intel

The 8086 Family User's Manual

Numerics Supplement

July 1980

Additional copies of this manual or other Intel literature may be obtained from:

Literature Department
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

The information in this document is subject to change without notice.

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update nor to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.

The following are trademarks of Intel Corporation and may be used only to identify Intel products:

BXP	Intellec	Multibus
CREDIT	iSBC	Multimodule
i	iSBX	PROMPT
ICE	Library Manager	Promware
iCS	MCS	RMX
Insite	Megachassis	UPI
Intel	Micromap	μScope
Intelevision		

and the combination of ICE, iCS, iSBC, iSBX, MCS, or RMX and a numerical suffix.

PREFACE

This supplement provides detailed information on the 8087 Numeric Coprocessor extension to the iAPX 86/10 and 88/10 CPUs. These processors and their support circuits are described in detail in the 8086 Family User's Manual, and discussions in this supplement frequently reference this manual. Below is a brief overview of architectural concepts used throughout the family and the system nomenclature used to describe particular system configurations built around the four iAPX 86, 88 family processors.

Microsystem 80 Nomenclature

Over the last several years, the increase in microcomputer system and software complexity has given birth to a new family of microprocessor products oriented towards solving these increasingly complex problems. This new generation of microprocessors is both powerful and flexible and includes many processor enhancements, such as numeric floating point extensions, I/O processors, and operating system functionality in silicon.

As Intel's product line has grown and evolved, its microprocessor product numbering system has become inadequate to name VLSI solutions involving the above enhancements.

In order to accommodate these new VLSI systems, we've allowed the 8086 family name to evolve into a more comprehensive numbering scheme, while still including the basis of the previous 8086 nomenclature.

We've adopted the following prefixes to provide differentiation and consistency among our Microsystem 80 related product lines:

iAPX — Processor Series
iRMX — Operating Systems
iSBC — Single Board Computers
iSBX — MULTIMODULE Boards

Concentrating on the iAPX Series, two Processor Families are defined:

iAPX 86—8086 CPU based system
iAPX 88—8088 CPU based system

With additional suffix information, configuration options within each iAPX system can be identified, for example:

iAPX 86/10	CPU Alone (8086)
iAPX 86/11	CPU + IOP (8086 + 8089)
iAPX 88/20	CPU + Math Extension (8088 + 8087)
iAPX 88/21	CPU + Math Extension + IOP (8088 + 8087 + 8089)

This nomenclature is intended as an addition to rather than a replacement for, Intel's current part numbers. These new series level descriptions are used to describe the functional capabilities provided by specific configurations of the processors in the 8086 Family. The hardware used to implement each functional configuration is still described by referring to the parts involved (as is the case for the majority of the 8087 information described in this supplement.)

This improved nomenclature provides a more meaningful view of system capability and performance within the evolving Microsystem 80 architecture.

iAPX 86, 88 Architecture

The components in the iAPX 86, 88 product lines have been designed to operate together in diverse combinations within the framework of the family architecture, as shown in figure i. In this way a single family of components can be used to solve a wide array of microcomputing problems. A component mix can be tailored to fit the performance needs of an application precisely, without having to pay for unneeded capabilities that may be bundled into more monolithic, CPU-centered architectures. Using the same family of components across multiple systems limits the learning curve problem and builds on past experience. Finally, the modular structure of the family architecture provides an orderly way for systems to grow and change.

The iAPX 86, 88 product line architecture is characterized by three major principles:

1. System functions are distributed among specialized components.

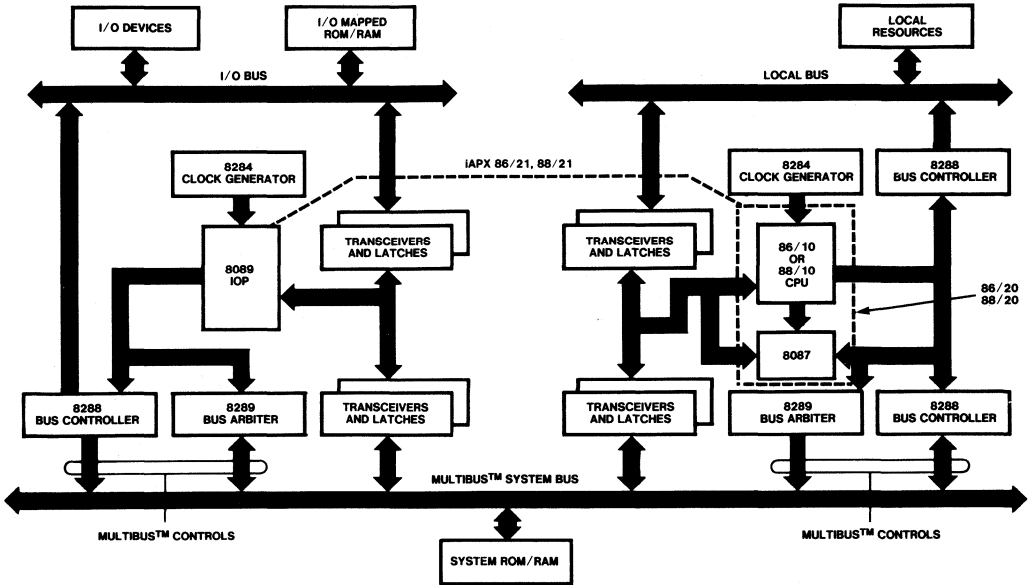


Figure i. iAPX 86, 88 Multiprocessing System

2. Multiprocessing capabilities are inherent in the hardware.
3. A hierarchical bus organization provides for the complex data flows required by high-performance systems without burdening simpler systems with unneeded capabilities.

Microprocessors

At the core of the product line are four microprocessors that share these characteristics:

- 5 MHz (200 ns cycle time) and 8 MHz (86/10) are available.
- Systems can be constructed for both 8 & 16 bit data paths.
- Processors operate on 8-, 16-, 32-, 64-, 80-bit character and string data types; internal data paths are at least 16 bits wide.
- Up to 1 megabyte of memory can be addressed, along with a separate 64K byte I/O space.
- The address/data and status interfaces of the processors are compatible (the address and data buses are time multiplexed at the processor, allowing each to be compatible with a common set of bipolar bus support components.

Multiprocessing

Employing multiple processors in medium to large systems offers several significant advantages over the centralized approach that relies on a single CPU and extremely fast memory:

- System tasks may be allocated to special-purpose processors whose designs are optimized to perform specific (or classes of) tasks simply and efficiently;
- Very high levels of performance can be attained when processors can execute simultaneously (parallel/distributed processing);
- Reliability can be improved by isolating system functions so that a failure or error in one part of the system has a limited effect on the rest of the system;
- The natural partitioning of the system promotes parallel development of sub-systems, breaks the application into smaller, more manageable tasks, and helps isolate the effects of system modifications.

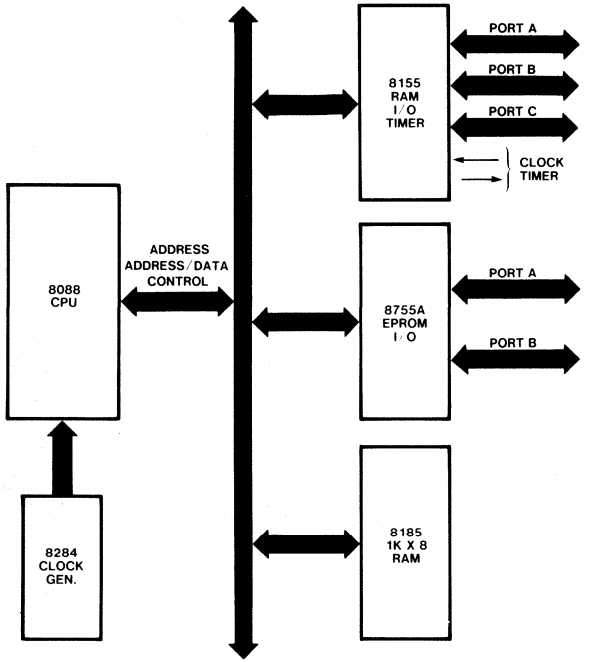


Figure ii. iAPX 88 Allows for a Highly Integrated, High Performance System Using 8085 Family of Components

The iAPX 86, 88 product line architecture is explicitly designed to simplify the development of multiple processor systems by providing facilities for coordinating the interaction of the processors.

The architecture supports two types of processors: independent processors and coprocessors. An independent processor executes its own instruction stream. The iAPX 86/10, 88/10, and IOP are examples of independent processors. An iAPX 86/10 or iAPX 88/10 typically executes a program in response to an interrupt. The IOP starts its channels in response to an interrupt-like signal called a channel attention; this signal is typically issued by a CPU.

The iAPX 86, 88 product line architecture also supports a second type of processor, called a coprocessor. Coprocessor "hooks" have been designed into the iAPX 86/10 and iAPX 88/10 to allow this type of processor to be accommodated. The coprocessor adds additional register, datatype, and instruction resources directly to the system. A coprocessor, in effect, extends the instruction set (and architecture) of its host processor.

The iAPX 86, 88 provides built-in solutions to two classic multiprocessing coordination problems: bus arbitration and mutual exclusion. Bus arbitration may be performed by the bus request/grant logic contained in each of the processors (local bus arbitration), by 8289 bus arbiters (system bus arbitration), or by a combination of the two when processors have access to multiple shared buses. In all cases, the arbitration mechanism operates invisibly to software.

For mutual exclusion, each processor has a LOCK (bus lock) signal which a program may activate to prevent other processors from obtaining a shared system bus. The IOP may lock the bus during a DMA transfer to ensure both that the transfer completes in the shortest possible time and that another processor does not access the target of the transfer (e.g., a buffer) while it is being updated. Each of the processors has an

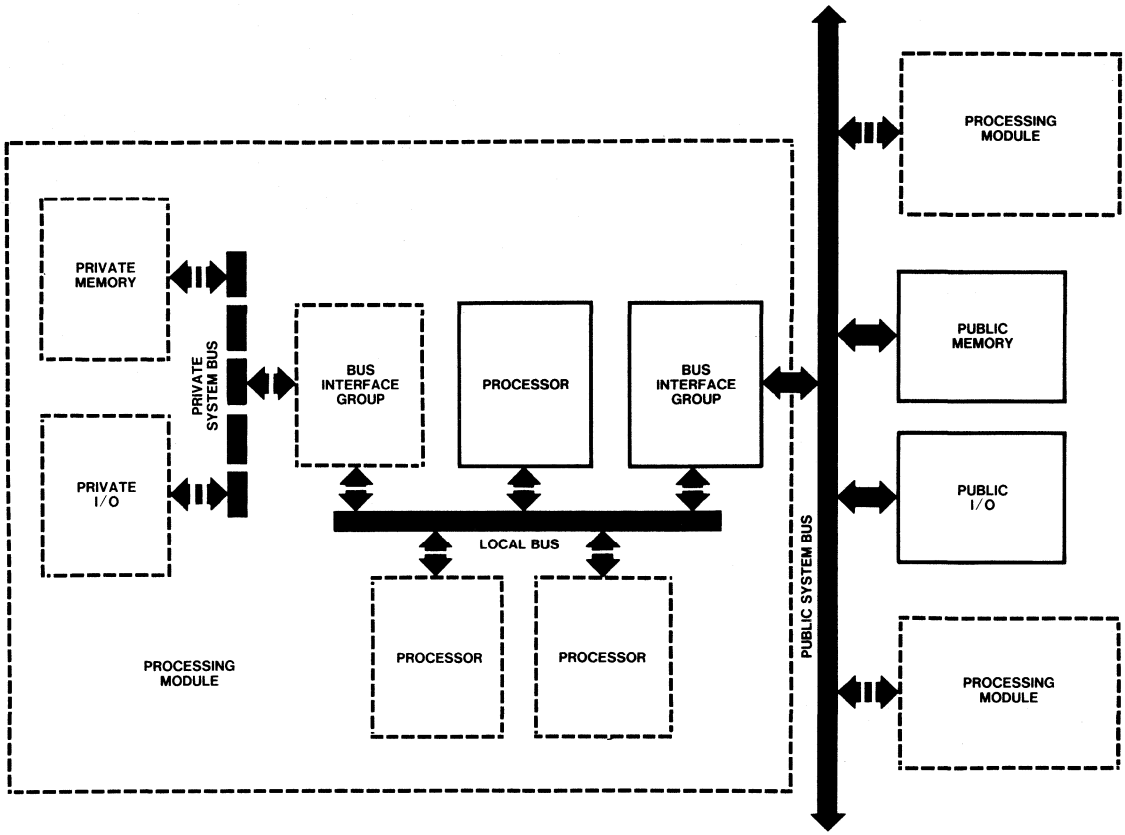


Figure iii. Generalized iAPX 86, 88 Bus Structure

instruction that examines and updates a memory byte with the bus locked. This instruction can be used to implement a semaphore mechanism for controlling the access of multiple processors to shared resources. (A semaphore is a variable that indicates whether a resource, such as a buffer or a pointer, is "available" or "in use".)

Bus Organization

Figure iii summarizes the iAPX 86, 88 bus structure. There are two different types of buses: system and local. Both buses may be shared by multiple processors, i.e., both are multimaster buses. Microprocessors are always connected to a local bus, and memory and I/O components usually reside on a system bus. The iAPX 86, 88 bus interface components link a local bus to a system bus.

Local Bus

The local bus is optimized for use by the iAPX 86, 88 microprocessors. Since standard memory and I/O components are not attached to the local bus, information can be multiplexed and encoded to make very efficient use of processor pins (certain MCS-85 peripheral components can be directly connected to the local bus). This allows several pins to be dedicated to coordinating the activity of multiple processors sharing the local bus. Multiple processors connected to the same local

bus are said to be local to each other; processors on different local buses are said to be remote to each other, or configured remotely. Both independent processors and coprocessors may share a local bus; on-chip arbitration logic determines which processor drives the bus. Because the processors on the local bus share the same bus interface components, the local configuration of multiple processors provides a compact and inexpensive multiprocessing system.

System Bus

A full implementation of an iAPX 86, 88 system bus consists of the following five sets of signals: address bus, data bus, control lines, interrupt lines and arbitration lines. A group of bus interface components transforms the signals of a local bus into a system bus. The number of bus interface components required to generate a system bus depends on the size and complexity of the system; reduced application needs translate directly into reduced component counts. These main variables determine the configuration of a bus interface group: address space size (number of latches), data bus width (number of transceivers), and arbitration needs (presence of a bus arbiter).

The iAPX 86, 88 system bus is functionally and electrically compatible with the MULTIBUS multimaster system bus used in Intel's iSBC line of single board computing products. This compatiblity gives system designers access to a wide variety of computer, memory, communications and other modules that may be incorporated into products, used for evaluation or for test vehicles.

Processing Modules and Bus Topology

The processor(s) and bus interface group(s) that are connected by a local bus constitute a processing module. A simple processing module could consist of a single CPU and one bus interface group. A more complex module would contain multiple processors, such as two IOPs, a CPU and one or two IOPs, or a CPU with a coprocessor with/without IOP. One bus interface group typically links the processors in the module to a public system bus. If there are multiple processing modules in the system, all memory or I/O connected to the public bus is accessible to all processing modules on the public bus. 8289 bus arbiters in each processing module control the access of the modules to the public bus and hence to the public memory and I/O.

A second bus interface group may be connected to a processing module's local bus, generating a demultiplexed bus. This bus can provide the processing module with a private address space that is not accessible to other processing modules. Distributing memory and I/O resources in this manner can improve system reliability by isolating the effects of failures. It can also increase system throughput dramatically. If processor programs and local data are placed in private memory, contention for use of the public system bus can be held to a minimum to ensure that shared resources are quickly available when they are needed. In addition, processors in separate modules can simultaneously fetch instructions from private memory spaces to allow multiple system tasks to proceed in parallel.

Table of Contents

Tables

Illustrations

8087 Numeric Data Processor

THE 8087 NUMERIC DATA PROCESSOR

This supplement describes the 8087 Numeric Data Processor (NDP). Its organization is similar to chapters 2 and 3 of *The 8086 Family User's Manual*:

1. Processor Overview
2. Processor Architecture
3. Computation Fundamentals
4. Memory
5. Multiprocessing Features
6. Processor Control and Monitoring
7. Instruction Set
8. Programming Facilities
9. Special Features
10. Programming Examples

Section 1 covers both hardware and software topics at a general level. Sections 2 and 4 through 6 are largely hardware-oriented, while sections 3 and 7 through 10 are of greatest interest to programmers. Section 9 describes features of the NDP that will be of interest to specialized groups of users; it is not necessary to understand this section to successfully use the 8087 in most applications. Hardware coverage in this supplement is limited to discussing processor facilities in functional terms. Timing, electrical characteristics, and other physical interface data may be found in Appendix B, as well as in Chapter 4 of *The 8086 Family User's Manual*.

Note that throughout this supplement the term "CPU" refers to either an 8086 or 8088 configured in maximum mode. To make best use of the material in this publication, readers should have a good understanding of the operation of the 8086/8088 CPUs.

S.1 Processor Overview

The 8087 Numeric Data Processor is a coprocessor that performs arithmetic and comparison operations on a variety of numeric data types; it also executes numerous built-in transcendental functions (e.g., tangent and log functions). As a coprocessor to a maximum mode 8086 or 8088, the NDP effectively extends the register and instruction sets of the host CPU and adds several new data types as well. The programmer generally does not perceive the 8087 as a separate device; instead, the computational capabilities of the CPU appear greatly expanded.

The 8087 is the only chip required to add extensive high-speed numeric processing capabilities to an 8086- or 8088-based system. It is specifically designed to deliver stable, correct results when used in a straightforward fashion by programmers who are not expert in numerical analysis. Its applicability to accounting and financial environments, in addition to scientific and engineering settings, further distinguishes the 8087 from the "floating point accelerators" employed in many computer systems, including minicomputers and mainframes. The NDP is housed in a standard 40-pin dual in-line package (figure S-1) and requires a single +5V power source.

The description of the 8087 in this section deliberately omits some operating details in order to provide a coherent overall view of the processor's capabilities. Subsequent sections of the supplement describe these capabilities, and others, in more detail.

Evolution

The performance of first- and second-generation microprocessor-based systems was limited in three principal areas: storage capacity, input/output speed, and numeric computation. The 8086 and 8088 CPUs broke the 64k memory barrier, allowing larger and more time-critical applications to be undertaken. The 8089 Input/Output Processor eliminated many of the I/O bottlenecks and permitted microprocessors to be employed effectively in I/O-intensive designs. The 8087 Numeric Data Processor clears the third roadblock by enabling applications with significant computational requirements to be implemented with microprocessor technology.

Figure S-2 illustrates the progression of Intel numeric products and events that have led to the development of the 8087. In the mid-1970's, Intel

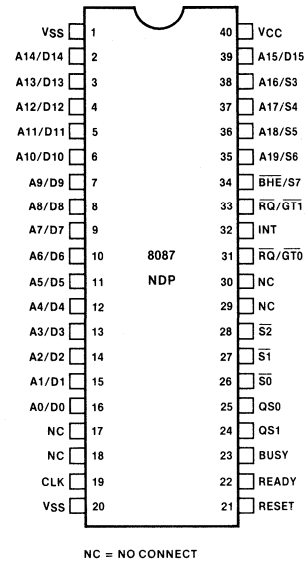


Figure S-1. 8087 Numeric Data Processor Pin Diagram

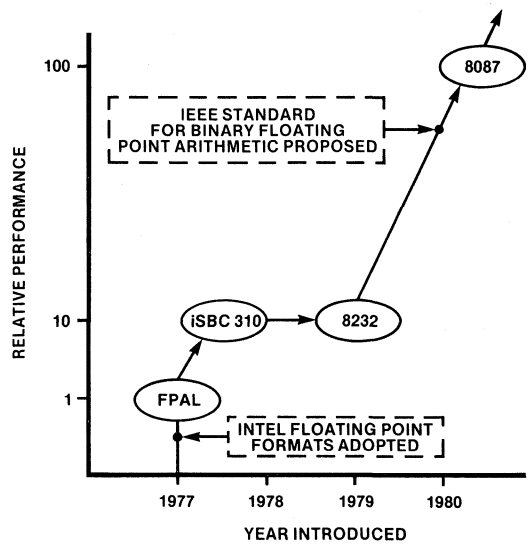


Figure S-2. 8087 Evolution and Relative Performance

made the commitment to expand the computational capabilities of microprocessors from addition and subtraction of integers to an array of widely useful operations on real numbers. (Real numbers encompass integers, fractions, and irrational numbers such as π and $\sqrt{2}$.) In 1977, the corporation adopted a standard for representing real numbers in a "floating point" format. Intel's Floating Point Arithmetic Library (FPAL) was the first product to utilize this standard format. FPAL is a set of subroutines for the 8080/8085 microprocessors. These routines perform arithmetic and limited standard functions on single precision (32-bit) real numbers; an FPAL multiply executes in about 1.5 ms (1.6 MHz 8080A CPU). The next product, the iSBC 310™ High Speed Math Unit, essentially implements FPAL in a single iSBC™ card, reducing a single-precision multiply to about 100 μs. The Intel® 8232 is a single-chip arithmetic processor for the 8080/8085 family. The 8232 accepts double precision (64-bit) operands as well as single precision numbers. It performs a single precision multiply in about 100 μs and multiplies double precision numbers in about 875 μs (2 MHz version).

In 1979, a working committee of the Institute for Electrical and Electronic Engineers (IEEE) proposed an industry standard for minicomputer and microcomputer floating point arithmetic*. The intent of the standard is to promote portability of numeric programs between computers and to provide a uniform programming environment that encourages the development of accurate, reliable software. The proposed standard specifies requirements and options for number formats as well as the results of computations on these numbers. The floating point number formats are identical to those previously adopted by Intel and used in the products described in this section.

