Real Arithmetic Chapter Eleven

11.1 Chapter Overview

This chapter discusses the implementation of floating point arithmetic computation in assembly language. By the conclusion of this chapter you should be able to translate arithmetic expressions and assignment statements involving floating point operands from high level languages like Pascal and C/C++ into 80x86 assembly language.

11.2 Floating Point Arithmetic

When the 8086 CPU first appeared in the late 1970's, semiconductor technology was not to the point where Intel could put floating point instructions 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)[1]. 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 chips provide a high-performance floating point unit directly on the CPU. There is no (Intel) floating point coprocessor available for the Pentium chip.

Collectively, we will refer to all these chips as the 80x87 FPU. Given the obsolescence of the 8086, 80286, 8087, 80287, 80387, and 80487 chips, this text will concentrate on the Pentium and later chips. There are some differences between the 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.

11.2.1 FPU Registers

The 80x86 FPUs add 13 registers to the 80x86 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 FPU handles certain degenerate cases like rounding of inaccurate computations, it contains bits that 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 FPU. 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 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.

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

11.2.1.1 FPU Data Registers

The 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. HLA refers to these registers as ST0, ST1, …, ST7.

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 FPU, however, the register set is an eight element stack of 80 bit floating point values (see Figure 11.1).

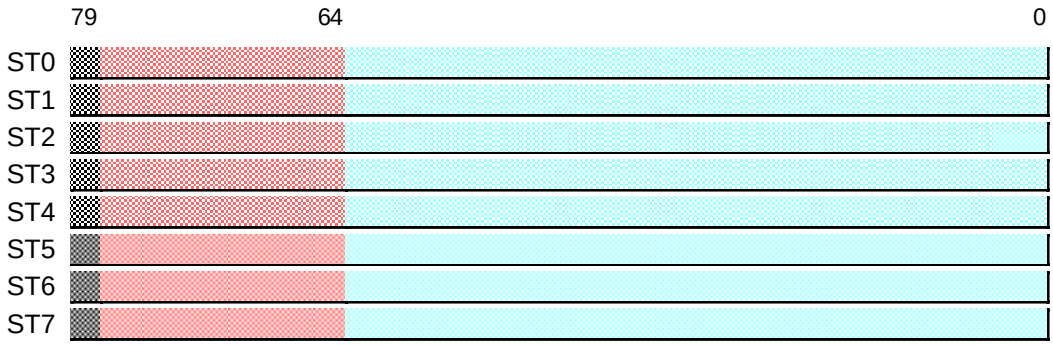

Figure 11.1 FPU Floating Point Register Stack

ST0 refers to the item on the top of the stack, ST1 refers to the next item on the stack, and so on. Many floating point instructions push and pop items on the stack; therefore, ST1 will refer to the previous contents of ST0 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.

11.2.1.2 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 design 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 incompatible. 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 their FPU.

The 80x87 control register contains 16 bits organized as shown in Figure 11.2.

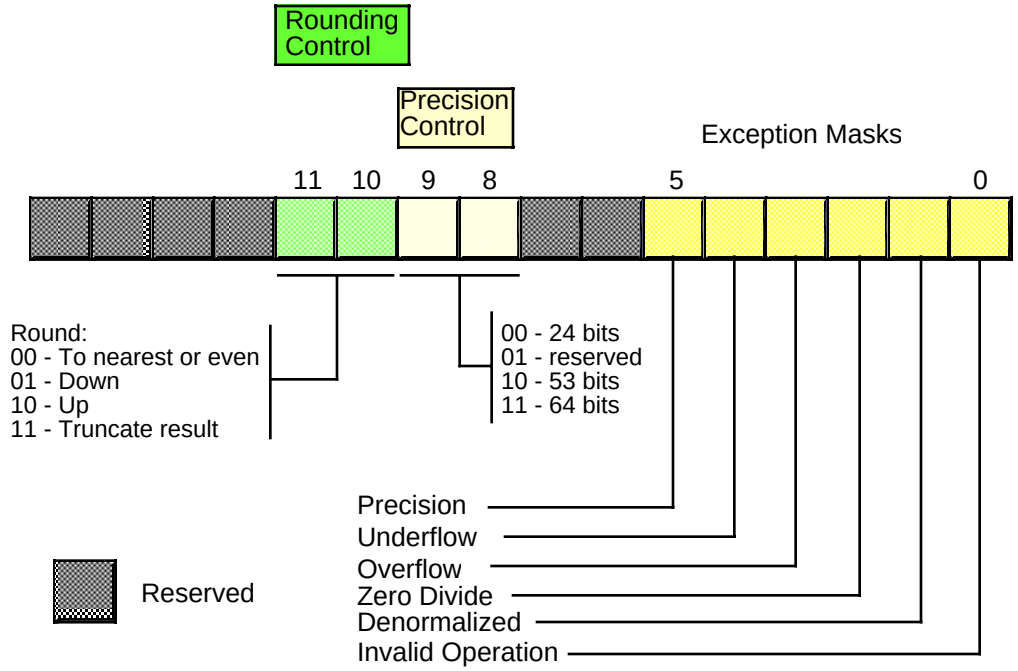

Figure 11.2 FPU Control Register

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

Table 1: 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 FPU 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 of the least significant bit, the FPU 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, then 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 FPU, you might use this option to help when porting the software. One place where this option is extremely useful is when converting a floating point value to an integer. Since most software expects floating point to integer conversions to truncate the result, you will need to use the truncation rounding mode to achieve this.

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

Table 2: Mantissa Precision Control Bits

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

Some CPUs may operate faster with floating point values whose precision is 53 bits (i.e., 64-bit floating point format) rather than 64 bits (i.e., 80-bit floating point format). Please see the documentation for your specific processor for details. Generally, the CPU defaults these bits to %11 to select the 64-bit mantissa precision.

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 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. Problems 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 that occurs whenever you try to manipulate denormalized values. Denormalized exceptions 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. If you enable this exception and the FPU generates this interrupt, the HLA run-time system raises the *ex.fDenormal* 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. It's probably a good idea to enable this exception by programming a zero into this bit. Note that if your program generates this interrupt, the HLA run-time system will raise the *ex.fDivByZero* exception.

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). If you enable this exception and the FPU generates this interrupt, the HLA run-time system raises the *ex.fOverflow* exception.

Bit four, if set, masks the *underflow* exception. Underflow occurs when the result is too *small* to fit in the destination 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. If you enable this exception and the FPU generates this interrupt, the HLA run-time system raises the *ex.fUnderflow* exception.

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. If you enable this exception and the FPU generates this interrupt, the HLA run-time system raises the *ex.InexactResult* exception.

