A user-written exception handler can itself cause an unending wait. When the exception handler starts to run, the 8087 is suspended with its BUSY line active, waiting for the exception to be cleared, and interrupts on the CPU are disabled. If, in this condition, the exception handler issues any 8087 instruction, other than a no-wait form, the result will be an unending wait. To prevent this, the exception handler should clear the exception on the 8087 and enable interrupts on the CPU before executing any instruction that is preceded by a WAIT.

More generally, an instruction that is preceded by a WAIT (or an FWAIT instruction) should never be executed when CPU interrupts are disabled and there is any possibility that the 8087's BUSY line is active.

Status Lines

When the 8087 has control of the local bus, it emits signals on status lines S2-S0 to identify the type of bus cycle it is running. The 8087 generates the restricted (compared to a CPU) set of encodings shown in table S-8. These lines correspond exactly to the signals output by the 8086 and 8088 CPU's, and are normally decoded by an 8288 Bus Controller.

Table S-8. Bus Cycle Status Signals

$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	Type of Bus Cycle
1	0	1	Read Memory
1	1	0	Write Memory
1	1	1	Passive; no bus cycle

Status line S7 is currently identical to BHE of the same bus cycle, while S4 and S3 are both currently 1; however, these signals are reserved by Intel for possible future use. Status line S6 emits 1 and S5 emits 0.

S.7 Instruction Set

This section describes the operation of each of the 8087's 69 instructions. The first part of the section describes the function of each instruction in detail. For this discussion, the instructions are divided into six functional groups: data transfer, arithmetic, comparison, transcendental, constant, and processor control. The second part provides instruction attributes such as execution speed, bus transfers, and exceptions, as well as a coding example for each combination of operands accepted by the instruction. This information is concentrated in a table, organized alphabetically by instruction mnemonic, for easy reference.

Throughout this section, the instruction set is described as it appears to the ASM-86 programmer who is coding a program. Appendix A covers the actual machine instruction encodings, which are principally of use to those reading unformatted memory dumps, monitoring instruction fetches on the bus, or writing exception handlers.

The instruction descriptions in this section concentrate on describing the normal function of each operation. Table S-19 lists the exceptions that can occur for each instruction and table S-32 details the causes of exceptions as well as the 8087's masked responses.

The typical NDP instruction accepts one or two operands as "inputs", operates on these, and produces a result as an "output". Operands are

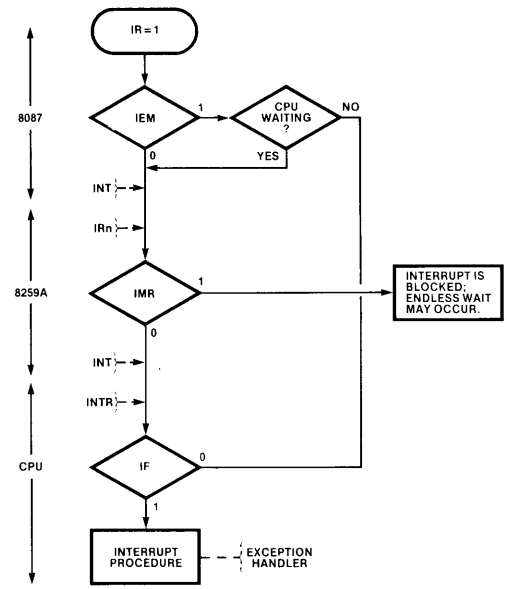


Figure S-17. Interrupt Request Path

most often (the contents of) register or memory locations. The operands of some instructions are predefined; for example, FSQRT always takes the square root of the number in the top stack element. Others allow, or require, the programmer to explicitly code the operand(s) along with the instruction mnemonic. Still others accept one explicit operand and one implicit operand, which is usually the top stack element.

Whether supplied by the programmer or utilized automatically, there are two basic types of operands, *sources* and *destinations*. A source operand simply supplies one of the "inputs" to an instruction; it is not altered by the instruction. Even when an instruction converts the source operand from one format to another (e.g., real to integer), the conversion is actually performed in an internal work area to avoid altering the source operand. A destination operand may also provide an "input" to an instruction. It is distinguished from a source operand, however, because its content may be altered when it receives the result produced by the operation; that is, the destination is replaced by the result.

Many instructions allow their operands to be coded in more than one way. For example, FADD (add real) may be written without operands, with only a source or with a destination and a source. The instruction descriptions in this section employ the simple convention of separating alternative operand forms with slashes; the slashes, however, are not coded. Consecutive slashes indicate an option of no explicit operands. The operands for FADD are thus described as:

//source/destination, source

This means that FADD may be written in any of three ways:

FADD
FADD *source*
FADD *destination, source*

When reading this section, it is important to bear in mind that memory operands may be coded with any of the CPU's memory addressing modes. To review these modes—direct, register indirect, based, indexed, based indexed—refer to sections 2.8 and 2.9. Table S-22 in this chapter also provides several addressing mode examples.

Data Transfer Instructions

These instructions (summarized in table S-9) move operands among elements of the register stack, and between the stack top and memory. Any of the seven data types can be converted to temporary real and loaded (pushed) onto the stack in a single operation; they can be stored to memory in the same manner. The data transfer instructions automatically update the 8087 tag word to reflect the register contents following the instruction.

FLD *source*

FLD (load real) loads (pushes) the source operand onto the top of the register stack. This is done by decrementing the stack pointer by one and then copying the content of the source to the new stack top. The source may be a register on the stack (ST(i)) or any of the real data types in memory. Short and long real source operands are converted to temporary real automatically. Coding FLD ST(0) duplicates the stack top.

Table S-9. Data Transfer Instructions

Real Transfers	
FLD	Load real
FST	Store real
FSTP	Store real and pop
FXCH	Exchange registers
Integer Transfers	
FILD	Integer load
FIST	Integer store
FISTP	Integer store and pop
Packed Decimal Transfers	
FBLD	Packed decimal (BCD) load
FBSTP	Packed decimal (BCD) store and pop

FST *destination*

FST (store real) transfers the stack top to the destination, which may be another register on the stack or a short or long real memory operand. If the destination is short or long real, the significand is rounded to the width of the destination

according to the RC field of the control word, and the exponent is converted to the width and bias of the destination format.

If, however, the stack top is tagged special (it contains ∞, a NAN, or a denormal) then the stack top's significand is not rounded but is chopped (on the right) to fit the destination. Neither is the exponent converted, but it also is chopped on the right and transferred "as is". This preserves the value's identification as ∞ or a NAN (exponent all ones) or a denormal (exponent all zeros) so that it can be properly loaded and tagged later in the program if desired.

FSTP *destination*

FSTP (store real and pop) operates identically to FST except that the stack is popped following the transfer. This is done by tagging the top stack element empty and then incrementing ST. FSTP permits storing to a temporary real memory variable while FST does not. Coding FSTP ST(0) is equivalent to popping the stack with no data transfer.

FXCH *//destination*

FXCH (exchange registers) swaps the contents of the destination and the stack top registers. If the destination is not coded explicitly, ST(1) is used. Many 8087 instructions operate only on the stack top; FXCH provides a simple means of effectively using these instructions on lower stack elements. For example, the following sequence takes the square root of the third register from the top:

```
FXCH  ST(3)
FSQRT
FXCH  ST(3)
```

FILD *source*

FILD (integer load) converts the source memory operand from its binary integer format (word, short, or long) to temporary real and loads (pushes) the result onto the stack. The (new) stack top is tagged zero if all bits in the source were zero, and is tagged valid otherwise.

FIST *destination*

FIST (integer store) rounds the content of the stack top to an integer according to the RC field of the control word and transfers the result to the destination. The destination may define a word or short integer variable. Negative zero is stored in the same encoding as postive zero: 0000...00.

FISTP *destination*

FISTP (integer store and pop) operates like FIST and also pops the stack following the transfer. The destination may be any of the binary integer data types.

FBLD *source*

FBLD (packed decimal (BCD) load) converts the content of the source operand from packed decimal to temporary real and loads (pushes) the result onto the stack. The sign of the source is preserved, including the case where the value is negative zero. FBLD is an exact operation; the source is loaded with no rounding error.

The packed decimal digits of the source are assumed to be in the range 0-9H. The instruction does not check for invalid digits (A-FH) and the result of attempting to load an invalid encoding is undefined.

FBSTP *destination*

FBSTP (packed decimal (BCD) store and pop) converts the content of the stack top to a packed decimal integer, stores the result at the destination in memory, and pops the stack. FBSTP produces a rounded integer from a non-integral value by adding 0.5 to the value and then chopping. Users who are concerned about rounding may precede FBSTP with FRNDINT.

Arithmetic Instructions

The 8087's arithmetic instruction set (table S-10) provides a wealth of variations on the basic add, subtract, multiply, and divide operations, and a number of other useful functions. These range from a simple absolute value to a square root instruction that executes faster than ordinary divi-

Table S-10. Arithmetic Instructions

Addition	
FADD	Add real
FADDP	Add real and pop
FIADD	Integer add
Subtraction	
FSUB	Subtract real
FSUBP	Subtract real and pop
FISUB	Integer subtract
FSUBR	Subtract real reversed
FSUBRP	Subtract real reversed and pop
FISUBR	Integer subtract reversed
Multiplication	
FMUL	Multiply real
FMULP	Multiply real and pop
FIMUL	Integer multiply
Division	
FDIV	Divide real
FDIVP	Divide real and pop
FIDIV	Integer divide
FDIVR	Divide real reversed
FDIVRP	Divide real reversed and pop
FIDIVR	Integer divide reversed
Other Operations	
FSQRT	Square root
FSCALE	Scale
FPREM	Partial remainder
FRNDINT	Round to integer
FXTRACT	Extract exponent and significand
FABS	Absolute value
FCHS	Change sign

sion; 8087 programmers no longer need to spend valuable time eliminating square roots from algorithms because they run too slowly. Other arithmetic instructions perform exact modulo division, round real numbers to integers, and scale values by powers of two.

The 8087's basic arithmetic instructions (addition, subtraction, multiplication, and division) are designed to encourage the development of very efficient algorithms. In particular, they allow the programmer to minimize memory references and to make optimum use of the NDP register stack.

Table S-11 summarizes the available operation/operand forms that are provided for basic arithmetic. In addition to the four normal operations, two "reversed" instructions make subtraction and division "symmetrical" like addition and multiplication. The variety of instruction and operand forms give the programmer unusual flexibility:

- operands may be located in registers or memory;
- results may be deposited in a choice of registers;
- operands may be a variety of NDP data types: temporary real, long real, short real, short integer or word integer, with automatic conversion to temporary real performed by the 8087.

Five basic instruction forms may be used across all six operations, as shown in table S-11. The classical stack form may be used to make the 8087 operate like a classical stack machine. No operands are coded in this form, only the instruction mnemonic. The NDP picks the source operand from the stack top and the destination from the next stack element. It then pops the stack, performs the operation, and returns the result to the new stack top, effectively replacing the operands by the result.

The register form is a generalization of the classical stack form; the programmer specifies the stack top as one operand and any register on the stack as the other operand. Coding the stack top as the destination provides a convenient way to access a constant, held elsewhere in the stack, from the stack top. The converse coding (ST is the source operand) allows, for example, adding the top into a register used as an accumulator.

Often the operand in the stack top is needed for one operation but then is of no further use in the computation. The register pop form can be used to pick up the stack top as the source operand, and then discard it by popping the stack. Coding operands of ST(1),ST with a register pop mnemonic is equivalent to a classical stack operation: the top is popped and the result is left at the new top.