The 8087 Numeric Data Processor is the most advanced development in Intel's continuing effort to provide improved tools for numerically-oriented microprocessor applications. It is a single-chip hardware implementation of the proposed IEEE standard, including all its options for single and double precision numbers. As such, it is compatible with previous Intel numerics products; programs written for the 8087 will be transportable to future products that conform to

* J. Coonen, W. Kahan, J. Palmer, T. Pittman, D. Stevenson, "A Proposed Standard for Binary Floating Point Arithmetic," *ACM SIGNUM Newsletter*, October 1979.

the proposed IEEE standard. The NDP also provides many additional functions that are extensions to the proposed standard.

Performance

As figure S-2 indicates, the 8087 provides about 10 times the instruction speed of the 8232 and a 100-fold improvement over FPAL. The 8087 multiplies 32-bit and 64-bit real numbers in about 19 μs and 27 μs, respectively. Of course, the actual performance of the NDP in a given system depends on numerous application-specific factors.

Table S-1 compares the execution times of several 8087 instructions with the equivalent operations executed in software on a 5 MHz 8086. The software equivalents are highly optimized assembly language procedures from the 8087 emulator, an NDP development tool discussed later in this section.

The performance figures quoted in this section are for operations on real (floating point) numbers. The 8087 also has instructions that enable it to utilize fixed point binary and decimal integers of up to 64 bits and 18 digits, respectively. Using an 8087, rather than multiple precision software algorithms for integer operations, can provide speed improvements of 10-100 times.

The 8087's unique coprocessor interface to the CPU can yield an additional performance increment beyond that of simple instruction speed. No overhead is incurred in setting up the device for a computation; the 8087 decodes its own instructions automatically in parallel with the CPU. Moreover, built-in coordination facilities allow the CPU to proceed with other instructions while the 8087 is simultaneously executing its numeric instruction. Programs can exploit this processor parallelism to increase total system throughput.

Usability

Viewed strictly from the standpoint of raw speed, the 8087 enables serious computation-intensive tasks to be performed by microprocessors for the first time. The 8087 offers more than just high performance, however. By synthesizing advances made by numerical analysts in the past several years, the NDP provides a level of usability that surpasses existing minicomputer and mainframe arithmetic units. In fact, the charter of the 8087 design team was first to achieve exceptional functionality and then to obtain high performance.

The 8087 is explicitly designed to deliver stable, accurate results when programmed using straightforward "pencil and paper" algorithms. While this statement may seem trivial, experienced users of "floating point processors" will

Table S-1. 8087 Emulator Speed Comparison

Instruction	Approximate Execution Time (μs) (5 MHz Clock)	
	8087	8086 Emulation
Multiply (single precision)	19	1,600
Multiply (double precision)	27	2,100
Add	17	1,600
Divide (single precision)	39	3,200
Compare	9	1,300
Load (single precision)	9	1,700
Store (single precision)	18	1,200
Square root	36	19,600
Tangent	90	13,000
Exponentiation	100	17,100

recognize its fundamental importance. For example, most computers can overflow when two single precision floating point numbers are multiplied together and then divided by a third, even if the final result is a perfectly valid 32-bit number. The 8087 delivers the correctly rounded result. Other typical examples of undesirable machine behavior in straightforward calculations occur when solving for the roots of a quadratic equation:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

or computing financial rate of return, which involves the expression: $(1+i)^n$. Straightforward algorithms will not deliver consistently correct results (and will not indicate when they are incorrect) on most machines. To obtain correct results on traditional machines under all conditions usually requires sophisticated numerical techniques that are foreign to most programmers. General application programmers using straightforward algorithms will produce much more reliable programs on the 8087. This simple fact greatly reduces the software investment required to develop safe, accurate computation-based products.

Beyond traditional numerics support for "scientific" applications, the 8087 has built-in facilities for "commerical" computing. It can process decimal numbers of up to 18 digits without round-off errors, and it performs *exact arithmetic* on integers as large as 2^{64}. Exact arithmetic is vital in accounting applications where rounding errors may introduce money losses that cannot be reconciled.

The NDP contains a number of facilities that can optionally be invoked by sophisticated users. Examples of these advanced features include two models of infinity, directed rounding, gradual underflow, and traps to user-written exception handling software.

Applications

The NDP's versatility and performance make it appropriate to a broad array of numerically-oriented applications. In general, applications that exhibit any of the following characteristics can benefit by implementing numeric processing on the 8087:

- Numeric data vary over a wide range of values, or include non-integral values;
- Algorithms produce very large or very small intermediate results;
- Computations must be very precise, i.e., a large number of significant digits must be maintained;
- Performance requirements exceed the capacity of traditional microprocessors;
- Consistently safe, reliable results must be delivered using a programming staff that is not expert in numerical techniques.

Note also that the 8087 can reduce software development costs and improve the performance of systems that do not utilize real numbers but operate on multi-precision binary or decimal integer values.

A few examples, which show how the 8087 might be utilized in specific numerics applications, are described below. In many cases, these types of systems have been implemented in the past with minicomputers. The advent of the 8087 brings the size and cost savings of microprocessor technology to these applications for the first time.

- Business data processing—The NDP's ability to accept decimal operands and produce exact decimal results of up to 18 digits greatly simplifies accounting programming. Financial calculations which use power functions can take advantage of the 8087's exponentiation and logarithmic instructions.
- Process control—The 8087 solves dynamic range problems automatically and its extended precision allows control functions to be fine-tuned for more accurate and efficient performance. Control algorithms implemented with the NDP also contribute to improved reliability and safety, while the 8087's speed can be exploited in real-time operations.
- Numerical control—The 8087 can move and position machine tool heads with extreme accuracy. Axis positioning also benefits from the hardware trigonometric support provided by the 8087.

- Robotics—Coupling small size and modest power requirements with powerful computational abilities, the NDP is ideal for on-board six-axis positioning.
- Navigation—Very small, light weight, and accurate inertial guidance systems can be implemented with the 8087. Its built-in trigonometric functions can speed and simplify the calculation of position from bearing data.
- Graphics terminals—The 8087 can be used in graphics terminals to locally perform many functions which normally demand the attention of a main computer; these include rotation, scaling, and interpolation. By also including an 8089 Input/Output Processor to perform high speed data transfers, very powerful and highly self-sufficient terminals can be built from a relatively small number of 8086 family parts.
- Data acquisition—The 8087 can be used to scan, scale, and reduce large quantities of data as it is collected, thereby lowering storage requirements as well as the time required to process the data for analysis.

The preceding examples are oriented toward "traditional" numerics applications. There are, in addition, many other types of systems that do not appear to the end user as "computational," but can employ the 8087 to advantage. Indeed, the 8087 presents the imaginative system designer with an opportunity similar to that created by the introduction of the microprocessor itself. Many applications can be viewed as numerically-based if sufficient computational power is available to support this view. This is analogous to the thousands of successful products that have been built around "buried" microprocessors, even though the products themselves bear little resemblance to computers.

Programming Interface

The combination of an 8086 or 8088 CPU and an 8087 generally appears to the programmer as a single machine. The 8087, in effect, adds new data types, registers, and instructions to the CPU. The programming languages and the coprocessor architecture take care of most interprocessor coordination automatically.

Table S-2 lists the seven 8087 data types. Internally, the 8087 holds all numbers in the temporary real format; the extended range and precision of this format are key contributors to the NDP's ability to consistently deliver stable, expected results. The 8087's load and store instructions convert operands between the other formats and temporary real. The fact that these conversions are made, and that calculations may be performed on converted numbers, is transparent to the programmer. Integer operands, whether binary or decimal, yield correct integer results, just as real operands yield correct real results. Moreover, a rounding error does not occur when a number in an external format is converted to temporary real.

Computations in the 8087 center on the processor's register stack. These eight 80-bit registers provide the equivalent capacity of 40 of the 16-bit registers found in typical CPUs. This generous register space allows more constants and intermediate results to be held in registers during calculations, reducing memory access and consequently improving execution speed as well as bus availability. The 8087 register set is unique in that it can be accessed both as a stack, with instructions operating implicitly on the top one or two stack elements, and as a fixed register set, with instructions operating on explicitly designated registers.

Table S-3 lists the 8087's major instructions by class. Assembly language programs are written in ASM-86, the 8086/8088/8087 common assembly language. ASM-86 provides directives for defining all 8087 data types and mnemonics for all instructions. The fact that some instructions in a program are executed by the 8087 and others by the CPU is usually of no concern to the programmer. All 8086/8088 addressing modes may be used to access memory-based 8087 operands, enabling convenient processing of numeric arrays, structures, based variables, etc.

NDP routines may also be written in PL/M-86, Intel's high-level language for the 8086 and 8088 CPUs. PL/M-86 provides the programmer with access to many 8087 facilities while reducing the programmer's need to understand the architecture of the chip.

Two features of the 8087 hardware further simplify numeric application programming. First, the 8087 is invoked directly by the programmer's instructions. There is no need to write instructions

Table S-2. Data Types

Data Type	Bits	Significant Digits (Decimal)	Approximate Range (Decimal)
Word integer	16	4	$-32{,}768 \leq X \leq +32{,}767$
Short integer	32	9	$-2\times10^{9} \leq X \leq +2\times10^{9}$
Long integer	64	18	$-9\times10^{18} \leq X \leq +9\times10^{18}$
Packed decimal	80	18	$-99\ldots99 \leq X \leq +99\ldots99$ (18 digits)
Short real*	32	6-7	$8.43\times10^{-37} \leq \lvert X\rvert \leq 3.37\times10^{38}$
Long real*	64	15-16	$4.19\times10^{-307} \leq \lvert X\rvert \leq 1.67\times10^{308}$
Temporary real	80	19	$3.4\times10^{-4932} \leq \lvert X\rvert \leq 1.2\times10^{4932}$

*The short and long real data types correspond to the single and double precision data types defined in other Intel numerics products.

Table S-3. Principal Instructions

Class	Instructions
Data Transfer	Load (all data types), Store (all data types), Exchange
Arithmetic	Add, Subtract, Multiply, Divide, Subtract Reversed, Divide Reversed, Square Root, Scale, Remainder, Integer Part, Change Sign, Absolute Value, Extract
Comparison	Compare, Examine, Test
Transcendental	Tangent, Arctangent, 2^X-1, $Y\bullet\text{Log}_2(X+1)$, $Y\bullet\text{Log}_2(X)$
Constants	0, 1, π, $\text{Log}_{10}2$, Log_e2, Log_210, Log_2e
Processor Control	Load Control Word, Store Control Word, Store Status Word, Load Environment, Store Environment, Save, Restore, Enable Interrupts, Disable Interrupts, Clear Exceptions, Initialize

that "address" the NDP as an "I/O device", or to incur the overhead of setting up a DMA operation to perform data transfers. Second, the NDP automatically detects exception conditions that can potentially damage a calculation at run-time. On-chip exception handlers are automatically invoked by default to field these exceptions so that a reasonable result is produced and execution may proceed without program intervention. Alternatively, the 8087 can interrupt the CPU and thus trap to a user procedure when an exception is detected.

Besides the assembler and compiler, Intel provides a software emulator for the 8087. The 8087 emulator (E8087) is a software package that provides the functional equivalent of an 8087; it executes entirely on an 8086 or 8088 CPU. The emulator allows 8087 routines to be developed and checked out on an 8086/8088 execution vehicle before prototype 8087 hardware is operational. At the source code level, there is no difference between a routine that will ultimately run on an 8087 or on a CPU emulation of an 8087. At link time, the decision is made whether to use the NDP or the software emulator; no re-compilation or re-assembly is necessary. Source programs are independent of the numeric execution vehicle: except for timing, the operation of the emulated NDP is the same as for "real hardware". The emulator also makes it simple for a product to offer the NDP as a "plug-in" performance option without the necessity of maintaining two sets of source code.

Hardware Interface

As a coprocessor to an 8086 or 8088, the 8087 is wired directly to the CPU as shown in figure S-3. The CPU's queue status lines (QS0 and QS1) enable the NDP to obtain and decode instructions in synchronization with the CPU. The NDP's BUSY signal informs the CPU that the NDP is executing; the CPU WAIT instruction tests this signal to ensure that the NDP is ready to execute a subsequent instruction. The NDP can interrupt the CPU when it detects an exception. The NDP's interrupt request line is typically routed to the CPU through an 8259A Programmable Interrupt Controller.

The NDP uses one of its host CPU's request/grant lines to obtain control of the local bus for data transfers (loads and stores). The other CPU request/grant line is available for general system use, for example, by a local 8089 Input/Output Processor. A local 8089 may also be connected to the 8087's $\overline{RQ}/\overline{GT1}$ line. In this configuration, the 8087 passes the request/grant handshake signals between the CPU and the IOP when the 8087 is not in control of the local bus. When it is in control of the bus, the 8087 relinquishes the bus (at the end of the current bus cycle) upon a request from the connected IOP, giving the IOP higher priority than itself. In this way, two local 8089's can be configured in a module that also includes a CPU and an 8087.

All processors utilize the same clock generator and system bus interface components (bus controller, latches, transceivers, and bus arbiter). Thus, no additional hardware beyond the 8087 is required to add powerful computational capabilities to an 8086- or 8088-based system.

S.2 Processor Architecture

As shown in figure S-4, the NDP is internally divided into two processing elements, the control unit (CU) and the numeric execution unit (NEU). In essence, the NEU executes all numeric instructions, while the CU fetches instructions, reads and writes memory operands, and executes the processor control class of instructions. The two

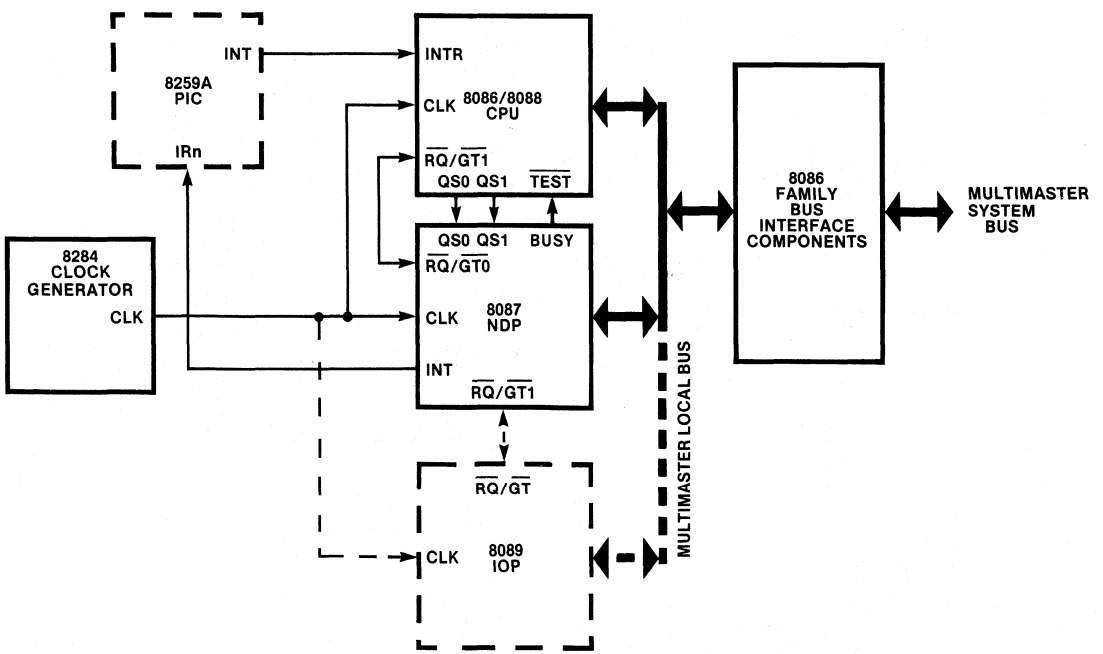


Figure S-3. NDP Interconnect

elements are able to operate independently of one another, allowing the CU to maintain synchronization with the CPU while the NEU executes numeric instructions.

Control Unit

The CU keeps the 8087 operating in synchronization with its host CPU. 8087 instructions are intermixed with CPU instructions in a single instruction stream fetched by the CPU. By monitoring the status signals emitted by the CPU, the NDP control unit can determine when an instruction is being fetched. When the instruction byte or word becomes available on the local bus, the CU taps the bus in parallel with the CPU and obtains that portion of the instruction.

The CU maintains an instruction queue that is identical to the queue in the host CPU. By monitoring the CPU's queue status lines, the CU is able to obtain and decode instructions from the queue in synchronization with the CPU. In effect, both processors fetch and decode the instruction stream in parallel.

The two processors execute the instruction stream differently, however. The first five bits of all 8087 machine instructions are identical; these bits designate the coprocessor escape (ESC) class of instructions. The control unit ignores all instructions that do not match these bits, since these instructions are directed to the CPU only. When the CU decodes an instruction containing the escape code, it either executes the instruction itself, or passes it to the NEU, depending on the type of instruction.

The CPU distinguishes between ESC instructions that reference memory and those that do not. If the instruction refers to a memory operand, the CPU calculates the operand's address and then performs a "dummy read" of the word at that location. This is a normal read cycle, except that the CPU ignores the data it receives. If the ESC instruction does not contain a memory reference, the CPU simply proceeds to the next instruction.

A given 8087 instruction (an ESC to the CPU) will either require loading an operand from memory into the 8087, or will require storing an operand from the 8087 into memory, or will not reference

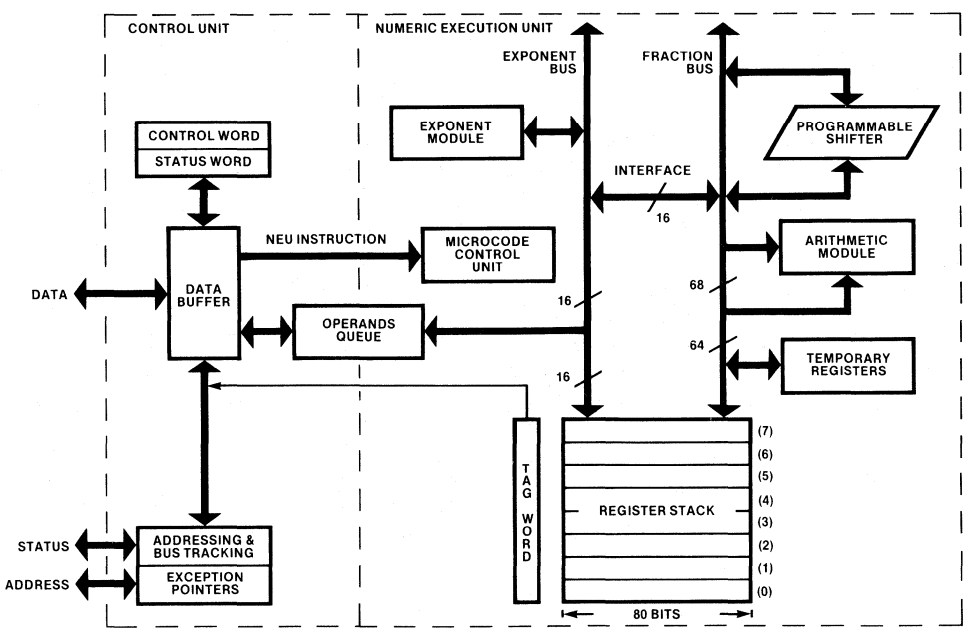


Figure S-4. 8087 Block Diagram

memory at all. In the first two cases, the CU makes use of the "dummy read" cycle initiated by the CPU. The CU captures and saves the operand address that the CPU places on the bus early in the "dummy read". If the instruction is an 8087 load, the CU additionally captures the first (and possibly only) word of the operand when it becomes available on the bus. If the operand to be loaded is longer than one word, the CU immediately obtains the bus from the CPU and reads the rest of the operand in consecutive bus cycles. In a store operation, the CU captures and saves the operand address as in a load, and ignores the data word that follows in the "dummy read" cycle. When the 8087 is ready to perform the store, the CU obtains the bus from the CPU and writes the operand at the saved address using as many consecutive bus cycles as are necessary to store the operand.

Numeric Execution Unit

The NEU executes all instructions that involve the register stack; these include arithmetic, comparison, transcendental, constant, and data transfer instructions. The data path in the NEU is 68 bits wide and allows internal operand transfers to be performed at very high speeds.

Register Stack

Each of the eight registers in the 8087's register stack is 80 bits wide, and each is divided into the "fields" shown in figure S-5. This format corresponds to the NDP's temporary real data type that is used for all calculations. Section S.3 describes in detail how numbers are represented in the temporary real format.

At a given point in time, the ST field in the status word (described shortly) identifies the current top-of-stack register. A load ("push") operation decrements ST by 1 and loads a value into the new top register. A store-and-pop operation stores the value from the current top register and then increments ST by 1. Thus, like 8086/8088 stacks in memory, the 8087 register stack grows "down" toward lower-addressed registers.

Instructions may address registers either implicitly or explicitly. Many instructions operate on the register at the top of the stack. These instructions implicitly address the register pointed to by ST.

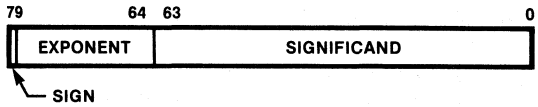


Figure S-5. Register Structure

For example, the ASM-86 instruction FSQRT replaces the number at the top of the stack with its square root; this instruction takes no operands because the top-of-stack register is implied as the operand. Other instructions allow the programmer to explicitly specify the register that is to be used. Explicit register addressing is "top-relative" where the ASM-86 expression ST denotes the current stack top and ST(*i*) refers to the *i*th register from ST in the stack ($0 \leqslant i \leqslant 7$). For example, if ST contains 011B (register 3 is the top of the stack), the following instruction would add registers 3 and 5:

```
FADD  ST, ST(2)
```

In typical use, the programmer may conceptually "divide" the registers into a fixed group and an adjustable group. The fixed registers are used like the conventional registers in a CPU, to hold constants, accumulations, etc. The adjustable group is used like a stack, with operands pushed on and results popped off. After loading, the registers in the fixed group are addressed explicitly, while those in the adjustable group are addressed implicitly. Of course, all registers may be addressed using either mode, and the "definition" of the fixed versus the adjustable areas may be altered at any time. Section S.8 contains a programming example that illustrates typical register stack use.

