CHAPTER 6

Using Floating-Point and Binary Coded Decimal Numbers

MASM requires different techniques for handling floating-point (real) numbers and binary coded decimal (BCD) numbers than for handling integers. You have two choices for working with real numbers—a math coprocessor or emulation routines.

Math coprocessors—the 8087, 80287, and 80387 chips—work with the main processor to handle real-number calculations. The 80486 processor performs floating-point operations directly. All information in this chapter pertaining to the 80387 coprocessor applies to the 80486DX processor as well. It does not apply to the 80486SX, which does not provide an on-chip coprocessor.

This chapter begins with a summary of the directives and formats of floating-point data that you need to allocate memory storage and initialize variables before you can work with floating-point numbers.

The chapter then explains how to use a math coprocessor for floating-point operations. It covers:

- The architecture of the registers.
- The operands for the coprocessor instruction formats.
- The coordination of coprocessor and main processor memory access.
- The basic groups of coprocessor instructions—for loading and storing data, doing arithmetic calculations, and controlling program flow.

The next main section describes emulation libraries. The emulation routines provided with all Microsoft high-level languages enable you to use coprocessor instructions as though your computer had a math coprocessor. However, some coprocessor instructions are not handled by emulation, as this section explains.

Finally, because math coprocessor and emulation routines can also operate on BCD numbers, this chapter includes the instruction set for these numbers.

Using Floating-Point Numbers

Before using floating-point data in your program, you need to allocate the memory storage for the data. You can then initialize variables either as real numbers in decimal form or as encoded hexadecimals. The assembler stores allocated data in 10-byte IEEE format. This section covers floating-point declarations and floating-point data formats.

Declaring Floating-Point Variables and Constants

You can allocate real constants using the **REAL4**, **REAL8**, and **REAL10** directives. These directives allocate the following floating-point numbers:

Directive	Size
REAL4	Short (32-bit) real numbers
REAL8	Long (64-bit) real numbers
REAL10	10-byte (80-bit) real numbers and BCD numbers

Table 6.1 lists the possible ranges for floating-point variables. The number of significant digits can vary in an arithmetic operation as the least-significant digit may be lost through rounding errors. This occurs regularly for short and long real numbers, so you should assume the lesser value of significant digits shown in Table 6.1. Ten-byte real numbers are much less susceptible to rounding errors for reasons described in the next section. However, under certain circumstances, 10-byte real operations can have a precision of only 18 digits.

Table 6.1 Ranges of Floating-Point Variables

Data Type	Bits	Significant Digits	Approximate Range
Short real	32	6–7	1.18×10^{-38} to 3.40×10^{38}
Long real	64	15–16	2.23×10^{-308} to 1.79×10^{308}
10-byte real	80	19	3.37×10^{-4932} to 1.18×10^{4932}

With versions of MASM prior to 6.0, the **DD**, **DQ**, and **DT** directives could allocate real constants. MASM 6.1 still supports these directives, but the variables are integers rather than floating-point values. Although this makes no difference in the assembly code, CodeView displays the values incorrectly.

You can specify floating-point constants either as decimal constants or as encoded hexadecimal constants. You can express decimal real-number constants in the form:

[[+ | –]] *integer*[[*fraction*]][[**E**[[+ | –]]*exponent*]]

For example, the numbers `2.523E1` and `-3.6E-2` are written in the correct decimal format. You can use these numbers as initializers for real-number variables.

The assembler always evaluates digits of real numbers as base 10. It converts real-number constants given in decimal format to a binary format. The sign, exponent, and decimal part of the real number are encoded as bit fields within the number.

You can also specify the encoded format directly with hexadecimal digits (0–9 plus A–F). The number must begin with a decimal digit (0–9) and end with the real-number designator (R). It cannot be signed. For example, the hexadecimal number `3F800000r` can serve as an initializer for a doubleword-sized variable.

The maximum range of exponent values and the number of digits required in the hexadecimal number depend on the directive. The number of digits for encoded numbers used with **REAL4**, **REAL8**, and **REAL10** must be 8, 16, and 20 digits, respectively. If the number has a leading zero, the number must be 9, 17, or 21 digits.

Examples of decimal constant and hexadecimal specifications are shown here:

```
; Real numbers
short   REAL4    25.23             ; IEEE format
double  REAL8    2.523E1           ; IEEE format
tenbyte REAL10   2523.0E-2         ; 10-byte real format

; Encoded as hexadecimals
ieeeshort       REAL4    3F800000r             ; 1.0 as IEEE short
ieeedouble      REAL8    3FF0000000000000r     ; 1.0 as IEEE long
temporary       REAL10   3FFF8000000000000000r ; 1.0 as 10-byte
                                                ;   real
```

The section “Storing Numbers in Floating-Point Format,” following, explains the IEEE formats—the way the assembler actually stores the data.

Pascal or C programmers may prefer to create language-specific **TYPEDEF** declarations, as illustrated in this example:

```
; C-language specific
float           TYPEDEF REAL4
double          TYPEDEF REAL8
long_double     TYPEDEF REAL10
; Pascal-language specific
SINGLE          TYPEDEF REAL4
DOUBLE          TYPEDEF REAL8
EXTENDED        TYPEDEF REAL10
```

For applications of **TYPEDEF**, see "Defining Pointer Types with TYPEDEF," page 75.

Storing Numbers in Floating-Point Format

The assembler stores floating-point variables in the IEEE format. MASM 6.1 does not support **.MSFLOAT** and Microsoft binary format, which are available in version 5.1 and earlier. Figure 6.1 illustrates the IEEE format for encoding short (4-byte), long (8-byte), and 10-byte real numbers. Although this figure places the most significant bit first for illustration, low bytes actually appear first in memory.