Table S-11. Basic Arithmetic Instructions and Operands

Instruction Form	Mnemonic Form	Operand Forms destination, source	ASM-86 Example
Classical stack	F*op*	{ST(1),ST}	FADD
Register	F*op*	ST(i),ST or ST,ST(i)	FSUB ST,ST(3)
Register pop	F*op*P	ST(i),ST	FMULP ST(2),ST
Real memory	F*op*	{ST,} short-real/long-real	FDIV AZIMUTH
Integer memory	FI*op*	{ST,} word-integer/short-integer	FIDIV N_PULSES

NOTES: Braces { } surround *implicit* operands; these are not coded, and are shown here for information only.

op = ADD destination ← destination + source
SUB destination ← destination − source
SUBR destination ← source − destination
MUL destination ← destination • source
DIV destination ← destination ÷ source
DIVR destination ← source ÷ destination

The two memory forms increase the flexibility of the 8087's arithmetic instructions. They permit a real number or a binary integer in memory to be used directly as a source operand. This is a very useful facility in situations where operands are not used frequently enough to justify holding them in registers. Note that any memory addressing mode may be used to define these operands, so they may be elements in arrays, structures or other data organizations, as well as simple scalars.

The six basic operations are discussed further in the next paragraphs, and descriptions of the remaining seven arithmetic operations follow.

Addition
FADD *//source/destination,source*
FADDP *destination,source*
FIADD *source*

The addition instructions (add real, add real and pop, integer add) add the source and destination operands and return the sum to the destination. The operand at the stack top may be doubled by coding:

```
FADD  ST,ST(0)
```

Normal Subtraction
FSUB *//source/destination,source*
FSUBP *destination,source*
FISUB *source*

The normal subtraction instructions (subtract real, subtract real and pop, integer subtract) subtract the source operand from the destination and return the difference to the destination.

Reversed Subtraction
FSUBR *//source/destination,source*
FSUBRP *destination,source*
FISUBR *source*

The reversed subtraction instructions (subtract real reversed, subtract real reversed and pop, integer subtract reversed) subtract the destination from the source and return the difference to the destination.

Multiplication
FMUL *//source/destination,source*
FMULP *destination,source*
FIMUL *source*

The multiplication instructions (multiply real, multiply real and pop, integer multiply) multiply the source and destination operands and return

the product to the destination. Coding FMUL ST,ST(0) squares the content of the stack top.

Normal Division

FDIV *//source/destination,source*
FDIVP *destination,source*
FIDIV *source*

The normal division instructions (divide real, divide real and pop, integer divide) divide the destination by the source and return the quotient to the destination.

Reversed Division

FDIVR *//source/destination,source*
FDIVRP *destination,source*
FIDIVR *source*

The reversed division instructions (divide real reversed, divide real reversed and pop, integer divide reversed) divide the source operand by the destination and return the quotient to the destination.

FSQRT

FSQRT (square root) replaces the content of the top stack element with its square root. (Note: the square root of −0 is defined to be −0.)

FSCALE

FSCALE (scale) interprets the value contained in ST(1) as an integer, and adds this value to the exponent of the number in ST. This is equivalent to:

$$ST \leftarrow ST \bullet 2^{ST(1)}$$

thus, FSCALE provides rapid multiplication or division by integral powers of 2. It is particularly useful for scaling the elements of a vector.

Note that FSCALE assumes the scale factor in ST(1) is an integral value in the range $-2^{15} \leqslant X < 2^{15}$. If the value is not integral, but is in-range and is greater in magnitude than 1, FSCALE uses the nearest integer smaller in magnitude, i.e., it chops the value toward 0. If the value is out of range, or $0 < |X| < 1$, the instruction will produce an undefined result and will not signal an exception. The recommended practice is to load the scale factor from a word integer to ensure correct operation.

FPREM

FPREM (partial remainder) performs modulo division of the top stack element by the next stack element, i.e., ST(1) is the modulus. FPREM produces an *exact* result; the precision exception does not occur. The sign of the remainder is the same as the sign of the original dividend.

FPREM operates by performing successive scaled subtractions; obtaining the exact remainder when the operands differ greatly in magnitude can consume large amounts of execution time. Since the 8087 can only be preempted between instructions, the remainder function could seriously increase interrupt latency in these cases. Accordingly, the instruction is designed to be executed iteratively in a software-controlled loop.

FPREM can reduce a magnitude difference of up to 2^{64} in one execution. If FPREM produces a remainder that is less than the modulus, the function is complete and bit C2 of the status word condition code is cleared. If the function is incomplete, C2 is set to 1; the result in ST is then called the partial remainder. Software can inspect C2 by storing the status word following execution of FPREM and re-execute the instruction (using the partial remainder in ST as the dividend), until C2 is cleared. Alternatively, a program can determine when the function is complete by comparing ST to ST(1). If ST>ST(1) then FPREM must be executed again; if ST=ST(1) then the remainder is 0; if ST<ST(1) then the remainder is ST. A higher priority interrupting routine which needs the 8087 can force a context switch between the instructions in the remainder loop.

An important use for FPREM is to reduce arguments (operands) of periodic transcendental functions to the range permitted by these instructions. For example, the FPTAN (tangent) instruction requires its argument to be less than $\pi/4$. Using $\pi/4$ as a modulus, FPREM will reduce an argument so that it is in range of FPTAN. Because FPREM produces an exact result, the argument reduction does *not* introduce roundoff error into the calculation, even if several iterations are required to bring the argument into range. (The rounding of π does not create the effect of a rounded argument, but of a rounded period.)

FPREM also provides the least-significant three bits of the quotient generated by FPREM (in C_3, C_1, C_0). This is also important for transcendental argument reduction since it locates the original angle in the correct one of eight $\pi/4$ segments of the unit circle.

FRNDINT

FRNDINT (round to integer) rounds the top stack element to an integer. For example, assume that ST contains the 8087 real number encoding of the decimal value 155.625. FRNDINT will change the value to 155 if the RC field of the control word is set to down or chop, or to 156 if it is set to up or nearest.

FXTRACT

FXTRACT (extract exponent and significand) "decomposes" the number in the stack top into two numbers that represent the actual value of the operand's exponent and significand fields. The "exponent" replaces the original operand on the stack and the "significand" is pushed onto the stack. Following execution of FXTRACT, ST (the new stack top) contains the value of the original significand expressed as a real number: its sign is the same as the operand's, its exponent is 0 true (16,383 or 3FFFH biased), and its significand is identical to the original operand's. ST(1) contains the value of the original operand's true (unbiased) exponent expressed as a real number. If the original operand is zero, FXTRACT produces zeros in ST and ST(1) and *both* are signed as the original operand.

To clarify the operation of FXTRACT, assume ST contains a number whose true exponent is +4 (i.e., its exponent field contains 4003H). After executing FXTRACT, ST(1) will contain the real number +4.0; its sign will be positive, its exponent field will contain 4001H (+2 true) and its significand field will contain $1_\Delta 00...00B$. In other words, the value in ST(1) will be $1.0 \times 2^2 = 4$. If ST contains an operand whose true exponent is −7 (i.e., its exponent field contains 3FF8H), then FXTRACT will return an "exponent" of −7.0; after the instruction executes, ST(1)'s sign and exponent fields will contain C001H (negative sign, true exponent of 2) and its significand will be $1_\Delta 1100...00B$. In other words the value in ST(1) will be $-1.11 \times 2^2 = -7.0$. In both cases, following FXTRACT, ST's sign and significand fields will be the same as the original operand's, and its exponent field will contain 3FFFH, (0 true).

FXTRACT is useful in conjunction with FBSTP for converting numbers in 8087 temporary real format to decimal representations (e.g., for printing or displaying). It can also be useful for debugging since it allows the exponent and significand parts of a real number to be examined separately.

FABS

FABS (absolute value) changes the top stack element to its absolute value by making its sign positive.

FCHS

FCHS (change sign) complements (reverses) the sign of the top stack element.

Comparison Instructions

Each of these instructions (table S-12) analyzes the top stack element, often in relationship to another operand, and reports the result in the status word condition code. The basic operations are compare, test (compare with zero), and examine (report tag, sign, and normalization). Special forms of the compare operation are provided to optimize algorithms by allowing direct comparisons with binary integers and real numbers in memory, as well as popping the stack after a comparison.

The FSTSW (store status word) instruction may be used following a comparison to transfer the condition code to memory for inspection. Section S.10 contains an example of using this technique to implement conditional branching.

Note that instructions other than those in the comparison group may update the condition code. To insure that the status word is not altered inadvertently, store it immediately following a comparison operation.

FCOM *//source*

FCOM (compare real) compares the stack top to the source operand. The source operand may be a register on the stack, or a short or long real memory operand. If an operand is not coded, ST is compared to ST(1). Positive and negative forms of zero compare identically as if they were unsigned. Following the instruction, the condition codes reflect the order of the operands as follows:

C3	C0	Order
0	0	ST > source
0	1	ST < source
1	0	ST = source
1	1	ST ? source

NANs and ∞ (projective) cannot be compared and return C3=C0=1 as shown above.

Table S-12. Comparison Instructions

FCOM	Compare real
FCOMP	Compare real and pop
FCOMPP	Compare real and pop twice
FICOM	Integer compare
FICOMP	Integer compare and pop
FTST	Test
FXAM	Examine

FCOMP *//source*

FCOMP (compare real and pop) operates like FCOM, and in addition pops the stack.

FCOMPP

FCOMPP (compare real and pop twice) operates like FCOM and additionally pops the stack twice, discarding both operands. The comparison is of the stack top to ST(1); no operands may be explicitly coded.

FICOM *source*

FICOM (integer compare) converts the source operand, which may reference a word or short binary integer variable, to temporary real and compares the stack top to it.

FICOMP *source*

FICOMP (integer compare and pop) operates identically to FICOM and additionally discards the value in ST by popping the stack.

FTST

FTST (test) tests the top stack element by comparing it to zero. The result is posted to the condition codes as follows:

C3	C0	Result
0	0	ST is positive and nonzero
0	1	ST is negative and nonzero
1	0	ST is zero (+ or −)
1	1	ST is not comparable (i.e., it is a NAN or projective ∞)

FXAM

FXAM (examine) reports the content of the top stack element as positive/negative and NAN/unnormal/denormal/normal/zero, or empty. Table S-13 lists and interprets all the condition code values that FXAM generates. Although four different encodings may be returned for an empty register, bits C3 and C0 of the condition code are both 1 in all encodings. Bits C2 and C1 should be ignored when examining for empty.

Transcendental Instructions

The instructions in this group (table S-14) perform the time-consuming *core calculations* for all common trigonometric, inverse trigonometric, hyperbolic, inverse hyperbolic, logarithmic and exponential functions. Prologue and epilogue software may be used to reduce arguments to the range accepted by the instructions and to adjust the result to correspond to the original arguments if necessary. The transcendentals operate on the top one or two stack elements and they return their results to the stack also.

Table S-13. FXAM Condition Code Settings

Condition Code				Interpretation
C3	C2	C1	C0	
0	0	0	0	+ Unnormal
0	0	0	1	+ NAN
0	0	1	0	− Unnormal
0	0	1	1	− NAN
0	1	0	0	+ Normal
0	1	0	1	+ ∞
0	1	1	0	− Normal
0	1	1	1	− ∞
1	0	0	0	+ 0
1	0	0	1	Empty
1	0	1	0	− 0
1	0	1	1	Empty
1	1	0	0	+ Denormal
1	1	0	1	Empty
1	1	1	0	− Denormal
1	1	1	1	Empty

Table S-14. Transcendental Instructions

FPTAN	Partial tangent
FPATAN	Partial arctangent
F2XM1	2^X-1
FYL2X	$Y \bullet \log_2 X$
FYL2XP1	$Y \bullet \log_2(X+1)$

The transcendental instructions assume that their operands are *valid and in-range*. The instruction descriptions in this section provide the range of each operation. To be considered valid, an operand to a transcendental must be normalized; denormals, unnormals, infinities and NANs are considered invalid. (Zero operands are accepted by some functions and are considered out-of-range by others.) If a transcendental operand is invalid or out-of-range, the instruction will produce an undefined result without signalling an exception. It is the programmer's responsibility to ensure that operands are valid and in-range before executing a transcendental. For periodic functions, FPREM may be used to bring a valid operand into range.