The stack organization and top-relative addressing of the registers simplify subroutine programming. Passing subroutine parameters on the register stack eliminates the need for the subroutine to "know" which registers actually contain the parameters and allows different routines to call the same subroutine without having to observe a convention for passing parameters in dedicated registers. So long as the stack is not full, each routine simply loads the parameters on the stack and calls the subroutine. The subroutine addresses the parameters as ST, ST(1), etc., even though ST may, for example, refer to register 3 in one invocation and register 5 in another.

Status Word

The status word reflects the overall condition of the 8087; it may be examined by storing it into memory with an NDP instruction and then inspecting it with CPU code. The status word is divided into the fields shown in figure S-6. The busy field (bit 15) indicates whether the NDP is executing an instruction (B=1) or is idle (B=0).

Several 8087 instructions (for example, the comparison instructions) post their results to the condition code (bits 14 and 10-8 of the status word). The principal use of the condition code is for conditional branching. This may be accomplished by executing an instruction that sets the condition code, storing the status word in memory and then examining the condition code with CPU instructions.

Bits 13-11 of the status word point to the 8087 register that is the current stack top (ST). Note that if ST=000B, a "push" operation, which decrements ST, produces ST=111B; similarly, popping the stack with ST=111B yields ST=000B.

Bit 7 is the interrupt request field. The NDP sets this field to record a pending interrupt to the CPU.

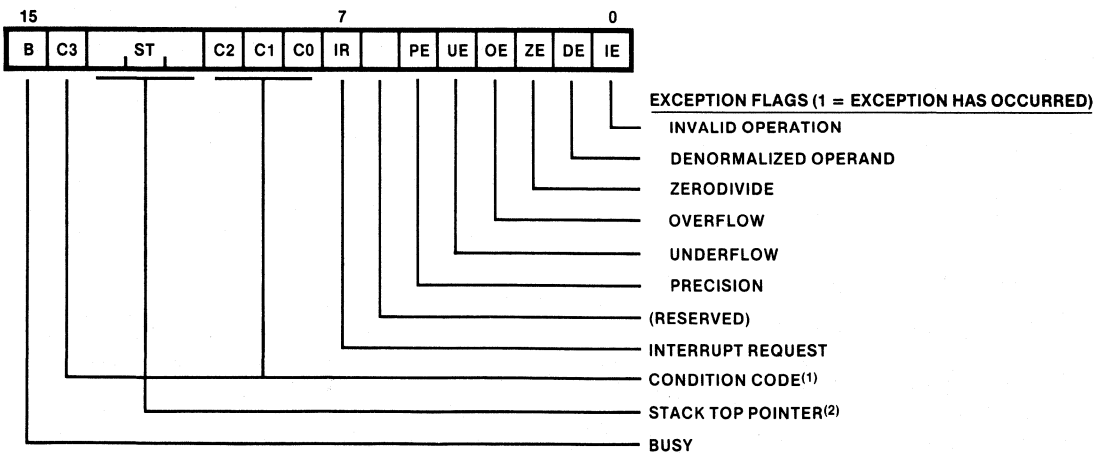
Bits 5-0 are set to indicate that the NEU has detected an exception while executing an instruction. Section S.3 explains these exceptions.

Control Word

To satisfy a broad range of application requirements, the NDP provides several processing options which are selected by loading a word from memory into the control word. Figure S-7 shows the format and encoding of the fields in the control word; it is provided here for reference. Section S.3 explains the use of each of these 8087 facilities except the interrupt-enable control field, which is covered in section S.6.

Tag Word

The tag word marks the content of each register as shown in figure S-8. The principal function



(1) See descriptions of compare, test, examine and remainder instructions in section S.7 for condition code interpretation.

(2) ST values:
000 = register 0 is stack top
001 = register 1 is stack top
•
•
•
111 = register 7 is stack top

Figure S-6. Status Word Format

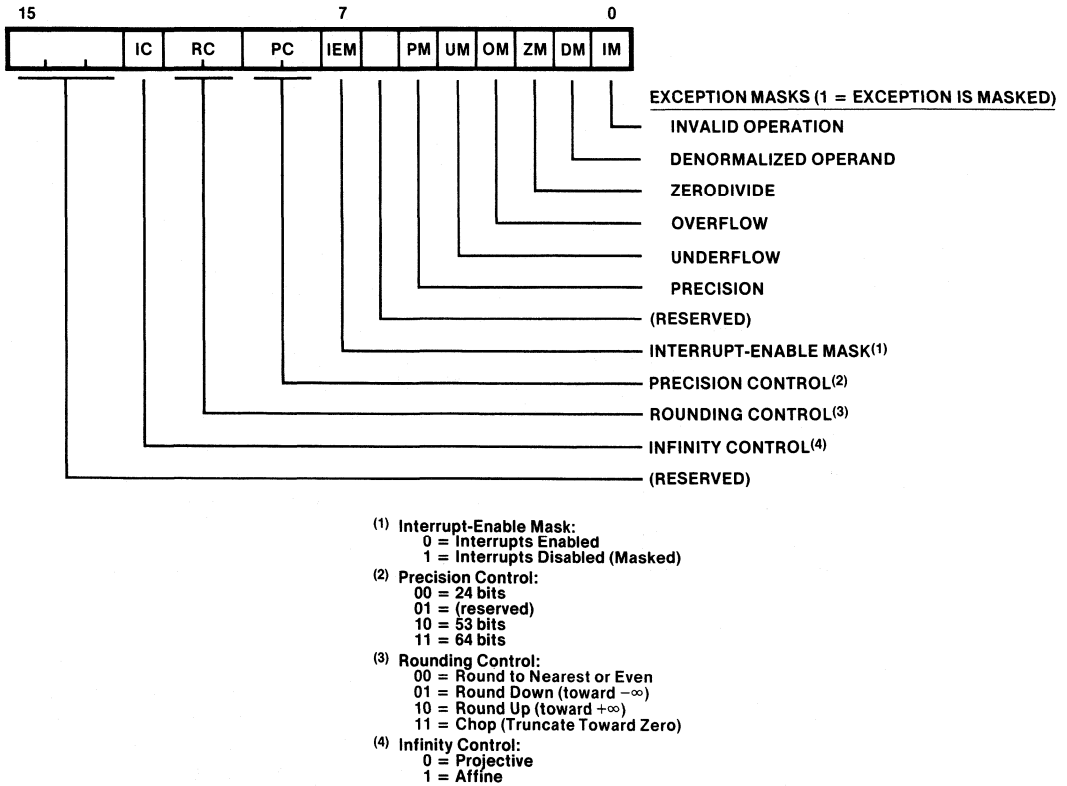


Figure S-7. Control Word Format

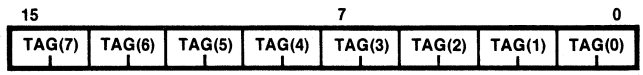
of the tag word is to optimize the NDP's performance under certain circumstances and programmers ordinarily need not be concerned with it.

Exception Pointers

The exception pointers (see figure S-9) are provided for user-written exception handlers. Whenever the 8087 executes an instruction, the CU saves the instruction address and the instruction opcode in the exception pointers. In addition, if the instruction references a memory operand, the address of the operand is retained also. An exception handler can store these pointers in memory and thus obtain information concerning the instruction that caused the exception.

S.3 Computation Fundamentals

This section covers 8087 programming concepts that are common to all applications. It describes the 8087's internal number system and the various types of numbers that can be employed in NDP programs. The most commonly used options for rounding, precision and infinity (selected by fields in the control word) are described, with exhaustive coverage of less frequently used facilities deferred to section S.9. Exception conditions which may arise during execution of NDP instructions are also described along with the options that are available for responding to these exceptions.



Tag values:
00 = Valid (Normal or Unnormal)
01 = Zero (True)
10 = Special (Not-A-Number, ∞, or Denormal)
11 = Empty

Figure S-8. Tag Word Format

(1) 20-bit physical address

(2) 11 least significant bits of opcode; 5 most significant bits are always 8087 hook (11011B)

Figure S-9. Exception Pointers Format

Number System

The system of real numbers that people use for pencil and paper calculations is conceptually infinite and continuous. There is no upper or lower limit to the magnitude of the numbers one can employ in a calculation, or to the precision (number of significant digits) that the numbers can represent. When considering any real number, there are always an infinity of numbers both larger and smaller. There is also an infinity of numbers between (i.e., with more significant digits than) any two real numbers. For example, between 2.5 and 2.6 are 2.51, 2.5897, 2.500001, etc.

While ideally it would be desirable for a computer to be able to operate on the entire real number system, in practice this is not possible. Computers, no matter how large, ultimately have fixed-size registers and memories that limit the system of numbers that can be accommodated. These limitations proscribe both the range and the precision of numbers. The result is a set of numbers that is finite and discrete, rather than infinite and continuous. This sequence is a subset of the real numbers which is designed to form a useful *approximation* of the real number system.

Figure S-10 superimposes the basic 8087 real number system on a real number line (decimal numbers are shown for clarity, although the 8087 actually represents numbers in binary). The dots indicate the subset of real numbers the 8087 can represent as data and final results of calculations. The 8087's range is approximately $\pm 4.19 \times 10^{-307}$ to $\pm 1.67 \times 10^{308}$. Applications that are required to deal with data and final results outside this range are rare. By comparison, the range of the IBM 370 is about $\pm 0.54 \times 10^{-78}$ to $\pm 0.72 \times 10^{76}$.

The finite spacing in figure S-10 illustrates that the NDP can represent a great many, but not all, of the real numbers in its range. There is always a "gap" between two "adjacent" 8087 numbers, and it is possible for the result of a calculation to fall in this space. When this occurs, the NDP rounds the true result to a number that it can represent. Thus, a real number that requires more digits than the 8087 can accommodate (e.g., a 20 digit number) is represented with some loss of accuracy. Notice also that the 8087's representable numbers are not distributed evenly along the real number line. There are, in fact, an equal number of representable numbers between successive powers of 2 (i.e., there are as many representable numbers between 2 and 4 as between 65,536 and 131,072). Therefore, the "gaps" between representable numbers are

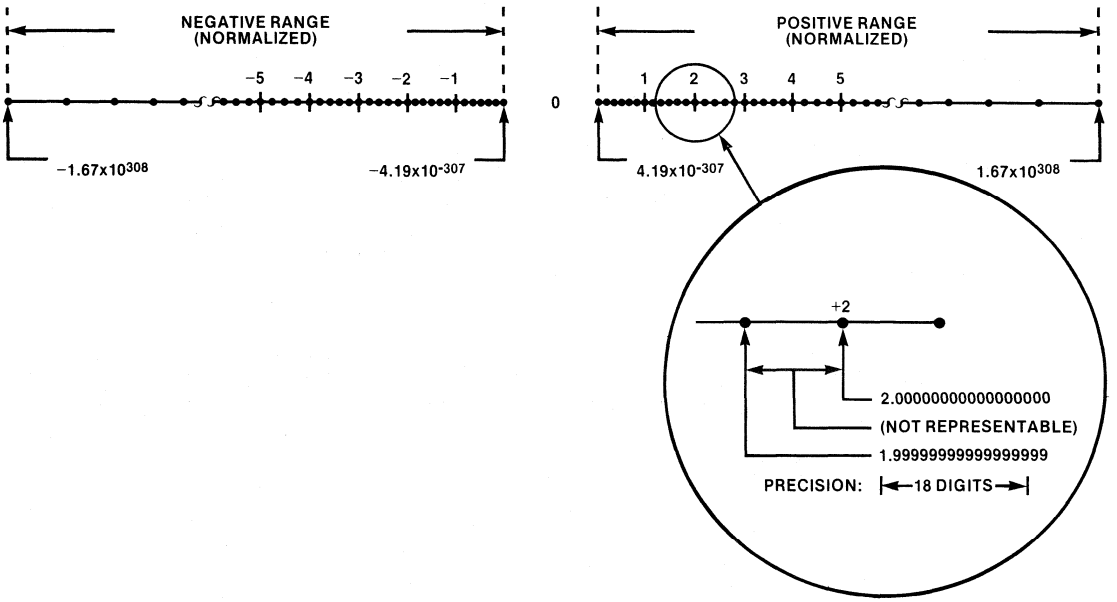


Figure S-10. 8087 Number System

"larger" as the numbers increase in magnitude. All integers in the range $\pm 2^{64}$, however, are exactly representable.

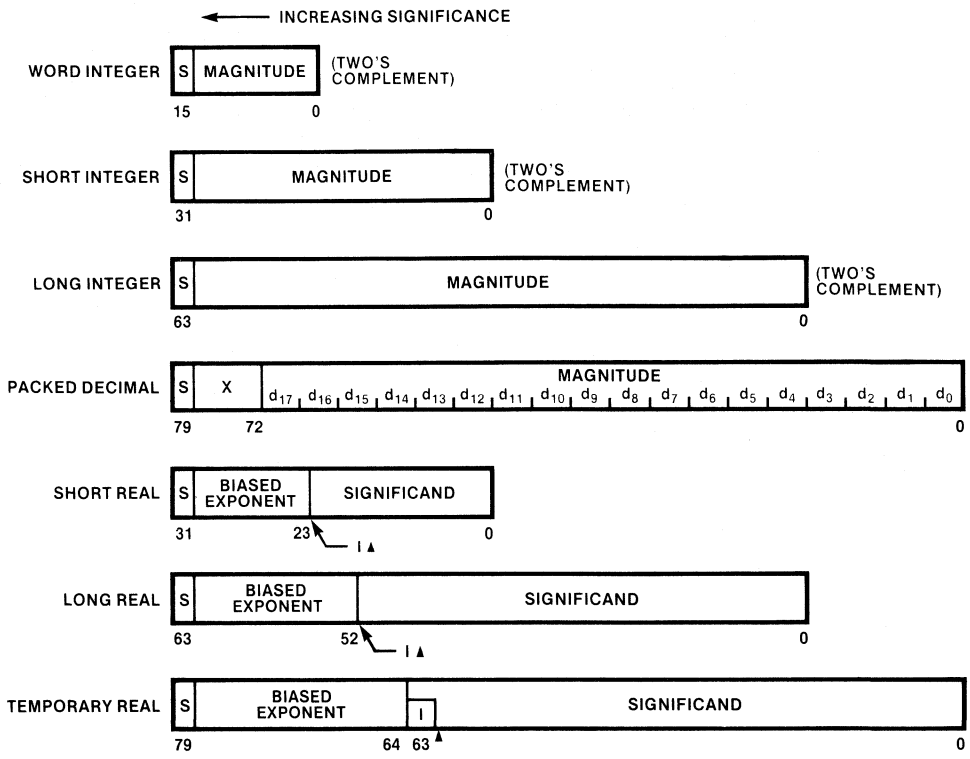
In its internal operations, the 8087 actually employs a number system that is a substantial superset of that shown in figure S-10. The internal format (called temporary real) extends the 8087's range to about $\pm 3.4 \times 10^{-4932}$ to $\pm 1.2 \times 10^{4932}$, and its precision to about 19 (equivalent decimal) digits. This format is designed to provide extra range and precision for constants and intermediate results, and is not normally intended for data or final results.

From a practical standpoint, the 8087's set of real numbers is sufficiently "large" and "dense" so as not to limit the vast majority of microprocessor applications. Compared to most computers, including mainframes, the NDP provides a very good approximation of the real number system. It is important to remember, however, that it is not an exact representation, and that arithmetic on real numbers is inherently approximate.

Conversely, and equally important, the 8087 does perform exact arithmetic on its integer subset of the reals. That is, an operation on two integers returns an exact integral result, provided that the true result is an integer and is in range. For example, $4 \div 2$ yields an exact integer, $1 \div 3$ does not, and $2^{40} \times 2^{30} + 1$ does not, because the result requires greater than 64 bits of precision.

Data Types and Formats

The 8087 recognizes seven numeric data types, divided into three classes: binary integers, packed decimal integers, and binary reals. Section S.4 describes how these formats are stored in memory (the sign is always located in the highest- addressed byte). Figure S-11 summarizes the format of each data type. In the figure, the most significant digits of all numbers (and fields within numbers) are the leftmost digits. Table S-2 provides the range and number of significant (decimal) digits that each format can accommodate.



NOTES:
S = Sign bit (0 = positive, 1 = negative)
d_n = Decimal digit (two per byte)
X = Bits have no significance; 8087 ignores when loading, zeros when storing.
▲ = Position of implicit binary point
I = Integer bit of significand; stored in temporary real, implicit in short and long real
Exponent Bias (normalized values):
Short Real: 127 (7FH)
Long Real: 1023 (3FFH)
Temporary Real: 16383 (3FFFH)

Figure S-11. Data Formats

Binary Integers

The three binary integer formats are identical except for length, which governs the range that can be accommodated in each format. The leftmost bit is interpreted as the number's sign: 0=positive and 1=negative. Negative numbers are represented in standard two's complement notation (the binary integers are the only 8087 format to use two's complement). The quantity zero is represented with a positive sign (all bits are 0). The 8087 word integer format is identical to the 16-bit signed integer data type of the 8086 and 8088.

Decimal Integers

Decimal integers are stored in packed decimal notation, with two decimal digits "packed" into each byte, except the leftmost byte, which carries the sign bit (0 = positive, 1 = negative). Negative

numbers are not stored in two's complement form and are distinguished from positive numbers only by the sign bit. The most significant digit of the number is the leftmost digit. All digits must be in the range 0H-9H.

Real Numbers

The 8087 stores real numbers in a three-field binary format that resembles scientific, or exponential, notation. The number's significant digits are held in the *significand* field, the *exponent* field locates the binary point within the significant digits (and therefore determines the number's magnitude), and the *sign* field indicates whether the number is positive or negative. (The exponent and significand are analogous to the terms "characteristic" and "mantissa" used to describe floating point numbers on some computers.) Negative numbers differ from positive numbers only in their sign bits.

Table S-4 shows how the real number 178.125 (decimal) is stored in the 8087 short real format. The table lists a progression of equivalent notations that express the same value to show how a number can be converted from one form to another. The ASM-86 and PL/M-86 language translators perform a similar process when they encounter programmer-defined real number constants. Note that not every decimal fraction has an exact binary equivalent. The decimal number 1/10, for example, cannot be expressed exactly in binary (just as the number 1/3 cannot be expressed exactly in decimal). When a translator encounters such a value, it produces a rounded binary approximation of the decimal value.

The NDP usually carries the digits of the significand in normalized form. This means that, except for the value zero, the significand is an *integer* and a *fraction* as follows:

$$1_{\Delta}fff...ff$$

where Δ indicates an assumed binary point. The number of fraction bits varies according to the real format: 23 for short, 52 for long and 63 for temporary real. By normalizing real numbers so that their integer bit is always a 1, the 8087 eliminates leading zeros in small values ($|x| < 1$). This technique maximizes the number of significant digits that can be accommodated in a significand of a given width. Note that in the short and long real formats the integer bit is *implicit* and is not actually stored; the integer bit is physically present in the temporary real format only.

If one were to examine only the significand with its assumed binary point, all normalized real numbers would have values between 1 and 2. The exponent field locates the *actual* binary point in the significant digits. Just as in decimal scientific notation, a positive exponent has the effect of moving the binary point to the right and a negative exponent effectively moves the binary point to the left, inserting leading zeros as necessary. An unbiased exponent of zero

Table S-4. Real Number Notation

Notation	Value		
Ordinary Decimal	178.125		
Scientific Decimal	$1_{\Delta}78125E2$		
Scientific Binary	$1_{\Delta}0110010001E111$		
Scientific Binary (Biased Exponent)	1_{Δ} 0110010001E10000110		
8087 Short Real (Normalized)	**Sign**	**Biased Exponent**	**Significand**
	0	10000110	01100100010000000000000 ↑ 1_{Δ} (implicit)

indicates that the position of the assumed binary point is also the position of the actual binary point. The exponent field, then, determines a real number's magnitude.

In order to simplify comparing real numbers (e.g., for sorting), the 8087 stores exponents in a biased form. This means that a constant is added to the *true exponent* described above. The value of this bias is different for each real format (see figure S-11). It has been chosen so as to force the *biased exponent* to be a positive value. This allows two real numbers (of the same format and sign) to be compared as if they are unsigned binary integers. That is, when comparing them bitwise from left to right (beginning with the left-most exponent bit), the first bit position that differs orders the numbers; there is no need to proceed further with the comparison. A number's true exponent can be determined simply by subtracting the bias value of its format.

The short and long real formats exist in memory only. If a number in one of these formats is loaded into a register, it is automatically converted to temporary real, the format used for all internal operations. Likewise, data in registers can be converted to short or long real for storage in memory. The temporary real format may be used in memory also, typically to store intermediate results that cannot be held in registers.

Most applications should use the long real form to store real number data and results; it provides sufficient range and precision to return correct results with a minimum of programmer attention. The short real format is appropriate for applications that are constrained by memory, but it should be recognized that this format provides a smaller margin of safety. It is also useful for debugging algorithms because roundoff problems will manifest themselves more quickly in this format. The temporary real format should normally be reserved for holding intermediate results, loop accumulations, and constants. Its extra length is designed to shield final results from the effects of rounding and overflow/underflow in intermediate calculations. When the temporary real format is used to hold data or to deliver final results, the safety features built into the 8087 are compromised. Furthermore, the range and precision of the long real form are adequate for most microcomputer applications.