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 FPU 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. The syntax for these instructions is

```
fldcw( mem16 );
fstcw( mem16 );
```

Here's some example code that sets the rounding control to "truncate result" and sets the rounding precision to 24 bits:

```
static
    fcw16: word;
        .
        .
        .
        fstcw( fcw16 );
        mov( fcw16, ax );
        and( $f0ff, ax );      // Clears bits 8-11.
        or( $0c00, ax );       // Rounding control=%11, Precision = %00.
        mov( ax, fcw16 );
        fldcw( fcw16 );
```

11.2.1.3 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 a word variable. The status register is a 16 bit register, its layout appears in Figure 11.3.

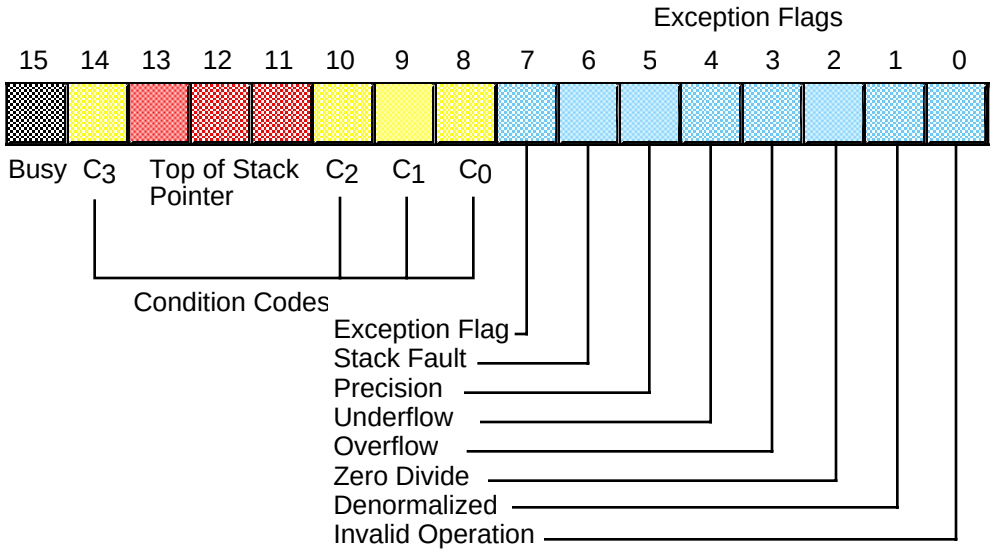

Figure 11.3 The FPU Status Register

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 FPU sets and clears these bits regardless of the corresponding mask setting.

Bit six 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 3: FPU Condition Code Bits

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

Table 3: 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		+ 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,	0	0	X	0	ST > source
fucomp,	0	0	X	1	ST < source
fucompp	1	0	X	0	ST = source
	1	1	X	1	Unordered
	X = Don’t care				

Table 4: Condition Code Interpretations

Instruction(s)	Condition Code Bits			
	C_0	C_3	C_2	C_1
fcom, fcomp, fcmpp, ftst, fucom, fucomp, fucompp, ficom, ficomp	Result of comparison. See previous table.	Result of comparison. See previous table.	Operands are not comparable	Result of comparison. See previous table. Also denotes stack overflow/underflow if stack exception bit is set.

Table 4: Condition Code Interpretations

Instruction(s)	Condition Code Bits			
	C_0	C_3	C_2	C_1
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.

Bits 11-13 of the FPU status register provide the register number of the top of stack. During computations, the FPU 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.

11.2.2 FPU Data Types

The FPU supports seven different data types: three integer types, a packed decimal type, and three floating point types. The integer type provides for 64-bit integers, although it is often faster to do the 64-bit arithmetic using the integer unit of the CPU (see the chapter on Advanced Arithmetic). Certainly it is often faster to do 16-bit and 32-bit integer arithmetic using the standard integer registers. The packed decimal type provides a 17 digit signed decimal (BCD) integer. The primary purpose of the BCD format is to convert between strings and floating point values. 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 11.4, Figure 11.5, and Figure 11.6.