FPTAN

FPTAN (partial tangent) computes the function $Y/X = TAN(\Theta)$. Θ is taken from the top stack element; it must lie in the range $0 < \Theta < \pi/4$. The result of the operation is a ratio; Y replaces Θ in the stack and X is pushed, becoming the new stack top.

The ratio result of FPTAN and the ratio argument of FPATAN are designed to optimize the calculation of the other trigonometric functions, including SIN, COS, ARCSIN and ARCCOS. These can be derived from TAN and ARCTAN via standard trigonometric identities.

FPATAN

FPATAN (partial arctangent) computes the function $\Theta = ARCTAN(Y/X)$. X is taken from the top stack element and Y from ST(1). Y and X must observe the inequality $0 < Y < X < \infty$. The instruction pops the stack and returns Θ to the (new) stack top, overwriting the Y operand.

F2XM1

F2XM1 (2 to the X minus 1) calculates the function $Y = 2^X - 1$. X is taken from the stack top and must be in the range $0 \leqslant X \leqslant 0.5$. The result Y replaces X at the stack top.

This instruction is designed to produce a very accurate result even when X is close to zero. To obtain $Y=2^X$, add 1 to the result delivered by F2XM1.

The following formulas show how values other than 2 may be raised to a power of X:

$$10^X = 2^{X \bullet LOG_2 10}$$

$$e^X = 2^{X \bullet LOG_2 e}$$

$$y^X = 2^{X \bullet LOG_2 y}$$

As shown in the next section, the 8087 has built-in instructions for loading the constants $LOG_2 10$ and $LOG_2 e$, and the FYL2X instruction may be used to calculate $X \bullet LOG_2 Y$.

FYL2X

FYL2X (Y log base 2 of X) calculates the function $Z = Y \bullet LOG_2 X$. X is taken from the stack top and Y from ST(1). The operands must be in the ranges $0 < X < \infty$ and $-\infty < Y < +\infty$. The instruction pops the stack and returns Z at the (new) stack top, replacing the Y operand.

This function optimizes the calculation of log to any base other than two since a multiplication is always required:

$$LOG_n 2 \bullet LOG_2 X$$

FYL2XP1

FYL2XP1 (Y log base 2 of (X + 1)) calculates the function $Z = Y \bullet LOG_2 (X+1)$. X is taken from the stack top and must be in the range $0 < |X| < (1 - (\sqrt{2}/2))$. Y is taken from ST(1) and must be in the range $-\infty < Y < \infty$. FYL2XP1 pops the stack and returns Z at the (new) stack top, replacing Y.

This instruction provides improved accuracy over FYL2X when computing the log of a number very close to 1, for example $1 + \varepsilon$ where $\varepsilon << 1$. Providing ε rather than $1 + \varepsilon$ as the input to the function allows more significant digits to be retained.

Constant Instructions

Each of these instructions (table S-15) loads (pushes) a commonly-used constant onto the stack. The values have full temporary real precision (64 bits) and are accurate to approximately 19 decimal digits. Since a temporary real constant occupies 10 memory bytes, the constant instructions, which are only two bytes long, save storage and improve execution speed, in addition to simplifying programming.

Table S-15. Constant Instructions

FLDZ	Load +0.0
FLD1	Load +1.0
FLDPI	Load π
FLDL2T	Load $\log_2 10$
FLDL2E	Load $\log_2 e$
FLDLG2	Load $\log_{10} 2$
FLDLN2	Load $\log_e 2$

FLDZ

FLDZ (load zero) loads (pushes) +0.0 onto the stack.

FLD1

FLD1 (load one) loads (pushes) +1.0 onto the stack.

FLDPI

FLDPI (load π) loads (pushes) π onto the stack.

FLDL2T

FLDL2T (load log base 2 of 10) loads (pushes) the value $LOG_2 10$ onto the stack.

FLDL2E

FLDL2E (load log base 2 of e) loads (pushes) the value $LOG_2 e$ onto the stack.

FLDLG2

FLDLG2 (load log base 10 of 2) loads (pushes) the value $LOG_{10} 2$ onto the stack.

FLDLN2

FLDLN2 (load log base e of 2) loads (pushes) the value $LOG_e 2$ onto the stack.

Processor Control Instructions

Most of these instructions (table S-16) are not used in computations; they are provided principally for system-level activities. These include initialization, exception handling and task switching.

As shown in table S-16, an alternate mnemonic is available for many of the processor control instructions. This mnemonic, distinguished by a second character of "N", instructs the assembler to *not* prefix the instruction with a CPU WAIT instruction (instead, a CPU NOP precedes the instruction). This "no-wait" form is intended for use in critical code regions where a WAIT instruction might precipitate an endless wait. Thus, when CPU interrupts are disabled, and the NDP can potentially generate an interrupt, the no-wait form should be used. When CPU interrupts are enabled, as will normally be the case when an application task is running, the "wait" forms of these instructions should be used.

Except for FNSTENV and FNSAVE, all instructions which provide a no-wait mnemonic are self-synchronizing and can be executed back-to-back in any combination without intervening FWAITs. These instructions can be executed by the 8087 CU while the NEU is busy with a previously decoded instruction. To insure that the processor control instruction executes after completion of any operation in progress in the NEU, the "wait" form of that instruction should be used.

FINIT/FNINIT

FINIT/FNINIT (initialize processor) performs the functional equivalent of a hardware RESET (see section S.6), except that it does not affect the instruction fetch synchronization of the 8087 and its CPU.

For compatibility with the 8087 emulator, a system should call the INIT87 procedure in lieu of executing FINIT/FNINIT when the processor is first initialized (see section S.8 for details). Note that if FNINIT is executed while a previous 8087 memory referencing instruction is running, 8087 bus cycles in progress will be aborted.

FDISI/FNDISI

FDISI/FNDISI (disable interrupts) sets the interrupt enable mask in the control word and prevents the NDP from issuing an interrupt request.

Table S-16. Processor Control Instructions

FINIT/FNINIT	Initialize processor
FDISI/FNDISI	Disable interrupts
FENI/FNENI	Enable interrupts
FLDCW	Load control word
FSTCW/FNSTCW	Store control word
FSTSW/FNSTSW	Store status word
FCLEX/FNCLEX	Clear exceptions
FSTENV/FNSTENV	Store environment
FLDENV	Load environment
FSAVE/FNSAVE	Save state
FRSTOR	Restore state
FINCSTP	Increment stack pointer
FDECSTP	Decrement stack pointer
FFREE	Free register
FNOP	No operation
FWAIT	CPU wait

FENI/FNENI

FENI/FNENI (enable interrupts) clears the interrupt enable mask in the control word, allowing the 8087 to generate interrupt requests.

FLDCW *source*

FLDCW (load control word) replaces the current processor control word with the word defined by the source operand. This instruction is typically used to establish, or change, the 8087's mode of operation. Note that if an exception bit in the status word is set, loading a new control word that unmasks that exception and clears the interrupt enable mask will generate an immediate interrupt request before the next instruction is executed. When changing modes, the recommended procedure is to first clear any exceptions and then load the new control word.

FSTCW/FNSTCW *destination*

FSTCW/FNSTCW (store control word) writes the current processor control word to the memory location defined by the destination.

FSTSW/FNSTSW *destination*

FSTSW/FNSTCW (store status word) writes the current value of the 8087 status word to the destination operand in memory. The instruction has many uses:

- to implement conditional branching following a comparison or FPREM instruction (FSTSW);
- to poll the 8087 to determine if it is busy (FNSTSW);
- to invoke exception handlers in environments that do not use interrupts (FSTSW).

FCLEX/FNCLEX

FCLEX/FNCLEX (clear exceptions) clears all exception flags, the interrupt request flag and the busy flag in the status word. As a consequence, the 8087's INT and BUSY lines go inactive. An exception handler must issue this instruction before returning to the interrupted computation, or another interrupt request will be generated immediately, and an endless loop may result.

FSAVE/FNSAVE *destination*

FSAVE/FNSAVE (save state) writes the full 8087 state—environment plus register stack—to the memory location defined by the destination operand. Figure S-18 shows the layout of the 94-byte save area; typically the instruction will be coded to save this image on the CPU stack. If an instruction is executing in the 8087 NEU when FNSAVE is decoded, the CPU queues the FNSAVE and delays its execution until the running instruction completes normally or encounters an unmasked exception. Thus, the save image reflects the state of the NDP following the completion of any running instruction. After writing the state image to memory, FSAVE/FNSAVE initializes the 8087 as if FINIT/FNINIT had been executed.

FSAVE/FNSAVE is useful whenever a program wants to save the current state of the NDP and initialize it for a new routine. Three examples are:

- an operating system needs to perform a context switch (suspend the task that had been running and give control to a new task);
- an interrupt handler needs to use the 8087;
- an application task wants to pass a "clean" 8087 to a subroutine.

FNSAVE must be "protected" by executing it in a critical region, i.e., with CPU interrupts disabled. This prevents an interrupt handler from executing a second FNSAVE (or other "no-wait" processor control instruction that references memory) which could destroy the first FNSAVE if it is queued in the 8087. An FWAIT should be executed before CPU interrupts are enabled or any subsequent 8087 instruction is executed. (Because the FNSAVE initializes the NDP, there is no danger of the FWAIT causing an endless wait.) Other CPU instructions may be executed between the FNSAVE and the FWAIT; this parallel execution will reduce interrupt latency if the FNSAVE is queued in the 8087.

FRSTOR *source*

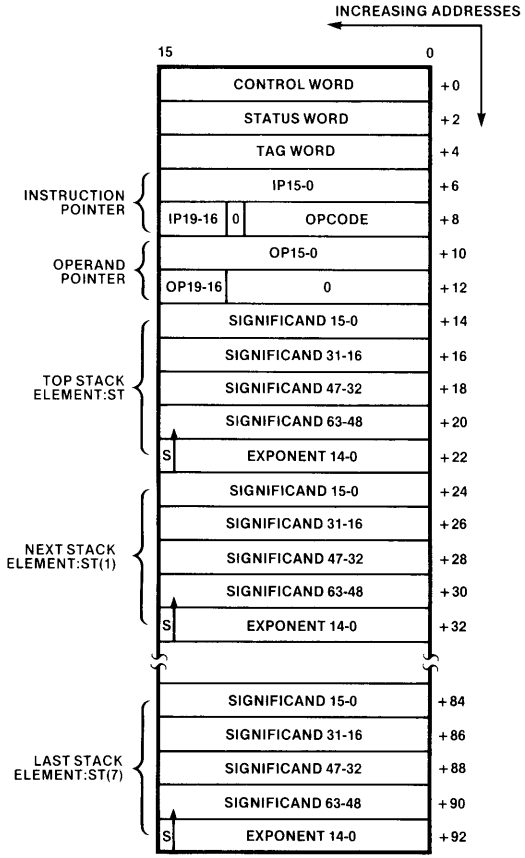
FRSTOR (restore state) reloads the 8087 from the 94-byte memory area defined by the source operand. This information should have been written by a previous FSAVE/FNSAVE instruction and not altered by any other instruction. CPU instructions (that do not reference the save image) may immediately follow FRSTOR, but no NDP instruction should be without an intervening FWAIT or an assembler-generated WAIT.

Note that the 8087 "reacts" to its new state at the conclusion of the FRSTOR; it will for example, generate an immediate interrupt request if the exception and mask bits in the memory image so indicate.

FSTENV/FNSTENV *destination*

FSTENV/FNSTENV (store environment) writes the 8087's basic status—control, status and tag words, and exception pointers—to the memory location defined by the destination operand. Typically the environment is saved on the CPU stack. FSTENV/FNSTENV is often used by

exception handlers because it provides access to the exception pointers which identify the offending instruction and operand. After saving the environment, FSTENV/FNSTENV sets all exception masks in the processor; it does not affect the interrupt-enable mask. Figure S-19 shows the format of the environment data in memory. If FNSTENV is decoded while another instruction is executing concurrently in the NEU, the 8087 queues the FNSTENV and does not store the environment until the other instruction has completed. Thus, the data saved by the instruction reflects the 8087 after any previously decoded instruction has been executed.



