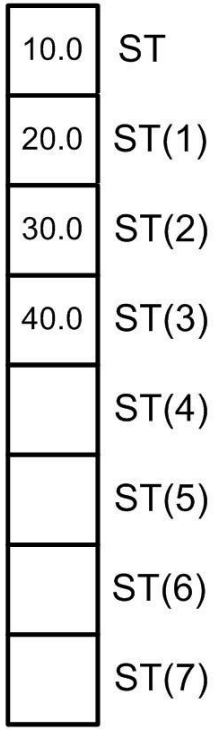
Figure 10.4: Floating-point addition instructions

Chapter10: Floating-Point Arithmetic
10.1 80x86 Floating-Point Architecture

A few examples illustrate how the floating-point addition instructions work. Suppose that the data segment contains the directives

```
fpValue     REAL4   5.0
intValue    DWORD   1
```

and that the floating-point register stack contains

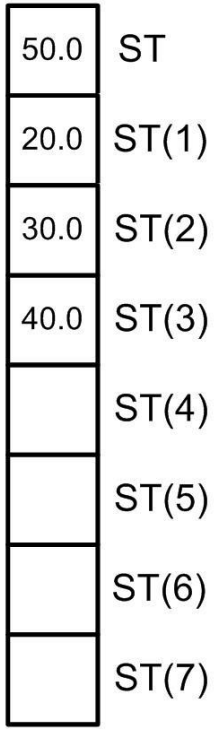


Chapter10: Floating-Point Arithmetic
10.1 80x86 Floating-Point Architecture

After the instruction

```
fadd   st,st(3)
```

is executed, the stack contains

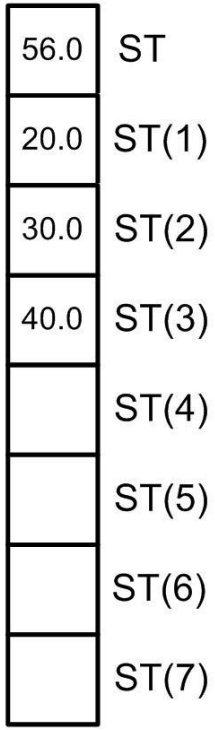


Chapter10: Floating-Point Arithmetic
10.1 80x86 Floating-Point Architecture

Starting with these stack values, after the two instructions

```
fadd    fpValue
fiadd   intValue
```

are executed, the contents of the stack are

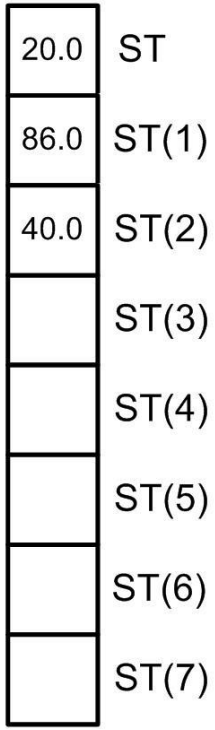


10.1 80x86 Floating-Point Architecture

Finally, if the instruction

```
faddp   st(2),st
```

is executed, the stack will contain



Chapter10: Floating-Point Arithmetic
10.1 80x86 Floating-Point Architecture

Subtraction instructions are displayed in Fig. 10.5. The first six instructions are very similar to the corresponding addition instructions. The second six subtraction instructions are the same except that the operands are subtracted in the opposite order.