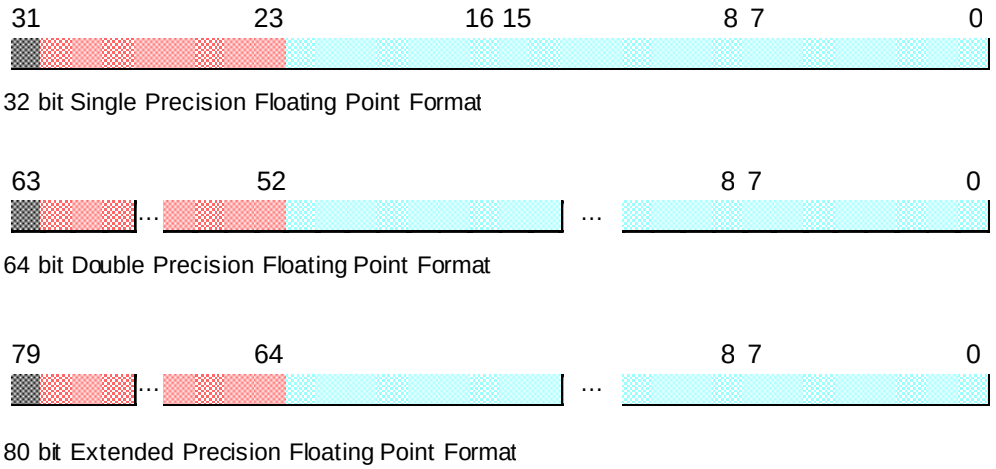

Figure 11.4 FPU Floating Point Formats

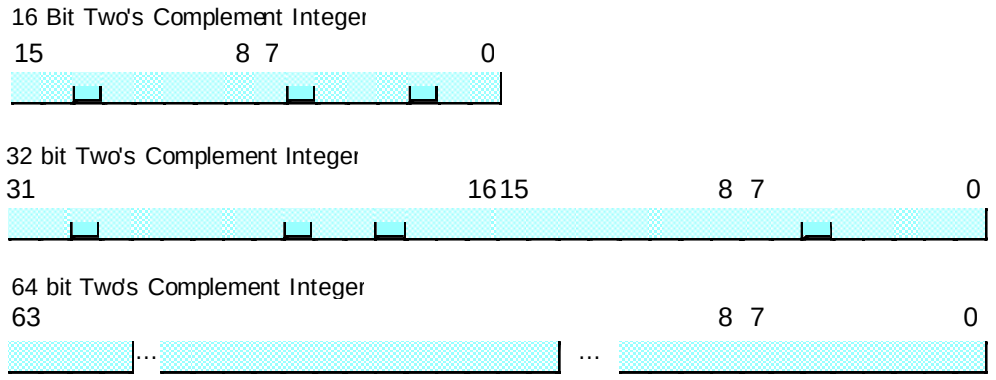

Figure 11.5 FPU Integer Formats

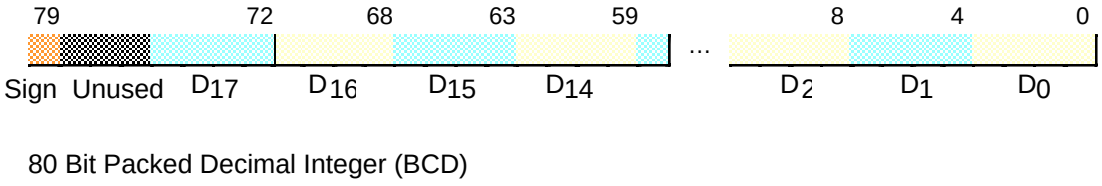

Figure 11.6 FPU Packed Decimal Format

The FPU generally stores values in a *normalized* format. When a floating point number is normalized, the H.O. bit of the mantissa is always one. In the 32 and 64 bit floating point formats, the FPU does not actually store this bit, the FPU 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 FPU does *not* assume that the H.O. bit of the mantissa is one, the H.O. bit of the mantissa 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 *cannot* 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 FPUs support a special 80-bit form known as *denormalized* values. Denormalized values allow the FPU to encode very small values it cannot encode using normalized values, but at a price. Denormalized values offer fewer 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 FPU status register contains a bit you can use to detect when the FPU uses a denormalized value in a computation.

11.2.3 The FPU Instruction Set

The 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.

11.2.4 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 instruction 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). The remaining instructions do not affect the number of items on the stack.

11.2.4.1 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 operands to an 80 bit extended precision value before pushing the value onto the floating point stack.

The FLD instruction first decrements the top of stack (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 FPU uses for the load operation is the register number *before* decrementing the tos pointer. Therefore, "fld(st0);" duplicates the value on the top of the stack.

The FLD instruction sets the stack fault bit if stack overflow occurs. It sets 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( st1 );
fld( real32_variable );
fld( real64_variable );
fld( real80_variable );
fld( real_constant );
```

Note that there is no way to directly load a 32-bit integer register onto the floating point stack, even if that register contains a REAL32 value. To accomplish this, you must first store the integer register into a memory location then you can push that memory location onto the FPU stack using the FLD instruction. E.g.,


```
mov( eax, tempReal32 );      // Save REAL32 value in EAX to memory.
fld( tempReal32 );           // Push that real value onto the FPU stack.
```

Note: loading a constant via FLD is actually an HLA extension. The FPU doesn't support this instruction type. HLA creates a REAL80 object in the "constants" segment and uses the address of this memory object as the true operand for FLD.

11.2.4.2 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 ST0. 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(st0);" instruction effectively pops the data off the top of stack with no data transfer. Examples:

```
fst( real32_variable );
fst( real64_variable );
fst( realArray[ ebx*8 ] );
fst( real80_variable );
fst( st2 );
fstp( st1 );
```

The last example above effectively pops ST1 while leaving ST0 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 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[2]. 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).

Note: Because of an idiosyncrasy in the FPU instruction set related to the encoding of the instructions, you cannot use the FST instruction to store data into a real80 memory variable. You may, however, store 80-bit data using the FSTP instruction.

11.2.4.3 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, the second without any operands. The first form exchanges the top of stack (tos) with the specified register. The second form of FXCH swaps the top of stack with ST1.

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

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

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 ST2:

```
fxch( st2 );
fsqrt();
fxch( st2 );
```

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.

11.2.5 Conversions

The FPU 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 FPU provides five other instructions that convert to or from integer or binary coded decimal (BCD) format when moving data. These instructions are FILD, FIST, FISTP, FBLD, and FBSTP.

11.2.5.1 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. You cannot specify one of the 80x86'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( word_variable );
fild( dword_val[ ecx*4 ] );
fild( qword_variable );
```

11.2.5.2 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 ST0). 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( word_var[ ebx*2 ] );
fist( qword_var );
fistp( dword_var );
```


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 rounding 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, e.g.,

```
static
    fcw16:        word;
    fcw16_2:      word;
    IntResult:    int32;
        .
        .
        .
        fstcw( fcw16 );
        mov( fcw16, ax );
        or( $0c00, ax );       // Rounding control=%11 (truncate).
        mov( ax, fcw16_2 );    // Store into memory and reload the ctrl word.
        fldcw( fcw16_2 );

        fistp( IntResult );      // Truncate ST0 and store as int32 object.

        fldcw( fcw16 );          // Restore original rounding control
```

11.2.5.3 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( st0 );
        fbstp( tbyte_var );

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

        fbld( tbyte_var );
        fist( qword_var );
```

11.2.6 Arithmetic Instructions

The arithmetic instructions make up a small, but important, subset of the FPU'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.

11.2.6.1 The FADD and FADDP Instructions

These two instructions take the following forms:

```
fadd()
faddp()
fadd( st0, sti );
fadd( sti, st0 );
faddp( st0, sti );
fadd( mem_32_64 );
fadd( real_constant );
```

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 source register operand to the value in the destination register operand. Note that one of the register operands must be ST0.

The FADDP instruction with two operands adds ST0 (which must always be the source operand) to the destination operand and then pops ST0. 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 ST0. 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.

Like FLD(real_constant), the FADD(real_constant) instruction is an HLA extension. Note that it creates a 64-bit variable holding the constant value and emits the FADD(*mem64*) instruction, specifying the read-only object it creates in the constants segment.

11.2.6.2 The FSUB, FSUBP, FSUBR, and FSUBRP Instructions

These four instructions take the following forms:

```
fsub()
fsubp()
fsubr()
fsubrp()

fsub( st0, sti )
fsub( sti, st0 );
fsubp( st0, sti );
fsub( mem_32_64 );
fsub( real_constant );

fsubr( st0, sti )
fsubr( sti, st0 );
fsubrp( st0, sti );
fsubr( mem_32_64 );
fsubr( real_constant );
```

With no operands, the FSUB and FSUBP instructions operate identically. They pop ST0 and ST1 from the register stack, compute ST1-ST0, 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 ST0-ST1 and push that difference.

With two register operands (*source, destination*) the FSUB instruction computes *destination := destination - source*. One of the two registers must be ST0. With two registers as operands, the FSUBP also com-

putes *destination := destination - source* and then it pops ST0 off the stack after computing the difference. For the FSUBP instruction, the source operand must be ST0.

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 ST0 (FSUB) or subtract ST0 from this value (FSUBR) and store the result back into ST0.

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.

Note: the instructions that have real constants as operands aren't true FPU instructions. These are extensions provided by HLA. HLA generates a constant segment memory object initialized with the constant's value.