Special Values

Besides being able to represent positive and negative numbers, the 8087 data formats may be used to describe other entities. These special values provide extra flexibility but most users do not need to understand them in detail to use the 8087 successfully. Accordingly, they are discussed here only briefly; expanded coverage, including the bit encoding of each value, is provided in section S.9.

The value zero may be signed positive or negative in the real and decimal integer formats; the sign of a binary integer zero is always positive. The fact that zero may be signed, however, is transparent to the programmer.

The real number formats allow for the representation of the special values $+\infty$ and $-\infty$. The 8087 may generate these values as its built-in response to exceptions such as division by zero, or the attempt to store a result that exceeds the upper range limit of the destination format. Infinities may participate in arithmetic and comparison operations, and in fact the processor provides two different conceptual models for handling these special values.

If a programmer attempts an operation for which the 8087 cannot deliver a reasonable result, it will, at the programmer's discretion, either request an interrupt, or return the special value *indefinite*. Taking the square root of a negative number is an example of this type of invalid operation. The recommended action in this situation is to stop the computation by trapping to a user-written exception handler. If, however, the programmer elects to continue the computation, the specially coded *indefinite* value will propagate through the calculation and thus flag the erroneous computation when it is eventually delivered as the result. Each format has an encoding that represents the special value *indefinite*.

In the real formats, a whole range of special values, both positive and negative, is designated to represent a class of values called NAN (Not-A-Number). The special value *indefinite* is a reserved NAN encoding, but all other encodings are made available to be defined in any way by application software. Using a NAN as an operand raises the invalid operation exception, and can trap to a user-written routine to process the NAN. Alternatively, the 8087's built-in exception

Table S-5. Rounding Modes

RC Field	Rounding Mode	Rounding Action
00	Round to nearest	Closer to *b* of *a* or *c*; if equally close, select even number (the one whose least significant bit is zero).
01	Round down (toward $-\infty$)	*a*
10	Round up (toward $+\infty$)	*c*
11	Chop (toward 0)	Smaller in magnitude of *a* or *c*

Note: $a < b < c$; *a* and *c* are representable, *b* is not.

handler will simply return the NAN itself as the result of the operation; in this way NANs, including *indefinite*, may be propagated through a calculation and delivered as a final, special-valued, result. One use for NANs is to detect uninitialized variables.

As mentioned earlier, the 8087 stores non-zero real numbers in "normalized floating point" form. It also provides for storing and operating on reals that are not normalized, i.e., whose significands contain one or more leading zeros. Nonnormals arise when the result of a calculation yields a value that is too small to be represented in normal form. The leading zeros of nonnormals permit smaller numbers to be represented, at the cost of some lost precision (the number of significant digits is reduced by the leading zeros). In typical algorithms, extremely small values are most likely to be generated as intermediate, rather than final results. By using the NDP's temporary real format for holding intermediates, values as small as $\pm 3.4 \times 10^{-4932}$ can be represented; this makes the occurrence of nonnormal numbers a rare phenomenon in 8087 applications. Nevertheless, the NDP can load, store and operate on nonnormalized real numbers.

Rounding Control

Internally, the 8087 employs three extra bits (guard, round and sticky bits) which enable it to represent the infinitely precise true result of a computation; these bits are not accessible to programmers. Whenever the destination can represent the infinitely precise true result, the 8087 delivers it. Rounding occurs in arithmetic and store operations when the format of the destination cannot exactly represent the infinitely precise true result. For example, a real number may be rounded if it is stored in a shorter real format, or in an integer format. Or, the infinitely precise true result may be rounded when it is returned to a register.

The NDP has four rounding modes, selectable by the RC field in the control word (see figure S-7). Given a true result *b* that cannot be represented by the target data type, the 8087 determines the two representable numbers *a* and *c* that most closely bracket *b* in value ($a < b < c$). The processor then rounds (changes) *b* to *a* or to *c* according to the mode selected by the RC field as shown in table S-5. Rounding introduces an error in a result that is less than one unit in the last place to which the result is rounded. "Round to nearest" is the default mode and is suitable for most applications; it provides the most accurate and statistically unbiased estimate of the true result. The "chop" mode is provided for integer arithmetic applications.

"Round up" and "round down" are termed directed rounding and can be used to implement interval arithmetic. Interval arithmetic generates a certifiable result independent of the occurrence of rounding and other errors. The upper and lower bounds of an interval may be computed by executing an algorithm twice, rounding up in one pass and down in the other.

Precision Control

The 8087 allows results to be calculated with 64, 53, or 24 bits of precision as selected by the PC field of the control word. The default setting, and

the one that is best-suited for most applications, is the full 64 bits. The other settings are required by the proposed IEEE standard, and are provided to obtain compatibility with the specifications of certain existing programming languages. Specifying less precision nullifies the advantages of the temporary real format's extended fraction length, and does not improve execution speed. When reduced precision is specified, the rounding of the fraction zeros the unused bits on the right.

Infinity Control

The 8087's system of real numbers may be closed by either of two models of infinity. These two means of closing the number system, projective and affine closure, are illustrated schematically in figure S-12. The setting of the IC field in the control word selects one model or the other. The default means of closure is projective, and this is recommended for most computations. When projective closure is selected, the NDP treats the special values $+\infty$ and $-\infty$ as a single unsigned infinity (similar to its treatment of signed zeros). In the affine mode the NDP respects the signs of $+\infty$ and $-\infty$.

While affine mode may provide more information than projective, there are occasions when the sign may in fact represent misinformation. For example, consider an algorithm that yields an intermediate result x of +0 and −0 (the same numeric value) in different executions. If 1/x were then computed in affine mode, two entirely different values ($+\infty$ and $-\infty$) would result from numerically identical values of x. Projective mode, on the other hand, provides less information but never returns misinformation. In general, then, projective mode should be used globally, with affine mode reserved for local computations where the programmer can take advantage of the sign and knows for certain that the nature of the computation will not produce a misleading result.

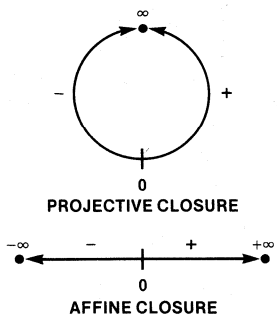


Figure S-12. Projective Versus Affine Closure

Exceptions

During the execution of most instructions, the 8087 checks for six classes of exception conditions.

The 8087 reports *invalid operation* if any of the following occurs:

- An attempt to load a register that is not empty, (e.g., stack overflow),
- An attempt to pop an operand from an empty register (e.g., stack underflow),
- An operand is a NAN,
- The operands cause the operation to be indeterminate (0/0, square root of a negative number, etc.).

An invalid operation generally indicates a program error.

If the exponent of the true result is too large for the destination real format, the 8087 signals *overflow*. Conversely, a true exponent that is too small to be represented results in the *underflow* exception. If either of these occur, the result of the operation is outside the range of the destination real format.

Typical algorithms are most likely to produce extremely large and small numbers in the calculation of intermediate, rather than final, results. Because of the great range of the temporary real format (recommended as the destination format for intermediates), overflow and underflow are relatively rare events in most 8087 applications.

If division of a finite non-zero operand by zero is attempted, the 8087 reports the *zerodivide* exception.

If an instruction attempts to operate on a denormal, the NDP reports the *denormalized* exception. This exception is provided for users who wish to implement, in software, an option of the proposed IEEE standard which specifies that operands must be prenormalized before they are used.

If the result of an operation is not exactly representable in the destination format, the 8087 rounds the number and reports the *precision* exception. This exception occurs frequently and indicates that some (generally acceptable) accuracy has been lost; it is provided for applications that need to perform exact arithmetic only.

Invalid operation, zerodivide, and denormalized exceptions are detected before an operation begins, while overflow, underflow, and precision exceptions are not raised until a true result has been computed. When a "before" exception is detected, the register stack and memory have not yet been updated, and appear as if the offending instruction has not been executed. When an "after" exception is detected, the register stack and memory appear as if the instruction has run to completion, i.e., they may be updated. (However, in a store or store and pop operation, unmasked over/underflow is handled like a "before" exception; memory is not updated and the stack is not popped.) In cases where multiple exceptions arise simultaneously, one exception is signalled according to the following precedence sequence:

- Denormalized (if unmasked),
- Invalid operation,
- Zerodivide,
- Denormalized (if masked),
- Over/underflow,
- Precision.

(The terms "masked" and "unmasked" are explained shortly.) This means, for example, that zero divided by zero will result in an invalid operation and not a zerodivide exception.

The 8087 reports an exception by setting the corresponding flag in the status word to 1. It then checks the corresponding exception mask in the control word to determine if it should "field" the exception (mask=1), or if it should issue an interrupt request to invoke a user-written exception handler (mask=0). In the first case, the exception is said to be *masked* (from user software) and the NDP executes its on-chip *masked response* for that exception. In the second case, the exception is *unmasked*, and the processor performs its *unmasked response*. The masked response always produces a standard result and then proceeds with the instruction. The unmasked response always traps to user software by interrupting the CPU (assuming the interrupt path is clear). These responses are summarized in table S-6. Section S.9 contains a complete description of all exception conditions and the NDP's masked responses.

Note that when exceptions are masked, the NDP may detect multiple exceptions in a single instruction, since it continues executing the instruction after performing its masked response. For example, the 8087 could detect a denormalized operand, perform its masked response to this exception, and then detect an underflow.

By writing different values into the exception masks of the control word, the user can accept responsibility for handling exceptions, or delegate this to the NDP. Exception handling software is often difficult to write, and the 8087's masked responses have been tailored to deliver the most "reasonable" result for each condition. The majority of applications will find that masking all exceptions other than invalid operation will yield satisfactory results with the least programming investment. An invalid operation exception normally indicates a fatal error in a program that must be corrected; this exception should not normally be masked.

The exception flags are "sticky" and can be cleared only by executing the FCLEX (clear exceptions) instruction, by reinitializing the processor, or by overwriting the flags with an FRSTOR or FLDENV instruction. This means that the flags can provide a cumulative record of the exceptions encountered in a long calculation. A program can therefore mask all exceptions (except, typically, invalid operation), run the calculation and then inspect the status word to see if any exceptions were detected at any point in the calculation. Note that the 8087 has another set of internal exception flags that it clears before each instruction. It is these flags and not those in the status word that actually trigger the 8087's exception response. The flags in the status word provide a cumulative record of exceptions for the programmer only.

If the NDP executes an unmasked response to an exception, it is assumed that a user exception handler will be invoked via an interrupt from the 8087. The 8087 sets the IR (interrupt request) bit in the status word, but this, in itself, does not guarantee an immediate CPU interrupt. The interrupt request may be blocked by the IEM (interrupt-enable mask) in the 8087 control word,

Table S-6. Exception and Response Summary

Exception	Masked Response	Unmasked Response
Invalid Operation	If one operand is NAN, return it; if both are NANs, return NAN with larger absolute value; if neither is NAN, return *indefinite*.	Request interrupt.
Zerodivide	Return ∞ signed with "exclusive or" of operand signs.	Request interrupt.
Denormalized	Memory operand: proceed as usual. Register operand: convert to valid unnormal, then re-evaluate for exceptions.	Request interrupt.
Overflow	Return properly signed ∞.	Register destination: adjust exponent*, store result, request interrupt. Memory destination: request interrupt.
Underflow	Denormalize result.	Register destination: adjust exponent*, store result, request interrupt. Memory destination: request interrupt.
Precision	Return rounded result.	Return rounded result, request interrupt.

*On overflow, 24,576 decimal is *subtracted* from the true result's exponent; this forces the exponent back into range and permits a user exception handler to ascertain the true result from the adjusted result that is returned. On underflow, the same constant is *added* to the true result's exponent.

by the 8259A Programmable Interrupt Controller, or by the CPU itself. *If any exception flag is unmasked, it is imperative that the interrupt path to the CPU is eventually cleared so that the user's software can field the exception and the offending task can resume execution.* Interrupts are covered in detail in section S.6.

A user-written exception handler takes the form of an 8086/8088 interrupt procedure. Although exception handlers will vary widely from one application to the next, most will include these basic steps:

- Store the 8087 environment (control, status and tag words, operand and instruction pointers) as it existed at the time of the exception;
- Clear the exception bits in the status word;
- Enable interrupts on the CPU;
- Identify the exception by examining the status and control words in the saved environment;
- Take application-dependent action;
- Return to the point of interruption, resuming normal execution.

Possible "application-dependent actions" include:

- Incrementing an exception counter for later display or printing;
- Printing or displaying diagnostic information (e.g., the 8087 environment and registers);
- Aborting further execution of the calculation causing the exception;
- Aborting all further execution;
- Using the exception pointers to build an instruction that will run without exception and executing it.
- Storing a diagnostic value (a NAN) in the result and continuing with the computation.

Notice that an exception may or may not constitute an error depending on the application. For example, an invalid operation caused by a stack overflow could signal an ambitious exception handler to extend the register stack to memory and continue running.

S.4 Memory

The 8087 can access any location in its host CPU's megabyte memory space. Because it relies on the CPU to generate the addresses of memory operands, the NDP can take advantage of the CPU's memory addressing modes and its ability to relocate code and data during execution.

Data Storage

Figures S-13 and S-14 show how the 8087 data types are stored in memory. The sign bit is always located in the highest-addressed byte. The least significant binary or decimal digits in a number

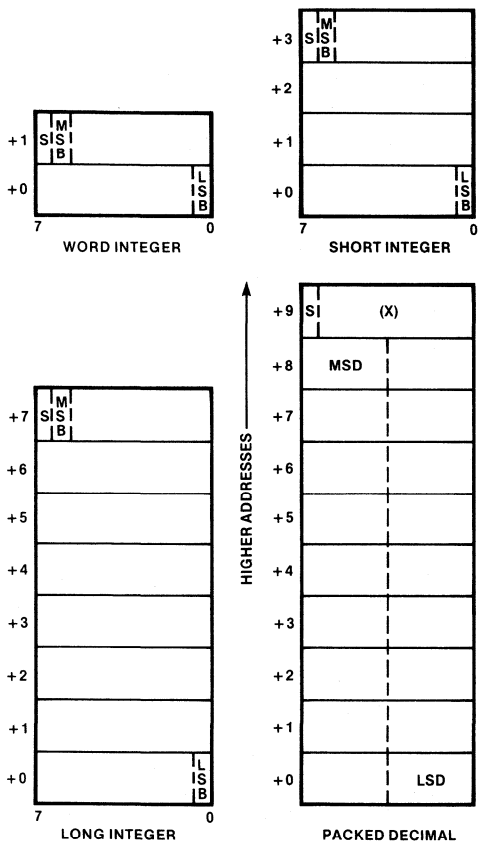


Figure S-13. Storage of Integer Data Types

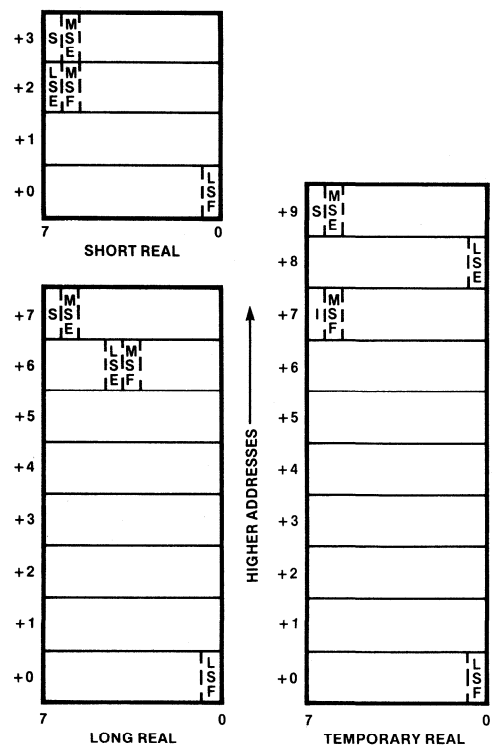


Figure S-14. Storage of Real Data Types

(or in a field in the case of reals) are those with the lowest addresses. The word integer format is stored exactly like an 8086/8088 16-bit signed integer, and is directly usable by instructions executed on either the CPU or the NDP.

A few special instructions access memory to load or store formatted processor control and state data. The formats of these memory operands are provided with the discussions of the instructions in section S.7.

Storage Access

The host CPU always generates the address of the first (lowest-addressed) byte of a memory operand. The CPU interprets an 8087 instruction that references memory as an ESC (escape), and generates the operand's effective and physical addresses normally as discussed in section 2.3. Any 8086/8088 memory addressing mode—direct, register indirect, based, indexed or based indexed—can be used to access an 8087 operand in memory. This makes the NDP easy to use with data structures such as arrays, structures, and lists.

When the CPU emits the 20-bit physical address of the memory operand, the 8087 captures the address and saves it. If the instruction loads information into the NDP, the 8087 captures the lowest-addressed word when it becomes available on the bus as a result of the CPU's "dummy read." (The "dummy read" may require either one or two bus cycles depending on the CPU type and the alignment of the operand.) If the operand is longer than one word (all 8087 operands are an integral number of words), the 8087 immediately requests use of the local bus by activating its CPU request/grant ($\overline{RQ}/\overline{GT0}$) line, as described in section S.6. When the NDP obtains the bus, it runs consecutive bus cycles incrementing the saved address until the rest of the operand has been obtained, returns the local bus to the CPU, and then executes the instruction.

If an operation stores data from the NDP to memory, the NDP and the CPU both ignore the data placed on the bus by the CPU's "dummy read." The NDP does not request the bus from the CPU until it is ready to write the result of the instruction to memory. When it obtains the bus, the NDP writes the operand in successive bus cycles, incrementing the saved address as in a load.

As described in section S.6, the 8087 automatically determines the identity of its host CPU. When the NDP is wired to an 8088, it transfers one byte per bus cycle in the same manner as the CPU. When used with an 8086, the NDP again operates like the CPU, accessing odd-addressed words in two bus cycles and even-addressed words in one bus cycle. If the 8087 is reading or writing more than one word of an odd-addressed operand in 8086 memory, it optimizes the transfer by accessing a byte on the first transfer, forcing the address to even, and then transferring words up to the last byte of the operand.

To minimize operand transfer time and 8087 use of the system bus, it is advantageous to align 8087 memory operands on even addresses when the CPU is an 8086. Following the same practice for 8088-based systems will ensure top performance without reprogramming if the application is transferred to an 8086. The ASM-86 EVEN directive can be used to force word alignment.

Dynamic Relocation

Since the host CPU takes care of both instruction fetching and memory operand addressing, the NDP may be utilized in systems that alter program addresses during execution. The only restriction on the CPU is that it should not change the address of an 8087 operand while the 8087 is executing an instruction which stores a result to that address. If this is done, the 8087 will store to the operand's old address (the one it picked up during the "dummy read").

Dedicated and Reserved Memory Locations

The 8087 does not require any addresses in memory to be set aside for special purposes. Care should be taken, however, to respect the dedicated and reserved areas associated with the CPU and the IOP (see sections 2.3 and 3.3). Using any of these areas may inhibit compatibility with current or future Intel hardware and software products.

S.5 Multiprocessing Features

As a coprocessor to an 8086 or 8088 CPU, the NDP is by definition always used in a multiprocessing environment. This section

describes the facilities built into the 8087 that simplify the coordinaton of multiple processor systems. Included are descriptions of instruction synchronization, local and system bus arbitration, and shared resource access control.

Instruction Synchronization

In the execution of a typical NDP instruction, the CPU will complete the ESC long before the 8087 finishes its interpretation of the same machine instruction. For example, the NDP performs a square root in about 180 clocks, while the CPU will execute its interpretation of this same instruction in 2 clocks. Upon completion of the ESC, the CPU will decode and execute the next instruction, and the NDP's CU, tracking the CPU, will do the same. (The NDP "executes" a CPU instruction by ignoring it). If the CPU has work to do that does not affect the NDP, it can proceed with a series of instructions while the NDP is executing in parallel; the NDP's CU will ignore these CPU-only instructions as they do not contain the 8087 escape code. This asynchronous execution of the processors can substantially improve the performance of systems that can be designed to exploit it.

There are two cases, however, when it is necessary to synchronize the execution of the CPU to the NDP:

1. An NDP instruction that is executed by the NEU must not be started if the NEU is still busy executing a previous instruction.
2. The CPU should not execute an instruction that accesses a memory operand being referenced by the NDP until the NDP has actually accessed the location.

The 8086/8088 WAIT instruction allows software to synchronize the CPU to the NDP so that the CPU will not execute the following instruction until the NDP is finished with its current (if any) instruction.

Whenever the 8087 is executing an instruction, it activates its BUSY line. This signal is wired to the CPU's $\overline{\text{TEST}}$ input as shown in figure S-3. The NDP ignores the WAIT instruction, and the CPU executes it. The CPU interprets the WAIT instruction as "wait while $\overline{\text{TEST}}$ is active." The CPU examines the $\overline{\text{TEST}}$ pin every 5 clocks; if $\overline{\text{TEST}}$ is inactive, execution proceeds with the instruction following the WAIT. If $\overline{\text{TEST}}$ is active, the CPU examines the pin again. Thus, the effective execution time of a WAIT can stretch from 3 clocks (3 clocks are required for decoding and setup) to infinity, as long as $\overline{\text{TEST}}$ remains active. The WAIT instruction, then, prevents the CPU from decoding the next instruction until the 8087 is not busy. The instruction following a WAIT is decoded simultaneously by both processors.