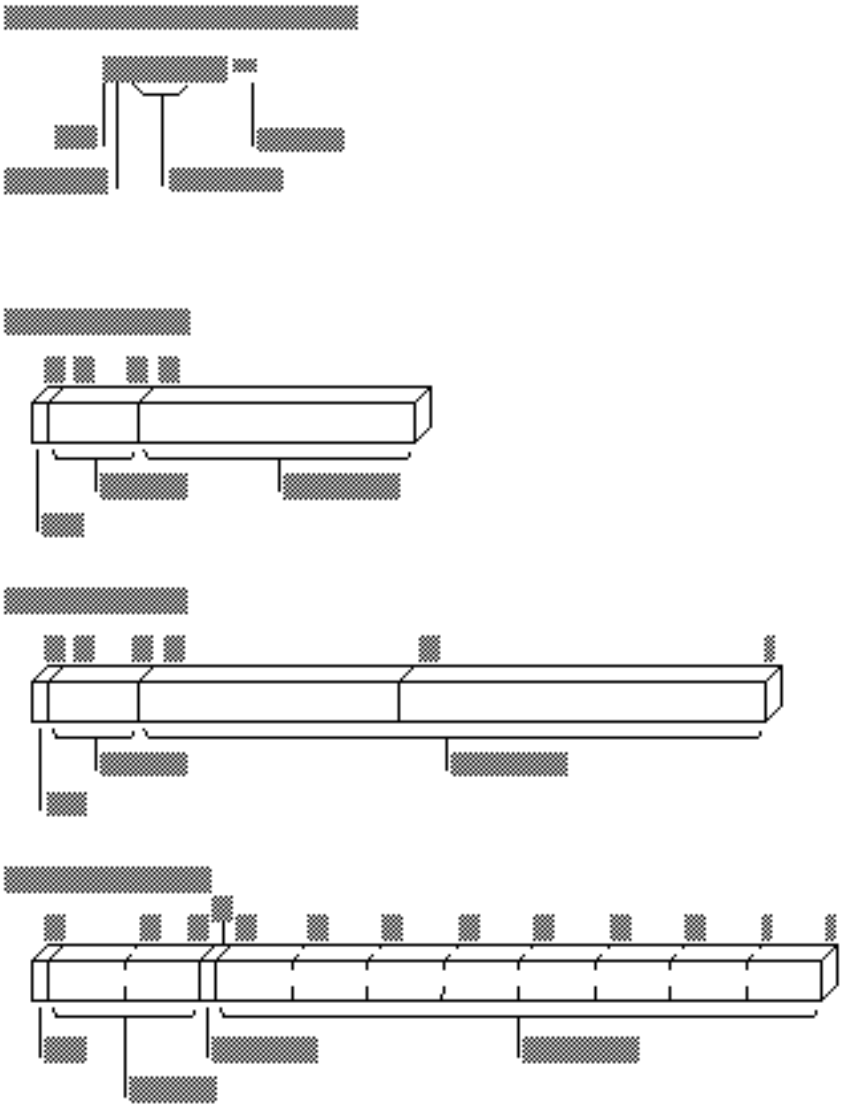

Figure 6.1 Encoding for Real Numbers in IEEE Format

The following list explains how the parts of a real number are stored in the IEEE format. Each item in the list refers to an item in Figure 6.1.

- Sign bit (0 for positive or 1 for negative) in the upper bit of the first byte.
- Exponent in the next bits in sequence (8 bits for a short real number, 11 bits for a long real number, and 15 bits for a 10-byte real number).

- The integer part of the significand in bit 63 for the 10-byte real format. By absorbing carry values, this bit allows 10-byte real operations to preserve precision to 19 digits. The integer part is always 1 in short and long real numbers; consequently, these formats do not provide a bit for the integer, since there is no point in storing it.
- Decimal part of the significand in the remaining bits. The length is 23 bits for short real numbers, 52 bits for long real numbers, and 63 bits for 10-byte real numbers.

The exponent field represents a multiplier 2^n. To accommodate negative exponents (such as 2^{-6}), the value in the exponent field is biased; that is, the actual exponent is determined by subtracting the appropriate bias value from the value in the exponent field. For example, the bias for short real numbers is 127. If the value in the exponent field is 130, the exponent represents a value of $2^{130-127}$, or 2^3. The bias for long real numbers is 1,023. The bias for 10-byte real numbers is 16,383.

Once you have declared floating-point data for your program, you can use coprocessor or emulator instructions to access the data. The next section focuses on the coprocessor architecture, instructions, and operands required for floating-point operations.

Using a Math Coprocessor

When used with real numbers, packed BCD numbers, or long integers, coprocessors (the 8087, 80287, 80387, and 80486) calculate many times faster than the 8086-based processors. The coprocessor handles data with its own registers. The organization of these registers can be one of the four formats for using operands explained in "Instruction and Operand Formats," later in this section.

This section describes how the coprocessor transfers data to and from the coprocessor, coordinates processor and coprocessor operations, and controls program flow.

Coprocessor Architecture

The coprocessor accesses memory as the CPU does, but it has its own data and control registers—eight data registers organized as a stack and seven control registers similar to the 8086 flag registers. The coprocessor's instruction set provides direct access to these registers.

The eight 80-bit data registers of the 8087-based coprocessors are organized as a stack, although they need not be used as a stack. As data items are pushed into the top register, previous data items move into higher-numbered registers, which are lower on the stack. Register 0 is the top of the stack; register 7 is the bottom. The syntax for specifying registers is:

ST [[(*number*)]]

The *number* must be a digit between 0 and 7 or a constant expression that evaluates to a number from 0 to 7. **ST** is another way to refer to **ST(0)**.

All coprocessor data is stored in registers in the 10-byte real format. The registers and the register format are shown in Figure 6.2.