11.2.6.3 The FMUL and FMULP Instructions

The FMUL and FMULP instructions multiply two floating point values. These instructions allow the following forms:

```
fmul()
fmulp()

fmul( sti, st0 );
fmul( st0, sti );
fmul( mem_32_64 );
fmul( real_constant );

fmulp( st0, sti );
```

With no operands, FMUL and FMULP both do the same thing – they pop ST0 and ST1, 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 ST0.

The FMULP(ST0, ST*i*) instruction computes ST*i* := ST*i* * ST0 and then pops ST0. This instruction uses the value for *i* before popping ST0. 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 ST0 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.

Note: the instruction that has a real constant as its operand isn't a true FPU instruction. It is an extension provided by HLA (see the note at the end of the previous section for details).

11.2.6.4 The FDIV, FDIVP, FDIVR, and FDIVRP Instructions

These four instructions allow the following forms:

```
fdiv()
fdivp()
fdivr()
fdivrp()

fdiv( sti, st0 );
fdiv( st0, sti );
fdivp( st0, sti );
```



```
fdivr( sti, st0 );
fdivr( st0, sti );
fdivrp( st0, sti );

fdiv( mem_32_64 );
fdivr( mem_32_64 );
fdiv( real_constant );
fdivr( real_constant );
```

With no operands, the FDIV and FDIVP instructions pop ST0 and ST1, compute ST1/ST0, and push the result back onto the stack. The FDIVR and FDIVRP instructions also pop ST0 and ST1 but compute ST0/ST1 before pushing the quotient onto the stack.

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

```
fdiv( sti, st0 );     // ST0 := ST0/STi
fdiv( st0, sti );     // STi := STi/ST0
fdivp( st0, sti );    // STi := STi/ST0  then pop ST0
fdivr( st0, sti );    // ST0 := ST0/STi
fdivrp( st0, sti );   // STi := ST0/STi then pop ST0
```

The FDIVP and FDIVRP instructions also pop ST0 after performing the division operation. The value for *i* in these two instructions is computed before popping ST0.

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.

Note: the instructions that have real constants as operands aren't true FPU instructions. These are extensions provided by HLA.

11.2.6.5 The FSQRT Instruction

The FSQRT routine does not allow any operands. It computes the square root of the value on top of stack (TOS) and replaces ST0 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( st0 );   // Duplicate X on TOS.
      fmul();       // Compute X**2.

      fld( y );     // Load Y
      fld( st0 );   // Duplicate Y.
      fmul();       // Compute Y**2.

      fadd();       // Compute X**2 + Y**2.
      fsqrt();      // Compute sqrt( X**2 + Y**2 ).
      fstp( z );    // Store result away into Z.
```


11.2.6.6 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 ST0/ST1. If the difference between the exponents of ST0 and ST1 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 ST0 and the original divisor in ST1 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 );
        repeat

            fprem1();
            fstsw( ax );     // Get condition code bits into AX.
            and( 1, ah );    // See if C2 is set.

        until( @z );        // Repeat until C2 is clear.
        fstp( z );          // Store away the remainder.
        fstp( st0 );        // Pop old Y value.
```

11.2.6.7 The FRNDINT Instruction

The FRNDINT instruction rounds the value on the top of stack (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. Note that the result on tos is still a floating point value, it simply does not have a fractional component.

11.2.6.8 The FABS Instruction

FABS computes the absolute value of ST0 by clearing the mantissa sign bit of ST0. 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 );
```

11.2.6.9 The FCHS Instruction

FCHS changes the sign of ST0's value by inverting the mantissa sign bit (that is, this is the floating point negation instruction). 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 );
```

11.2.7 Comparison Instructions

The FPU provides several instructions for comparing real values. The FCOM, FCOMP, and FCOMPP 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.

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 615) 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 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 FPU status bits into the sign or overflow flags, you cannot use signed comparison instructions. Instead, use unsigned operations (e.g., SETA, SETB) when testing the results of a floating point comparison. *Yes, these instructions normally test unsigned values and floating point numbers are signed values*. However, use the unsigned operations anyway; the FSTSW and SAHF instructions set the 80x86 flags register as though you had compared unsigned values with the CMP instruction.

The Pentium II and (upwards) compatible processors provide an extra set of floating point comparison instructions that directly affect the 80x86 condition code flags. These instructions circumvent having to use FSTSW and SAHF to copy the FPU status into the 80x86 condition codes. These instructions include FCOMI and FCOMIP. You use them just like the FCOM and FCOMP instructions except, of course, you do not have to manually copy the status bits to the FLAGS register. Do be aware that these instructions are not available on many processors in common use today (as of 1/1/2000). However, as time passes it may be safe to begin assuming that everyone's CPU supports these instructions. Since this text assumes a minimum Pentium CPU, it will not discuss these two instructions any further.

11.2.7.1 The FCOM, FCOMP, and FCOMPP Instructions

The FCOM, FCOMP, and FCOMPP instructions compare ST0 to the specified operand and set the corresponding FPU condition code bits based on the result of the comparison. The legal forms for these instructions are


```
fcom()
fcomp()
fcompp()

fcom( sti )
fcomp( sti )

fcom( mem_32_64 )
fcomp( mem_32_64 )
fcom( real_constant )
fcomp( real_constant )
```

With no operands, FCOM, FCOMP, and FCOMPP compare ST0 against ST1 and set the processor flags accordingly. In addition, FCOMP pops ST0 off the stack and FCOMPP pops both ST0 and ST1 off the stack.

With a single register operand, FCOM and FCOMP compare ST0 against the specified register. FCOMP also pops ST0 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 ST0 against this value, setting the condition code bits accordingly. FCOMP also pops ST0 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.

Note: the instructions that have real constants as operands aren't true FPU instructions. These are extensions provided by HLA. When HLA encounters such an instruction, it creates a real64 read-only variable in the constants segment and initializes this variable with the specified constant. Then HLA translates the instruction to one that specifies a real64 memory operand. *Note that because of the precision differences (64 bits vs. 80 bits), if you use a constant operand in a floating point instruction you may not get results that are as precise as you would expect.*

Example of a floating point comparison:

```
fcompp();
fstsw( ax );
sahf();
setb( al );   // AL = true if ST1 < ST0.
    .
    .
    .
```

Note that you cannot compare floating point values in an HLA run-time boolean expression (e.g., within an IF statement).