Mnemonic	Operand	Action
fsub	none	pops ST and ST(1); calculates ST(1)- ST; pushes difference onto the stack
fsub	st(num),st	calculates ST(num)-ST; replaces ST(num) by the differences
fsub	st,st(num)	calculates ST-ST(num); replaces ST by the differences
fsub	memory(real)	calculates ST −real number from memory; replaces ST by the difference
fisub	memory(integer)	calculates ST −integer from memory; replaces ST by the difference
fsubp	st(num),st	calculates ST(num)-ST; replaces ST(num) by the differences, pops ST from the stack

Figure 10.5: Floating-point subtraction instructions

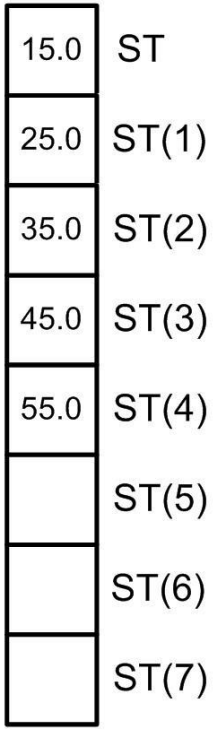
10.1 80x86 Floating-Point Architecture

Mnemonic	Operand	Action
fsubr	none	pops ST and ST(1); calculates ST-ST(1); pushes difference onto the stack
fsubr	st(num),st	calculates ST-ST(num); replaces ST(num) by the differences
fsub	st,st(num)	calculates ST(num)-ST; replaces ST by the differences
fsubr	memory(real)	calculates real number from memory-ST; replaces ST by the difference
fisubr	memory(integer)	calculates integer from memory-ST; replaces ST by the difference
fsubpr	st(num),st	calculates ST-ST(num); replaces ST(num) by the differences, pops ST from the stack

Figure 10.5: Floating-point subtraction instructions (cont.)

Chapter10: Floating-Point Arithmetic
10.1 80x86 Floating-Point Architecture

An example illustrates the difference between the parallel subtraction instructions. Suppose that the floating-point register stack contains



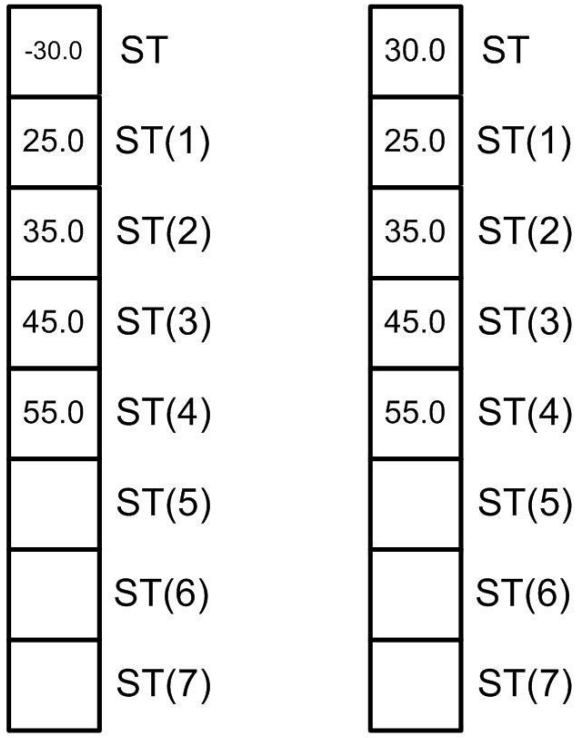
Chapter10: Floating-Point Arithmetic
10.1 80x86 Floating-Point Architecture

The two diagrams below show the results after executing the instructions

```
fsub    st,st(3)
```
(left)

and

```
fsubr   st,st(3)
```
(right)



Chapter10: Floating-Point Arithmetic
10.1 80x86 Floating-Point Architecture

Multiplication and division instructions are listed in Figs. 10.6 and 10.7, respectively. Multiplication instructions have the same forms as the addition instructions in Fig. 10.4.