NOTES:
S = Sign
Bit 0 of each field is rightmost, least significant bit of corresponding register field.
Bit 63 of significand is integer bit (assumed binary point is immediately to the right).

Figure S-18. FSAVE/FRSTOR Memory Layout

FSTENV/FNSTENV must be allowed to complete before any other 8087 instruction is decoded. When FSTENV is coded, an explicit FWAIT, or assembler-generated WAIT, should precede any subsequent 8087 instruction. An FNSTENV must be executed in a critical region that is protected from interruption, in the same manner as FNSAVE. (There is no risk of the following FWAIT causing an endless wait, because FNSTENV masks all exceptions, thereby preventing an interrupt request from the 8087.)

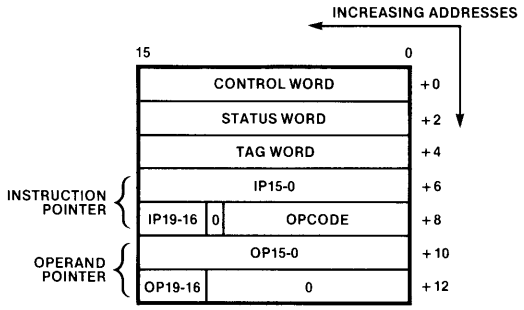


Figure S-19. FSTENV/FLDENV Memory Layout

FLDENV *source*

FLDENV (load environment) reloads the 8087 environment from the memory area defined by the source operand. This data should have been written by a previous FSTENV/FNSTENV instruction. CPU instructions (that do not reference the environment image) may immediately follow FLDENV, but no subsequent NDP instruction should be executed without an intervening FWAIT or assembler-generated WAIT.

Note that loading an environment image that contains an unmasked exception will cause an immediate interrupt request from the 8087 (assuming IEM=0 in the environment image).

FINCSTP

FINCSTP (increment stack pointer) adds 1 to the stack top pointer (ST) in the status word. It does not alter tags or register contents, nor does it transfer data. It is not equivalent to popping the stack since it does not set the tag of the previous stack top to empty. Incrementing the stack pointer when ST=7 produces ST=0.

FDECSTP

FDECSTP (decrement stack pointer) subtracts 1 from ST, the stack top pointer in the status word. No tags or registers are altered, nor is any data transferred. Executing FDECSTP when ST=0 produces ST=7.

FFREE *destination*

FFREE (free register) changes the destination register's tag to empty; the content of the register is unaffected.

FNOP

FNOP (no operation) stores the stack top to the stack top (FST ST,ST(0)) and thus effectively performs no operation.

FWAIT (CPU instruction)

FWAIT is not actually an 8087 instruction, but an alternate mnemonic for the CPU WAIT instruction described in section 2.8. The FWAIT mnemonic should be coded whenever the programmer wants to synchronize the CPU to the NDP, that is, to suspend further instruction decoding until the NDP has completed the current instruction. *A CPU instruction should not attempt to access a memory operand that has been read or written by a previous 8087 instruction until the 8087 instruction has completed*. The following coding shows how FWAIT can be used to force the CPU instruction to wait for the 8087:

```
FNSTSW   STATUS
FWAIT    ;Wait for FNSTSW
MOV      AX,STATUS
```

Programmers should not code WAIT to synchronize the CPU and the NDP. The routines that alter an object program for 8087 emulation eliminate FWAITs (and assembler-generated WAITs) but do not change any explicitly coded WAITs. The program will wait forever if a WAIT is encountered in emulated execution, since there is no 8087 to drive the CPU's $\overline{\text{TEST}}$ pin active.

Instruction Set Reference Information

Table S-19 lists the operating characteristics of all the 8087 instructions. There is one table entry for each instruction mnemonic; the entries are in alphabetical order for quick lookup. Each entry provides the general operand forms accepted by the instruction as well as a list of all exceptions that may be detected during the operation.

There is one entry for each combination of operand types that can be coded with the mnemonic. Table S-17 explains the operand identifiers allowed in table S-19. Following this entry are columns that provide execution time in clocks, the number of bus transfers run during the operation, the length of the instruction in bytes, and an ASM-86 coding sample.

Table S-17. Key to Operand Types

Identifier	Explanation
ST	Stack top; the register currently at the top of the stack.
ST(i)	A register in the stack i ($0 \leq i \leq 7$) stack elements from the top. ST(1) is the next-on-stack register, ST(2) is below ST(1), etc.
Short-real	A short real (32 bits) number in memory.
Long-real	A long real (64 bits) number in memory.
Temp-real	A temporary real (80 bits) number in memory.
Packed-decimal	A packed decimal integer (18 digits, 10 bytes) in memory.
Word-integer	A word binary integer (16 bits) in memory.
Short-integer	A short binary integer (32 bits) in memory.
Long-integer	A long binary integer (64 bits) in memory.
nn-bytes	A memory area nn bytes long.

Execution Time

The execution of an 8087 instruction involves three principal activities, each of which may contribute to the total duration (execution time) of the operation:

- Instruction fetch
- Instruction execution
- Operand transfer

The CPU and NDP simultaneously prefetch and queue their common instruction stream from memory. This activity is performed during spare bus cycles and proceeds in parallel with the execution of instructions from the queue. Because of their complexity, 8087 instructions typically take much longer to execute than to fetch. This means that in a typical sequence of 8087 instructions the processors have a relatively large amount of time available to maintain full instruction queues. Instruction fetching is therefore fully overlapped with execution and does not contribute to the overall duration of a series of instructions. Fetch time does become apparent when a CPU jump or call instruction alters the normal sequential execution. This empties the queues and delays execution of the target instruction until it is fetched from memory. The time required to fetch the instruction depends on its length, the type of CPU, and, if the CPU is an 8086, whether the instruction is located at an even or odd address. (Slow memories, which force the insertion of wait states in bus cycles, and the bus activities of other processors in the system, may also lengthen fetch time.) Section 2.7 covers this topic in more detail.

Table S-19 quotes a typical execution time and a range for each instruction. Dividing the figures in the table by 5 (assuming a 5 MHz clock) produces execution time in microseconds. The typical case is an estimate for operand values that normally characterize most applications. The range encompasses best- and worst-case operand values that may be found in extreme circumstances. Where applicable, the figures *include* all overhead incurred by the CPU's execution of the ESC instruction, local bus arbitration (request/grant time), and the average overhead imposed by a preceding WAIT instruction (half of the 5-clock cycle that it uses to examine the $\overline{\text{TEST}}$ pin).

The execution times assume that no exceptions are detected. Invalid operation, denormalized (unmasked), and zerodivide exceptions usually decrease execution time from the typical figure, but it will still fall within the quoted range. The precision exception has no effect on execution time. Unmasked overflow and underflow, and masked denormalized exceptions, impose the penalties shown in table S-18. Absolute worst-case execution time is therefore the high range figure plus the largest penalty that may be encountered.

For instructions that transfer operands to or from memory, the execution times in table S-19 show that the time required for the CPU to calculate the operand's effective address (EA) should be added. Effective address calculation time varies according to addressing mode; table 2-20 supplies the figures.

Table S-18. Execution Penalties

Exception	Additional Clocks
Overflow (unmasked)	14
Underflow (unmasked)	16
Denormalized (masked)	33

Bus Transfers

Instructions that reference memory execute bus cycles to transfer operands. Each transfer requires one bus cycle. The number of transfers depends on the length of the operand, the type of CPU, and the alignment of the operand if the CPU is an 8086. The figures in table S-19 *include* the "dummy read" transfer(s) performed by the CPU in its execution of the escape instruction that corresponds to the 8087 instruction. The first 8086 figure is for even-addressed operands, and the second is for odd-addressed operands.

A bus cycle (transfer) consumes four clocks if the bus is immediately available and if the memory is running at processor speed, without wait states. Additional time is required if slow memories are employed, because these insert wait states into the bus cycle. In multiprocessor environments, the bus may not be available immediately if a higher priority processor is using it; this also can increase effective transfer time.

Instruction Length

Instructions that do not reference memory are two bytes long. Memory reference instructions vary between two and four bytes. The third and fourth bytes are used for 8- or 16-bit displacement values; the assembler generates the short displacement whenever possible. No displacements are required in memory references that use only CPU register contents to calculate an operand's effective address.

Note that the lengths quoted in table S-19 do not include the one byte CPU WAIT instruction that the assembler automatically inserts in front of all NDP instructions (except those coded with a "no-wait" mnemonic).

Table S-19. Instruction Set Reference Data

FABS

FABS (no operands)
Absolute value

Exceptions: I

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	14	10-17	0	0	2	FABS

FADD

FADD //source/destination,source
Add real

Exceptions: I, D, O, U, P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
//ST,ST(i)/ST(i),ST	85	70-100	0	0	2	FADD ST,ST(4)
short-real	105+EA	90-120+EA	2/4	4	2-4	FADD AIR_TEMP [SI]
long-real	110+EA	95-125+EA	4/6	8	2-4	FADD [BX].MEAN

FADDP

FADDP destination,source
Add real and pop

Exceptions: I, D, O, U, P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
ST(i),ST	90	75-105	0	0	2	FADDP ST(2),ST

FBLD

FBLD source
Packed decimal (BCD) load

Exceptions: I

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
packed-decimal	300+EA	290-310+EA	5/7	10	2-4	FBLD YTD_SALES

Table S-19. Instruction Set Reference Data (Cont'd.)

FBSTP

FBSTP destination
Packed decimal (BCD) store and pop

Exceptions: I

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
packed-decimal	530+EA	520-540+EA	6/8	12	2-4	FBSTP [BX].FORECAST

FCHS

FCHS (no operands)
Change sign

Exceptions: I

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	15	10-17	0	0	2	FCHS

FCLEX/FNCLEX

FCLEX (no operands)
Clear exceptions

Exceptions: None

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	5	2-8	0	0	2	FNCLEX

FCOM

FCOM //source
Compare real

Exceptions: I, D

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
//ST(i)	45	40-50	0	0	2	FCOM ST(1)
short-real	65+EA	60-70+EA	2/4	4	2-4	FCOM [BP].UPPER_LIMIT
long-real	70+EA	65-75+EA	4/6	8	2-4	FCOM WAVELENGTH

FCOMP

FCOMP //source
Compare real and pop

Exceptions: I, D

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
//ST(i)	47	42-52	0	0	2	FCOMP ST(2)
short-real	68+EA	63-73+EA	2/4	4	2-4	FCOMP [BP+2].N_READINGS
long-real	72+EA	67-77+EA	4/6	8	2-4	FCOMP DENSITY

Table S-19. Instruction Set Reference Data (Cont'd.)

FCOMPP

FCOMPP (no operands)
Compare real and pop twice

Exceptions: I, D

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	50	45-55	0	0	2	FCOMPP

FDECSTP

FDECSTP (no operands)
Decrement stack pointer

Exceptions: None

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	9	6-12	0	0	2	FDECSTP

FDISI/FNDISI

FDISI (no operands)
Disable interrupts

Exceptions: None

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	5	2-8	0	0	2	FDISI

FDIV

FDIV //source/destination,source
Divide real

Exceptions: I, D, Z, O, U, P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
//ST(i),ST	198	193-203	0	0	2	FDIV
short-real	220+EA	215-225+EA	2/4	4	2-4	FDIV DISTANCE
long-real	225+EA	220-230+EA	4/6	8	2-4	FDIV ARC [DI]

FDIVP

FDIVP destination,source
Divide real and pop

Exceptions: I, D, Z, O, U, P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
ST(i),ST	202	197-207	0	0	2	FDIVP ST(4),ST

Table S-19. Instruction Set Reference Data (Cont'd.)