11.2.7.2 The FTST Instruction

The FTST instruction compares the value in ST0 against 0.0. It behaves just like the FCOM instruction would if ST1 contained 0.0. Note that this instruction does not differentiate -0.0 from +0.0. If the value in ST0 is either of these values, ftst will set C_3 to denote equality. Note that this instruction does *not* pop st(0) off the stack. Example:

```
ftst();
fstsw( ax );
sahf();
sete( al );       // Set AL to 1 if TOS = 0.0
```


11.2.8 Constant Instructions

The FPU provides several instructions that let you load commonly used constants onto the FPU's register stack. These instructions set the stack fault, invalid operation, 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).
```

11.2.9 Transcendental Instructions

The FPU provides eight transcendental (log and trigonometric) instructions to compute sin, cos, 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.

11.2.9.1 The F2XM1 Instruction

F2XM1 computes $2^{st0}-1$. The value in ST0 must be in the range $-1.0 \le ST0 \le +1.0$. If ST0 is out of range F2XM1 generates an undefined result but raises no exceptions. The computed value replaces the value in ST0. 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.

11.2.9.2 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(ST0) in ST0 and sin(ST0) in ST1.

These instructions assume ST0 specifies an angle in radians and this angle must be in the range $-2^{63} < ST0 < +2^{63}$. If the original operand is out of range, these instructions set the C_2 flag and leave ST0 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.

11.2.9.3 The FPTAN Instruction

FPTAN computes the tangent of ST0 and pushes this value and then it pushes 1.0 onto the stack. Like the FSIN and FCOS instructions, the value of ST0 is assumed to be in radians and must be in the range $-2^{63}<ST0<+2^{63}$. 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.

11.2.9.4 The FPATAN Instruction

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

$ST0 = \tan^{-1}(ST1 / ST0)$

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.

11.2.9.5 The FYL2X Instruction

This instruction expects two operands on the FPU stack: y is found in ST1 and x is found in ST0. This function computes:

$ST0 = ST1 * \log_2(ST0)$

This instruction has no operands (to the instruction itself). The instruction uses the following syntax:

```
fyl2x();
```

Note that this instruction computes the base two logarithm. Of course, it is a trivial matter to compute the log of any other base by multiplying by the appropriate constant.

11.2.9.6 The FYL2XP1 Instruction

This instruction expects two operands on the FPU stack: y is found in ST1 and x is found in ST0. This function computes:

$ST0 = ST1 * \log_2(ST0 + 1.0)$

The syntax for this instruction is

```
fyl2xp1();
```

Otherwise, the instruction is identical to FYL2X.

11.2.10Miscellaneous instructions

The 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, FLDCW, FSTCW, FCLEX/FNCLEX, and FSTSW.

11.2.10.1 The FINIT and FNINIT Instructions

The FINIT instruction initializes the FPU for proper operation. Your applications should execute this instruction before executing any other FPU instructions. This instruction initializes the control register to 37Fh (see "The FPU Control Register" on page 612), the status register to zero (see "The FPU Status Register" on page 615) and the tag word to 0FFFFh. The other registers are unaffected. Examples:

```
FINIT();
FNINIT();
```

The difference between FINIT and FNINIT is that FINIT first checks for any pending floating point exceptions before initializing the FPU; FNINIT does not.

11.2.10.2 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 612) 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.

11.2.10.3 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 615). Examples:

```
fclex();
fnclex();
```

The difference between these instructions is the same as FINIT and FNINIT.

11.2.10.4 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 615) 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 (specifically, AX). 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. The difference between FSTSW and FNSTSW is the same as for FCLEX and FNCLEX.

11.2.11Integer 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_16_32 )
fisub( int_16_32 )
fisubr( int_16_32 )
fimul( int_16_32 )
fidiv( int_16_32 )
fidivr( int_16_32 )

ficom( int_16_32 )
ficomp( int_16_32 )
```

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 ST0 as the destination operand.

11.3 Converting Floating Point Expressions to Assembly Language

Because the FPU register organization is different than the 80x86 integer register set, translating arithmetic expressions involving floating point operands is a little different than the techniques for translating integer expressions. Therefore, it makes sense to spend some time discussing how to manually translate floating point expressions into assembly language.

In one respect, it's actually easier to translate floating point expressions into assembly language. The stack architecture of the Intel FPU eases the translation of arithmetic expressions into assembly language. If you've ever used a Hewlett-Packard calculator, you'll be right at home on the FPU because, like the HP calculator, the FPU uses *reverse polish notation*, or *RPN*, for arithmetic calculations. Once you get used to using RPN, it's actually a bit more convenient for translating expressions because you don't have to worry about allocating temporary variables - they always wind up on the FPU stack.

RPN, as opposed to standard *infix notation*, places the operands before the operator. The following examples give some simple examples of infix notation and the corresponding RPN notation:

```
infix notation        RPN notation
   5 + 6                 5  6  +
   7 - 2                 7  2  -
   x * y                 x  y  *
   a / b                 a  b  /
```

An RPN expression like "5 6 +" says "push five onto the stack, push six onto the stack, then pop the value off the top of stack (six) and add it to the new top of stack." Sound familiar? This is exactly what the FLD and FADD instructions do. In fact, you can calculate this using the following code:

```
fld( 5.0 );
fld( 6.0 );
fadd();       // 11.0 is now on the top of the FPU stack.
```

As you can see, RPN is a convenient notation because it's very easy to translate this code into FPU instructions.

One advantage to RPN (or *postfix notation*) is that it doesn't require any parentheses. The following examples demonstrate some slightly more complex infix to postfix conversions:

```
infix notation              postfix notation
(x + y) * 2                 x  y + 2 *
x * 2 - (a + b)             x 2 * a b + -
(a + b) * (c + d)           a b + c d + *
```

The postfix expression "x y + 2 *" says "push x, then push y; next, add those values on the stack (producing X+Y on the stack). Next, push 2 and then multiply the two values (two and X+Y) on the stack to produce two times the quantity X+Y." Once again, we can translate these postfix expressions directly into assembly language. The following code demonstrates the conversion for each of the above expressions:

```
//      x y + 2 *

        fld( x );
        fld( y );
        fadd();
        fld( 2.0 );
        fmul();

//      x 2 * a b + -

        fld( x );
        fld( 2.0 );
        fmul();
        fld( a );
        fld( b );
        fadd();
        fsub();

//      a b + c d + *

        fld( a );
        fld( b );
        fadd();
        fld( c );
        fld( d );
        fadd();
        fmul();
```

11.3.1 Converting Arithmetic Expressions to Postfix Notation

Since the process of translating arithmetic expressions into assembly language involves postfix (RPN) notation, converting arithmetic expressions into postfix notation seems like the right place to start. This section will concentrate on that conversion.

For simple expressions, those involving two operands and a single expression, the translation is trivial. Simply move the operator from the infix position to the postfix position (that is, move the operator from inbetween the operands to after the second operand). For example, "5 + 6" becomes "5 6 +". Other than separating your operands so you don't confuse them (i.e., is it "5" and "6" or "56"?) there isn't much to converting simple infix expressions into postfix notation.

For complex expressions, the idea is to convert the simple sub-expressions into postfix notation and then treat each converted subexpression as a single operand in the remaining expression. The following discussion will surround completed conversions in square brackets so it is easy to see which text needs to be treated as a single operand in the conversion.