Mnemonic	Operand	Action
fmul	none	pops ST and ST(1); multiplies these values; pushes product onto the stack
fmul	st(num),st	multiplies ST(num) and ST; replaces ST(num) by the product
fmul	st,st(num)	multiplies ST and ST(num); replaces ST by the product
fmul	memory(real)	multiplies ST and real number from memory; replaces ST by the product
fimul	memory(integer)	multiplies ST and integer from memory; replaces ST by the product
fmulp	st(num),st	multiplies ST (num) and ST; replaces ST (num) by the product; pops ST from stack

Figure 10.6: Floating-point multiplication instructions

Chapter10: Floating-Point Arithmetic
10.1 80x86 Floating-Point Architecture

Division instructions have the same forms as subtraction instructions in Fig. 10.5, that is, the *R* versions reverse the operands' dividend and divisor roles.

Mnemonic	Operand	Action
fdiv	none	pops ST and ST(1); calculates ST(1) / ST; pushes quotient onto the stack
fdiv	st(num),st	calculates ST(num) / ST; replaces ST(num) by the quotient
fdiv	st,st(num)	calculates ST / ST(num); replaces ST by the quotient
fdiv	memory(real)	calculates ST / real number from memory; replaces ST by the quotient
fidiv	memory(integer)	calculates ST / integer from memory; replaces ST by the quotient
fdivp	st(num),st	calculates ST (num) / ST; replaces ST (num) by the quotient; pops ST from the stack

Figure 10.7: Floating-point division instructions

Chapter10: Floating-Point Arithmetic
10.1 80x86 Floating-Point Architecture

Mnemonic	Operand	Action
fdivr	none	pops ST and ST(1); calculates ST / ST(1); pushes quotient onto the stack
fdivr	st(num),st	calculates ST / ST(num); replaces ST(num) by the quotient
fdivr	st,st(num)	calculates ST(num) / ST; replaces ST by the quotient
fdivr	memory(real)	calculates real number from memory / ST; replaces ST by the quotient
fidivr	memory(integer)	calculates integer from memory / ST; replaces ST by the quotient
fdivpr	st(num),st	calculates ST / ST (num); replaces ST (num) by the quotient; pops ST from the stack

Figure 10.7: Floating-point division instructions (cont.)

Chapter10: Floating-Point Arithmetic
10.1 80x86 Floating-Point Architecture

Figure 10.8 describes four additional floating-point instructions. Additional instructions that calculate tangent, arctangent, exponent, and logarithm functions are not covered here.

Mnemonic	Operand	Action
fabs	none	ST := | ST | (absolute value)
fchs	none	ST := – ST (change sign)
frndint	none	rounds ST to an integer value
fsqrt	none	replace the contents of ST by its square root

Figure 10.8: Additional Floating-point instructions

Chapter10: Floating-Point Arithmetic
10.1 80x86 Floating-Point Architecture

The floating-point unit provides a collection of instructions to compare the stack top ST to a second operand. These are listed in Fig. 10.9.

Mnemonic	Operand	Action
fcom	none	compares ST and ST(1)
fcom	st(num)	compares ST and ST(num)
fcom	memory(real)	compares ST and real number in memory
ficom	memory(integer)	compares ST and integer in memory
ftst	none	compares ST and 0.0
fcomp	none	compares ST and ST(1), then pops stack
fcomp	st(num)	compares ST and ST(num) , then pops stack
fcomp	memory(real)	compares ST and real number in memory, then pops stack
ficomp	memory(integer)	compares ST and integer in memory, then pops stack
fcompp	st(num),st	compares ST and ST(1), then pops stack twice

Figure 10.0: Floating-point comparison instructions

Chapter10: Floating-Point Arithmetic
10.1 80x86 Floating-Point Architecture

Recall that the floating point has a 16-bit control register called the status word. The comparison instructions assign values to bits 14, 10, and 8 in the status word; these "condition code" bits are named C3, C2, and C0, respectively. These flags are set as follows:

result of comparison	*C3*	*C2*	*C0*
ST > second operand	**0**	**0**	**0**
ST < second operand	**0**	**0**	**1**
ST = second operand	**1**	**0**	**0**

Another possibility is that the operands are not comparable. This can occur if one of the operands is the IEEE representation for infinity or NaN (not a number). In this case, all three bits are set to 1.