FDIVR

FDIVR //source/destination,source
Divide real reversed

Exceptions: I, D, Z, O, U, P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
//ST,ST(i)/ST(i),ST	199	194-204	0	0	2	FDIVR ST(2),ST
short-real	221+EA	216-226+EA	2/4	6	2-4	FDIVR [BX].PULSE_RATE
long-real	226+EA	221-231+EA	4/6	8	2-4	FDIVR RECORDER.FREQUENCY

FDIVRP

FDIVRP destination,source
Divide real reversed and pop

Exceptions: I, D, Z, O, U, P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
ST(i),ST	203	198-208	0	0	2	FDIVRP ST(1),ST

FENI/FNENI

FENI (no operands)
Enable interrupts

Exceptions: None

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	5	2-8	0	0	2	FNENI

FFREE

FFREE destination
Free register

Exceptions: None

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
ST(i)	11	9-16	0	0	2	FFREE ST(1)

FIADD

FIADD source
Integer add

Exceptions: I, D, O, P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
word-integer	120+EA	102-137+EA	1/2	2	2-4	FIADD DISTANCE_TRAVELLED
short-integer	125+EA	108-143+EA	2/4	4	2-4	FIADD PULSE_COUNT [SI]

Table S-19. Instruction Set Reference Data (Cont'd.)

FICOM

FICOM source
Integer compare

Exceptions: I, D

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
word-integer	80+EA	72-86+EA	1/2	2	2-4	FICOM TOOL.N_PASSES
short-integer	85+EA	78-91+EA	2/4	4	2-4	FICOM [BP+4].PARM_COUNT

FICOMP

FICOMP source
Integer compare and pop

Exceptions: I, D

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
word-integer	82+EA	74-88+EA	1/2	2	2-4	FICOMP [BP].LIMIT [SI]
short-integer	87+EA	80-93+EA	2/4	4	2-4	FICOMP N_SAMPLES

FIDIV

FIDIV source
Integer divide

Exceptions: I, D, Z, O, U, P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
word-integer	230+EA	224-238+EA	1/2	2	2-4	FIDIV SURVEY.OBSERVATIONS
short-integer	236+EA	230-243+EA	2/4	4	2-4	FIDIV RELATIVE_ANGLE [DI]

FIDIVR

FIDIVR source
Integer divide reversed

Exceptions: I, D, Z, O, U, P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
word-integer	230+EA	225-239+EA	1/2	2	2-4	FIDIVR [BP].X_COORD
short-integer	237+EA	231-245+EA	2/4	4	2-4	FIDIVR FREQUENCY

FILD

FILD source
Integer load

Exception: I

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
word-integer	50+EA	46-54+EA	1/2	2	2-4	FILD [BX].SEQUENCE
short-integer	56+EA	52-60+EA	2/4	4	2-4	FILD STANDOFF [DI]
long-integer	64+EA	60-68+EA	4/6	8	2-4	FILD RESPONSE.COUNT

Table S-19. Instruction Set Reference Data (Cont'd.)

FIMUL

FIMUL source
Integer multiply

Exceptions: I, D, O, P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
word-integer	130+EA	124-138+EA	1/2	2	2-4	FIMUL BEARING
short-integer	136+EA	130-144+EA	2/4	4	2-4	FIMUL POSITION.Z_AXIS

FINCSTP

FINCSTP (no operands)
Increment stack pointer

Exceptions: None

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	9	6-12	0	0	2	FINCSTP

FINIT/FNINIT

FINIT (no operands)
Initialize processor

Exceptions: None

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	5	2-8	0	0	2	FINIT

FIST

FIST destination
Integer store

Exceptions: I, P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
word-integer	86+EA	80-90+EA	2/4	4	2-4	FIST OBS.COUNT [SI]
short-integer	88+EA	82-92+EA	3/5	6	2-4	FIST [BP].FACTORED_PULSES

FISTP

FISTP destination
Integer store and pop

Exceptions: I, P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
word-integer	88+EA	82-92+EA	2/4	4	2-4	FISTP [BX].ALPHA_COUNT [SI]
short-integer	90+EA	84-94+EA	3/5	6	2-4	FISTP CORRECTED_TIME
long-integer	100+EA	94-105+EA	5/7	10	2-4	FISTP PANEL.N_READINGS

Table S-19. Instruction Set Reference Data (Cont'd.)

FISUB

FISUB source
Integer subtract

Exceptions: I, D, O, P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
word-integer	120+EA	102-137+EA	1/2	2	2-4	FISUB BASE_FREQUENCY
short-integer	125+EA	108-143+EA	2/4	4	2-4	FISUB TRAIN_SIZE [DI]

FISUBR

FISUBR source
Integer subtract reversed

Exceptions: I, D, O, P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
word-integer	120+EA	103-139+EA	1/2	2	2-4	FISUBR FLOOR [BX] [SI]
short-integer	125+EA	109-144+EA	2/4	4	2-4	FISUBR BALANCE

FLD

FLD source
Load real

Exceptions: I, D

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
ST(i)	20	17-22	0	0	2	FLD ST(0)
short-real	43+EA	38-56+EA	2/4	4	2-4	FLD READING [SI].PRESSURE
long-real	46+EA	40-60+EA	4/6	8	2-4	FLD [BP].TEMPERATURE
temp-real	57+EA	53-65+EA	5/7	10	2-4	FLD SAVEREADING

FLDCW

FLDCW source
Load control word

Exceptions: None

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
2-bytes	10+EA	7-14+EA	1/2	2	2-4	FLDCW CONTROL_WORD

FLDENV

FLDENV source
Load environment

Exceptions: None

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
14-bytes	40+EA	35-45+EA	7/9	14	2-4	FLDENV [BP+6]

Table S-19. Instruction Set Reference Data (Cont'd.)

FLDLG2

FLDLG2 (no operands)
Load $\log_{10} 2$

Exceptions: I

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	21	18-24	0	0	2	FLDLG2

FLDLN2

FLDLN2 (no operands)
Load $\log_e 2$

Exceptions: I

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	20	17-23	0	0	2	FLDLN2

FLDL2E

FLDL2E (no operands)
Load $\log_2 e$

Exceptions: I

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	18	15-21	0	0	2	FLDL2E

FLDL2T

FLDL2T (no operands)
Load $\log_2 10$

Exceptions: I

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	19	16-22	0	0	2	FLDL2T

FLDPI

FLDPI (no operands)
Load π

Exceptions: I

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	19	16-22	0	0	2	FLDPI

Table S-19. Instruction Set Reference Data (Cont'd.)

FLDZ

FLDZ (no operands)
Load +0.0

Exceptions: I

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	14	11-17	0	0	2	FLDZ

FLD1

FLD1 (no operands)
Load +1.0

Exceptions: I

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	18	15-21	0	0	2	FLD1

FMUL

FMUL //source/destination,source
Multiply real

Exceptions: I, D, O, U, P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
//ST(i),ST/ST,ST(i)[1]	97	90-105	0	0	2	FMUL ST,ST(3)
//ST(i),ST/ST,ST(i)	138	130-145	0	0	2	FMUL ST,ST(3)
short-real	118+EA	110-125+EA	2/4	4	2-4	FMUL SPEED_FACTOR
long-real[1]	120+EA	112-126+EA	4/6	8	2-4	FMUL [BP].HEIGHT
long-real	161+EA	154-168+EA	4/6	8	2-4	FMUL [BP].HEIGHT

[1] occurs when one or both operands is "short"—it has 40 trailing zeros in its fraction (e.g., it was loaded from a short-real memory operand).

FMULP

FMULP destination,source
Multiply real and pop

Exceptions: I, D, O, U, P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
ST(i),ST[1]	100	94-108	0	0	2	FMULP ST(1),ST
ST(i),ST	142	134-148	0	0	2	FMULP ST(1),ST

[1] occurs when one or both operands is "short"—it has 40 trailing zeros in its fraction (e.g., it was loaded from a short-real memory operand).

Table S-19. Instruction Set Reference Data (Cont'd.)

FNOP

FNOP (no operands)
No operation

Exceptions: None

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	13	10-16	0	0	2	FNOP

FPATAN

FPATAN (no operands)
Partial arctangent

Exceptions: U, P (operands not checked)

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	650	250-800	0	0	2	FPATAN

FPREM

FPREM (no operands)
Partial remainder

Exceptions: I, D, U

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	125	15-190	0	0	2	FPREM

FPTAN

FPTAN (no operands)
Partial tangent

Exceptions: I, P (operands not checked)

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	450	30-540	0	0	2	FPTAN

FRNDINT

FRNDINT (no operands)
Round to integer

Exceptions: I, P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	45	16-50	0	0	2	FRNDINT

Table S-19. Instruction Set Reference Data (Cont'd.)

FRSTOR

FRSTOR source
Restore saved state

Exceptions: None

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
94-bytes	210+EA	205-215+EA	47/49	96	2-4	FRSTOR [BP]

FSAVE/FNSAVE

FSAVE destination
Save state

Exceptions: None

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
94-bytes	210+EA	205-215+EA	48/50	94	2-4	FSAVE [BP]

FSCALE

FSCALE (no operands)
Scale

Exceptions: I, O, U

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	35	32-38	0	0	2	FSCALE

FSQRT

FSQRT (no operands)
Square root

Exceptions: I, D, P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	183	180-186	0	0	2	FSQRT

FST

FST destination
Store real

Exceptions: I, O, U, P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
ST(i)	18	15-22	0	0	2	FST ST(3)
short-real	87+EA	84-90+EA	3/5	6	2-4	FST CORRELATION [DI]
long-real	100+EA	96-104+EA	5/7	10	2-4	FST MEAN_READING

Table S-19. Instruction Set Reference Data (Cont'd.)

FSTCW/FNSTCW

FSTCW destination
Store control word

Exceptions: None

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
2-bytes	15+EA	12-18+EA	2/4	4	2-4	FSTCW SAVE_CONTROL

FSTENV/FNSTENV

FSTENV destination
Store environment

Exceptions: None

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
14-bytes	45+EA	40-50+EA	8/10	16	2-4	FSTENV [BP]

FSTP

FSTP destination
Store real and pop

Exceptions: I, O, U, P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
ST(i)	20	17-24	0	0	2	FSTP ST(2)
short-real	89+EA	86-92+EA	3/5	6	2-4	FSTP [BX].ADJUSTED_RPM
long-real	102+EA	98-106+EA	5/7	10	2-4	FSTP TOTAL_DOSAGE
temp-real	55+EA	52-58+EA	6/8	12	2-4	FSTP REG_SAVE [SI]

FSTSW/FNSTSW

FSTSW destination
Store status word

Exceptions: None

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
2-bytes	15+EA	12-18+EA	2/4	4	2-4	FSTSW SAVE_STATUS

FSUB

FSUB //source/destination,source
Subtract real

Exceptions: I,D,O,U,P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
//ST,ST(i)/ST(i),ST	85	70-100	0	0	2	FSUB ST,ST(2)
short-real	105+EA	90-120+EA	2/4	4	2-4	FSUB BASE_VALUE
long-real	110+EA	95-125+EA	4/6	8	2-4	FSUB COORDINATE.X

Table S-19. Instruction Set Reference Data (Cont'd.)

FSUBP

FSUBP destination,source
Subtract real and pop

Exceptions: I,D,O,U,P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
ST(i),ST	90	75-105	0	0	2	FSUBP ST(2),ST

FSUBR

FSUBR //source/destination,source
Subtract real reversed

Exceptions: I,D,O,U,P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
//ST,ST(i)/ST(i),ST	87	70-100	0	0	2	FSUBR ST,ST(1)
short-real	105+EA	90-120+EA	2/4	4	2-4	FSUBR VECTOR[SI]
long-real	110+EA	95-125+EA	4/6	8	2-4	FSUBR [BX].INDEX

FSUBRP

FSUBRP destination,source
Subtract real reversed and pop

Exceptions: I,D,O,U,P

Operands	Executon Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
ST(i),ST	90	75-105	0	0	2	FSUBRP ST(1),ST

FTST

FTST (no operands)
Test stack top against +0.0

Exceptions: I, D

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	42	38-48	0	0	2	FTST

FWAIT

FWAIT (no operands)
(CPU) Wait while 8087 is busy

Exceptions: None (CPU instruction)

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	3+5n*	3+5n*	0	0	1	FWAIT

*n = number of times CPU examines $\overline{TEST}$ line before 8087 lowers BUSY.