As for integer expression conversion, the best place to start is in the inner-most parenthetical sub-expression and then work your way outward considering precedence, associativity, and other parenthetical sub-expressions. As a concrete working example, consider the following expression:

```
x = ((y-z)*a) - ( a + b * c )/3.14159
```

A possible first translation is to convert the subexpression "(y-z)" into postfix notation. This is accomplished as follows:

```
x = ([y z -] * a) - ( a + b * c )/3.14159
```

Square brackets surround the converted postfix code just to separate it from the infix code. These exist only to make the partial translations more readable. Remember, for the purposes of conversion we will treat the text inside the square brackets as a single operand. Therefore, you would treat "[y z -]" as though it were a single variable name or constant.

The next step is to translate the subexpression "([y z -] * a)" into postfix form. This yields the following:

```
x = [y z - a *] - ( a + b * c )/3.14159
```

Next, we work on the parenthetical expression "(a + b * c)." Since multiplication has higher precedence than addition, we convert "b*c" first:

```
x = [y z - a *] - ( a + [b c *])/3.14159
```

After converting "b*c" we finish the parenthetical expression:

```
x = [y z - a *] - [a b c * +]/3.14159
```

This leaves only two infix operators: subtraction and division. Since division has the higher precedence, we'll convert that first:

```
x = [y z - a *] - [a b c * + 3.14159 /]
```

Finally, we convert the entire expression into postfix notation by dealing with the last infix operation, subtraction:

```
x = [y z - a *] [a b c * + 3.14159 /] -
```

Removing the square brackets to give us true postfix notation yields the following RPN expression:

```
x = y z - a * a b c * + 3.14159 / -
```

Here is another example of an infix to postfix conversion:

```
a = (x * y - z + t)/2.0
```

Step 1: Work inside the parentheses. Since multiplication has the highest precedence, convert that first:

```
a = ( [x y *] - z + t)/2.0
```

Step 2: Still working inside the parentheses, we note that addition and subtraction have the same precedence, so we rely upon associativity to determine what to do next. These operators are left associative, so we must translate the expressions in a left to right order. This means translate the subtraction operator first:

```
a = ( [x y * z -] + t)/2.0
```

Step 3: Now translate the addition operator inside the parentheses. Since this finishes the parenthetical operators, we can drop the parentheses:


```
a = [x y * z - t +]/2.0
```

Step 4: Translate the final infix operator (division). This yields the following:

```
a = [x y * z - t + 2.0 / ]
```

Step 5: Drop the square brackets and we're done:

```
a = x y * z - t + 2.0 /
```

11.3.2 Converting Postfix Notation to Assembly Language

Once you've translated an arithmetic expression into postfix notation, finishing the conversion to assembly language is especially easy. All you have to do is issue an FLD instruction whenever you encounter an operand and issue an appropriate arithmetic instruction when you encounter an operator. This section will use the completed examples from the previous section to demonstrate how little there is to this process.

```
x = y z - a * a b c * + 3.14159 / -
```

- Step 1: Convert *y* to FLD(y);
- Step 2: Convert *z* to FLD(z);
- Step 3: Convert "-" to FSUB();
- Step 4: Convert *a* to FLD(a);
- Step 5: Convert "*" to FMUL();
- Steps 6-n: Continuing in a left-to-right fashion, generate the following code for the expression:

```
fld( y );
fld( z );
fsub();
fld( a );
fmul();
fld( a );
fld( b );
fld( c );
fmul();
fadd();
fldpi();      // Loads pi (3.14159)
fdiv();
fsub();

fstp( x );    // Store result away into x.
```

Here's the translation for the second example in the previous section:

```
a = x y * z - t + 2.0 /
```

```
fld( x );
fld( y );
fmul();
fld( z );
fsub();
fld( t );
fadd();
fld( 2.0 );
fdiv();

fstp( a );    // Store result away into a.
```


As you can see, the translation is fairly trivial once you've converted the infix notation to postfix notation. Also note that, unlike integer expression conversion, you don't need any explicit temporaries. It turns out that the FPU stack provides the temporaries for you[3]. For these reasons, conversion of floating point expressions into assembly language is actually easier than converting integer expressions.

11.3.3 Mixed Integer and Floating Point Arithmetic

Throughout the previous sections on floating point arithmetic an unstated assumption was made: all operands in the expressions were floating point variables or constants. In the real world, you'll often need to mix integer and floating point operands in the same expression. Thanks to the FILD instruction, this is a trivial exercise.

Of course, the FPU cannot operate on integer operands. That is, you cannot push an integer operand (in integer format) onto the FPU stack and add this integer to a floating point value that is also on the stack. Instead, you use the FILD instruction to load and translate the integer value; this winds up pushing the floating point equivalent of the integer onto the FPU stack. Once the value is converted to a floating point number, you continue the calculation using the standard real arithmetic operations.

Embedding a floating point value in an integer expression is a little more problematic. In this case you must convert the floating point value to an integer value for use in the integer expression. To do this, you must use the FIST instruction. FIST converts the floating point value on the top of stack into an integer value according to the setting of the rounding bits in the floating point control register (See "The FPU Control Register" on page 612). By default, FIST will round the floating point value to the nearest integer before storing the value into memory; if you want to use the more common fraction truncation mode, you will need to change the value in the FPU control register. You compute the integer expression using the techniques from the previous chapter (see "Complex Expressions" on page 600). The FPU participates only to the point of converting the floating point value to an integer.

```
static
    intVal1 : uns32 := 1;
    intVal2 : uns32 := 2;
    realVal : real64;
        .
        .
        .
        fild( intVal1 );
        fild( intVal2 );
        fadd();
        fstp( realVal );
        stdout.put( "realVal = ", realVal, nl );
```

11.4 HLA Standard Library Support for Floating Point Arithmetic

The HLA Standard Library provides several routines that support the use of real number on the FPU. In Volume One you saw, with one exception, how the standard input and output routines operate. This section will not repeat that discussion, see "HLA Support for Floating Point Values" on page 93 for more details. One input function that Volume One only mentioned briefly was the *stdin.getf* function. This section will elaborate on that function. The HLA Standard Library also includes the "math.hhf" module that provides several mathematical functions that the FPU doesn't directly support. This section will discuss those functions, as well.

3. Assuming, of course, that your calculations aren't so complex that you exceed the eight-element limitation of the FPU stack.

11.4.1 The stdin.getf and fileio.getf Functions

The *stdin.getf* function reads a floating point value from the standard input device. It leaves the converted value in ST0 (i.e., on the top of the floating point stack). The only reason Chapter Two did not discuss this function thoroughly was because you hadn't seen the FPU and FPU registers at that point.

The *stdin.getf* function accepts the same inputs that "stdin.get(fp_variable);" would except. The only difference between the two is where these functions store the floating point value.