To satisfy the first case mentioned above, every 8087 instruction that affects the NEU should be preceded by a WAIT to ensure that the NEU is ready. All instructions except the processor control class affect the NEU. To simplify programming, the 8086 family language translators provide the WAIT automatically. When an assembly language programmer codes:

```
FMUL    ;(multiply)
FDIV    ;(divide)
```

the assembler produces *four* machine instructions, as if the programmer had written:

```
WAIT
FMUL
WAIT
FDIV
```

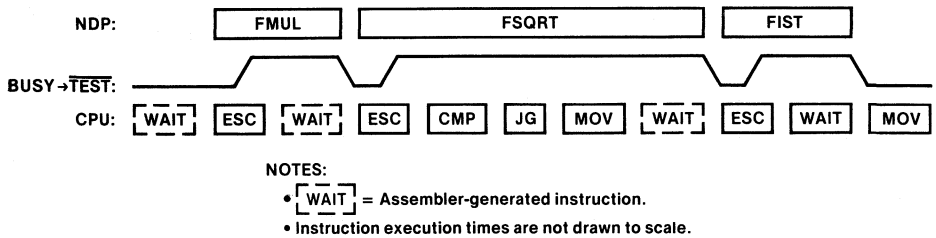
This ensures that the multiply runs to completion before the CPU and the 8087 CU decode the divide.

To satisfy the second case, the programmer should explicitly code the FWAIT instruction immediately before a CPU instruction that accesses a memory operand read or written by a previous 8087 instruction. This will ensure that the 8087 has read or written the memory operand before the CPU attempts to use it. (The FWAIT mnemonic causes the assembler to create a CPU WAIT instruction that can be eliminated at link time if the program is to run on an 8087 emulator. See section S.8 for details.)

Figure S-15 is a hypothetical sequence of instructions that illustrates the effect of the WAIT instruction and parallel execution of the NDP with a CPU.

The first two instructions in the sequence (FMUL and FSQRT) are 8087 instructions that illustrate the ASM-86 assembler's automatic generation of

```
                ;ASSUME 8087 REGISTER STACK IS LOADED WITH OPERANDS,
                ;       NEU IS NOT BUSY,
                ;       AND THAT 'ALPHA' AND 'BETA' ARE WORD
                ;       INTEGERS.
                ;
                FMUL                    ;MULTIPLY TOP STACK
                                        ;ELEMENTS
                FSQRT                   ;SQUARE ROOT OF PRODUCT
                CMP     ALPHA,100       ;ALPHA > 100?
                JG      CONTINUE        ;YES, LEAVE UNALTERED
                MOV     ALPHA,100       ;NO, SET TO 100
CONTINUE:       FIST    BETA            ;STORE ROOT AS INTEGER WORD
                FWAIT                   ;WAIT FOR 8087 TO COMPLETE
                                        ;STORE OF BETA
                MOV     AX,BETA         ;PROCEED TO PROCESS BETA
```



NOTES:
- [WAIT] = Assembler-generated instruction.
- Instruction execution times are not drawn to scale.

Figure S-15. Synchronizing Execution With WAIT

a preceding WAIT, and the effect of the WAIT when the NDP is, and is not, busy. Since the NDP is not busy when the first WAIT is encountered, the CPU executes it and immediately proceeds to the next instruction; the NDP ignores the WAIT. The next instruction is decoded simultaneously by both processors. The NDP starts the multiplication and raises its BUSY line. The CPU executes the ESC and then the second WAIT. Since $\overline{\text{TEST}}$ is active (it is tied to BUSY), the CPU effectively stretches execution of this WAIT until the NDP signals completion of the multiply by lowering BUSY. The next instruction is interpreted as a square root by the NDP and another escape by the CPU. The CPU finishes the ESC well before the NDP completes the FSQRT. This time, instead of waiting, the CPU executes three instructions (compare, jump if greater, and move) while the 8087 is working on the FSQRT. The 8087 ignores these CPU-only instructions. The CPU then encounters the third WAIT, generated by the assembler immediately preceding the FIST (store stack top into integer word). When the NDP finishes the FSQRT, both processors proceed to the next instruction, FIST to the NDP and ESC to the CPU. The CPU completes the escape quickly and then executes an explicit programmer-coded FWAIT to ensure that the 8087 has updated BETA before it moves BETA's new value to register AX.

The 8087 CU can execute most processor control instructions by itself regardless of what the NEU is doing: thus the 8087 can, in these cases, potentially execute two instructions at once. The ASM-86 assembler provides separate "wait" and "no wait" mnemonics for these instructions. For example, the instruction that sets the 8087 interrupt enable mask, and thus disables interrupts, can be coded as FDISI or FNDISI. The assembler does *not* generate a preceding WAIT if the second form is coded, so that interrupts can be disabled while the NEU is busy executing a previous instruction. The no-wait forms are principally used in exception handlers and operating systems.

Local Bus Arbitration

Whenever an NDP instruction writes data to memory, or reads more than one word from memory, the NDP forces the CPU to relinquish the local bus. It does this by means of the request/grant facility built into all 8086 family processors. For memory reads, the NDP requests the bus immediately upon the CPU's completion of its "dummy read" cycle; it follows from this that the CPU may "immediately" update a variable read by the NDP in the previous instruction with the assurance that the NDP will have obtained the old value before the CPU has altered it. For memory writes, the NDP performs as

much processing as possible before requesting the bus. In all cases, the 8087 transfers the data in back-to-back bus cycles and then immediately releases the bus.

The 8087's $\overline{RQ}/\overline{GT0}$ line is wired to one of the CPU's request/grant lines. Connecting it to $\overline{RQ}/\overline{GT1}$ on the CPU (see figure S-3) leaves the higher priority $\overline{RQ}/\overline{GT0}$ open for possible attachment of a local 8089 to the CPU. Note that an 8089 on $\overline{RQ}/\overline{GT0}$ will obtain the bus if it requests it simultaneously with an 8087 attached to $\overline{RQ}/\overline{GT1}$; it cannot, however, preempt the 8087 if the 8087 has the bus. The NDP requests the local bus by pulsing its $\overline{RQ}/\overline{GT0}$ line. If the CPU has the bus, it will grant it to the NDP by pulsing the same request/grant line. The CPU grants the bus immediately unless it is running a bus cycle, in which case the grant is delayed until the bus cycle is completed. The NDP releases the bus back to the CPU by sending a final pulse on $\overline{RQ}/\overline{GT0}$ when it has completed the transfer.

The 8087 provides a second request/grant line, $\overline{RQ}/\overline{GT1}$, that may be used to service local bus requests from an 8089 Input/Output Processor (see figure S-3). By using this line, a CPU, two IOPs (one is attached directly to the CPU) and an NDP can all reside on the same local bus, sharing a single set of system bus interface components.

When the 8087 detects a bus request pulse on $\overline{RQ}/\overline{GT1}$, its response depends on whether it is idle, executing, or running a bus cycle. If it is idle or executing, the 8087 passes the bus request through to the CPU via $\overline{RQ}/\overline{GT0}$. The subsequent grant and release pulses are also passed between the CPU and the requesting device. If the 8087 is running a bus cycle (or a series of bus cycles), it has already obtained the bus from the CPU so it grants the bus directly at the end of the current bus cycle rather than passing the request on to the CPU. When the 8089 releases the bus, the 8087 resumes the series of bus cycles it was running before it granted the bus to the 8089. Thus, to an 8089 attached to the 8087's $\overline{RQ}/\overline{GT1}$ line, the NDP appears to be a CPU. An IOP attached to an NDP also effectively has higher local bus priority than the NDP, since it can force the NDP to relinquish the bus even in the midst of a multi-cycle transfer. This satisfies the typical system requirement for I/O transfers to be serviced as soon as possible.

System Bus Arbitration

A single 8288 Bus Controller (plus latches and tranceivers as required) links both the host CPU and the NDP to the system bus. The 8087 performs system bus transfers exactly the same as its CPU; status, address, and data signals and timing are identical.

In systems that allow multiple processing modules on separate local buses common access to a public system bus, the 8087 also shares its host CPU's 8289 Bus Arbiter. The 8289 operates identically regardless of whether the system bus request is initiated by the CPU or the NDP. Since only one of the processors in the module will have control of the local bus at the time of a request to access the system bus, the transfer will be between the controlling processor and the system bus. If the 8289 does not obtain the system bus immediately, it causes the bus to appear "not ready" (as if a slow memory were being accessed), and the 8087 will stretch the bus cycle by adding the wait states.

Because it presents the same system bus interface as a maximum mode 8086 family CPU, the NDP is also electrically compatible with Intel's Multibus™ shared system bus architecture. This means that the 8087 can be utilized in systems that are based on the broad line of iSBC™ single board computers, controllers, and memories.

Controlled Variable Access

If an 8087 and a processor other than its host CPU can both update a variable, access to that variable should be controlled so that one processor at a time has exclusive rights to it. This may be implemented by a semaphore convention as described in section 2.5. However, since the 8087 has no facility for locking the system bus during an instruction, the host CPU should obtain exclusive rights to the variable before the 8087 accesses it. This can be done using an XCHG instruction prefixed by LOCK as discussed in section 2.5. When the NDP no longer needs the controlled variable the CPU should clear the semaphore to signal other processors that the variable is again available for use.

S.6 Processor Control and Monitoring

Initialization

The NDP may be initialized by hardware or software. Hardware initialization occurs in response to a pulse on the 8087's RESET line. When the processor detects RESET going active, it suspends all activities. When RESET subsequently goes inactive, the NDP initializes itself. The state of the NDP following initialization is shown in table S-7. Hardware initialization also causes the 8087 to identify its host CPU and begin to track its instruction fetches and execution. Initialization does not affect the content of the registers or of the exception pointers (these have indeterminate values immediately following power up). However, since the stack is effectively emptied by initialization (ST = 0, all registers tagged empty), the contents of the registers should normally be considered "destroyed" by initialization.

The FINIT (intialize) and FSAVE (save state) instructions also initialize the processor. Unlike a RESET pulse, software initialization does not affect the 8087's tracking of the CPU.

CPU Identification

The 8087's bidirectional $\overline{BHE}$ (bus high enable) line is tied to pin 34 of the CPU ($\overline{BHE}$ on the 8086, SS0 on the 8088). The 8088 always holds SS0 = 1 . The 8086 emits a 0 on BHE whenever it is accessing an even-addressed word or an odd-addressed byte.

Following RESET, the CPU always performs a word fetch of its first instruction from the dedicated memory location: FFFF0H. The 8087 identifies its host CPU by monitoring $\overline{BHE}$ during the CPU's first fetch following RESET. If $\overline{BHE}$ =1, the CPU is an 8088; if $\overline{BHE}$ =0, the CPU is an 8086 (because the first fetch is an even-addressed word). Note that to ensure proper operation, the same pulse must reset both the 8087 and its host CPU.

Table S-7. Processor State Following Initialization

Field	Value	Interpretation
Control Word		
Infinity Control	0	Projective
Rounding Control	00	Round to nearest
Precision Control	11	64 bits
Interrupt-enable Mask	1	Interrupts disabled
Exception Masks	111111	All exceptions masked
Status Word		
Busy	0	Not busy
Condition Code	????	(Indeterminate)
Stack Top	000	Empty stack
Interrupt Request	0	No interrupt
Exception Flags	000000	No exceptions
Tag Word		
Tags	11	Empty
Registers	N.C.	Not changed
Exception Pointers		
Instruction Code	N.C.	Not changed
Instruction Address	N.C.	Not changed
Operand Address	N.C.	Not changed

Interrupt Requests

The 8087 can request an interrupt of its host CPU via the 8087 INT (interrupt request) pin. This signal is normally routed to the CPU's INTR input via an 8259A Programmable Interrupt Controller (PIC). The 8087 should not be tied to the CPU's NMI (non-maskable interrupt) line.

All 8087 interrupt requests originate in the detection of an exception. The interrupt request logic is illustrated in figure S-16. The interrupt request is made if the exception is unmasked *and* 8087 interrupts are enabled, i.e., both the relevant exception mask and the interrupt-enable mask are clear (0). If the exception is masked, the processor executes its masked response and does not set the interrupt request bit.

If the exception is unmasked but interrupts are disabled (IEM = 1), the 8087's action depends on whether the CPU is waiting (the 8087 "knows" if the CPU is waiting because it decodes the WAIT instruction in parallel with the CPU). If the CPU is *not* waiting, the 8087 assumes that the CPU does not want to be interrupted at present and that it will enable interrupts on the 8087 when it does. The 8087 sets the interrupt request bit and holds its BUSY line active. The 8087 CU continues to track the CPU, and if an 8087 instruction (without a preceding WAIT) comes along, it will be executed. Normally in this situation the instruction would be FNENI (enable interrupts without waiting). This will clear the interrupt-enable mask and the 8087 will then activate INT. However, any instruction will be executed, and it is therefore conceivably possible to abort the interrupt request before it is ever handled. Aborting an interrupt request in this manner, however, would normally be considered a program error.

If the CPU is waiting, then the processors are in danger of entering an endless wait condition (discussed shortly). To prevent this condition, the 8087 *ignores* the fact that interrupts are disabled and activates INT even though the interrupt-enable mask is set.

The interrupt request bit remains set until it is explicitly cleared (if INT is not disabled by IEM, it will remain active also). This can be done by the FNCLEX, FNSAVE, or FNINT instructions. The interrupt procedure that fields the 8087's interrupt request, i.e., the exception handler, must clear the interrupt request bit before returning to normal execution on the 8087. If it does not, the interrupt will immediately be generated again and the program will enter an endless loop.

Interrupt Priority

Most systems can be viewed as consisting of two distinct classes of software: interrupt handlers and application tasks. Interrupt handlers execute in response to external events; in the 8086 family they are implemented as interrupt service procedures. (Of course, the CPU interrupt instructions allow interrupt handlers to respond to internal "events" also.) A hardware interrupt controller, such as the 8259A, usually monitors the external events and invokes the appropriate interrupt handler by activating the CPU INTR line, and passing a code to the CPU that identifies the interrupt handler that is to service the event. Since the 8259A typically monitors several events, a priority-resolving technique is used to select one

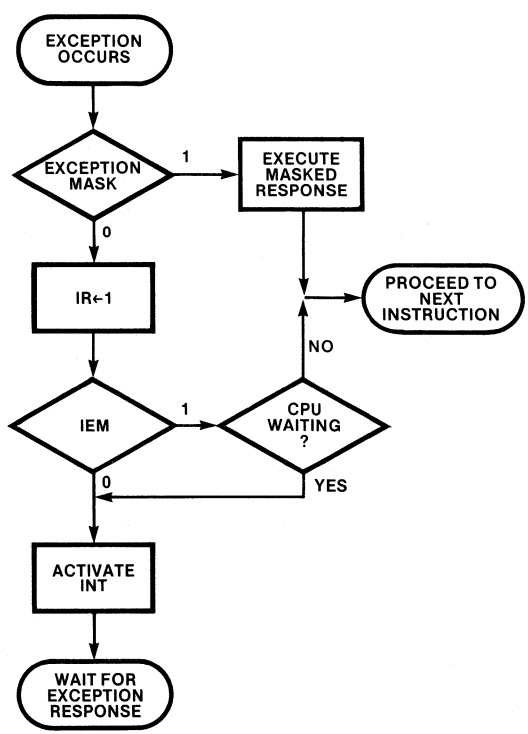


Figure S-16. Interrupt Request Logic

event when several occur simultaneously. Many systems allow higher-priority interrupts to preempt lower-priority interrupt handlers. The 8259A supports several priority-resolving techniques; a system will normally select one of these by programming the 8259A at initialization time.

Application tasks execute only when no external event needs service, i.e., when no interrupt handler is running. Application tasks are invoked by software, rather than hardware; typically a scheduling or dispatching algorithm is used to select one task for execution. In effect, any interrupt handler has higher priority than any application task, since the recognition of an interrupt will invoke the interrupt handler, preempting the application task that was running.

There are two important questions to consider when assigning a priority to the 8087's interrupt request:

- Who can cause 8087 exceptions—only application tasks, or interrupt handlers as well?
- Who should be preempted by NDP exceptions—only applications tasks, or interrupt handlers as well?

Given these considerations, the 8087 should normally be assigned the lowest priority of any interrupting device in the system. This allows the interrupt handler (i.e., the NDP exception handler) to preempt any application task that generates an 8087 exception, and at the same time prevents the exception NDP handler from interfering with other interrupt handlers.

If an *interrupt handler* uses the 8087 and requires the service of the exception handler, it can effectively "raise" the priority of the exception handler by disabling all interrupts lower than itself and higher than the 8087. Then, any unmasked exception caused by the interrupt handler will be fielded without interference from lower-priority interrupts.

If, for some reason, the 8087 must be given higher priority than another interrupt source, the interrupt handler that services the lower-priority device may want to prevent interrupts from the 8087 (which may originate in a long instruction still running on the 8087 when the interrupt handler is invoked) from preempting it. This should be done by executing the FNSTCW and FNDISI instructions before enabling CPU interrupts. Before returning, the interrupt handler should restore the original control word in the 8087 by executing FLDCW.

Users should consult "*Using the 8259A Programmable Interrupt Controller*", Intel Application Note No. AP-59, for a description of the 8259A's various modes of operation.

Endless Wait

The 8087 and its host CPU can enter an endless wait condition when the CPU is executing a WAIT instruction and a pending interrupt request from the 8087 is prevented from being recognized by the CPU. Thus, the CPU will wait for the 8087 to lower its BUSY line, while the NDP will wait for the CPU to invoke the exception handler interrupt procedure, and the task which has generated the exception will be blocked from further execution.

Figure S-17 shows the typical path of an interrupt request from the 8087 to the interrupt procedure which is designated to field NDP exceptions. The interrupt request can be potentially blocked at three points along the path, creating an endless wait if the CPU is executing a WAIT instruction. The first block can occur at the 8087's interrupt-enable mask (IEM). If this mask is set, the interrupt request is blocked except that the 8087 will override the mask if the CPU is waiting (the 8087 decodes the WAIT instruction simultaneously with the CPU). Thus, the 8087 detects and prevents one of the endless wait conditions.

A given interrupt request, IRn, can be masked on the 8259A by setting the corresponding bit in the PIC's interrupt mask register (IMR). This will prevent a request from the 8087 from being passed to the CPU. (The 8259A's normal priority-resolving activity can also block an interrupt request.) Finally, the CPU can exclude all interrupts tied to INTR by clearing its interrupt-enable flag (IF). In these two cases, the CPU can "escape" the endless wait only if another interrupt is recognized (if IF is cleared, the interrupt must arrive on NMI, the CPU's non-maskable interrupt line). Following execution of the interrupt procedure and resumption of the WAIT, the endless wait will be entered again, unless, as part of its response to the interrupt it recognizes, the CPU clears the interrupt path from the 8087.

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

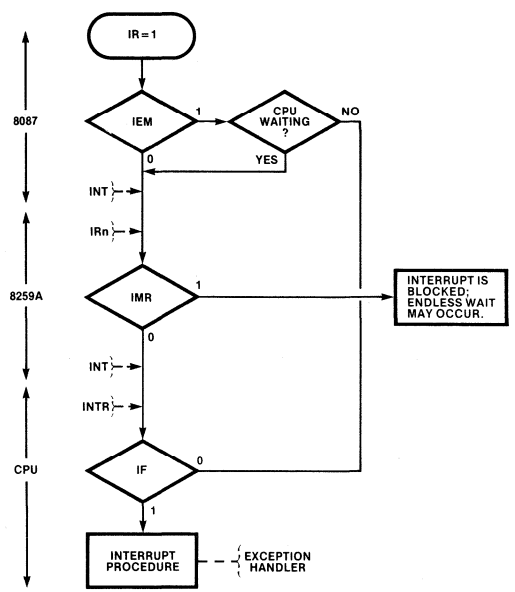


Figure S-17. Interrupt Request Path

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_Δ00...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_Δ1100...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*

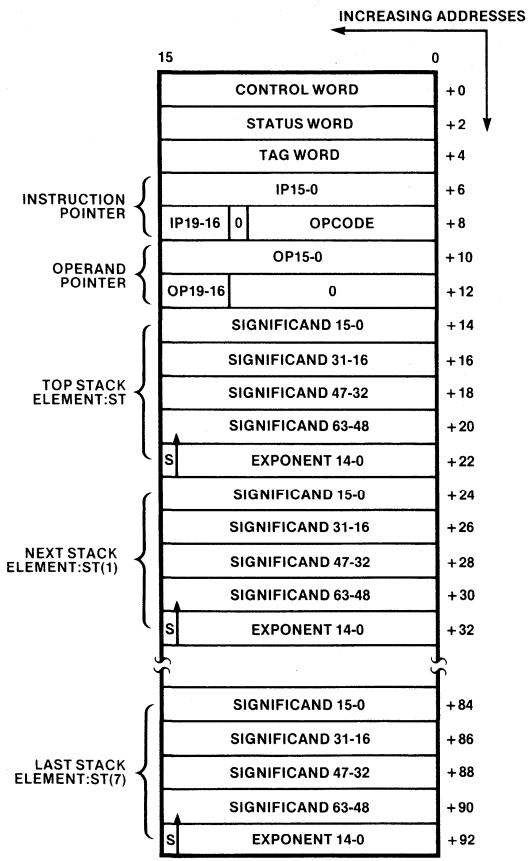
FRSTOR (restore state) reloads the 8087 from the 94-byte memory area defined by the source operand. This information should have been written by a previous FSAVE/FNSAVE instruction and not altered by any other instruction. CPU instructions (that do not reference the save image) may immediately follow 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.)

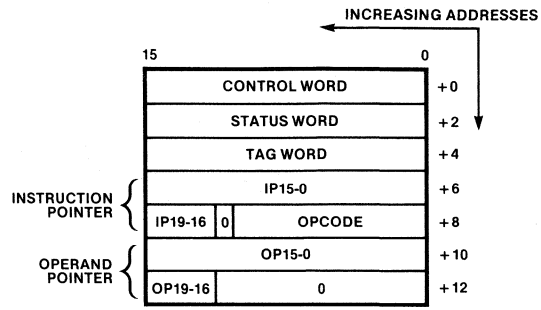


Figure S-19. FSTENV/FLDENV Memory Layout

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 ($0 \leqslant i \leqslant 7$) 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 $\overline{\text{TEST}}$ 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  1111111111000010B ; 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:  :F0: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:  :FO: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.