Table S-19. Instruction Set Reference Data (Cont'd.)

FXAM

FXAM (no operands)
Examine stack top

Exceptions: None

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	17	12-23	0	0	2	FXAM

FXCH

FXCH //destination
Exchange registers

Exceptions: I

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
//ST(i)	12	10-15	0	0	2	FXCH ST(2)

FXTRACT

FXTRACT (no operands)
Extract exponent and significand

Exceptions: I

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	50	27-55	0	0	2	FXTRACT

FYL2X

FYL2X (no operands)
$Y \bullet \log_2 X$

Exceptions: P (operands not checked)

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	950	900-1100	0	0	2	FYL2X

FYL2XP1

FYL2XP1 (no operands)
$Y \bullet \log_2 (X+1)$

Exceptions: P (operands not checked)

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	850	700-1000	0	0	2	FYL2XP1

Table S-19. Instruction Set Reference Data (Cont'd.)

F2XM1	F2XM1 (no operands) 2^X-1				Exceptions: U, P (operands not checked)	
Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	500	310-630	0	0	2	F2XM1

S.8 Programming Facilities

Writing programs for the 8087 is a natural extension of the process described in section 2.9, just as the NDP itself is an extension to the CPU. This section describes how PL/M-86 and ASM-86 programmers work with the 8087 in these languages. It also covers the 8087 software emulators provided for both translators.

The level of detail in this section is intended to give programmers a basic understanding of the software tools that can be used with the 8087, but this information is not sufficient to document the full capabilities of these facilities. The definitive description of ASM-86 and the full 8087 emulator is provided in *MCS-86 Assembly Language Reference Manual*, Order No. 9800640, and *MCS-86 Assembler Operating Instructions for ISIS-II Users*, Order No. 9800641. PL/M-86 and the partial emulator are documented in *PL/M-86 Programming Manual*, Order No. 9800466 and *ISIS-II PL/M-86 Compiler Operator's Manual*, Order No. 9800478. These publications may be ordered from Intel's Literature Department.

Readers should be familiar with section 2.9 of the *8086 Family User's Manual* in order to benefit from the material in this section.

PL/M-86

High level language programmers can access a useful subset of the 8087's (real or emulated) capabilities. The PL/M-86 REAL data type corresponds to the NDP's short real (32-bit) format. This data type provides a range of about $8.43*10^{-37} \leq |X| \leq 3.38*10^{38}$, with about seven significant decimal digits. This representation is adequate for the data manipulated by many microcomputer applications.

The utility of the REAL data type is extended by the PL/M-86 compiler's practice of holding intermediate results in the 8087's temporary real format. This means that the full range and precision of the processor may be utilized for intermediate results. Underflow, overflow, and rounding errors are most likely to occur during intermediate computations rather than during calculation of an expression's final result. Holding intermediate results in temporary real format greatly reduces the likelihood of overflow and underflow and eliminates roundoff as a serious source of error until the final assignment of the result is performed.

The compiler generates 8087 code to evaluate expressions that contain REAL data types, whether variables or constants or both. This means that addition, subtraction, multiplication, division, comparison, and assignment of REALs will be performed by the NDP. INTEGER expressions, on the other hand, are evaluated on the CPU.

Five built-in procedures (table S-20) give the PL/M-86 programmer access to 8087 functions manipulated by the processor control instructions. Prior to any arithmetic operations, a typical PL/M-86 program will setup the NDP after power up using the INIT$REAL$MATH $UNIT procedure and then issue SET$REAL$MODE to configure the NDP. SET$REAL$MODE loads the 8087 control word, and its 16-bit parameter has the format shown in figure S-7. The recommended value of this parameter is 033EH (projective closure, round to nearest, 64-bit precision, interrupts enabled, all exceptions masked except invalid operation). Other settings may be used at the programmer's discretion.

Table S-20. PL/M-86 Built-In Procedures

Procedure	8087 Instruction	Description
INIT$REAL$MATH$UNIT[1]	FINIT	Initialize processor.
SET$REAL$MODE	FLDCW	Set exception masks, rounding precision, and infinity controls.
GET$REAL$ERROR[2]	FNSTSW & FNCLEX	Store, then clear, exception flags.
SAVE$REAL$STATUS	FNSAVE	Save processor state.
RESTORE$REAL$STATUS	FRSTOR	Restore processor state.

[1]Also initializes interrupt pointers for emulation.

[2]Returns low-order byte of status word.

If any exceptions are unmasked, an exception handler must be provided in the form of an interrupt procedure that is designated to be invoked by CPU interrupt pointer (vector) number 16. The exception handler can use the GET$REAL $ERROR procedure to obtain the low-order byte of the 8087 status word and to then clear the exception flags. The byte returned by GET$REAL$ERROR contains the exception flags; these can be examined to determine the source of the exception.

The SAVE$REAL$STATUS and RESTORE $REAL$STATUS procedures are provided for multi-tasking environments where a running task that uses the 8087 may be preempted by another task that also uses the 8087. It is the responsibility of the preempting task to issue SAVE$REAL$STATUS before it executes any statements that affect the 8087; these include the INIT$REAL$MATH$UNIT and SET$REAL $MODE procedures as well as arithmetic expressions. SAVE$REAL$STATUS saves the 8087 state (registers, status, and control words, etc.) on the CPU's stack. RESTORE$REAL$STATUS reloads the state information; the preempting task must invoke this procedure before terminating in order to restore the 8087 to its state at the time the running task was preempted. This enables the preempted task to resume execution from the point of its preemption.

Note that the PL/M-86 compiler prefixes *every* 8087 instruction with a CPU WAIT. Therefore, programmers should not code PL/M-86 statements that generate 8087 instructions if the NDP can request an interrupt and that interrupt is blocked (this may result in the endless wait condition described in section S.6.)

ASM-86

The ASM-86 assembly language provides a single uniform set of facilities for all combinations of the 8086/8088/8087 processors. Assembly language programs can be written to be completely independent of the processor set on which they are destined to execute. This means that a program written originally for an 8088 alone will execute on an 8086/8087 combination without re-assembling. The programmer's view of the hardware is a single machine with these resources:

- 160 instructions
- 12 data types
- 8 general registers
- 4 segment registers
- 8 floating-point registers, organized as a stack

The combination of the assembly language and the 8087 emulator decouple the source code from the execution vehicle. For example, the assembler automatically inserts CPU WAIT instructions in front of those 8087 instructions that require them. If the program actually runs with the emulator rather than the 8087, the WAITs are automatically removed at link time (since there is no NDP for which to wait).

Defining Data

The ASM-86 directives shown in table S-21 allocate storage for 8087 variables and constants. As with other storage allocation directives, the assembler associates a type with any variable defined with these directives. The type value is equal to the length of the storage unit in bytes (10 for DT, 8 for DQ, etc.). The assembler checks the type of any variable coded in an instruction to be certain that it is compatible with the instruction. For example, the coding FIADD ALPHA will be flagged as an error if ALPHA's type is not 2 or 4, because integer addition is only available for word and short integer data types. The operand's type also tells the assembler which machine instruction to produce; although to the programmer there is only an FIADD instruction, a different machine instruction is required for each operand type.

On occasion it is desirable to use an instruction with an operand that has no declared type. For example, if register BX points to a short integer variable, a programmer may want to code FIADD [BX]. This can be done by informing the assembler of the operand's type in the instruction, coding FIADD DWORD PTR [BX]. The corresponding overrides for the other storage allocations are WORD PTR, QWORD PTR, and TBYTE PTR.

The assembler does not, however, check the types of operands used in processor control instructions. Coding FRSTOR [BP] implies that the programmer has set up register BP to point to the stack location where the processor's 94-byte state record has been previously saved.

The initial values for 8087 constants may be coded in several different ways. Binary integer constants may be specified as bit strings, decimal integers, octal integers, or hexadecimal strings. Packed decimal values are normally written as decimal integers, although the assembler will accept and convert other representations of integers. Real values may be written as ordinary decimal real numbers (decimal point required), as decimal numbers in scientific notation, or as hexadecimal strings. Using hexadecimal strings is primarily intended for defining special values such as infinities, NANs, and nonnormalized numbers. Most programmers will find that ordinary decimal and scientific decimal provide the simplest way to initialize 8087 constants. Figure S-20 compares several ways of setting the various 8087 data types to the same initial value.

Note that preceding 8087 variables and constants with the ASM-86 EVEN directive ensures that the operands will be word-aligned in memory. This will produce the best performance in 8086-based systems, and is good practice even for 8088 software, in the event that the programs are transferred to an 8086. All 8087 data types occupy integral numbers of words so that no storage is "wasted" if blocks of variables are defined together and preceded by a single EVEN declarative.

Records and Structures

The ASM-86 RECORD and STRUC (structure) declaratives can be very useful in NDP programming. The record facility can be used to define the bit fields of the control, status, and tag words. Figure S-21 shows one definition of the status word and how it might be used in a routine that polls the 8087 until it has completed an instruction.

Because structures allow different but related data types to be grouped together, they often provide a natural way to represent "real world" data organizations. The fact that the structure template may be "moved" about in memory adds to its flexibility. Figure S-22 shows a simple struc-

Table S-21. 8087 Storage Allocation Directives

Directive	Interpretation	8087 Data Types
DW	Define Word	Word integer
DD	Define Doubleword	Short integer, short real
DQ	Define Quadword	Long integer, long real
DT	Define Tenbyte	Packed decimal, temporary real

```
; THE FOLLOWING ALL ALLOCATE THE CONSTANT: -126
; NOTE TWO'S COMPLEMENT STORAGE OF NEGATIVE BINARY INTEGERS.
;
; EVEN                                  ; FORCE WORD ALIGNMENT
WORD_INTEGER    DW  1111111110000010B ; BIT STRING
SHORT_INTEGER   DD  0FFFFFF82H        ; HEX STRING MUST START WITH DIGIT
LONG_INTEGER    DQ  -126              ; ORDINARY DECIMAL
SHORT_REAL      DD  -126.0            ; NOTE PRESENCE OF '.'
LONG_REAL       DD  -1.26E2           ; ''SCIENTIFIC''
PACKED_DECIMAL DT  -126               ; ORDINARY DECIMAL INTEGER
; IN THE FOLLOWING, SIGN AND EXPONENT IS 'C005',
;     SIGNIFICAND IS '7E00...00', 'R' INFORMS ASSEMBLER THAT
;     THE STRING REPRESENTS A REAL DATA TYPE.
;
TEMP_REAL       DT  0C0057E00000000000000R  ; HEX STRING
```

Figure S-20. Sample 8087 Constants

```
; RESERVE SPACE FOR STATUS WORD
STATUS_WORD                 DW ?
; LAY OUT STATUS WORD FIELDS
STATUS RECORD
&     BUSY:                 1,
&     COND_CODE3:           1,
&     STACK_TOP:            3,
&     COND_CODE2:           1,
&     COND_CODE1:           1,
&     COND_CODE0:           1,
&     INT_REQ:              1,
&     RESERVED:             1,
&     P_FLAG:               1,
&     U_FLAG:               1,
&     O_FLAG:               1,
&     Z_FLAG:               1,
&     D_FLAG:               1,
&     I_FLAG:               1
; POLL STATUS WORD UNTIL 8087 IS NOT BUSY
POLL:   FNSTSW  STATUS_WORD
        TEST    STATUS_WORD, MASK BUSY
        JNZ     POLL
```

Figure S-21. Status Word RECORD Definition

```
SAMPLE      STRUC

   N_OBS            DD     ?    ;SHORT INTEGER
   MEAN             DQ     ?    ;LONG REAL
   MODE             DW     ?    ;WORD INTEGER
   STD_DEV          DQ     ?    ;LONG REAL
   ;ARRAY OF OBSERVATIONS -- WORD INTEGER
   TEST_SCORES      DW     1000 DUP (?)
SAMPLE ENDS
```

Figure S-22. Structure Definition

ture that might be used to represent data consisting of a series of test score samples. A structure could also be used to define the organization of the information stored and loaded by the FSTENV and FLDENV instructions.