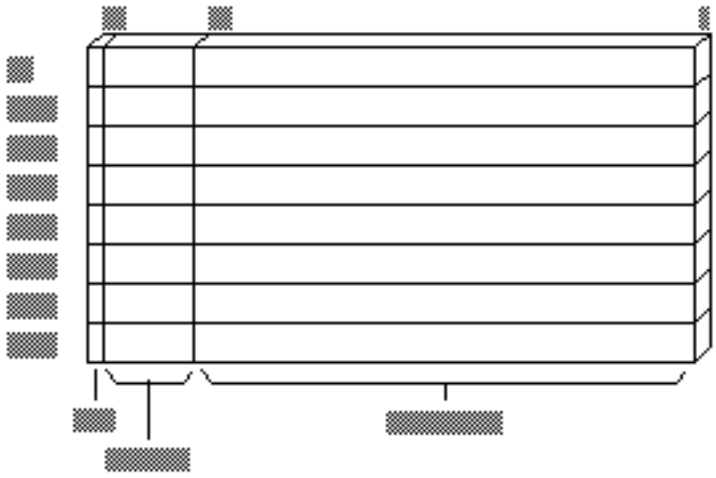

Figure 6.2 Coprocessor Data Registers

Internally, all calculations are done on numbers of the same type. Since 10-byte real numbers have the greatest precision, lower-precision numbers are guaranteed not to lose precision as a result of calculations. The instructions that transfer values between the main memory and the coprocessor automatically convert numbers to and from the 10-byte real format.

Instruction and Operand Formats

Because of the stack organization of registers, you can consider registers either as elements on a stack or as registers much like 8086-family registers. Table 6.2 lists the four main groups of coprocessor instructions and the general syntax for each. The names given to the instruction format reflect the way the instruction uses the coprocessor registers. The instruction operands are placed in the coprocessor data registers before the instruction executes.

Table 6.2 Coprocessor Operand Formats

Instruction Format	Syntax	Implied Operands	Example
Classical stack	**F***instruction*	ST, ST(1)	`fadd`
Memory	**F***instruction memory*	ST	`fadd memloc`
Register	**F***instruction* **ST(***num***), ST**	—	`fadd st(5), st`
	F*instruction* **ST, ST(***num***)**		`fadd st, st(3)`
Register pop	**F***instruction***P ST(***num***), ST**	—	`faddp st(4), st`

You can easily recognize coprocessor instructions because, unlike all 8086-family instruction mnemonics, they start with the letter **F**. Coprocessor instructions can never have immediate operands and, with the exception of the **FSTSW** instruction, they cannot have processor registers as operands.

Classical-Stack Format

Instructions in the classical-stack format treat the coprocessor registers like items on a stack—thus its name. Items are pushed onto or popped off the top elements of the stack. Since only the top item can be accessed on a traditional stack, there is no need to specify operands. The first (top) register (and the second, if the instruction needs two operands) is always assumed.

ST (the top of the stack) is the source operand in coprocessor arithmetic operations. ST(1), the second register, is the destination. The result of the operation replaces the destination operand, and the source is popped off the stack. This leaves the result at the top of the stack.

The following example illustrates the classical-stack format; Figure 6.3 shows the status of the register stack after each instruction.

```
fld1                ; Push 1 into first position
fldpi               ; Push pi into first position
fadd                ; Add pi and 1 and pop
```

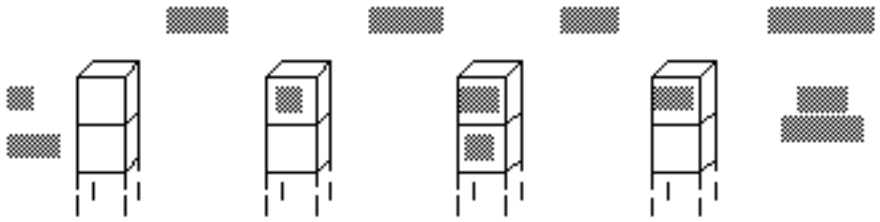

Figure 6.3 Status of the Register Stack

Memory Format

Instructions that use the memory format, such as data transfer instructions, also treat coprocessor registers like items on a stack. However, with this format, items are pushed from memory onto the top element of the stack, or popped from the top element to memory. You must specify the memory operand.

Some instructions that use the memory format specify how a memory operand is to be interpreted—as an integer (**I**) or as a binary coded decimal (**B**). The letter **I** or **B** follows the initial **F** in the syntax. For example, **FILD** interprets its operand as an integer and **FBLD** interprets its operand as a BCD number. If the instruction name does not include a type letter, the instruction works on real numbers.

You can also use memory operands in calculation instructions that operate on two values (see “Using Coprocessor Instructions,” later in this section). The memory operand is always the source. The stack top (ST) is always the implied destination.

The result of the operation replaces the destination without changing its stack position, as shown in this example and in Figure 6.4:

```
        .DATA
m1      REAL4  1.0
m2      REAL4  2.0
        .CODE
        .
        .
        .
        fld    m1      ; Push m1 into first position
        fld    m2      ; Push m2 into first position
        fadd   m1      ; Add m2 to first position
        fstp   m1      ; Pop first position into m1
        fst    m2      ; Copy first position to m2
```

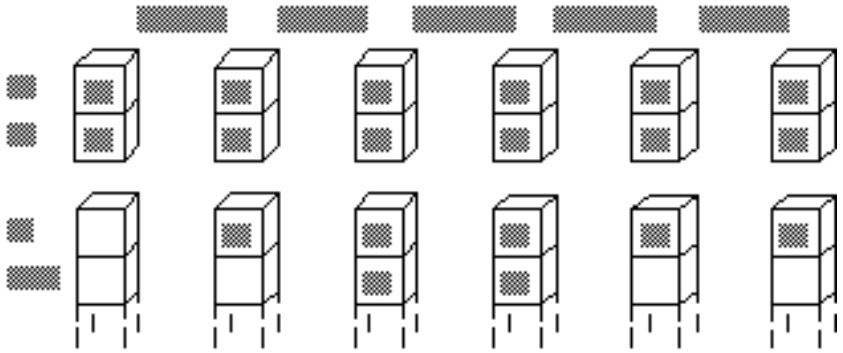

Figure 6.4 Status of the Register Stack and Memory Locations

Register Format

Instructions that use the register format treat coprocessor registers as registers rather than as stack elements. Instructions that use this format require two register operands; one of them must be the stack top (ST).