As you'd probably surmise, there is a corresponding *fileio.getf* function as well. This function reads the floating point value from the file whose file handle is the single parameter in this function call. It, too, leaves the converted result on the top of the FPU stack.

11.4.2 Trigonometric Functions in the HLA Math Library

The FPU provides a small handful of trigonometric functions. It does not, however, support the full range of trig functions. The HLA MATH.HHF module fills in most of the missing functions. The trigonometric functions that HLA provides include

- ACOS(arc cosine)
- ACOT (arc cotangent)
- ACSC(arc cosecant)
- ASEC (arc secant)
- ASIN (arc sin)
- COT (cotangent)
- CSC (cosecant)
- SEC (secant)

The HLA Standard Library actually provides five different routines you can call for each of these functions. For example, the prototypes for the first four COT (cotangent) routines are:

```
procedure cot32( r32: real32 );
procedure cot64( r64: real64 );
procedure cot80( r80: real80 );
procedure _cot();
```

The first three routines push their parameter onto the FPU stack and compute the cotangent of the result. The fourth routine above (*_cot*) computes the cotangent of the value in ST0.

The fifth routine is actually an overloaded procedure that calls one of the four routines above depending on the parameter. This call uses the following syntax:

```
cot();        // Calls _cot() to compute cot(ST0).
cot( r32 );   // Calls cot32 to compute the cotangent of r32.
cot( r64 );   // Calls cot64 to compute the cotangent of r64.
cot( r80 );   // Calls cot80 to compute the cotangent of r80.
```

Using this fifth form is probably preferable since it is much more convenient. Note that there is no efficiency loss when you used *cot* rather than one of the other cotangent routines. HLA actually translates this statement directly into one of the other calls.

The HLA trigonometric functions that require an angle as a parameter expect that angle to be expressed in radians, not degrees. Keep in mind that some of these functions produce undefined results for certain input values. If you've enabled exceptions on the FPU, these functions will raise the appropriate FPU exception if an error occurs.

11.4.3 Exponential and Logarithmic Functions in the HLA Math Library

The HLA MATH.HHF module provides several exponential and logarithmic functions in addition to the trigonometric functions. Like the trig functions, the exponential and logarithmic functions provide five different interfaces to each function depending on the size and location of the parameter. The functions that MATH.HHF supports are

- TwoToX (raise 2.0 to the specified power).
- TenToX (raise 10.0 to the specified power).
- exp (raises e [2.718281828...] to the specified power).
- YtoX (raises first parameter to the power specified by the second parameter).
- log (computes base 10 logarithm).
- ln (computes base e logarithm).

Except for the *YtoX* function, all these functions provide the same sort of interface as the *cot* function mentioned in the previous section. For example, the *exp* function provides the following prototypes:

```
procedure exp32( r32: real32 );
procedure exp64( r64: real64 );
procedure exp80( r80: real80 );
procedure _exp();
```

The exp function, by itself, automatically calls one of the above functions depending on the parameter type (and presence of a parameter):

```
exp();        // Calls _exp() to compute exp(ST0).
exp( r32 );   // Calls exp32 to compute the e**r32.
exp( r64 );   // Calls exp64 to compute the e**r64.
exp( r80 );   // Calls exp80 to compute the e**r80.
```

The lone exception to the above is the *YtoX* function. *YtoX* has its own rules because it has two parameters rather than one (Y and X). *YtoX* provides the following function prototypes:

```
procedure YtoX32( y: real32; x: real32 );
procedure YtoX64( y: real64; x: real64 );
procedure YtoX80( y: real80; x: real80 );
procedure _YtoX();
```

The *_YtoX* function computes ST1**ST0 (i.e., ST1 raised to the ST0 power).

The YtoX function provides the following interface:

```
YtoX();              // Calls _YtoX() to compute exp(ST0).
YtoX( y32, x32);     // Calls YtoX32 to compute y32**x32.
YtoX( y64, x64 );    // Calls YtoX64 to compute y64**x64.
YtoX( y80, x80 );    // Calls YtoX80 to compute y80**x80.
```

11.5 Sample Program

This chapter presents a simple “Reverse Polish Notation” calculator that demonstrates the use of the 80x86 FPU. In addition to demonstrating the use of the FPU, this sample program also introduces a few new routines from the HLA Standard Library, specifically the *arg.c*, *arg.v*, and *conv.strToFlt* routines.

The HLA Standard Library conversions module (“conv.hhf”) contains dozens of procedures that translate data between various formats. A large percentage of these routines convert data between some internal numeric form and a corresponding string format. The *conv.strToFlt* routine, as its name suggests, converts string data to a floating point value. The prototype for this function is the following:


```
procedure conv.strToFlt( s:string; index:dword );
```

The first parameter is the string containing the textual representation of the floating point value. The second parameter contains an index into the string where the floating point text actually begins (usually the *index* parameter is zero if the string contains nothing but the floating point text). The *conv.strToFlt* procedure will attempt to convert the specified string to a floating point number. If there is a problem, this function will raise an appropriate exception (e.g., *ex.ConversionError*). In fact, the HLA *stdin* routines that read floating point values from the user actually read string data and call this same procedure to convert that data to a floating point value; hence, you should protect this procedure call with a TRY..ENDTRY statement exactly the same way you protect a call to *stdin.get* or *stdin.getf*. If this routine is successful, it leaves the converted floating point value on the top of the FPU stack.

The HLA Standard Library contains numerous other procedures that convert textual data to the corresponding internal format. Examples include *conv.strToi8, conv.strToi16, conv.strToi32, conv.strToi64,* and more. See the HLA Standard Library documentation for more details.

This sample program also uses the *arg.c* and *arg.v* routines from the HLA Standard Library's command line arguments module ("args.hhf"). These functions provide access to the text following the program name when you run a program from the command line prompt. This calculator program, for example, expects the user to supply the desired calculation on the command line immediately after the program name. For example to add the two values 18 and 22 together, you'd specify the following command line:

```
rpncalc 18  22  +
```

The text "18 22 +" is an example of three *command line parameters*. Programs often use command line parameters to communicate filenames and other data to the application. For example, the HLA compiler uses command line parameters to pass the names of the source files to the compiler. The rpncalc program uses the command line to pass the RPN expression to the calculator.

The *arg.c* function ("argument count") returns the number of parameters on the command line. This function returns the count in the EAX register. It does not have any parameters. In general, you can probably assume that the maximum possible number of command line arguments is between 64 and 128. Note that the operating system counts the program's name on the command line in this argument count. Therefore, this value will always be one or greater. If *arg.c* returns one, then there are no extra command line parameters; the single item is the program's name.

A program that expects at least one command line parameter should always call *arg.c* and verify that it returns the value two or greater. Programs that process command line parameters typically execute a loop of some sort that executes the number of times specified by *arg.c*'s return value. Of course, when you use *arg.c*'s return value for this purpose, don't forget to subtract one from the return result to account for the program's name (unless you are treating the program name as one more parameter).