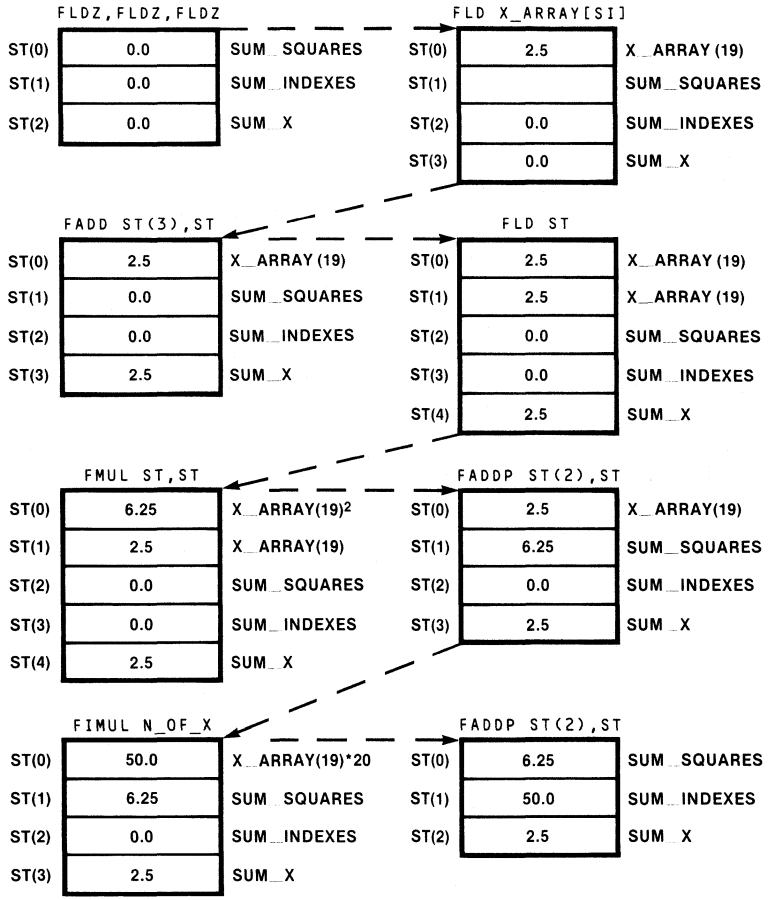


Figure S-25. Instructions and Register Stack

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_{Δ}01011100...00
Denormalize	0	−128	0_{Δ}101011100...00
Denormalize	0	−127	0_{Δ}0101011100...00
Denormalize	0	−126	0_{Δ}00101011100...00
Denormal Result(2)	0	−126	0_{Δ}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.

makes underflow very unlikely. Denormals are likely to arise only when an application generates a great many intermediates, so many that they cannot be held on the register stack or in temporary real memory variables. If storage limitations force the use of short or long reals for intermediates, and small values are produced, underflow may occur, and if masked, may generate denormals.

Accessing a denormal may produce an exception as shown in table S-24. (The denormalized exception signals that a denormal has been fetched.) Denormals may have reduced significance due to lost low-order bits, and an option of the proposed IEEE standard precludes operations on non-normalized operands. This option may be implemented in the form of an exception handler that responds to unmasked denormalized exceptions. Most users will mask this exception so that computation may proceed; any loss of accuracy will be analyzed by the user when the final result is delivered.

As table S-24 shows, the division and remainder operations do not accept denormal divisors and raise the invalid operation exception. Recall, also, that the transcendental instructions require normalized operands and do *not* check for exceptions. In all other cases, the NDP converts denormals to unnormals, and the unnormal arithmetic rules then apply.

Unnormals

An unnormal is the "descendent" of a denormal and therefore of a masked underflow response. An unnormal may exist only in the temporary real format; it may have any exponent that a normal may have, but it is distinguished from a normal by the integer bit of its significand, which is always 0. An unnormal in a register is tagged valid.

Unnormals allow arithmetic to continue following an underflow while still retaining their identity as numbers which may have reduced significance. That is, unnormal operands generate unnormal results, so long as their unnormality has a significant effect on the result. Unnormals are thus prevented from "masquerading" as normals, numbers which have full significance. On the other hand, if an unnormal has an insignificant effect on a calculation with a normal, the result will be normal. For example, adding a small unnormal to a large normal yields a normal result. The converse situation yields an unnormal.

Table S-25 shows how the instruction set deals with unnormal operands. Note that the unnormal may be the original operand or a temporary created by the 8087 from a denormal.

Table S-24. Exceptions Due to Denormal Operands

Operation	Exception	Masked Response
FLD (short/long real)	D	Load as equivalent unnormal
arithmetic (except following)	D	Convert (in a work area) denormal to equivalent unnormal and proceed
Compare and test	D	Convert (in a work area) denormal to equivalent unnormal and proceed
Division or FPREM with denormal divisor	I	Return real *indefinite*

Table S-25. Unnormal Operands and Results

Operation	Result
Addition/subtraction	Normalization of operand with larger absolute value determines normalization of result.
Multiplication	If either operand is unnormal, result is unnormal.
Division (unnormal dividend only)	Result is unnormal.
FPREM (unnormal dividend only)	Result is normalized.
Division/FPREM (unnormal divisor)	Signal invalid operation.
Compare/FTST	Normalize as much as possible before making comparison.
FRNDINT	Normalize as much as possible before rounding.
FSQRT	Signal invalid operation.
FST, FSTP (short/long real destination)	If value is above destination's underflow boundary, then signal invalid operation; else signal underflow.
FSTP (temporary real destination)	Store as usual.
FIST, FISTP, FBSTP	Signal invalid operation.
FLD	Load as usual.
FXCH	Exchange as usual.
Transcendental instructions	Undefined; operands must be normal and are not checked.

Zeros and Pseudo-Zeros

As discussed in section S.3, the real and packed decimal data types support signed zeros, while the binary integers represent a single zero, signed positive. The signed zeros behave, however, as though they are a single unsigned quantity. If necessary, the FXAM instruction may be used to determine a zero's sign.

The zeros discussed above are called true zeros; if one of them is loaded or generated in a register, the register is tagged zero. Table S-26 lists the results of instructions executed with zero operands and also shows how a true zero may be created from nonzero operands. (Nonzero operands are denoted "X" or "Y" in the table.)

Only the temporary real format may contain a special class of values called pseudo-zeros. A pseudo-zero is an unnormal whose significand is all zeros, but whose (biased) exponent is nonzero (true zeros have a zero exponent). Neither is a pseudo-zero's exponent all ones, since this encoding is reserved for infinities and NANs. A pseudo-zero result will be produced if two unnormals, containing a total of more than 64 leading zero bits in their significands, are multiplied together. This is a remote possibility in most applications, but it can happen.

Table S-26. Zero Operands and Results

Operation/Operands	Result	Operation/Operands	Result
FLD, FBLD [1]		Division	
+0	+0	±0 ÷ ±0	Invalid operation
−0	−0	±X ÷ ±0	Zerodivide
FILD [2]		+0 ÷ +X, −0 ÷ −X	+0
+0	+0	+0 ÷ −X, −0 ÷ +X	−0
FST, FSTP		−X ÷ −Y, +X ÷ +Y	+0, underflow [8]
+0	+0	−X ÷ +Y, +X ÷ −Y	−0, underflow [8]
−0	−0		
+X [3]	+0	FPREM	
−X [3]	−0	±0 rem ±0	Invalid operation
FBSTP		±X rem ±0	Invalid operation
+0	+0	+0 rem +X, +0 rem −X	+0
−0	−0	−0 rem +X, −0 rem −X	−0
FIST, FISTP		+X rem +Y, +X rem −Y	+0 [9]
+0	+0	−X rem −Y, −X rem +Y	−0 [9]
−0	+0		
+X [4]	+0	FSQRT	
−X [4]	+0	−0	−0
		+0	+0
Addition			
+0 plus +0	+0	Compare	
−0 plus −0	−0	±0 : +X	A < B
+0 plus −0, −0 plus +0	*0 [5]	±0 : ±0	A = B
−X plus +X, +X plus −X	*0 [5]	±0 : −X	A > B
±0 plus ±X, ±X plus ±0	†X [6]		
		FTST	
Subtraction		±0	Zero
+0 minus −0	+0	FCHS	
−0 minus +0	−0	+0	−0
+0 minus +0, −0 minus −0	*0 [5]	−0	+0
+X minus +X, −X minus −X	*0 [5]	FABS	
±0 minus ±X, ±X minus ±0	†X [6]	±0	+0
		F2XM1	
Multiplication		+0	+0
+0 • +0, −0 • −0	+0	−0	−0
+0 • −0, −0 • +0	−0	FRNDINT	
+0 • +X, +X • +0	+0	+0	+0
+0 • −X, −X • +0	−0	−0	−0
−0 • +X, +X • −0	−0	FXTRACT	
−0 • −X, −X • −0	+0	+0	Both +0
+X • +Y, −X • −Y	+0, underflow [7]	−0	Both −0
+X • −Y, −X • +Y	−0, underflow [7]		

Notes:

[1] Arithmetic and compare operations with real memory operands interpret the memory operand signs in the same way.

[2] Arithmetic and compare operations with binary integers interpret the integer sign in the same manner.

[3] Severe underflows in storing to short or long real may generate zeros.

[4] Small values ($|X| < 1$) stored into integers may round to zero.

[5] Sign is determined by rounding mode:
* = + for nearest, up or chop
* = − for down

[6] † = sign of X.

(7) Very small values of X and Y may yield zeros, after rounding of true result. NDP signals underflow to warn that zero has been yielded by nonzero operands.

(8) Very small X and very large Y may yield zero, after rounding of true result. NDP signals underflow to warn that zero has been yielded from nonzero operands.

(9) When Y divides into X exactly.

Pseudo-zero operands behave like unnormals, except in the following cases where they produce the same results as true zeros:

- compare and test instructions
- FRNDINT (round to integer)
- division, where the dividend is either a true zero or a pseudo-zero (the divisor is a pseudo-zero).

In addition and subtraction of a pseudo-zero and a true zero or another pseudo-zero, the pseudo-zero(s) behave like unnormals, except for the determination of the result's sign. The sign is determined as shown in table S-26 for two true zero operands.

Infinities

The real formats support signed representations of infinities. These values are encoded with a biased exponent of all ones and a significand of $1_{\Delta}00...00$; if the infinity is in a register, it is tagged special. The significand distinguishes infinities from NANs, including real *indefinite*.

A programmer may code an infinity, or it may be created by the NDP as its masked response to an overflow or a zerodivide exception. Note that when rounding is up or down, the masked response may create the largest valid value representable in the destination rather than infinity. See table S-33 for details. As operands, infinities behave somewhat differently depending on how the infinity control field in the control word is set (see table S-27). When the projective model of infinity is selected, the infinities behave as a single unsigned representation; because of this, infinity cannot be compared with any value except infinity. In affine mode, the signs of the infinities are observed, and comparisons are possible.

Table S-27. Infinity Operands and Results

Operation	Projective Result	Affine Result
Addition		
$+\infty$ plus $+\infty$	Invalid operation	$+\infty$
$-\infty$ plus $-\infty$	Invalid operation	$-\infty$
$+\infty$ plus $-\infty$	Invalid operation	Invalid operation
$-\infty$ plus $+\infty$	Invalid operation	Invalid operation
$\pm\infty$ plus $\pm X$	$*\infty$	$*\infty$
$\pm X$ plus $\pm\infty$	$*\infty$	$*\infty$
Subtraction		
$+\infty$ minus $-\infty$	Invalid operation	$+\infty$
$-\infty$ minus $+\infty$	Invalid operation	$-\infty$
$+\infty$ minus $+\infty$	Invalid operation	Invalid operation
$-\infty$ minus $-\infty$	Invalid operation	Invalid operation
$\pm\infty$ minus $\pm X$	$*\infty$	$*\infty$
$\pm X$ minus $\pm\infty$	$\dagger\infty$	$\dagger\infty$
Multiplication		
$\pm\infty \bullet \pm\infty$	$\oplus\infty$	$\oplus\infty$
$\pm\infty \bullet \pm Y$	$\oplus\infty$	$\oplus\infty$
$\pm 0 \bullet \pm\infty$, $\pm\infty * \pm 0$	Invalid operation	Invalid operation

Table S-27. Infinity Operands and Results (Cont'd.)

Operation	Projective Result	Affine Result
Division		
$\pm\infty \div \pm\infty$	Invalid operation	Invalid operation
$\pm\infty \div \pm X$	$\oplus\infty$	$\oplus\infty$
$\pm X \div \pm\infty$	$\oplus 0$	$\oplus 0$
FSQRT		
$-\infty$	Invalid operation	Invalid operation
$+\infty$	Invalid operaton	$+\infty$
FPREM		
$\pm\infty$ rem $\pm\infty$	Invalid operation	Invalid operation
$\pm\infty$rem $\pm X$	Invalid operation	Invalid operation
$\pm Y$ rem $\pm\infty$	*Y	*Y
± 0 rem $\pm\infty$	*0	*0
FRNDINT		
$\pm\infty$	*∞	*∞
FSCALE		
$\pm\infty$ scaled by $\pm\infty$	Invalid operation	Invalid operation
$\pm\infty$ scaled by $\pm X$	*∞	*∞
± 0 scaled by $\pm\infty$	*0	*0
$\pm Y$ scaled by $\pm\infty$	Invalid operation	Invalid operation
FXTRACT		
$\pm\infty$	Invalid operation	Invalid operation
Compare		
$\pm\infty : \pm\infty$	A = B	$-\infty < +\infty$
$\pm\infty : \pm Y$	A ? B (and) invalid operation	$-\infty < Y < +\infty$
$\pm\infty : \pm 0$	A ? B (and) invalid operation	$-\infty < 0 < +\infty$
FTST		
$\pm\infty$	A ? B (and) invalid operation	*∞

Notes: X = zero or nonzero operand

Y = nonzero operand

* = sign of original operand

† = sign is complement of original operand's sign

⊕= sign is "exclusive or" original operand signs (+ if operands had same sign, − if operands had different signs)

NANs

A NAN (Not-A-Number) is a member of a class of special values that exist in the real formats only. A NAN has an exponent of 11...11B, may have either sign, and may have any significand except $1_{\Delta}00...00B$, which is assigned to the infinities. A NAN in a register is tagged special.

The 8087 will generate the special NAN, real *indefinite*, as its masked response to an invalid operation exception. This NAN is signed negative; its significand is encoded $1_{\Delta}100...00$. All other NANs represent programmer-created values.

Whenever the NDP uses an operand that is a NAN, it signals invalid operation. Its masked response to this exception is to return the NAN as the operation's result. If both operands of an instruction are NANs, the result is the NAN with the larger absolute value. In this way, a NAN that enters a computation propagates through the computation and will eventually be delivered as

the final result. Note, however, that the transcendental instructions do not check their operands, and a NAN will produce an undefined result.

By unmasking the invalid operation exception, the programmer can use NANs to trap to the exception handler. The generality of this approach and the large number of NAN values that are available, provide the sophisticated programmer with a tool that can be applied to a variety of special situations.

For example, a compiler could use NANs to references to uninitialized (real) array elements. The compiler could pre-initialize each array element with a NAN whose significand contained the index (relative position) of the element. If an application program attempted to access an element that it had not initialized, it would use the NAN placed there by the compiler. If the invalid operation exception were unmasked, an interrupt would occur, and the exception handler would be invoked. The exception handler could determine which element had been accessed, since the operand address field of the exception pointers would point to the NAN, and the NAN would contain the index number of the array element.

NANs could also be used to speed up debugging. In its early testing phase a program often contains multiple errors. An exception handler could be written to save diagnostic information in memory whenever it was invoked. After storing the diagnsotic data, it could supply a NAN as the result of the erroneous instruction, and that NAN could point to its associated diagnostic area in memory. The program would then continue, creating a different NAN for each error. When the program ended, the NAN results could be used to access the diagnostic data saved at the time the errors occurred. Many errors could thus be diagnosed and corrected in one test run.

Data Type Encodings

Tables S-28 through S-31 summarize how various types of values are encoded in the seven NDP data types. In all tables, the less significant bits are to the right and are stored in the lowest memory addresses. The sign bit is always the left-most bit of the highest-addressed byte.

Notice that in every format one encoding is interpreted as representing the special value *indefinite*. The 8087 produces this encoding as its response to a masked invalid operation exception. In the case of the reals, *indefinite* can be loaded and stored like any NAN and it always retains its special identity; programmers are advised not to use this encoding for any other purpose. Packed decimal *indefinite* may be stored by the NDP in a FBSTP instruction; attempting to use this encoding in a FBLD instruction, however, will have an undefined result. In the binary integers, the same encoding may represent either *indefinite* or the largest negative number supported by the format (-2^{15}, -2^{31} or -2^{63}). The 8087 will store this encoding as its masked response to an invalid operation, or when the value in a source register represents, or rounds to, the largest negative integer representable by the destination. In situations where its origin may be ambiguous, the invalid operation exception flag can be examined to see if the value was produced by an exception response. When this encoding is loaded, or used by an integer arithmetic or compare operation, it is always interpreted as a negative number; thus *indefinite* cannot be loaded from a packed decimal or binary integer.

Table S-28. Binary Integer Encodings

	Class	Sign	Magnitude
Positives	(Largest)	0	11...11
	•	•	•
	•	•	•
	•	•	•
	(Smallest)	0	00...01
	Zero	0	00...00
Negatives	(Smallest)	1	11...11
	•	•	•
	•	•	•
	•	•	•
	(Largest/*Indefinite**)	1	00...00

Word: ←15 bits→
Short: ←31 bits→
Long: ←63 bits→

* If this encoding is used as a source operand (as in an integer load or integer arithmetic instruction), the 8087 interprets it as the largest negative number representable in the format: -2^{15}, -2^{31}, or -2^{63}. The 8087 will deliver this encoding to an integer destination in two cases:

1) if the result is the largest negative number,

2) as the response to a masked invalid operation exception, in which case it represents the special value *integer indefinite*.

Exception Handling Details

Table S-32 lists every exception condition that the NDP detects and describes the processor's response when the relevant exception mask is set. The unmasked responses are described in table S-6. Note that if an unmasked overflow or underflow occurs in an FST or FSTP instruction, no result if stored, and the stack and memory are left as they existed *before* the instruction was executed. This gives an exception handler the opportunity to examine the offending operand on the stack top.

When rounding is directed (the RC field of the control word is set to "up" or "down"), the 8087 handles a masked overflow differently than it does for the "nearest" or "chop" rounding modes. Table S-33 shows the NDP's masked response when the true result is too large to be represented in it's destination real format. For a normalized result, the essence of this response is to deliver ∞ or the largest valid number representable in the destination format, as dictated by the rounding mode and the sign of the true result. Thus, when RC=down, a positive overflow is rounded down to the largest positive number. Conversely, when RC=up, a negative overflow is rounded up to the largest negative number. A properly signed ∞ is returned for a positive overflow with RC=up, or a negative overflow with RC=down. For an unnormalized result, the action is similar except that the the unnormal character of the result is preserved if the sign and rounding mode do not indicate that ∞ should be delivered.

In all masked overflow responses for directed rounding, the overflow flag is *not* set, but the precision exception *is* raised to signal that the exact true result has not been returned.

Table S-29. Packed Decimal Encodings

Class		Sign		Magnitude																							
				digit				digit				digit				digit				. . .		digit					
Positives	(Largest)	0	0000000	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	. . .	1	0	0	1
		•	•														•								
		•	•														•								
		•	•														•								
	(Smallest)	0	0000000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	. . .	0	0	0	1
	Zero	0	0000000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	. . .	0	0	0	0
Negatives	Zero	1	0000000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	. . .	0	0	0	0
	(Smallest)	1	0000000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	. . .	0	0	0	1
		•	•														•								
		•	•														•								
		•	•														•								
	(Largest)	1	0000000	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	. . .	1	0	0	1
	*Indefinite**	1	1111111	1	1	1	1	1	1	1	1	U	U	U	U	U	U	U	U	. . .	U	U	U	U
		← 1 byte →		← 9 bytes →																							

* The *packed decimal indefinite* encoding is stored by FBSTP in response to a masked invalid operation exception. Attempting to load this value via FBLD produces an undefined result.
Note: "UUUU" means bit values are undefined and may contain any value.

Table S-30. Real and Long Real Encodings

Class			Sign	Biased Exponent	Significand* $_{\Delta}$ff...ff
Positives		NANs	0 • • • 0	11...11 • • • 11...11	11...11 • • • 00...01
		∞	0	11...11	00...00
	Reals	Normals	0 • • • 0	11...10 • • • 00...01	11...11 • • • 00...00
		Denormals	0 • • • 0	00...00 • • • 00...00	11...11 • • • 00...01
		Zero	0	00...00	00...00

Table S-30. Real and Long Real Encodings (Cont'd.)

Class			Sign	Biased Exponent	Significand* $_{\Delta}$ff...ff
Negatives	Reals	Zero	1	00...00	00...00
		Denormals	1 • • • 1	00...00 • • • 00...00	00...01 • • • 11...11
		Normals	1 • • • 1	00...01 • • • 11...10	00...00 • • • 11...11
	∞		1	11...11	00...00
	NANs		1 • • •	11...11 • • •	00...01 • • •
		Indefinite	1	11...11	10...00
			• • • 1	• • • 11...11	• • • 11...11
				Short: ← 8 bits →	← 23 bits →
				Long: ← 11 bits →	← 52 bits →

* Integer bit is implied and not stored.