Chapter10: Floating-Point Arithmetic
10.1 80x86 Floating-Point Architecture

If a comparison is made in order to determine program flow, simply setting flags in the status word is no help. Conditional jump instructions look at bits in the flag register in the 80x86, not the status word in the floating-point unit.

Consequently, the status word must be copied to memory or to the AX register before its bits can be examined by an 80x86 instruction, perhaps with a test instruction. The floating-point unit has two instructions to store the status word; these are summarized in Fig. 10.10. This table also shows the instructions for storing or setting the control word.

Mnemonic	Operand	Action
fstsw	memory word	copies status register to memory word
fstsw	AX	copies status register to AX
fstcw	memory word	copies control word register to memory word
fldcw	memory word	copies memory word to control word register

Figure 10.10: Misellaneous Floating-point instructions

Chapter10: Floating-Point Arithmetic

10.2 Programming with Floating-Point Instructions

This section gives three examples of coding with floating-point instructions.

- ✓ **The first is a program that calculates the square root of the sum of the squares of two numbers.** (Although we do not yet have any procedures to facilitate input/output of floating-point values, FPU operations can be viewed through Windbg)

- ✓ **The second and third examples show procedure to facilitate input/output of floating-point numbers.**

Figure 10.11 has a listing of the first example.

Floating-point values are assembled at *value1* and *value2*. The first instruction copies *value1* from memory to ST. The second instruction copies it from ST to ST, pushing down the first stack entry to ST(1). The third instruction gives *value1***value1* in ST, with "nothing" in ST(1).

The same sequence of instructions is repeated for *value2*.

Chapter10: Floating-Point Arithmetic

10.2 Programming with Floating-Point Instructions

```
; find the sum of the squares of two floating-point numbers
; Author: R. Detmer
; Date: 4/98 .386

.MODEL FLAT
.STACK 4096 ; reserve 4096-byte stack

.DATA ; reserve storage for data
value1 REAL4 0.5
value2 REAL4 1.2
sqrt REAL4 ?

.CODE _start:
            fld         value1              ;value1 in ST
            fld         st                  ;value1 in ST and ST(1)
            fmul                            ;value1*value1 in ST
            fld         value2              ;value2 in ST (value1*value1 in ST(1))
            fld         st                  ;value2 in ST and ST(1)
            fmul                            ;value2*value2 in ST
            fadd                            ;sum of squares in ST
            fsqrt                           ;square root of sum of squares in ST
            fstp        sqrt                ;store result
PUBLIC _start
END
```

Figure 10.11: Floating-point computations

Chapter10: Floating-Point Arithmetic

10.2 Programming with Floating-Point Instructions

Figure 10.12 shows Windbg's view of the CPU just before the second fmul is executed. At this point, there are copies of *value2* in both ST and ST(1) and *value1*value1* in ST(2). After the result is calculated in ST, it is stored in *sqrt* and popped from the stack, leaving the stack in its original state.