Addressing Modes

8087 memory data can be accessed with any of the CPU's five memory addressing modes. This means that 8087 data types can be incorporated in data aggregates ranging from simple to complex according to the needs of the application. The addressing modes, and the ASM-86 notation used to specify them in instructions, make the accessing of structures, arrays, arrays of structures, and other organizations direct and straightforward. Table S-22 gives several examples of 8087 instructions coded with operands that illustrate different addressing modes.

8087 Emulators

Intel offers two software products that provide the functional equivalent of an 8087, implemented in 8086/8088 software. The full emulator (E8087) emulates all 8087 instructions. The partial emulator (PE8087) is a smaller version that implements only the instructions needed to support PL/M-86 programs. The full emulator adds about 16k bytes to a program, while the partial emulator executes in about 8k. Any emulated program will deliver the same results (except for timing) if it is executed on 8087 hardware.

The emulators may be viewed as consisting of emulated hardware and emulated instructions. The emulators establish in CPU memory the equivalent of the 8087 register stack, control, and status words and all other programmer-accessible elements of the NDP architecture. The emulator instructions utilize the same algorithms as their hardware counterparts. Emulator instructions are actually implemented as CPU interrupt procedures. During relocation and linkage the 8087 machine instructions generated by the ASM-86 and PL/M-86 translators are changed to software interrupt (INT) instructions which invoke these procedures as the CPU processes its instruction stream.

Table S-22. Addressing Mode Examples

Coding		Interpretation
FIADD	ALPHA	ALPHA is a simple scalar (mode is direct).
FDIVR	ALPHA.BETA	BETA is a field in a structure that is "overlaid" on ALPHA (mode is direct).
FMUL	QWORD PTR [BX]	BX contains the address of a long real variable (mode is register indirect).
FSUB	ALPHA [SI]	ALPHA is an array and SI contains the offset of an array element from the start of the array (mode is indexed).
FILD	[BP].BETA	BP contains the address of a structure on the CPU stack and BETA is a field in the structure (mode is based).
FBLD	TBYTE PTR [BX] [DI]	BX contains the address of a packed decimal array and DI contains the offset of an array element (mode is based indexed).

Since the decision to produce real or emulated 8087 instructions is made at link time, a program may be switched from one mode to the other without retranslating the source code. When the PL/M-86 compiler or ASM-86 assembler places an 8087 machine instruction into an object module, it also inserts a special external reference. This reference is satisfied by linking the object module to one of two Intel-supplied libraries: the real library, or the emulator library. If the real library is specified, LINK-86 simply deletes the external references, leaving the original 8087 machine instructions.

To run on an emulated 8087, the object program is linked to the emulator library and to a file containing the code of either the full or the partial emulator. LINK-86 then adds the emulator code to the program and changes the 8087 machine instructions (and their preceding WAITs) to CPU software interrupt instructions. Any FWAIT instructions are also changed to CPU NOPs.

Note that an explicitly-coded CPU WAIT instruction will *not* be changed; if it is executed under emulation, the CPU will wait forever. This is why the FWAIT mnemonic should always be used when the external processor that the CPU is to wait for is an 8087.

In order to be compatible with E8087, ASM-86 programs should observe the following conventions:

- Their stack segment and class should be named STACK.
- Interrupt pointer (vector) 16 should be designated for the user's exception handler interrupt procedure.
- The external procedure INIT87 should be called in the program's initialization (power-up) sequence. If the emulator is being used, this procedure will initialize CPU interrupt pointers 20-31 to the addresses of emulator procedures and will execute an (emulated) FINIT instruction. If the program is not being emulated, INIT87 simply executes the FINIT instruction.

PL/M-86 automatically observes corresponding conventions.

Programming Example

Figures S-23 and S-24 show the PL/M-86 and ASM-86 code for a simple 8087 program, called ARRSUM. The program references an array (X$ARRAY), which contains 0-100 short real values; the integer variable NOFX indicates the number of array elements the program is to consider. ARRSUM steps through X$ARRAY accumulating three sums:

- SUM$X, the sum of the array values;
- SUM$INDEXES, the sum of each array value times its index, where the index of the first element is 1, the second is 2, etc.;
- SUM$SQUARES, the sum of each array element squared.

(A true program, of course, would go beyond these steps to store and use the results of these calculations.) The control word is set with the recommended values: projective closure, round to nearest, 64-bit precision, interrupts enabled, and all exceptions masked except invalid operation. It is assumed that an exception handler has been written to field the invalid operation, if it occurs, and that it is invoked by interrupt pointer 16. Either version of the program will run on an actual or an emulated 8087 without altering the code shown.

The PL/M-86 version of ARRSUM (figure S-23) is very straightforward and illustrates how easily the 8087 can be used in this language. After declaring variables the program calls built-in procedures to initialize the processor (or its emulator) and to load the control word. The program clears the sum variables and then steps through X$ARRAY with a DO-loop. The loop control takes into account PL/M-86's practice of considering the index of the first element of an array to be 0. In the computation of SUM$INDEXES, the built-in procedure FLOAT converts I+1 from integer to real because the language does not support "mixed mode" arithmetic. One of the strengths of the NDP, of

```
PL/M-86 COMPILER     ARRAYSUM

ISIS-II PL/M-86 DEBUG V2.1 COMPILATION OF MODULE ARRAYSUM
OBJECT MODULE PLACED IN :F4:ARRSUM.OBJ
COMPILER INVOKED BY:  :FO:PLM86 :F4:ARRSUM.P86 XREF


                  /****************************************
                  *                                       *
                  *   A R R A Y S U M . M O D             *
                  *                                       *
                  ****************************************/

    1             ARRAY$SUM:  DO;

    2   1         DECLARE (SUM$X,SUM$INDEXES,SUM$SQUARES) REAL;
    3   1         DECLARE X$ARRAY (100)   REAL;
    4   1         DECLARE (N$OF$X,I)  INTEGER;
    5   1         DECLARE CONTROL$87  LITERALLY  '033EH';

                  /* ASSUME X$ARRAY AND N$OF$X ARE INITIALIZED  */

                  /* PREPARE THE 8087, OR ITS EMULATOR */
    6   1         CALL INIT$REAL$MATH$UNIT;
    7   1         CALL SET$REAL$MODE(CONTROL$87);

                  /* CLEAR SUMS  */
    8   1         SUM$X, SUM$INDEXES, SUM$SQUARES = 0.0;

                  /* LOOP THROUGH X$ARRAY, ACCUMULATING SUMS */
    9   1         DO I = 0 TO N$OF$X - 1;
   10   2           SUM$X = SUM$X + X$ARRAY(I);
   11   2           SUM$INDEXES = SUM$INDEXES +
                              (X$ARRAY(I) * FLOAT(I + 1));
   12   2           SUM$SQUARES = SUM$SQUARES + (X$ARRAY(I) * X$ARRAY(I));
   13   2         END;

                  /* ETC...*/

   14   1         END ARRAY$SUM;
```

Figure S-23. Sample PL/M-86 Program

```
PL/M-86 COMPILER    ARRAYSUM

CROSS-REFERENCE LISTING
-----------------------

   DEFN  ADDR   SIZE  NAME, ATTRIBUTES, AND REFERENCES
  ----- ------ -----  --------------------------------

      1  0002H   151  ARRAYSUM . . . . .    PROCEDURE STACK=0002H
      5                CONTROL87. . . . .    LITERALLY          7
                       FLOAT. . . . . .      BUILTIN           11
      4  019EH     2  I. . . . . . . . .    INTEGER            9   10   11   12
                       INITREALMATHUNIT .    BUILTIN            6
      4  019CH     2  NOFX . . . . . . .    INTEGER            9
                       SETREALMODE. . . .    BUILTIN            7
      2  0004H     4  SUMINDEXES . . . .    REAL           8   11
      2  0008H     4  SUMSQUARES . . . .    REAL           8   12
      2  0000H     4  SUMX . . . . . . .    REAL           8   10
      3  000CH   400  XARRAY . . . . . .    REAL ARRAY(100)        10   11   12


MODULE INFORMATION:

     CODE AREA SIZE     = 0099H    153D
     CONSTANT AREA SIZE = 0004H      4D
     VARIABLE AREA SIZE = 01A0H    416D
     MAXIMUM STACK SIZE = 0002H      2D
     33 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

Figure S-23. Sample PL/M-86 Program (Cont'd.)

course, is that it *does* support arithmetic on mixed data types, and assembly language programmers can take advantage of this facility.

The ASM-86 version (figure S-24) defines the external procedure INIT87, which makes the different initialization requirements of the processor and its emulator transparent to the source code. After defining the data, and setting up the segment registers and stack pointer, the program calls INIT87 and loads the control word. The computation begins with the next three instructions, which clear three registers by loading (pushing) zeros onto the stack. As shown in figure S-25, these registers remain at the bottom of the stack throughout the computation while temporary values are pushed on and popped off the stack above them.

```
8086/8087/8088 MACRO ASSEMBLER    ARRSUM

ISIS-II 8086/8087/8088 MACRO ASSEMBLER V3.0 ASSEMBLY OF MODULE ARRSUM
OBJECT MODULE PLACED IN :F1:ARRSUM.OBJ
ASSEMBLER INVOKED BY:  :F0:ASM86 :F1:ARRSUM.A86 XREF

LOC  OBJ                  LINE     SOURCE

                             1     ;DEFINE INITIALIZATION ROUTINE
                             2             EXTRN   INIT87:FAR
                             3
                             4     ;ALLOCATE SPACE FOR DATA
----                         5             DATA            SEGMENT PUBLIC 'DATA'
0000 3E03                    6             CONTROL_87      DW      033EH
0002 ????                    7             N_OF_X          DW      ?
0004 (100                    8             X_ARRAY         DD      100 DUP (?)
     ????????
     )
0194 ????????                9             SUM_X           DD      ?
0198 ????????               10             SUM_INDEXES     DD      ?
019C ????????               11             SUM_SQUARES     DD      ?
----                        12             DATA            ENDS
```

Figure S-24. Sample ASM-86 Program