Table S-31. Temporary Real Encodings

Class		Sign	Biased Exponent	Significand I_{Δ}ff..ff
Positives	NANs	0 • • • 0	11...11 • • • 11...11	111...11 • • • 100...01
	∞	0	11...11	100...00

Table S-31. Temporary Real Encodings (Cont'd.)

Class			Sign	Biased Exponent	Significand $I_{\Delta}ff...ff$
Positives	Reals		0	11...10	Normals
			•	•	111...11
			•	•	•
			•	•	•
			•	•	•
			•	•	•
			•	•	100...00
			•	•	
			•	•	Unnormals
			•	•	
			•	•	011...11
			•	•	•
			•	•	•
			•	•	•
			0	00...01	000...00
					Denormals
			0	00...00	011...11
			•	•	•
			•	•	•
			•	•	•
			0	00...00	000...01
		Zero	0	00...00	000...00
Negatives		Zero	1	00...00	000...00
					Denormals
			1	00...00	000...01
			•	•	•
			•	•	•
			•	•	•
			1	00...00	011...11
			1	00...01	Unnormals
			•	•	000...00
			•	•	•
			•	•	•
			•	•	•
			•	•	011...11
			•	•	
			•	•	Normals
			•	•	
			•	•	100...00
			•	•	•
			•	•	•
			•	•	•
			1	11...10	111...11
	∞		1	11...11	100...00

Table S-31. Temporary Real Encodings (Cont'd.)

Class			Sign	Biased Exponent	Significand $I_{\Delta}ff...ff$
Negatives			1 • • •	11...11 • • •	100...00 • • •
	NANs	*Indefinite*	1	11...11	110...00
			• • • 1	• • • 11...11	• • • 111...11
				←15 bits→	←64 bits→

Table S-32. Exception Conditions and Masked Responses

Condition	Masked Response
Invalid Operation	
Source register is tagged empty (usually due to stack underflow).	Return real *indefinite*.
Destination register is not tagged empty (usually due to stack overflow).	Return real *indefinite* (overwrite destination value).
One or both operands is a NAN.	Return NAN with larger absolute value (ignore signs).
(Compare and test operations only): one or both operands is a NAN.	Set condition codes "not comparable".
(Addition operations only): closure is affine and operands are opposite-signed infinities; or closure is projective and both operands are ∞ (signs immaterial).	Return real *indefinite*
(Subtraction operations only): closure is affine and operands are like-signed infinities; or closure is projective and both operands are ∞ (signs immaterial).	Return real *indefinite*.
(Multiplication operations only): ∞ * 0; or 0 * ∞.	Return real *indefinite*.
(Division operations only): ∞ ÷ ∞; or 0 ÷ 0; or 0 ÷ pseudo-zero; or divisor is denormal or unnormal.	Return real *indefinite*.
(FPREM instruction only): modulus (divisor) is unnormal or denormal; or dividend is ∞.	Return real *indefinite*, set condition code = "complete remainder".
(FSQRT instruction only): operand is nonzero and negative; or operand is denormal or unnormal; or closure is affine and operand is −∞; or closure is projective and operand is ∞.	Return real *indefinite*.

Exception Conditions and Masked Responses (Cont'd.)

Invalid Operation	
(Compare operations only): closure is projective and ∞ is being compared with 0 or a normal, or ∞.	Set condition code = "not comparable"
(FTST instruction only): closure is projective and operand is ∞.	Set condition code = "not comparable".
(FIST, FISTP instructions only): source register is empty, or a NAN, or denormal, or unnormal, or ∞, or exceeds representable range of destination.	Store integer *indefinite*.
(FBSTP instruction only): source register is empty, or a NAN, or denormal, or unnormal, or ∞, or exceeds 18 decimal digits.	Store packed decimal *indefinite*.
(FST, FSTP instructions only): destination is short or long real and source register is an unnormal with exponent in range.	Store real *indefinite*.
(FXCH instruction only): one or both registers is tagged empty.	Change empty register(s) to real *indefinite* and then perform exchange.
Denormalized Operand	
(FLD instruction only): source operand is denormal.	No special action; load as usual.
(Arithmetic operations only): one or both operands is denormal.	Convert (in a work area) the operand to the equivalent unnormal and proceed.
(Compare and test operations only): one or both operands is denormal *or unnormal* (other than pseudo-zero).	Convert (in a work area) any denormal to the equivalent unnormal; normalize as much as possible, and proceed with operation.
Zerodivide	
(Division operations only): divisor = 0.	Return ∞ signed with "exclusive or" of operand signs.
Overflow	
(Arithmetic operations only): rounding is nearest or chop, and exponent of true result > 16,383.	Return properly signed ∞and signal precision exception.
(FST, FSTP instructions only): rounding is nearest or chop, and exponent of true result > +127 (short real destination) or > +1023 (long real destination).	Return properly signed ∞ and signal precision exception.

Exception Conditions and Masked Responses (Cont'd.)

Underflow	
(Arithmetic operations only): exponent of true result <−16,382 (true).	Denormalize until exponent rises to −16,382 (true), round significand to 64 bits. If denormalized rounded significand = 0, then return true 0; else, return denormal (tag = special, biased exponent =0).
(FST, FSTP instructions only): destination is short real and exponent of true result <− 126 (true).	Denormalize until exponent rises to −126 (true), round significand to 24 bits, store true 0 if denormalized rounded significand = 0; else, store denormal (biased exponent = 0).
(FST, FSTP instructions only): destination is long real and exponent of true result <−1022 (true).	Denormalize until exponent rises to −1022 (true), round significand to 53 bits, store true 0 if rounded denormalized significand = 0; else, store denormal (biased exponent = 0).
Precision	
True rounding error occurs.	No special action.
Masked response to overflow exception earlier in instruction.	No special action.

Table S-33. Masked Overflow Response for Directed Rounding

True Result		Rounding Mode	Result Delivered
Normalization	Sign		
Normal	+	Up	+∞
Normal	+	Down	Largest finite positive number(1)
Normal	−	Up	Largest finite negative number(1)
Normal	−	Down	−∞
Unnormal	+	Up	+∞
Unnormal	−	Down	Largest exponent, result's significand(2)
Unnormal	+	Up	Largest exponent, result's significand(2)
Unnormal	−	Down	−∞

(1) The largest valid representable reals are encoded:
exponent: 11...10B
significand: $(1)_{\Delta}11...10B$

(2) The significand retains its identity as an unnormal; the true result is rounded as usual (effectively chopped toward 0 in this case). The exponent is encoded 11...10B.

S.10 Programming Examples

Conditional Branching

As discussed in section S.7, the comparison instructions post their results to the condition code bits of the 8087 status word. Although there are many ways to implement conditional branching following a comparison, the basic approach is as follows:

- execute the comparison,
- store the status word,
- inspect the condition code bits,
- jump on the result.

Figure S-26 is a code fragment that illustrates how two memory-resident long real numbers might be compared (similar code could be used with the FTST instruction). The numbers are called A and B, and the comparison is A to B. The comparison itself simply requires loading A onto the top of the 8087 register stack and then comparing it to B and popping the stack in the same instruction. The status word is written to memory and the code waits for completion of the store before attempting to use the result.

There are four possible orderings of A and B, and bits C3 and C0 of the condition code indicate which ordering holds. These bits are positioned in the upper byte of the status word so as to correspond to the CPU's zero and carry flags (ZF and CF), if the byte is written into the flags (see figures 2-32 and S-6). The code fragment, then, sets ZF and CF to the values of C3 and C0 and then uses the CPU conditional jumps to test the flags. Table 2-15 shows how each conditional jump instruction tests the CPU flags.

The FXAM instruction updates all four condition code bits. Figure S-27 shows how a jump table can be used to determine the characteristics of the value examined. The jump table (FXAM_TBL) is initialized to contain the 16-bit displacement of 16 labels, one for each possible condition code setting. Note that four of the table entries contain the same value, since there are four condition code settings that correspond to "empty."

The program fragment performs the FXAM and stores the status word. It then manipulates the condition code bits to finally produce a number in register BX that equals the condition code times 2. This involves zeroing the unused bits in the byte that contains the code, shifting C3 to the right so that it is adjacent to C2, and then shifting the code to multiply it by 2. The resulting value is used as an index which selects one of the displacements from FXAM_TBL (the multiplication of the condition code is required because of the 2-byte length of each value in FXAM_TBL). The unconditional JMP instruction effectively vectors through the jump table to the labelled routine that contains code (not shown in the example) to process each possible result of the FXAM instruction.

```
            .
            .
            .
A           DQ      ?
B           DQ      ?
STAT_87     DW      ?
            .
            .
            .
            FLD     A           ;LOAD A ONTO TOP OF 87 STACK
            FCOMP   B           ;COMPARE A:B, POP A
            FSTSW   STAT_87     ;STORE RESULT
            FWAIT               ;WAIT FOR STORE
```

Figure S-26. Conditional Branching for Compares

```
        ;
        ;LOAD CPU REGISTER AH WITH BYTE OF
        ;    STATUS WORD CONTAINING CONDITION CODE
        MOV     AH, BYTE PTR STAT_87+1
        ;
        ;LOAD CONDITION CODES INTO CPU FLAGS
        SAHF
        ;
        ;USE CONDITIONAL JUMPS TO DETERMINE
        ;    ORDERING OF A AND B
        JB      A_LESS_OR_UNORDERED
        ;CF (C0) = 0
        JNE     A_GREATER
A_EQUAL:
        ;CF (C0) = 0, ZF (C3) = 1
        .
        .
        .
A_GREATER:
        ;CF (C0) = 0, ZF (C3) = 0
        .
        .
        .
A_LESS_OR_UNORDERED:
        ;CF (C0) = 1, TEST ZF (C3)
        JNE     A_LESS
A_B_UNORDERED:
        ;CF (C0) = 1, ZF (C3) = 1
        .
        .
        .
A_LESS:
        ;CF (C0) = 1, ZF (C3) = 0
        .
        .
        .
```

Figure S-26. Conditional Branching for Compares (Cont'd.)

```
        .
        .
        .
FXAM_TBL            DW POS_UNNORM, POS_NAN, NEG_UNNORM,
&                       NEG_NAN, POS_NORM, POS_INFINITY,
&                       NEG_NORM, NEG_INFINITY, POS_ZERO,
&                       EMPTY, NEG_ZERO, EMPTY, POS_DENORM,
&                       EMPTY, NEG_DENORM, EMPTY
STAT_87             DW ?
```

Figure S-27. Conditional Branching for FXAM

```
            .
            .
            .
        ;EXAMINE ST, STORE RESULT, WAIT FOR COMPLETION
            FXAM
            FSTSW       STAT_87
            FWAIT
        ;CLEAR UPPER HALF OF BX, LOAD CONDITION CODE
        ;   IN LOWER HALF
            MOV         BH,0
            MOV         BL, BYTE PTR STAT_87+1
        ;COPY ORIGINAL IMAGE
            MOV         AL,BL
        ;CLEAR ALL BITS EXCEPT C2-C0
            AND         BL,00000111B
        ;CLEAR ALL BITS EXCEPT C3
            AND         AL,01000000B
        ;SHIFT C3 TWO PLACES RIGHT
            SHR         AL,1
            SHR         AL,1
        ;SHIFT C2-C0 ONE PLACE LEFT (MULTIPLY BY 2)
            SAL         BX,1
        ;DROP C3 BACK IN ADJACENT TO C2 (000XXXX0)
            OR          BL,AL
        ;JUMP TO THE ROUTINE ''ADDRESSED'' BY CONDITION CODE
            JMP         FXAM_TBL[BX]
        ;
        ;HERE ARE THE JUMP TARGETS, ONE TO HANDLE
        ;   EACH POSSIBLE RESULT OF FXAM
POS_UNNORM:
            .
            .
POS_NAN:
            .
            .
NEG_UNNORM:
            .
            .
NEG_NAN:
            .
            .
POS_NORM:
            .
            .
POS_INFINITY:
            .
            .
NEG_NORM:
            .
            .
NEG_INFINITY:
            .
```

Figure S-27. Conditional Branching for FXAM (Cont'd.)

```
                .
POS_ZERO:       .
                .
EMPTY:
                .
                .
NEG_ZERO:       .
                .
POS_DENORM:
                .
                .
NEG_DENORM:     .
                .
                .
```

Figure S-27. Conditional Branching for FXAM (Cont'd.)

Exception Handlers

There are many approaches to writing exception handlers. One useful technique is to consider the exception handler interrupt procedure as consisting of "prologue," "body" and "epilogue" sections of code. (For compatibility with the 8087 emulators, this procedure should be invoked by interrupt pointer (vector) number 16.)

At the beginning of the prologue, CPU interrupts have been disabled by the CPU's normal interrupt response mechanism. The prologue performs all functions that must be protected from possible interruption by higher-priority sources. Typically this will involve saving CPU registers and transferring diagnostic information from the 8087 to memory. When the critical processing has been completed, the prologue may enable CPU interrupts to allow higher-priority interrupt handlers to preempt the exception handler.

The exception handler body examines the diagnostic information and makes a response that is necessarily application-dependent. This response may range from halting execution, to displaying a message, to attempting to repair the problem and proceed with normal execution.

The epilogue essentially reverses the actions of the prologue, restoring the CPU and the NDP so that normal execution can be resumed. The epilogue must *not* load an unmasked exception flag into the 8087 or another interrupt will be requested immediately (assuming 8087 interrupts are also loaded as unmasked).

Figures S-28 through S-30 show the ASM-86 coding of three skeleton exception handlers. They show how prologues and epilogues can be written for various situations, but only provide comments indicating where the application-dependent exception handling body should be placed.

Figures S-28 and S-29 are very similar; their only substantial difference is their choice of instructions to save and restore the 8087. The tradeoff here is between the increased diagnostic information provided by FNSAVE and the faster execution of FNSTENV. For applications that are sensitive to interrupt latency, or do not need to examine register contents, FNSTENV reduces the duration of the "critical region," during which the CPU will not recognize another interrupt request (unless it is a non-maskable interrupt).

After the exception handler body, the epilogues prepare the CPU and the NDP to resume execution from the point of interruption (i.e., the instruction following the one that generated the unmasked exception). Notice that the exception flags in the memory image that is loaded into the 8087 are cleared to zero prior to reloading (in fact, in these examples, the entire status word

image is cleared). The prologue also provides for indicating to the interrupt controller hardware (e.g., 8259A) that the interrupt has been processed. The actual processing done here is application-dependent, but might typically involve writing an "end of interrupt" command to the interrupt controller.

The examples in figures S-28 and S-29 assume that the exception handler itself will not cause an unmasked exception. Where this is a possibility, the general approach shown in figure S-30 can be employed. The basic technique is to save the full 8087 state and then to load a new control word in the prologue. Note that considerable care should be taken when designing an exception handler of this type to prevent the handler from being reentered endlessly.

```
SAVE_ALL          PROC
;
;SAVE CPU REGISTERS, ALLOCATE STACK SPACE
;FOR 8087 STATE IMAGE
    PUSH      BP
        .
        .
        .
    MOV       BP,SP
    SUB       SP,94
;SAVE FULL 8087 STATE, WAIT FOR COMPLETION,
;ENABLE CPU INTERRUPTS
    FNSAVE    [BP-94]
    FWAIT
    STI
;
;APPLICATION-DEPENDENT EXCEPTION HANDLING
;CODE GOES HERE
;
;CLEAR EXCEPTION FLAGS IN STATUS WORD
;RESTORE MODIFIED STATE
;IMAGE
    MOV       BYTE PTR [BP-92], 0H
    FRSTOR    [BP-94]
;WAIT FOR RESTORE TO FINISH BEFORE RELEASING MEMORY
    FWAIT
;DE-ALLOCATE STACK SPACE, RESTORE CPU REGISTERS
    MOV       SP,BP
        .
        .
        .
    POP       BP
;
;CODE TO SEND ''END OF INTERRUPT'' COMMAND TO
;8259A GOES HERE
;
;RETURN TO INTERRUPTED CALCULATION
    IRET
SAVE_ALL          ENDP
```

Figure S-28. Full State Exception Handler

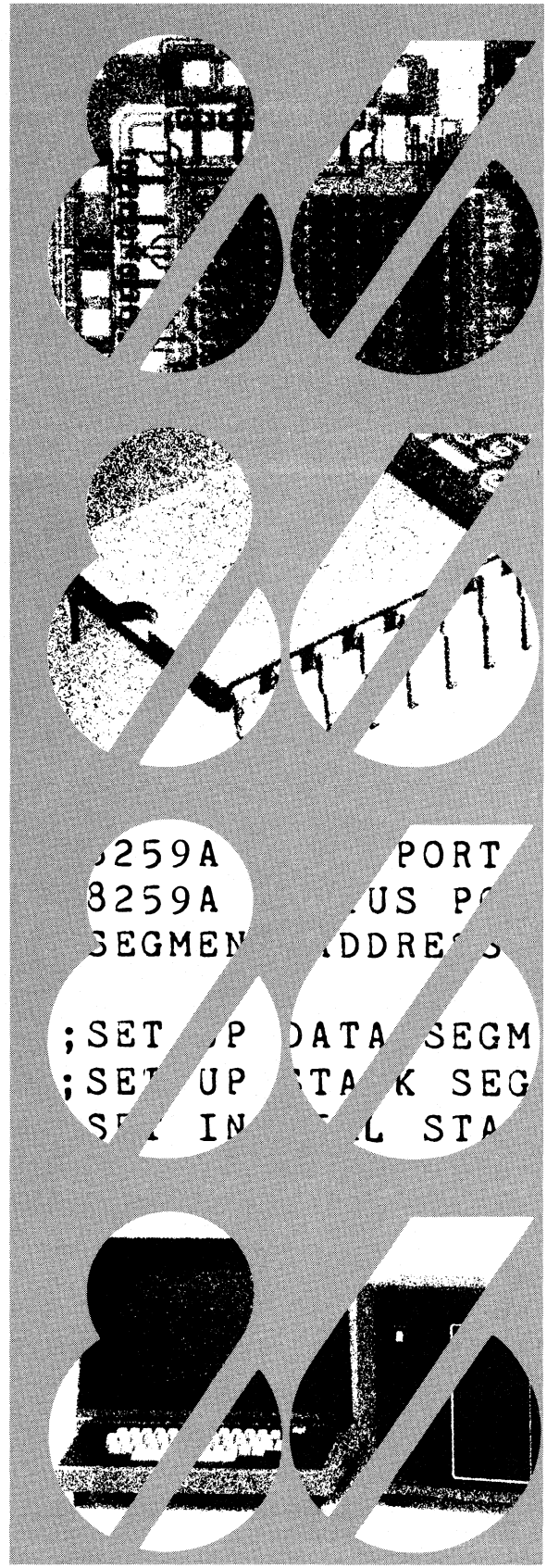
```
SAVE_ENVIRONMENT PROC
;
;SAVE CPU REGISTERS, ALLOCATE STACK SPACE
;FOR 8087 ENVIRONMENT
    PUSH    BP
      .
      .
      .
    MOV     BP,SP
    SUB     SP,14
;SAVE ENVIRONMENT, WAIT FOR COMPLETION,
;ENABLE CPU INTERRUPTS
    FNSTENV [BP-14]
    FWAIT
    STI
;
;APPLICATION EXCEPTION-HANDLING CODE GOES HERE
;
;CLEAR EXCEPTION FLAGS IN STATUS WORD
;RESTORE MODIFIED
;ENVIRONMENT IMAGE
    MOV     BYTE PTR [BP-12], 0H
    FLDENV  [BP-14]
;WAIT FOR LOAD TO FINISH BEFORE RELEASING MEMORY
    FWAIT
;DE-ALLOCATE STACK SPACE, RESTORE CPU REGISTERS
    MOV     SP,BP
      .
      .
      .
    POP     BP
;
;CODE TO SEND ''END OF INTERRUPT'' COMMAND TO
;8259A GOES HERE
;
;RETURN TO INTERRUPTED CALCULATION
    IRET
SAVE_ENVIRONMENT ENDP
```

Figure S-29. Reduced Latency Exception Handler

```
            .
            .
            .
      LOCAL_CONTROL  DW  ?  ;ASSUME INITIALIZED
            .
            .
            .
REENTRANT             PROC
;
;SAVE CPU REGISTERS, ALLOCATE STACK SPACE FOR
;8087 STATE IMAGE
    PUSH      BP
        .
        .
        .
    MOV       BP,SP
    SUB       SP,94
;SAVE STATE, LOAD NEW CONTROL WORD, WAIT
;FOR COMPLETION, ENABLE CPU INTERRUPTS
    FNSAVE    [BP-94]
    FLDCW     LOCAL_CONTROL
    FWAIT
    STI
;CODE TO SEND ''END OF INTERRUPT'' COMMAND TO
;8259A GOES HERE
        .
        .
        .
;APPLICATION EXCEPTION HANDLING CODE GOES HERE.
;AN UNMASKED EXCEPTION GENERATED HERE WILL
;CAUSE THE EXCEPTION HANDLER TO BE REENTERED.
;IF LOCAL STORAGE IS NEEDED, IT MUST BE
;ALLOCATED ON THE CPU STACK.
        .
        .
        .
;CLEAR EXCEPTION FLAGS IN STATUS WORD
;RESTORE MODIFIED STATE IMAGE
    MOV       BYTE PTR [BP-92], 0H
    FRSTOR    [BP-94]
;WAIT FOR RESTORE TO FINISH BEFORE RELEASING MEMORY
    FWAIT
;DE-ALLOCATE STACK SPACE, RESTORE CPU REGISTERS
    MOV       SP,BP
        .
        .
        .
    POP       BP
;RETURN TO POINT OF INTERRUPTION
    IRET
REENTRANT             ENDP
```

Figure S-30. Reentrant Exception Handler

Appendix A
Machine Instruction Encoding and Decoding



APPENDIX A
MACHINE INSTRUCTION ENCODING AND DECODING

8087 machine instructions assume one of five different forms as shown in table A-1. In all cases, the instructions are at least two bytes long and begin with the bit pattern 11011B, which identifies the escape class of instructions. Instructions which reference memory operands are encoded much like similar CPU instructions, since the CPU must calculate the operands effective address from the information contained in the instruction. Section 4.2 discusses this encoding scheme in more detail, and in particular, shows how each memory addressing mode is encoded.