In the register format, specify all operands by name. The first operand is the destination; its value is replaced with the result of the operation. The second operand is the source; it is not affected by the operation. The stack positions of the operands do not change.

The only instructions that use the register operand format are the **FXCH** instruction and arithmetic instructions for calculations on two values. With the **FXCH** instruction, the stack top is implied and need not be specified, as shown in this example and in Figure 6.5:

```
fadd    st(1), st   ; Add second position to first -
                    ;   result goes in second position
fadd    st, st(2)   ; Add first position to third -
                    ;   result goes in first position
fxch    st(1)       ; Exchange first and second positions
```

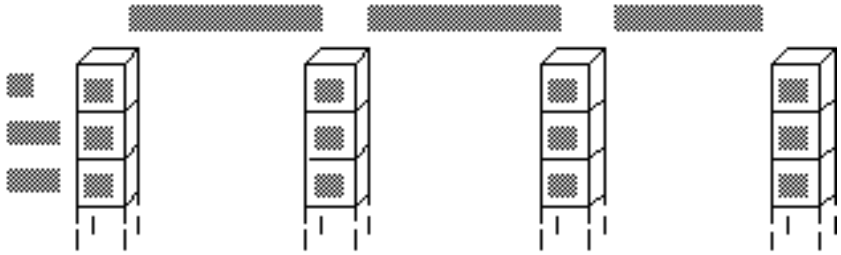

Figure 6.5 Status of the Previously Initialized Register Stack

Register-Pop Format

The register-pop format treats coprocessor registers as a modified stack. The source register must always be the stack top. Specify the destination with the register's name.

Instructions with this format place the result of the operation into the destination operand, and the top pops off the stack. The register-pop format is used only for instructions for calculations on two values, as in this example and in Figure 6.6:

```
faddp   st(2), st ; Add first and third positions and pop -
                  ;   first position destroyed;
                  ;   third moves to second and holds result
```

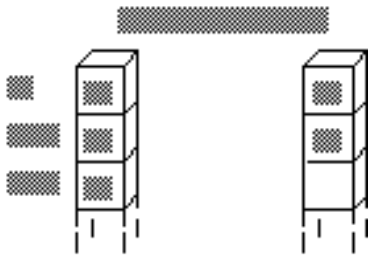

Figure 6.6 Status of the Already Initialized Register Stack

Coordinating Memory Access

The math coprocessor and main processor work simultaneously. However, since the coprocessor cannot handle device input or output, data originates in the main processor.

The main processor and the coprocessor have their own registers, which are separate and inaccessible to each other. They exchange data through memory, since memory is available to both.

When using the coprocessor, follow these three steps:

1. Load data from memory to coprocessor registers.
2. Process the data.
3. Store the data from coprocessor registers back to memory.

Step 2, processing the data, can occur while the main processor is handling other tasks. Steps 1 and 3 must be coordinated with the main processor so that the processor and coprocessor do not try to access the same memory at the same time; otherwise, problems of coordinating memory access can occur. Since the processor and coprocessor work independently, they may not finish working on memory in the order in which you give instructions. The two potential timing conflicts that can occur are handled in different ways.

One timing conflict results from a coprocessor instruction following a processor instruction. The processor may have to wait until the coprocessor finishes if the next processor instruction requires the result of the coprocessor's calculation. You do not have to write your code to avoid this conflict, however. The assembler coordinates this timing automatically for the 8088 and 8086 processors, and the processor coordinates it automatically on the 80186–80486 processors. This is the case shown in the first example that follows.

Another conflict results from a processor instruction that accesses memory following a coprocessor instruction that accesses the same memory. The processor can try to load a variable that is still being used by the coprocessor. You need careful synchronization to control the timing, and this synchronization is not automatic on the 8087 coprocessor. For code to run correctly on the 8087, you must include **WAIT** or **FWAIT** (mnemonics for the same instruction) to ensure that the coprocessor finishes before the processor begins, as shown in the second example.

In this situation, the processor does not generate the **FWAIT** instruction automatically.

```
; Processor instruction first - No wait needed
        mov     WORD PTR mem32[0], ax   ; Load memory
        mov     WORD PTR mem32[2], dx
        fild    mem32                   ; Load to register

; Coprocessor instruction first - Wait needed (for 8087)
        fist    mem32                   ; Store to memory
        fwait                           ; Wait until coprocessor
                                        ;   is done
        mov     ax, WORD PTR mem32[0]   ; Move to register
        mov     dx, WORD PTR mem32[2]
```

When generating code for the 8087 coprocessor, the assembler automatically inserts a **WAIT** instruction before the coprocessor instruction. However, if you use the **.286** or **.386** directive, the compiler assumes that the coprocessor instructions are for the 80287 or 80387 and does not insert the **WAIT** instruction. If your code does not need to run on an 8086 or 8088 processor, you can make your programs smaller and more efficient by using the **.286** or **.386** directive.

Using Coprocessor Instructions

The 8087 family of coprocessors has separate instructions for each of the following operations:

- Loading and storing data
- Doing arithmetic calculations
- Controlling program flow

The following sections explain the available instructions and show how to use them for each of these operations. For general syntax information, see "Instruction and Operand Formats," earlier in this section.

Loading and Storing Data

Data-transfer instructions copy data between main memory and the coprocessor registers or between different coprocessor registers. Two principles govern data transfers:

- The choice of instruction determines whether a value in memory is considered an integer, a BCD number, or a real number. The value is always considered a 10-byte real number once transferred to the coprocessor.

- The size of the operand determines the size of a value in memory. Values in the coprocessor always take up 10 bytes.

You can transfer data to stack registers using load commands. These commands push data onto the stack from memory or from coprocessor registers. Store commands remove data. Some store commands pop data off the register stack into memory or coprocessor registers; others simply copy the data without changing it on the stack.

If you use constants as operands, you cannot load them directly into coprocessor registers. You must allocate memory and initialize a variable to a constant value. That variable can then be loaded by using one of the load instructions in the following list.