The *arg.v* function returns a string containing one of the program's command line arguments. This function has the following prototype:

```
procedure arg.v( index:uns32 );
```

The *index* parameter specifies which command line parameter you wish to retrieve. The value zero returns a string containing the program's name. The value one returns the first command line parameter following the program's name. The value two returns a string containing the second command line parameter following the program's name. Etc. The value you provide as a parameter to this function must fall in the range 0..*arg.c()*-1 or *arg.v* will raise an exception.

The *arg.v* procedure allocates storage for the string it returns on the heap by calling stralloc. It returns a pointer to the allocated string in the EAX register. Don't forget to call *strfree* to return the storage to the system after you are done processing the command line parameter.

Well, without further ado, here is the RPN calculator program that uses the aforementioned functions.

```
// This sample program demonstrates how to use
```



```
// the FPU to create a simple RPN calculator.
// This program reads a string from the user
// and “parses” that string to figure out the
// calculation the user is requesting.  This
// program assumes that any item beginning
// with a numeric digit is a numeric operand
// to push onto the FPU stack and all other
// items are operators.
//
//  Example of typical user input:
//
//      calc 123.45 67.89 +
//
//  The program responds by printing
//
//      Result = 1.91340000000000000e+2
//
//
//  Current operators supported:
//
//      + - * /
//
//  Current functions supported:
//
//      sin sqrt


program RPNcalculator;
#include( “stdlib.hhf” )


static
    argc:       uns32;
    curOperand: string;
    ItemsOnStk: uns32;
    realRslt:   real80;


    // The following function converts an
    // angle (in ST0) from degrees to radians.
    // It leaves the result in ST0.

    procedure DegreesToRadians; @nodisplay;
    begin DegreesToRadians;

        fld( 2.0 ); // Radians = degrees*2*pi/360.0
        fmul();
        fldpi();
        fmul();
        fld( 360.0 );
        fdiv();

    end DegreesToRadians;




begin RPNcalculator;

    // Initialize the FPU.
```



```
finit();

// Okay, extract the items from the Windows
// CMD.EXE command line and process them.

arg.c();
if( eax <= 1 ) then

    stdout.put( "Usage: 'rpnCalc <rpn expression>'" nl );
    exit RPNcalculator;

endif;

// ECX holds the index of the current operand.
// ItemsOnStk keeps track of the number of numeric operands
//  pushed onto the FPU stack so we can ensure that each
//  operation has the appropriate number of operands.

mov( eax, argc );
mov( 1, ecx );
mov( 0, ItemsOnStk );

// The following loop repeats once for each item on the
// command line:

while( ecx < argc ) do

    // Get the string associated with the current item:

    arg.v( ecx );   // Note that this malloc's storage!
    mov( eax, curOperand );


    // If the operand begins with a numeric digit, assume
    // that it's a floating point number.

    if( (type char [eax]) in '0'..'9' ) then

        try

            // Convert this string representation of a numeric
            // value to the equivalent real value.  Leave the
            // result on the top of the FPU stack.  Also, bump
            // ItemsOnStk up by one since we're pushing a new
            // item onto the FPU stack.

            conv.strToFlt( curOperand, 0 );
            inc( ItemsOnStk );

          exception( ex.ConversionError )

            stdout.put("Illegal floating point constant" nl );
            exit RPNcalculator;

          anyexception

            stdout.put
            (
                "Exception ",
                (type uns32 eax ),
```



```
                ” while converting real constant”
                nl
            );
            exit RPNcalculator;

        endtry;



    // Handle the addition operation here.

    elseif( str.eq( curOperand, “+” )) then

        // The addition operation requires two
        // operands on the stack.  Ensure we have
        // two operands before proceeding.

        if( ItemsOnStk >= 2 ) then

            fadd();
            dec( ItemsOnStk );  // fadd() removes one operand.

        else

            stdout.put( “‘+’ operation requires two operands.” nl );
            exit RPNcalculator;

        endif;


    // Handle the subtraction operation here.  See the comments
    // for FADD for more details.

    elseif( str.eq( curOperand, “-” )) then

        if( ItemsOnStk >= 2 ) then

            fsub();
            dec( ItemsOnStk );

        else

            stdout.put( “‘-’ operation requires two operands.” nl );
            exit RPNcalculator;

        endif;


    // Handle the multiplication operation here.  See the comments
    // for FADD for more details.

    elseif( str.eq( curOperand, “*” )) then

        if( ItemsOnStk >= 2 ) then

            fmul();
            dec( ItemsOnStk );

        else

            stdout.put( “‘*’ operation requires two operands.” nl );
```



```
            exit RPNcalculator;

        endif;


    // Handle the division operation here.  See the comments
    // for FADD for more details.

    elseif( str.eq( curOperand, "/" )) then

        if( ItemsOnStk >= 2 ) then

            fdiv();
            dec( ItemsOnStk );

        else

            stdout.put( "'/' operation requires two operands." nl );
            exit RPNcalculator;

        endif;



    // Provide a square root operation here.

    elseif( str.eq( curOperand, "sqrt" )) then

        // Sqrt is a monadic (unary) function.  Therefore
        // we only require a single item on the stack.

        if( ItemsOnStk >= 1 ) then

            fsqrt();

        else

            stdout.put
            (
                "SQRT function requires at least one operand."
                nl
            );
            exit RPNcalculator;

        endif;


    // Provide the SINE function here.  See SQRT comments for details.

    elseif( str.eq( curOperand, "sin" )) then

        if( ItemsOnStk >= 1 ) then

            DegreesToRadians();
            fsin();

        else

            stdout.put( "SIN function requires at least one operand." nl );
            exit RPNcalculator;
```



```
                endif;

            else

                stdout.put( "'", curOperand, "' is an unknown operation." nl );
                exit RPNcalculator;

            endif;

            // Free the storage associated with the current item.

            strfree( curOperand );

            // Move on to the next item on the command line:

            inc( ecx );

        endwhile;
        if( ItemsOnStk = 1 ) then

            fstp( realRslt );
            stdout.put( "Result = ", realRslt, nl );

        else

            stdout.put( "Syntax error in expression. ItemsOnStk=", ItemsOnStk, nl );

        endif;


end RPNcalculator;
```

Program 11.1 A Floating Point Calculator Program

11.6 Putting It All Together

Between the FPU and the HLA Standard Library, floating point arithmetic is actually quite simple. In this chapter you learned about the floating point instruction set and you learned how to convert arithmetic expressions involving real arithmetic into a sequence of floating point instructions. This chapter also presented several transcendental functions that the HLA Standard Library provides. Armed with the information from this chapter, you should be able to deal with floating point expressions just as easily as integer expressions.