```
8086/8087/8088 MACRO ASSEMBLER    ARRSUM

LOC  OBJ                  LINE     SOURCE
                            13
                            14     ;ALLOCATE CPU STACK SPACE
----                        15             STACK           SEGMENT STACK 'STACK'
0000 (200                   16                             DW      200 DUP (?)
     ????
     )
                            17
                            18     ;LABEL INITIAL TOP OF STACK
0190                        19             STACK_TOP       LABEL   WORD
----                        20             STACK           ENDS
                            21
----                        22             CODE    SEGMENT PUBLIC 'CODE'
                            23             ASSUME  CS:CODE,DS:DATA,SS:STACK,ES:NOTHING
                            24
0000                        25     START:
0000 B8----          R      26             MOV     AX,DATA
0003 8ED8                   27             MOV     DS,AX
0005 B8----          R      28             MOV     AX,STACK
0008 8ED0                   29             MOV     SS,AX
000A BC9001          R      30             MOV     SP,OFFSET STACK_TOP
                            31
                            32     ;ASSUME X_ARRAY & N_OF_X ARE INITIALIZED.
                            33     ;       NOTE: PROGRAM ZEROS N_OF_X
                            34     ;PREPARE THE 8087 OR ITS EMULATOR.
                            35
000D 9A0000----      E      36             CALL    INIT87
0012 9BD92E0000      R      37             FLDCW   CONTROL_87
                            38
                            39     ;CLEAR 3 REGISTERS TO HOLD RUNNING SUMS.
                            40
0017 9BD9EE                 41             FLDZ
001A 9BD9EE                 42             FLDZ
001D 9BD9EE                 43             FLDZ
                            44
                            45     ;SETUP CX AS LOOP COUNTER & SI AS INDEX TO X_ARRAY.
                            46
0020 8B0E0200        R      47             MOV     CX,N_OF_X
0024 E329                   48             JCXZ    POP_RESULTS     ;EXIT EARLY IF X_ARRAY EMPTY
0026 B80400                 49             MOV     AX,TYPE X_ARRAY
0029 F7E9                   50             IMUL    CX
002B 8BF0                   51             MOV     SI,AX
                            52
                            53     ;SI NOW CONTAINS INDEX OF LAST ELEMENT + 1.
                            54     ;LOOP THRU X_ARRAY ACCUMULATING SUMS.
002D                        55     SUM_NEXT:
002D 83EE04                 56             SUB     SI,TYPE X_ARRAY       ;BACKUP ONE ELEMENT
0030 9BD9840400      R      57             FLD     X_ARRAY[SI]           ;PUSH IT ONTO STACK
0035 9BDCC3                 58             FADD    ST(3),ST              ;ADD INTO SUM OF X
0038 9BD9C0                 59             FLD     ST                    ;DUPLICATE X ON TOP
003B 9BDCC8                 60             FMUL    ST,ST                 ;SQUARE IT
003E 9BDEC2                 61             FADDP   ST(2),ST              ;ADD INTO SUM OF SQUARES
                            62                                           ;  AND DISCARD
0041 9BDE0E0200      R      63             FIMUL   N_OF_X                ;GET X TIMES ITS INDEX
0046 9BDEC2                 64             FADDP   ST(2),ST              ;ADD INTO SUM OF (INDEX * X)
                            65                                           ;   AND DISCARD
0049 FF0E0200        R      66             DEC     N_OF_X                ;REDUCE INDEX FOR NEXT ITERATION
004D E2DE                   67             LOOP    SUM_NEXT              ;CONTINUE
                            68
                            69     ;POP RUNNING SUMS INTO MEMORY
004F                        70     POP_RESULTS:
004F 9BD91E9C01      R      71             FSTP    SUM_SQUARES
0054 9BD91E9801      R      72             FSTP    SUM_INDEXES
0059 9BD91E9401      R      73             FSTP    SUM_X
                            74
                            75     ;
                            76     ;ETC...
                            77     ;
----                        78     CODE    ENDS
                            79
  0000                      80             END     START
```

Figure S-24. Sample ASM-86 Program (Cont'd.)

```
8086/8087/8088 MACRO ASSEMBLER    ARRSUM

XREF SYMBOL TABLE LISTING
---- ------ ----- -------

NAME          TYPE     VALUE   ATTRIBUTES, XREFS

??SEG . . .   SEGMENT          SIZE=0000H PARA PUBLIC
CODE. . . .   SEGMENT          SIZE=005EH PARA PUBLIC 'CODE'   22# 23 78
CONTROL_87.   V WORD   0000H   DATA  6# 37
DATA. . . .   SEGMENT          SIZE=01A0H PARA PUBLIC 'DATA'   5# 12 23 26
INIT87. . .   L FAR    0000H   EXTRN  2# 36
N_OF_X. . .   V WORD   0002H   DATA  7# 47 63 66
POP_RESULTS   L NEAR   004FH   CODE  48 70#
STACK . . .   SEGMENT          SIZE=0190H PARA STACK 'STACK'
STACK_TOP .   V WORD   0190H   STACK  19# 30
START . . .   L NEAR   0000H   CODE  25# 80
SUM_INDEXES   V DWORD  0198H   DATA  10# 72
SUM_NEXT. .   L NEAR   002DH   CODE  55# 67
SUM_SQUARES   V DWORD  019CH   DATA  11# 71
SUM_X . . .   V DWORD  0194H   DATA  9# 73
X_ARRAY . .   V DWORD  0004H   DATA  8# 49 56 57

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

Figure S-24. Sample ASM-86 Program (Cont'd.)

The program uses the CPU LOOP instruction to control its iteration through X_ARRAY; register CX, which LOOP automatically decrements, is loaded with N_OF_X, the number of array elements to be summed. Register SI is used to select (index) the array elements. The program steps through X_ARRAY from "back to front", so SI is initialized to point at the element just beyond the first element to be processed. The ASM-86 TYPE operator is used to determine the number of bytes in each array element. This permits changing X_ARRAY to a long real array by simply changing its definition (DD to DQ) and re-assembling.

Figure S-25 shows the effect of the instructions in the program loop on the NDP register stack. The figure assumes that the program is in its first iteration, that N_OF_X is 20, and that X_ARRAY(19) (the 20th element) contains the value 2.5. When the loop terminates, the three sums are left as the top stack elements so that the program ends by simply popping them into memory variables.

S.9 Special Topics

This section describes features of the 8087 which will be of interest to groups of users who have special requirements. Most users will not need to understand this material in detail in order to utilize the NDP successfully. Most readers, then, can either browse this section, or skip it altogether in favor of the programming examples in section S.10.

The first four topics in this section cover the 8087's generation and handling of nonnormalized real values, zeros, infinities and NANs. In the great majority of applications, these special values will either not appear at all, or in the case of zeros, will function according to the normal rules of arithmetic. Next the bit encodings of each data type are summarized in table form, including special values. This information may be of use to programmers who are sorting these data types or are decoding unformatted memory dumps or data monitored from the bus. At the end of the section is a table that lists all 8087 exception conditions by class, and the processor's masked response to each exception. This information will principally be of use to writers of exception handlers and to anyone else interested in ascertaining the exact conditions under which the NDP signals a given type of exception.

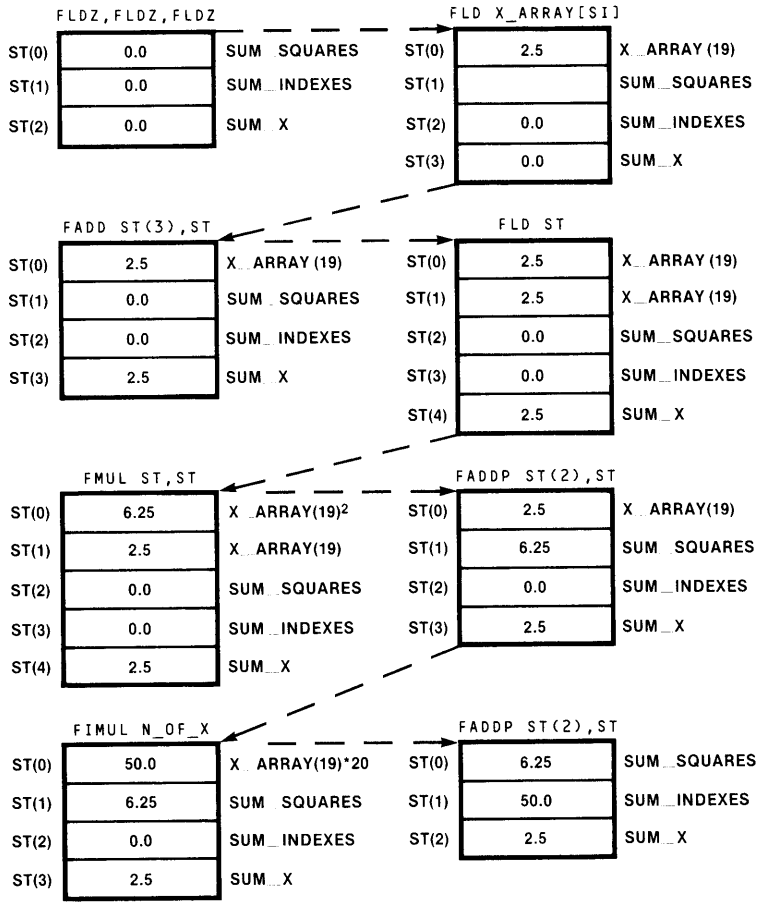


Figure S-25. Instructions and Register Stack

Nonnormal Real Numbers

As discussed in section S.3, the 8087 generally stores nonzero real numbers in normalized floating point form; that is, the integer (leading) bit of the significand is always a 1. This bit is explicitly stored in the temporary real format, and is implicit in the short and long real forms. Normalized storage allows the maximum number of significant digits to be held in a significand of a given width, because leading zeros are eliminated.

Denormals

A denormal is the result of the NDP's masked response to an underflow exception. Underflow occurs when the exponent of a true result is too small to be represented in the destination format. For example, a true exponent of −130 will cause underflow if the destination is short real, because −126 is the smallest exponent this format can accommodate. (No underflow would occur if the destination were long or temporary real since these can handle exponents down to −1023 and −16,383, respectively.)

The NDP's unmasked response to underflow is to stop and request an interrupt if the destination is a memory operand. If the destination is a register, the processor adds the constant 24,576 (decimal) to the true result's exponent, returns the result, and then requests an interrupt. The constant forces the exponent into the range of the temporary real format, and an exception handler can subtract out the constant to ascertain the true exponent. Thus, execution always stops when there is an unmasked underflow.

The intent of the masked response to underflow is to allow computation to continue without program intervention, while introducing an error that carries about the same risk of contaminating the final result as roundoff error. Roundoff (precision) errors occur frequently in real number calculations; sometimes they spoil the result of computation, but often they do not. Recognizing that roundoff errors are often non-fatal, computation usually proceeds and the programmer inspects the final result to see if these errors have had a significant effect. The 8087's masked underflow response allows programmers to treat underflows in a similar manner; the computation continues and the programmer can examine the final result to determine if an underflow has had important consequences. (If the underflow has had a significant effect, an invalid operation will probably be signalled later in the computation.)

Most computers underflow "abruptly"; they simply return a zero result, which is likely to produce an unacceptable final result if computation continues. The 8087, on the other hand, underflows "gradually" when the underflow exception is masked. Gradual underflow is accomplished by denormalizing the result until it is just within the exponent range of the destination. Denormalizing means incrementing the true result's exponent and inserting a corresponding leading zero in the significand, shifting the rest of the significand one place to the right. Table S-23 illustrates how a result might be denormalized to fit a short real destination.

Denormalization produces a denormal or a zero. Denormals are readily identified by their exponents, which are always the minimum for their formats; in biased form, this is always the bit string: 00...00. This same exponent value is also assigned to the zeros, but a denormal has a nonzero significand. A denormal in a register is tagged special.

The denormalization process may cause the loss of low-order significand bits as they are shifted off the right. In a severe case, *all* the significand bits of the true result are shifted out and replaced by the leading zeros. In this case, the result of denormalization is a true zero, and if the value is in a register, it is tagged as such. However, this is a comparatively rare occurrence, and in any case is no worse than "abrupt" underflow.

Denormals are rarely encountered in most applications. Typical debugged algorithms generate extremely small results during the evaluation of intermediate subexpressions; the final result is usually of an appropriate magnitude for its short or long real destination. If intermediate results are held in temporary real, as is recommended, the great range of this format

Table S-23. Denormalization Process

Operation	Sign	Exponent[1]	Significand
True Result	0	−129	$1_{\Delta}01011100...00$
Denormalize	0	−128	$0_{\Delta}101011100...00$
Denormalize	0	−127	$0_{\Delta}0101011100...00$
Denormalize	0	−126	$0_{\Delta}00101011100...00$
Denormal Result[2]	0	−126	$0_{\Delta}00101011100...00$

Notes:
[1]expressed as unbiased, decimal number

[2]Before storing, significand is rounded to 24 bits, integer bit is dropped, and exponent is biased by adding 126.