The math coprocessor offers a few special instructions for loading certain constants. You can load 0, 1, pi, and several common logarithmic values directly. Using these instructions is faster and often more precise than loading the values from initialized variables.

All instructions that load constants have the stack top as the implied destination operand. The constant to be loaded is the implied source operand.

The coprocessor data area, or parts of it, can also be moved to memory and later loaded back. You may want to do this to save the current state of the coprocessor before executing a procedure. After the procedure ends, restore the previous status. Saving coprocessor data is also useful when you want to modify coprocessor behavior by writing certain data to main memory, operating on the data with 8086-family instructions, and then loading it back to the coprocessor data area.

Use the following instructions for transferring numbers to and from registers:

Instruction(s)	Description
FLD, **FST**, **FSTP**	Loads and stores real numbers
FILD, **FIST**, **FISTP**	Loads and stores binary integers
FBLD	Loads BCD
FBSTP	Stores BCD
FXCH	Exchanges register values
FLDZ	Pushes 0 into ST
FLD1	Pushes 1 into ST
FLDPI	Pushes the value of pi into ST
FLDCW *mem2byte*	Loads the control word into the coprocessor
F[N]STCW *mem2byte*	Stores the control word in memory

FLDENV *mem14byte*	Loads environment from memory
F[N]STENV *mem14byte*	Stores environment in memory
Instruction(s)	**Description**
FRSTOR *mem94byte*	Restores state from memory
F[N]SAVE *mem94byte*	Saves state in memory
FLDL2E	Pushes the value of $\log_2 e$ into ST
FLDL2T	Pushes $\log_2 10$ into ST
FLDLG2	Pushes $\log_{10} 2$ into ST
FLDLN2	Pushes $\log_e 2$ into ST

The following example and Figure 6.7 illustrate some of these instructions:

```
        .DATA
m1      REAL4   1.0
m2      REAL4   2.0
        .CODE
        fld     m1      ; Push m1 into first item
        fld     st(2)   ; Push third item into first
        fst     m2      ; Copy first item to m2
        fxch    st(2)   ; Exchange first and third items
        fstp    m1      ; Pop first item into m1
```

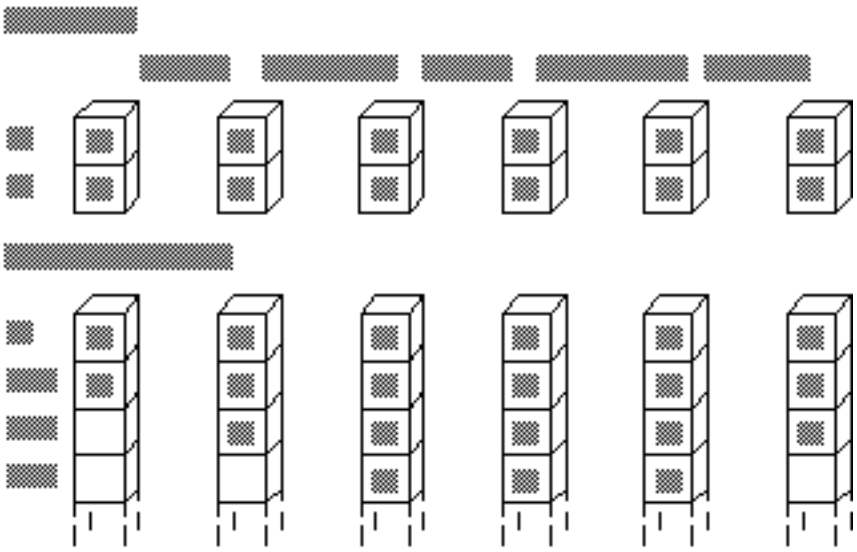

Figure 6.7 Status of the Register Stack: Main Memory and Coprocessor

Doing Arithmetic Calculations

Most of the coprocessor instructions for arithmetic operations have several forms, depending on the operand used. You do not need to specify the operand type in the

instruction if both operands are stack registers, since register values are always 10-byte real numbers. In most of the arithmetic instructions listed here, the result replaces the destination register. The instructions include:

Instruction	Description
FADD	Adds the source and destination
FSUB	Subtracts the source from the destination
FSUBR	Subtracts the destination from the source
FMUL	Multiplies the source and the destination
FDIV	Divides the destination by the source
FDIVR	Divides the source by the destination
FABS	Sets the sign of ST to positive
FCHS	Reverses the sign of ST
FRNDINT	Rounds ST to an integer
FSQRT	Replaces the contents of ST with its square root
FSCALE	Multiplies the stack-top value by 2 to the power contained in ST(1)
FPREM	Calculates the remainder of ST divided by ST(1)

80387 Only

Instruction	Description
FSIN	Calculates the sine of the value in ST
FCOS	Calculates the cosine of the value in ST
FSINCOS	Calculates the sine and cosine of the value in ST
FPREM1	Calculates the partial remainder by performing modulo division on the top two stack registers
FXTRACT	Breaks a number down into its exponent and mantissa and pushes the mantissa onto the register stack
F2XM1	Calculates 2^X-1
FYL2X	Calculates $Y * \log_2 X$
FYL2XP1	Calculates $Y * \log_2 (X+1)$
FPTAN	Calculates the tangent of the value in ST
FPATAN	Calculates the arctangent of the ratio Y/X
F[N]INIT	Resets the coprocessor and restores all the default conditions in the control and status words
F[N]CLEX	Clears all exception flags and the busy flag of the status word
FINCSTP	Adds 1 to the stack pointer in the status word
FDECSTP	Subtracts 1 from the stack pointer in the status word
FFREE	Marks the specified register as empty