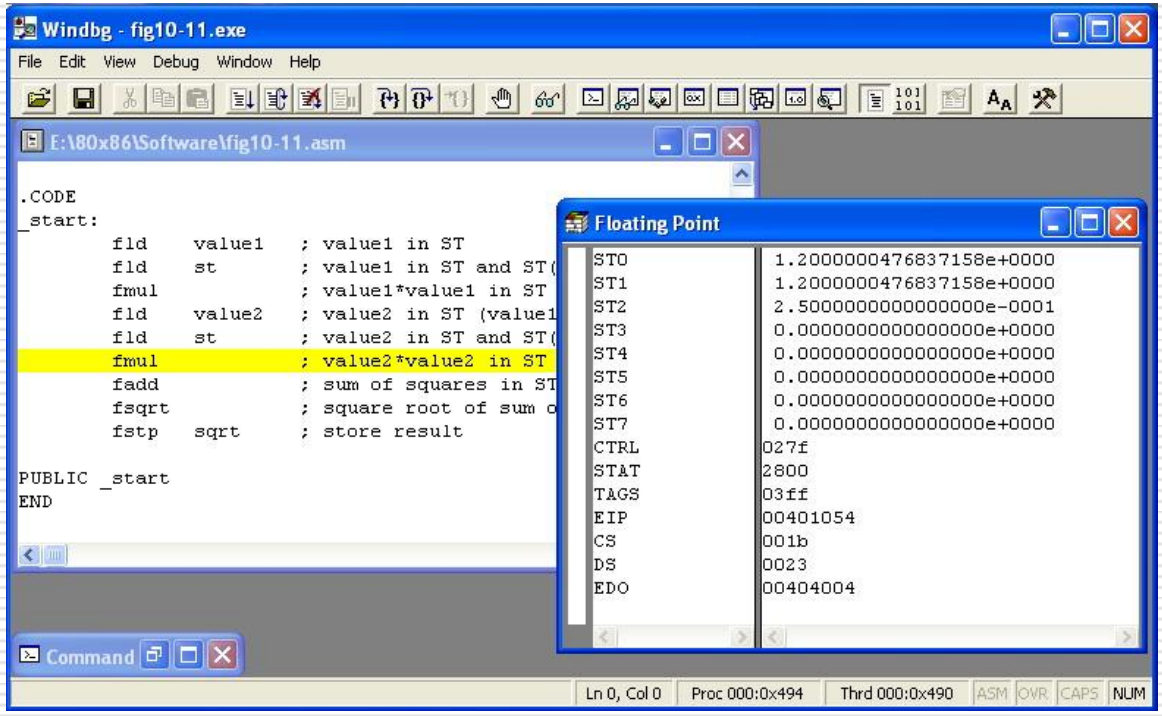


Figure 10.12: Execution of floating-point example

Notice that the value 1.2 is shown in Fig. 10.12 as 1.20000004768371 58e+0000.

The reason that there are nonzero digits after the decimal point is that 1.2 does not have an exact representation as a floating point number. The approximation used by the 32-bit REAL4 directive translates back to the number shown in 17-decimal-digit precision. You can get a better approximation by using a REAL8 or a REAL10 directive, but at the cost of extra bytes of storage.

Chapter10: Floating-Point Arithmetic

10.2 Programming with Floating-Point Instructions

The second example is an implementation of a

simple ASCII to floating-point conversion algorithm.

This algorithm, given in Fig. 10.13, is similar to the one used by the *atoi* and *atod* macros—it scans memory at the address given by its parameter, interpreting the characters as a floating point.

Chapter10: Floating-Point Arithmetic

10.2 Programming with Floating-Point Instructions

```
value := 0.0;
divisor := 1.0;
point := false;
minus := false;

point at first character of source string;
if source character = '-'
then
     minus := true;
     point at next character of source string;
end if;

while (source character is a digit or a decimal point) loop
     if source character = '.'
     then
          point := true;
     else
          convert ASCII digit to 2's complement digit;
          value := 10*value + float(digit);
          if point
          then
                    multiply divisor by 10;
          end if;
     end if;
     point at next character of source string;
end while;
```

```
value := value/divisor;

if minus
then
     value := -- value;
end if;
```

Figure 10.13: ASCII to floating-point algorithm

This algorithm is implemented in a NEAR32 procedure *atofproc*. This procedure has one parameter—the address of the string. It returns the floating-point value in ST. No flags are set to indicate illegal conditions, such as multiple minus signs or decimal points. The code appears in Fig. 10.14.

Chapter10: Floating-Point Arithmetic

10.2 Programming with Floating-Point Instructions

This implementation of the ASCII to floating-point algorithm uses ST(1) for *divisor* and ST for *value* except for one short segment where they are reversed in order to modify *divisor*. After the procedure entry code, the instructions

```
fld1                ;divisor := 1.0
fldz                ;value := 0.0
```

initialize these two variables. Note that the value 1.0 for *divisor* ends up in ST(1) since it is pushed down by the fldz instruction.