The following example illustrates several arithmetic instructions. The code solves quadratic equations, but does no error checking and fails for some values because it attempts to find the square root of a negative number. Both Help and the MATH.ASM sample file show a complete version of this procedure. The complete form uses the **FTST** (Test for Zero) instruction to check for a negative number or 0 before calculating the square root.

```
        .DATA
a       REAL4   3.0
b       REAL4   7.0
cc      REAL4   2.0
posx    REAL4   0.0
negx    REAL4   0.0

        .CODE
        .
        .
        .
; Solve quadratic equation - no error checking
; The formula is: -b +/- squareroot(b^2 - 4ac) / (2a)
        fld1                ; Get constants 2 and 4
        fadd    st,st       ; 2 at bottom
        fld     st          ; Copy it
        fmul    a           ; = 2a

        fmul    st(1),st    ; = 4a
        fxch                ; Exchange
        fmul    cc          ; = 4ac

        fld     b           ; Load b
        fmul    st,st       ; = b^2
        fsubr               ; = b^2 - 4ac
                            ; Negative value here produces error
        fsqrt               ; = square root(b^2 - 4ac)
        fld     b           ; Load b
        fchs                ; Make it negative
        fxch                ; Exchange

        fld     st          ; Copy square root
        fadd    st,st(2)    ; Plus version = -b + root(b^2 - 4ac)
        fxch                ; Exchange
        fsubp   st(2),st    ; Minus version = -b - root(b^2 - 4ac)

        fdiv    st,st(2)    ; Divide plus version
        fstp    posx        ; Store it
        fdivr               ; Divide minus version
        fstp    negx        ; Store it
```

Controlling Program Flow

The math coprocessor has several instructions that set control flags in the status word. The 8087-family control flags can be used with conditional jumps to direct program flow in the same way that 8086-family flags are used. Since the coprocessor does not have jump instructions, you must transfer the status word to memory so that the flags can be used by 8086-family instructions.

An easy way to use the status word with conditional jumps is to move its upper byte into the lower byte of the processor flags, as shown in this example:

```
fstsw   mem16           ; Store status word in memory
fwait                   ; Make sure coprocessor is done
mov     ax, mem16       ; Move to AX
sahf                    ; Store upper word in flags
```

The **SAHF** (Store AH into Flags) instruction in this example transfers AH into the low bits of the flags register.

You can save several steps by loading the status word directly to AX on the 80287 with the **FSTSW** and **FNSTSW** instructions. This is the only case in which data can be transferred directly between processor and coprocessor registers, as shown in this example:

```
fstsw   ax
```

The coprocessor control flags and their relationship to the status word are described in “Control Registers,” following.

The 8087-family coprocessors provide several instructions for comparing operands and testing control flags. All these instructions compare the stack top (ST) to a source operand, which may either be specified or implied as ST(1).

The compare instructions affect the C3, C2, and C0 control flags, but not the C1 flag. Table 6.3 shows the flags’ settings for each possible result of a comparison or test.

Table 6.3 Control-Flag Settings After Comparison or Test

After FCOM	After FTEST	C3	C2	C0
ST > *source*	ST is positive	0	0	0
ST < *source*	ST is negative	0	0	1
ST = *source*	ST is 0	1	0	0
Not comparable	ST is NAN or projective infinity	1	1	1

Variations on the compare instructions allow you to pop the stack once or twice and to compare integers and zero. For each instruction, the stack top is always

the implied destination operand. If you do not give an operand, ST(1) is the implied source. With some compare instructions, you can specify the source as a memory or register operand.

All instructions summarized in the following list have implied operands: either ST as a single-destination operand or ST as the destination and ST(1) as the source. Each instruction in the list has implied operands. Some instructions have a wait version and a no-wait version. The no-wait versions have **N** as the second letter. The instructions for comparing and testing flags include:

Instruction	Description
FCOM	Compares the stack top to the source. The source and destination are unaffected by the comparison.
FTST	Compares ST to 0.
FCOMP	Compares the stack top to the source and then pops the stack.
FUCOM, **FUCOMP**, **FUCOMPP**	Compares the source to ST and sets the condition codes of the status word according to the result (80386/486 only).
F[N]STSW *mem2byte*	Stores the status word in memory.
FXAM	Sets the value of the control flags based on the type of the number in ST.
FPREM	Finds a correct remainder for large operands. It uses the C2 flag to indicate whether the remainder returned is partial (C2 is set) or complete (C2 is clear). If the bit is set, the operation should be repeated. It also returns the least-significant three bits of the quotient in C0, C3, and C1.
FNOP	Copies the stack top onto itself, thus padding the executable file and taking up processing time without having any effect on registers or memory.
FDISI, **FNDISI**, **FENI**, **FNENI**	Enables or disables interrupts (8087 only).
FSETPM	Sets protected mode. Requires a **.286P** or **.386P** directive (80287, 80387, and 80486 only).

The following example illustrates some of these instructions. Notice how conditional blocks are used to enhance 80287 code.

```
        .DATA
down    REAL4   10.35       ; Sides of a rectangle
across  REAL4   13.07
diamtr  REAL4   12.93       ; Diameter of a circle
status  WORD    ?
```

```
P287    EQU     (@Cpu AND 00111y)
        .CODE
        .
        .
        .
; Get area of rectangle
        fld     across      ; Load one side
        fmul    down        ; Multiply by the other

; Get area of circle:  Area = PI * (D/2)2
        fld1                ; Load one and
        fadd    st, st      ;   double it to get constant 2
        fdivr   diamtr      ; Divide diameter to get radius
        fmul    st, st      ; Square radius
        fldpi               ; Load pi
        fmul                ; Multiply it

; Compare area of circle and rectangle
        fcompp              ; Compare and throw both away
        IF      p287
        fstsw   ax          ; (For 287+, skip memory)
        ELSE
        fnstsw  status      ; Load from coprocessor to memory
        mov     ax, status  ; Transfer memory to register
        ENDIF
        sahf                ; Transfer AH to flags register
        jp      nocomp      ; If parity set, can't compare
        jz      same        ; If zero set, they're the same
        jc      rectangle   ; If carry set, rectangle is bigger
        jmp     circle      ;   else circle is bigger

nocomp:                     ; Error handler
        .
        .
        .
same:                       ; Both equal
        .
        .
        .
rectangle:                  ; Rectangle bigger
        .
        .
        .
circle:                     ; Circle bigger
```

Additional instructions for the 80387/486 are **FLDENVD** and **FLDENVW** for loading the environment; **FNSTENVD**, **FNSTENVW**, **FSTENVD**, and **FSTENVW** for storing the environment state; **FNSAVED**, **FNSAVEW**, **FSAVED**, and **FSAVEW** for saving the coprocessor state; and **FRSTORD** and **FRSTORW** for restoring the coprocessor state.

The size of the code segment, not the operand size, determines the number of bytes loaded or stored with these instructions. The instructions ending with **W** store the 16-bit form of the control register data, and the instructions ending with **D** store the 32-bit form. For example, in 16-bit mode **FSAVEW** saves the 16-bit control register data. If you need to store the 32-bit form of the control register data, use **FSAVED**.

Control Registers

Some of the flags of the seven 16-bit control registers control coprocessor operations, while others maintain the current status of the coprocessor. In this sense, they are much like the 8086-family flags registers (see Figure 6.8).

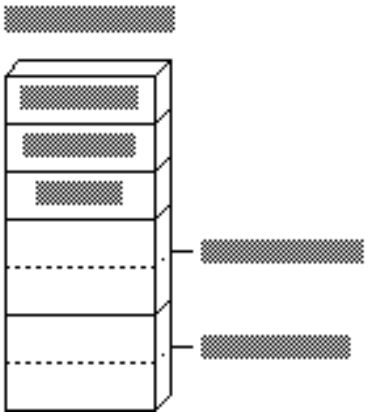

Figure 6.8 Coprocessor Control Registers

The status word register is the only commonly used control register. (The others are used mostly by systems programmers.) The format of the status word register is shown in Figure 6.9, which shows how the coprocessor control flags align with the processor flags. C3 overwrites the zero flag, C2 overwrites the parity flag, and C0 overwrites the carry flag. C1 overwrites an undefined bit, so it cannot be used directly with conditional jumps, although you can use the **TEST** instruction to

check C1 in memory or in a register. The status word register also overwrites the sign and auxiliary-carry flags, so you cannot count on their being unchanged after the operation.

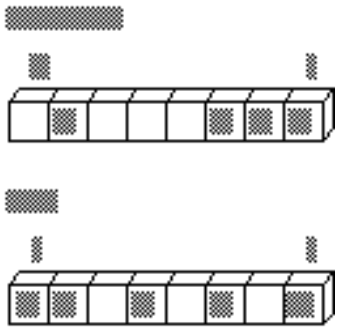

Figure 6.9 Coprocessor and Processor Control Flags

Using An Emulator Library

If you do not have a math coprocessor or an 80486 processor, you can do most floating-point operations by writing assembly-language procedures and accessing an emulator from a high-level language. All Microsoft high-level languages come with emulator libraries for all memory models.

To use emulator functions, first write your assembly-language procedure using coprocessor instructions. Then assemble the module with the /FPi option and link it with your high-level–language modules. You can enter options in the Programmer's WorkBench (PWB) environment, or you can use the **OPTION EMULATOR** in your source code.

In emulation mode, the assembler generates instructions for the linker that the Microsoft emulator can use. The form of the **OPTION** directive in the following example tells the assembler to use emulation mode. This option (introduced in Chapter 1) can be defined only once in a module.

```
OPTION EMULATOR
```

You can use emulator functions in a stand-alone assembler program by assembling with the /Cx command-line option and linking with the appropriate emulator library. The following fragment outlines a small-model program that contains floating-point instructions served by an emulator:

```
        .MODEL  small, c
        OPTION  EMULATOR
        .
        .
        .
        PUBLIC  main
        .CODE
main:                           ; Program entry point must
        .STARTUP                ;   have name 'main'
        .
        fadd    st, st          ; Floating-point instructions
        fldpi                   ;   emulated
```

Emulator libraries do not allow for all of the coprocessor instructions. The following floating-point instructions are not emulated:

FBLD	**FLDENV**	**FSAVE**	**FSTENV**
FBSTP	**FNOP**	**FSAVEW**	**FUCOM**
FCOS	**FPREM1**	**FSAVED**	**FUCOMP**
FDECSTP	**FRSTOR**	**FSETPM**	**FUCOMPP**
FINCSTP	**FRSTORW**	**FSIN**	**FXTRACT**
FINIT	**FRSTORD**	**FSINCOS**	

For information about writing assembly-language procedures for high-level languages, see Chapter 12, "Mixed-Language Programming."

Using Binary Coded Decimal Numbers

Binary coded decimal (BCD) numbers allow calculations on large numbers without rounding errors. This characteristic makes BCD numbers a common choice for monetary calculations. Although BCDs can represent integers of any precision, the 8087-based coprocessors accommodate BCD numbers only in the range ±999,999,999,999,999,999.

This section explains how to define BCD numbers, how to access them with a math coprocessor or emulator, and how to perform simple BCD calculations on the main processor.

Defining BCD Constants and Variables

Unpacked BCD numbers are made up of bytes containing a single decimal digit in the lower 4 bits of each byte. Packed BCD numbers are made up of bytes

containing two decimal digits: one in the upper 4 bits and one in the lower 4 bits. The leftmost digit holds the sign (0 for positive, 1 for negative).

Packed BCD numbers are encoded in the 8087 coprocessor's packed BCD format. They can be up to 18 digits long, packed two digits per byte. The assembler zero-pads BCDs initialized with fewer than 18 digits. Digit 20 is the sign bit, and digit 19 is reserved.

When you define an integer constant with the **TBYTE** directive and the current radix is decimal (`t`), the assembler interprets the number as a packed BCD number.

The syntax for specifying packed BCDs is the same as for other integers.

```
pos1    TBYTE   1234567890  ; Encoded as 0000000001234567890h
neg1    TBYTE   -1234567890 ; Encoded as 8000000001234567890h
```

Unpacked BCD numbers are stored one digit to a byte, with the value in the lower
4 bits. They can be defined using the **BYTE** directive. For example, an unpacked BCD number could be defined and initialized as follows:

```
unpackedr      BYTE    1,5,8,2,5,2,9   ; Initialized to 9,252,851
unpackedf      BYTE    9,2,5,2,8,5,1   ; Initialized to 9,252,851
```

As these two lines show, you can arrange digits backward or forward, depending on how you write the calculation routines that handle the numbers.