Chapter10: Floating-Point Arithmetic
10.2 Programming with Floating-Point Instructions

The design element

```
value := 10*value + float(digit);
```

is implemented by the code

```
fmul        ten             ;value := value * 10
fiadd       digit           ;value := value + digit
```

Note that a word-length 2's complement integer version of *digit* is stored in memory. The floating-point unit takes care of converting it to floating point as part of the fiadd instruction.

Chapter10: Floating-Point Arithmetic
10.2 Programming with Floating-Point Instructions

The design element

```
value := 10*value + float(digit);
```

is implemented by the code

```
fmul        ten             ;value := value * 10
fiadd       digit           ;value := value + digit
```

Note that a word-length 2's complement integer version of *digit* is stored in memory. The floating-point unit takes care of converting it to floating point as part of the fiadd instruction.

Chapter10: Floating-Point Arithmetic

10.2 Programming with Floating-Point Instructions

To implement "multiply divisor by 10," the number to be multiplied must be in ST. The instructions

```
fxch            ;put divisor in ST and value in ST(1)
fmul    ten     ;divisor := divisor * 10
fxch            ;value back to ST; divisor back to ST(1)
```

take care of swapping *divisor* and *value*, carrying out the multiplication in ST, and then swapping back.

When it is time to execute "*value* := *value* / *divisor*" the instruction]

```
fdivr                   ;value := value / divisor
```

pops *value* from ST and *divisor* from ST(1), computes the quotient, and pushes it back to ST. Notice that the fdiv version of this instruction would incorrectly compute "*divisor*/*value*." After the division instruction, ST(1) is no longer in use by this procedure. The instruction fchs changes the sign of *value* if a leading minus sign was noted in the ASCII string.

You can test *atofproc* with a simple test driver program such as the one shown in Fig. 10.15. The "output" of the procedure can be viewed using Windbg.

Chapter10: Floating-Point Arithmetic

10.2 Programming with Floating-Point Instructions

```
;test drive for atofproc
;Author: R. Detmer
;Date: 4/98

.386
.MODEL FLAT

ExitProcess PROTO NEAR32 stdcall, dwExitCode:DWORD
EXTRN atofproc:NEAR32
.STACK 4096                    ;reserve 4096-byte stack

.DATA
;reserve storage for data
String    BYTE      "435.75", 0
.CODE

;program code _
start:
          pushd     NEAR32    PTR String
          call atofproc
          INVOKE ExitProcess, 0
PUBLIC _start
END
```

Figure 10.15: Test driver for atofproc

Chapter10: Floating-Point Arithmetic

10.2 Programming with Floating-Point Instructions

Finally we come to a procedure to convert a floating-point parameter to "E notation." The procedure generates a 12-byte long ASCII string consisting of

- ✓ a leading minus sign or a blank
- ✓ a digit
- ✓ a decimal point
- ✓ five digits
- ✓ the letter *E*
- ✓ a plus sign or a minus sign
- ✓ two digits

This string represents the number in base 10 scientific notation. For example, for the decimal value 145.8798, the procedure would generate the string *b*1.45880E+02, where *b* represents a blank. Notice that the ASCII string has a rounded value.

Chapter10: Floating-Point Arithmetic

10.2 Programming with Floating-Point Instructions

Figure 10.16 displays the design for the floating to ASCII procedure. After the leading space or minus sign is generated, most of the work necessary to get the remaining characters is done before they are actually produced.

The value is repeatedly multiplied or divided by 10 until it is at least 1.0 but less than 10.0. Multiplication is used if the value is initially less than 1; the number of multiplications gives the negative power of 10 required for scientific notation. Division is used if the value is initially 10.0 or more; the number of divisions gives the positive power of 10 required for scientific notation.