BCD Calculations on a Coprocessor

As the previous section explains, BCDs differ from other numbers only in the way a program stores them in memory. Internally, a math coprocessor does not distinguish BCD integers from any other type. The coprocessor can load, calculate, and store packed BCD integers up to 18 digits long.

The coprocessor instruction

```
        fbld            bcd1
```

pushes the packed BCD number at `bcd1` onto the coprocessor stack. When your code completes calculations on the number, place the result back into memory in BCD format with the instruction

```
        fbstp           bcd1
```

which discards the variable from the stack top.

BCD Calculations on the Main Processor

The 8086-family of processors can perform simple arithmetic operations on BCD integers, but only one digit at a time. The main processor, like the coprocessor, operates internally on the number's binary value. It requires additional code to translate the binary result back into BCD format.

The main processor provides instructions specifically designed to translate to and from BCD format. These instructions are called "ASCII-adjust" and "decimal-adjust" instructions. They get their names from Intel mnemonics that use the term "ASCII" to refer to unpacked BCD numbers and "decimal" to refer to packed BCD numbers.

Unpacked BCD Numbers

When a calculation using two one-digit values produces a two-digit result, the instructions **AAA**, **AAS**, **AAM**, and **AAD** place the first digit in AL and the second in AH. If the digit in AL needs to carry to or borrow from the digit in AH, the instructions set the carry and auxiliary carry flags. The four ASCII-adjust instructions for unpacked BCDs are:

Instruction	Description
AAA	Adjusts after an addition operation.
AAS	Adjusts after a subtraction operation.
AAM	Adjusts after a multiplication operation. Always use with **MUL**, not with **IMUL**.
AAD	Adjusts before a division operation. Unlike other BCD instructions, **AAD** converts a BCD value to a binary value before the operation. After the operation, use **AAM** to adjust the quotient. The remainder is lost. If you need the remainder, save it in another register before adjusting the quotient. Then move it back to AL and adjust if necessary.

For processor arithmetic on unpacked BCD numbers, you must do the 8-bit arithmetic calculations on each digit separately, and assign the result to the AL register. After each operation, use the corresponding BCD instruction to adjust the result. The ASCII-adjust instructions do not take an operand and always work on the value in the AL register.

The following examples show how to use each of these instructions in BCD addition, subtraction, multiplication, and division.

```
; To add 9 and 3 as BCDs:
        mov     ax, 9       ; Load 9
        mov     bx, 3       ;   and 3 as unpacked BCDs
        add     al, bl      ; Add 09h and 03h to get 0Ch
        aaa                 ; Adjust 0Ch in AL to 02h,
                            ;   increment AH to 01h, set carry
                            ; Result 12 (unpacked BCD in AX)

; To subtract 4 from 13:
        mov     ax, 103h    ; Load 13
        mov     bx, 4       ;   and 4 as unpacked BCDs
        sub     al, bl      ; Subtract 4 from 3 to get FFh (-1)
        aas                 ; Adjust 0FFh in AL to 9,
                            ;   decrement AH to 0, set carry
                            ; Result 9 (unpacked BCD in AX)

; To multiply 9 times 3:
        mov     ax, 903h    ; Load 9 and 3 as unpacked BCDs
        mul     ah          ; Multiply 9 and 3 to get 1Bh
        aam                 ; Adjust 1Bh in AL
                            ;   to get 27 (unpacked BCD in AX)

; To divide 25 by 2:
        mov     ax, 205h    ; Load 25
        mov     bl, 2       ;   and 2 as unpacked BCDs
        aad                 ; Adjust 0205h in AX
                            ;   to get 19h in AX
        div     bl          ; Divide by 2 to get
                            ;   quotient 0Ch in AL
                            ;   remainder 1 in AH
        aam                 ; Adjust 0Ch in AL
                            ;   to 12 (unpacked BCD in AX)
                            ;   (remainder destroyed)
```

If you process multidigit BCD numbers in loops, each digit is processed and adjusted in turn.

Packed BCD Numbers

Packed BCD numbers are made up of bytes containing two decimal digits: one in the upper 4 bits and one in the lower 4 bits. The 8086-family processors provide instructions for adjusting packed BCD numbers after addition and subtraction. You must write your own routines to adjust for multiplication and division.

For processor calculations on packed BCD numbers, you must do the 8-bit arithmetic calculations on each byte separately, placing the result in the AL register. After each operation, use the corresponding decimal-adjust instruction to adjust the result. The decimal-adjust instructions do not take an operand and always work on the value in the AL register.

The 8086-family processors provide the instructions **DAA** (Decimal Adjust after Addition) and **DAS** (Decimal Adjust after Subtraction) for adjusting packed BCD numbers after addition and subtraction.

These examples use **DAA** and **DAS** to add and subtract BCDs.

```
;To add 88 and 33:
        mov     ax, 8833h   ; Load 88 and 33 as packed BCDs
        add     al, ah      ; Add 88 and 33 to get 0BBh
        daa                 ; Adjust 0BBh to 121 (packed BCD:)
                            ;   1 in carry and 21 in AL

;To subtract 38 from 83:
        mov     ax, 3883h   ; Load 83 and 38 as packed BCDs
        sub     al, ah      ; Subtract 38 from 83 to get 04Bh
        das                 ; Adjust 04Bh to 45 (packed BCD:)
                            ;   0 in carry and 45 in AL
```

Unlike the ASCII-adjust instructions, the decimal-adjust instructions never affect AH. The assembler sets the auxiliary carry flag if the digit in the lower 4 bits carries to or borrows from the digit in the upper 4 bits, and it sets the carry flag if the digit in the upper 4 bits needs to carry to or borrow from another byte.

Multidigit BCD numbers are usually processed in loops. Each byte is processed and adjusted in turn.