Only five digits are going to be displayed after the decimal point. The value between 1.0 and 10.0 is rounded by adding 0.000005; if the sixth digit after the decimal point is 5 or greater, this will be reflected in the digits that are actually displayed. It is possible that this addition gives a sum of 10.0 or more; if this happens, the value is divided by 10 again and the exponent is incremented.

Chapter10: Floating-Point Arithmetic

10.2 Programming with Floating-Point Instructions

Figure 10.17 shows this design implemented in a procedure named *ftoaproc*. The procedure has two parameters: first, the floating-point value to be converted and second, the address of the destination string.

Chapter10: Floating-Point Arithmetic

10.3 Floating-Point Emulation

Some 80x86 computer systems have no floating-point unit. Such a system can still do floating-point arithmetic. However, floating-point operations must be performed by software routines using memory and the general purpose registers, rather than by a floating-point unit.

This section describes procedures for multiplication and for addition of floating-point numbers. These could be useful for floating-point emulation, and they also provide a better understanding of the floating-point representation.

Chapter10: Floating-Point Arithmetic

10.3 Floating-Point Emulation

The procedures in this section manipulate floating-point values in the IEEE single format. Recall from Section 1.5 that this scheme includes the pieces that describe a number in "base two scientific notation":

- ✓ **a leading sign bit for the entire number, 0 for positive and 1 for negative**
- ✓ **an 8-bit biased exponent (or characteristic). This is the actual exponent plus a bias of 127_{10}**
- ✓ **23 bits that are the fraction (or mantissa) expressed with the leading 1 removed**

This is the format produced by the REAL4 directive

Chapter10: Floating-Point Arithmetic
10.3 Floating-Point Emulation

Each procedure combines the components of its parameters to yield a result in the structure fp3. Often this result is not normalized; that is, there are not exactly 24 significant fraction bits. The NEAR procedure *normalize* adjusts the fraction and exponent to recover the standard format.

Notice that there is a problem representing the number 0.0 using the normal IEEE scheme. There is no "binary scientific notation" zero with a 1 bit preceding the binary point of the fraction. The best that can be done is 1.0 2 127, which is small, but nonzero. According to the rules given previously, this value would have an IEEE representation consisting of 32 zero bits. However, the two bit patterns that end with 31 zeros are considered special cases, and each is interpreted as 0.0 instead of plus or minus 1.0 −2 −127. These special cases will be considered in the following multiplication and addition code.

Chapter10: Floating-Point Arithmetic
10.3 Floating-Point Emulation

In addition to a special bit pattern to represent 0.0, the IEEE standard describes three other distinctive situations. The pattern

s **11111111 00000000000000000000000**

(sign bit s, biased exponent 255, and fraction 0) represents plus or minus infinity. These values are used, for example, as quotients when a nonzero number is divided by zero. Another special case is called NaN (not a number) and is represented by any bit pattern with a biased exponent of 255 and a nonzero fraction. The quotient 0/0 should result in NaN, for example. The final special case is a denormalized number; when the biased exponent is zero and the fraction is nonzero, then no leading 1 is assumed for the fraction. This allows for representation of extra small numbers. Code in this section's floating-point procedures looks for the special zero representations wherever needed. However, other special number forms are ignored.

Chapter10: Floating-Point Arithmetic

10.3 Floating-Point Emulation

We will frequently need to extract the sign, exponent, and fraction of a floating-point number. For this purpose we will use a macro *expand*. This macro will have four parameters

- ✓ a 32-bit floating point number
- ✓ a byte to hold the sign (0 for positive, 1 for negative)
- ✓ a word to hold the unbiased (actual) exponent
- ✓ a doubleword to hold the fraction, including the leading 1 for a nonzero number.

Code for the macro *expand* appears in Fig. 10.18.

Chapter10: Floating-Point Arithmetic
10.3 Floating-Point Emulation

The