Note that all instructions (except those coded with a "no-wait" mnemonic) are preceded by an assembler-generated CPU WAIT instruction (encoding: 10011011B). Segment override prefixes may also precede 8087 instructions in the instruction stream.

Table A-1. Instruction Encoding

	Lower-addressed Byte								Higher-addressed Byte								0, 1, or 2 bytes
(1)	1	1	0	1	1	OP-A		1	MOD		1	OP-B		R/M			DISPLACEMENT
(2)	1	1	0	1	1	FORMAT		OP-A	MOD		OP-B			R/M			DISPLACEMENT
(3)	1	1	0	1	1	R	P	OP-A	1	1	OP-B			REG			
(4)	1	1	0	1	1	0	0	1	1	1	1	OP					
(5)	1	1	0	1	1	0	1	1	1	1	1	OP					
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	

(1) Memory transfers, including applicable processor control instructions; 0, 1, or 2 displacement bytes may follow.

(2) Memory arithmetic and comparison instructions; 0, 1, or 2 displacement bytes may follow.

(3) Stack arithmetic and comparison instructions.

(4) Constant, transcendental, some arithmetic instructions.

(5) Processor control instructions that do not reference memory.

OP, OP-A, OP-B: Instruction opcode, possibly split into two fields.

MOD: Same as CPU mode field; see table 4-8.

R/M: Sames as CPU register/memory field; see table 4-10.

Table A-1. Instruction Encoding (Cont'd.)

FORMAT: Defines memory operand
- 00 = short real
- 01 = short integer
- 10 = long real
- 11 = word integer

R: 0 = return result to stack top
- 1 = return result to other register

P: 0 = do not pop stack
- 1 = pop stack after operation

REG: register stack element
- 000 = stack top
- 001 = next on stack
- 010 = third stack element, etc.

Table A-2 lists all 8087 machine instructions in binary sequence. This table may be used to "disassemble" instructions in unformatted memory dumps or instructions monitored from the data bus. Users writing exception handlers may also find this information useful to identify the offending instruction.

Table A-2. Machine Instruction Decoding Guide

1st Byte		2nd Byte	Bytes 3, 4	ASM-86 Instruction Format	
Hex	Binary				
D8	1101 1000	MOD00 0R/M	(disp-lo),(disp-hi)	FADD	short-real
D8	1101 1000	MOD00 1R/M	(disp-lo),(disp-hi)	FMUL	short-real
D8	1101 1000	MOD01 0R/M	(disp-lo),(disp-hi)	FCOM	short-real
D8	1101 1000	MOD01 1R/M	(disp-lo),(disp-hi)	FCOMP	short-real
D8	1101 1000	MOD10 0R/M	(disp-lo),(disp-hi)	FSUB	short-real
D8	1101 1000	MOD10 1R/M	(disp-lo),(disp-hi)	FSUBR	short-real
D8	1101 1000	MOD11 0R/M	(disp-lo),(disp-hi)	FDIV	short-real
D8	1101 1000	MOD11 1R/M	(disp-lo),(disp-hi)	FDIVR	short-real
D8	1101 1000	1100 0REG		FADD	ST,ST(i)
D8	1101 1000	1100 1REG		FMUL	ST,ST(i)
D8	1101 1000	1101 0REG		FCOM	ST(i)
D8	1101 1000	1101 1REG		FCOMP	ST(i)
D8	1101 1000	1110 0REG		FSUB	ST,ST(i)
D8	1101 1000	1110 1REG		FSUBR	ST,ST(i)
D8	1101 1000	1111 0REG		FDIV	ST,ST(i)
D8	1101 1000	1111 1REG		FDIVR	ST,ST(i)
D9	1101 1001	MOD00 0R/M	(disp-lo),(disp-hi)	FLD	short-real
D9	1101 1001	MOD00 1R/M		reserved	
D9	1101 1001	MOD01 0R/M	(disp-lo),(disp-hi)	FST	short-real

Table A-2. Machine Instruction Decoding Guide (Cont'd.)

1st Byte		2nd Byte		Bytes 3,4	ASM-86 Instruction Format	
Hex	Binary					
D9	1101 1001	MOD01	1R/M	(disp-lo),(disp-hi)	FSTP	short-real
D9	1101 1001	MOD10	0R/M	(disp-lo),(disp-hi)	FLDENV	14-bytes
D9	1101 1001	MOD10	1R/M	(disp-lo),(disp-hi)	FLDCW	2-bytes
D9	1101 1001	MOD11	0R/M	(disp-lo),(disp-hi)	FSTENV	14-bytes
D9	1101 1001	MOD11	1R/M	(disp-lo),(disp-hi)	FSTCW	2-bytes
D9	1101 1001	1100	0REG		FLD	ST(i)
D9	1101 1001	1100	1REG		FXCH	ST(i)
D9	1101 1001	1101	0000		FNOP	
D9	1101 1001	1101	0001		reserved	
D9	1101 1001	1101	001-		reserved	
D9	1101 1001	1101	01--		reserved	
D9	1101 1001	1101	1REG		*(1)	
D9	1101 1001	1110	0000		FCHS	
D9	1101 1001	1110	0001		FABS	
D9	1101 1001	1110	001-		reserved	
D9	1101 1001	1110	0100		FTST	
D9	1101 1001	1110	0101		FXAM	
D9	1101 1001	1110	011-		reserved	
D9	1101 1001	1110	1000		FLD1	
D9	1101 1001	1110	1001		FLDL2T	
D9	1101 1001	1110	1010		FLDL2E	
D9	1101 1001	1110	1011		FLDPI	
D9	1101 1001	1110	1100		FLDLG2	
D9	1101 1001	1110	1101		FLDLN2	
D9	1101 1001	1110	1110		FLDZ	
D9	1101 1001	1110	1111		reserved	
D9	1101 1001	1111	0000		F2XM1	
D9	1101 1001	1111	0001		FYL2X	
D9	1101 1001	1111	0010		FPTAN	
D9	1101 1001	1111	0011		FPATAN	
D9	1101 1001	1111	0100		FXTRACT	
D9	1101 1001	1111	0101		reserved	
D9	1101 1001	1111	0110		FDECSTP	
D9	1101 1001	1111	0111		FINCSTP	
D9	1101 1001	1111	1000		FPREM	
D9	1101 1001	1111	1001		FYL2XP1	
D9	1101 1001	1111	1010		FSQRT	
D9	1101 1001	1111	1011		reserved	
D9	1101 1001	1111	1100		FRNDINT	
D9	1101 1001	1111	1101		FSCALE	
D9	1101 1001	1111	111-		reserved	
DA	1101 1010	MOD00	0R/M	(disp-lo),(disp-hi)	FIADD	short-integer
DA	1101 1010	MOD00	1R/M	(disp-lo),(disp-hi)	FIMUL	short-integer
DA	1101 1010	MOD01	0R/M	(disp-lo),(disp-hi)	FICOM	short-integer
DA	1101 1010	MOD01	1R/M	(disp-lo),(disp-hi)	FICOMP	short-integer
DA	1101 1010	MOD10	0R/M	(disp-lo),(disp-hi)	FISUB	short-integer
DA	1101 1010	MOD10	1R/M	(disp-lo),(disp-hi)	FISUBR	short-integer

Table A-2. Machine Instruction Decoding Guide (Cont'd.)

1st Byte		2nd Byte		Bytes 3,4	ASM-86 Instruction Format	
Hex	Binary					
DA	1101 1010	MOD11	0R/M	(disp-lo),(disp-hi)	FIDIV	short-integer
DA	1101 1010	MOD11	1R/M	(disp-lo),(disp-hi)	FIDIVR	short-integer
DA	1101 1010	11--	----		reserved	
DB	1101 1011	MOD00	0R/M	(disp-lo),(disp-hi)	FILD	short-integer
DB	1101 1011	MOD00	1R/M	(disp-lo),(disp-hi)	reserved	
DB	1101 1011	MOD01	0R/M	(disp-lo),(disp-hi)	FIST	short-integer
DB	1101 1011	MOD01	1R/M	(disp-lo),(disp-hi)	FISTP	short-integer
DB	1101 1011	MOD10	0R/M	(disp-lo),(disp-hi)	reserved	
DB	1101 1011	MOD10	1R/M	(disp-lo),(disp-hi)	FLD	temp-real
DB	1101 1011	MOD11	0R/M	(disp-lo),(disp-hi)	reserved	
DB	1101 1011	MOD11	1R/M	(disp-lo),(disp-hi)	FSTP	temp-real
DB	1101 1011	110-	----		reserved	
DB	1101 1011	1110	0000		FENI	
DB	1101 1011	1110	0001		FDISI	
DB	1101 1011	1110	0010		FCLEX	
DB	1101 1011	1110	0011		FINIT	
DB	1101 1011	1110	01--		reserved	
DB	1101 1011	1110	1---		reserved	
DB	1101 1011	1111	----		reserved	
DC	1101 1100	MOD00	0R/M	(disp-lo),(disp-hi)	FADD	long-real
DC	1101 1100	MOD00	1R/M	(disp-lo),(disp-hi)	FMUL	long-real
DC	1101 1100	MOD01	0R/M	(disp-lo),(disp-hi)	FCOM	long-real
DC	1101 1100	MOD01	1R/M	(disp-lo),(disp-hi)	FCOMP	long-real
DC	1101 1100	MOD10	0R/M	(disp-lo),(disp-hi)	FSUB	long-real
DC	1101 1100	MOD10	1R/M	(disp-lo),(disp-hi)	FSUBR	long-real
DC	1101 1100	MOD11	0R/M	(disp-lo),(disp-hi)	FDIV	long-real
DC	1101 1100	MOD11	1R/M	(disp-lo),(disp-hi)	FDIVR	long-real
DC	1101 1100	1100	0REG		FADD	ST(i),ST
DC	1101 1100	1100	1REG		FMUL	ST(i),ST
DC	1101 1100	1101	0REG		*(2)	
DC	1101 1100	1101	1REG		*(3)	
DC	1101 1100	1110	0REG		FSUB	ST(i),ST
DC	1101 1100	1110	1REG		FSUBR	ST(i),ST
DC	1101 1100	1111	0REG		FDIV	ST(i),ST
DC	1101 1100	1111	1REG		FDIVR	ST(i),ST
DD	1101 1101	MOD00	0R/M	(disp-lo),(disp-hi)	FLD	long-real
DD	1101 1101	MOD00	1R/M		reserved	
DD	1101 1101	MOD01	0R/M	(disp-lo),(disp-hi)	FST	long-real
DD	1101 1101	MOD01	1R/M	(disp-lo),(disp-hi)	FSTP	long-real
DD	1101 1101	MOD10	0R/M	(disp-lo),(disp-hi)	FRSTOR	94-bytes
DD	1101 1101	MOD10	1R/M	(disp-lo),(disp-hi)	reserved	
DD	1101 1101	MOD11	0R/M	(disp-lo),(disp-hi)	FSAVE	94-bytes
DD	1101 1101	MOD11	1R/M	(disp-lo),(disp-hi)	FSTSW	2-bytes
DD	1101 1101	1100	0REG		FFREE	ST(i)
DD	1101 1101	1100	1REG		*(4)	
DD	1101 1101	1101	0REG		FST	ST(i)
DD	1101 1101	1101	1REG		FSTP	ST(i)

Table A-2. Machine Instruction Decoding Guide (Cont'd.)

1st Byte		2nd Byte		Bytes 3,4	ASM-86 Instruction Format	
Hex	Binary					
DD	1101 1101	111-	----		reserved	
DE	1101 1110	MOD00	0R/M	(disp-lo),(disp-hi)	FIADD	word-integer
DE	1101 1110	MOD00	1R/M	(disp-lo),(disp-hi)	FIMUL	word-integer
DE	1101 1110	MOD01	0R/M	(disp-lo),(disp-hi)	FICOM	word-integer
DE	1101 1110	MOD01	1R/M	(disp-lo),(disp-hi)	FICOMP	word-integer
DE	1101 1110	MOD10	0R/M	(disp-lo),(disp-hi)	FISUB	word-integer
DE	1101 1110	MOD10	1R/M	(disp-lo),(disp-hi)	FISUBR	word-integer
DE	1101 1110	MOD11	0R/M	(disp-lo),(disp-hi)	FIDIV	word-integer
DE	1101 1110	MOD11	1R/M	(disp-lo),(disp-hi)	FIDIVR	word-integer
DE	1101 1110	1100	0REG		FADDP	ST(i),ST
DE	1101 1110	1100	1REG		FMULP	ST(i),ST
DE	1101 1110	1101	0---		*(5)	
DE	1101 1110	1101	1000		reserved	
DE	1101 1110	1101	1001		FCOMPP	
DE	1101 1110	1101	101-		reserved	
DE	1101 1110	1101	11--		reserved	
DE	1101 1110	1110	0REG		FSUBP	ST(i),ST
DE	1101 1110	1110	1REG		FSUBRP	ST(i),ST
DE	1101 1110	1111	0REG		FDIVP	ST(i),ST
DE	1101 1110	1111	1REG		FDIVRP	ST(i),ST
DF	1101 1111	MOD00	0R/M	(disp-lo),(disp-hi)	FILD	word-integer
DF	1101 1111	MOD00	1R/M	(disp-lo),(disp-hi)	reserved	
DF	1101 1111	MOD01	0R/M	(disp-lo),(disp-hi)	FIST	word-integer
DF	1101 1111	MOD01	1R/M	(disp-lo),(disp-hi)	FISTP	word-integer
DF	1101 1111	MOD10	0R/M	(disp-lo),(disp-hi)	FBLD	packed-decimal
DF	1101 1111	MOD10	1R/M	(disp-lo),(disp-hi)	FILD	long-integer
DF	1101 1111	MOD11	0R/M	(disp-lo),(disp-hi)	FBSTP	packed-decimal
DF	1101 1111	MOD11	1R/M	(disp-lo),(disp-hi)	FISTP	long-integer
DF	1101 1111	1100	0REG		*(6)	
DF	1101 1111	1100	1REG		*(7)	
DF	1101 1111	1101	0REG		*(8)	
DF	1101 1111	1101	1REG		*(9)	
DF	1101 1111	111-	----		reserved	

* The marked encodings are *not* generated by the language translators. If, however, the 8087 encounters one one these encodings in the instruction stream, it will execute it as follows:

(1) FSTP ST(i)

(2) FCOM ST(i)

(3) FCOMP ST(i)

(4) FXCH ST(i)

(5) FCOMP ST(i)

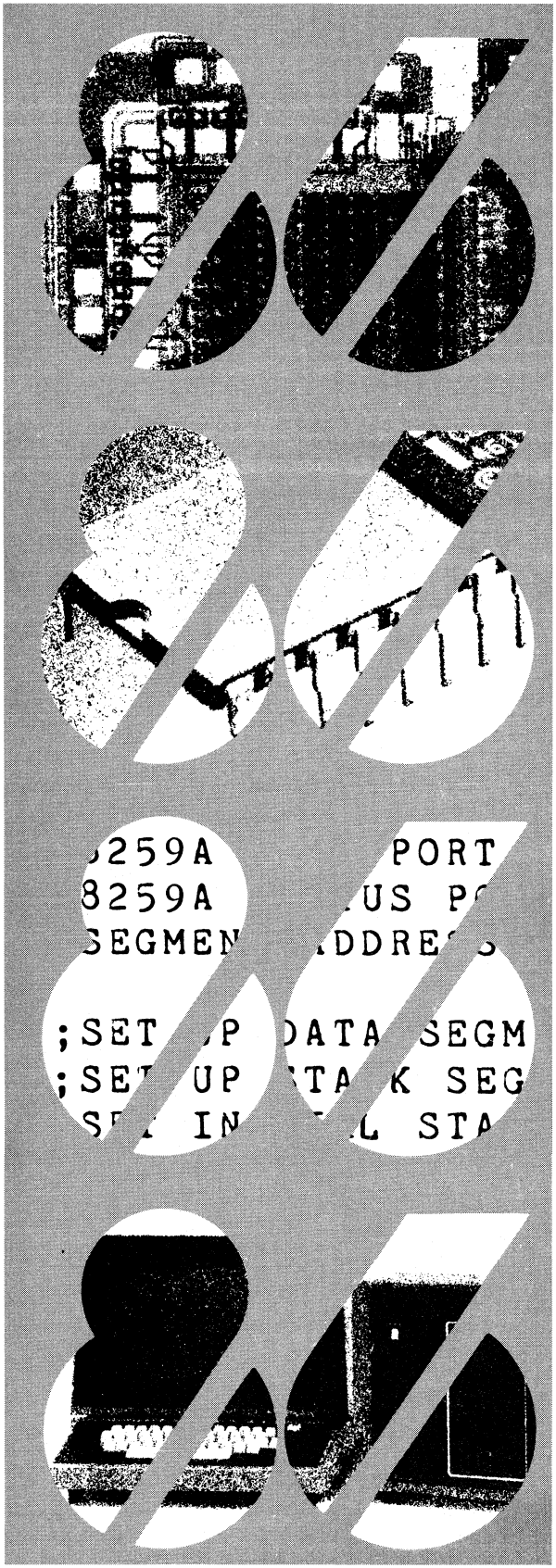
(6) FFREE ST(i) and pop stack

(7) FXCH ST(i)

(8) FSTP ST(i)

(9) FSTP ST(i)

Appendix B
Device Specification



8087
80-BIT HMOS
NUMERIC DATA PROCESSOR

- **Full Internal 80-Bit Architecture for High Performance**
- **Implements Proposed IEEE Floating Point Standard**
- **Total Numeric Support for iAPX 86/20, 88/20 Systems**
- **Expands iAPX 86/10 Datatypes to Include 32-, 64-Bit Integers, 32-, 64-, 80-Bit Floating Point, and 18-Digit BCD Operands**
- **All iAPX 86/10, 88/10 Addressing Modes Available**
- **Directly Extends iAPX 86/10, 88/10 Instruction Set to Trigonometric, Logarithmic, Exponential and Arithmetic Instructions for All Datatypes**
- **8 x 80-Bit, Individually Addressable, Numeric Register Stack**
- **Built-in Exception Handling Functions**

The Intel® 8087 is a high performance coprocessor that extends the iAPX 86/10, 88/10 architecture by providing all the required numerics support for iAPX 86/20, 88/20 systems. The 8087 is implemented in N-channel, depletion load, silicon gate technology (HMOS) and packaged in a 40-pin ceramic package. The iAPX 86/20 and 88/20 fully conform to the proposed IEEE Floating Point Standard. Using its coprocessor architecture, the 8087 adds over fifty opcodes directly into the iAPX 86/10 instruction set, making the iAPX 86/20, 88/20 a complete solution to high performance numeric processing.

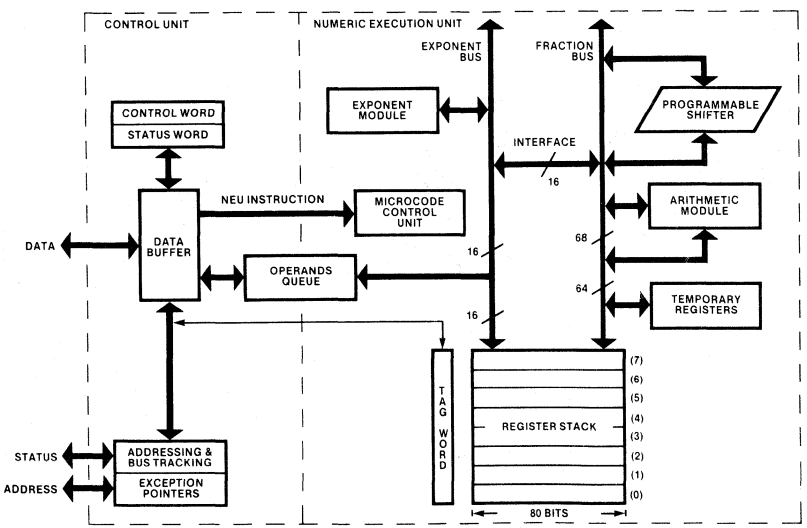


Figure 1. 8087 Block Diagram

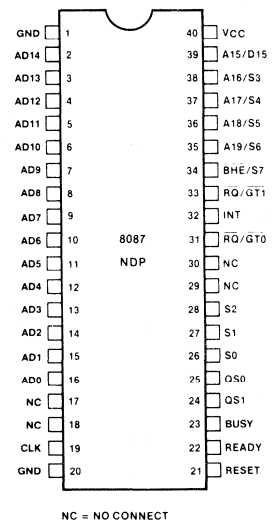


Figure 2. 8087 Pin Diagram

FUNCTIONAL DESCRIPTION

The 8087 Numeric Data Processor (NDP) is a processor extension that provides arithmetic and logical instruction support for a variety of numeric data types in iAPX 86/20, 88/20 systems. It also executes numerous built-in transcendental functions (e.g., tangent and log functions). The 8087 executes instructions as a coprocessor to a maximum mode 8086 or 8088. It effectively extends the register and instruction set of an iAPX 86/10 or 88/10 system for existing iAPX 86, 88 types and adds several new data types as well. Figure 3 presents this view of the iAPX 86/20 graphically. Essentially, the 8087 can be treated as an additional resource and an extension to the iAPX 86/10 or 88/10, providing register, datatype, control, and instruction capabilities at the hardware level that can be used as a single unified system, the iAPX 86/20, 88/20, at the programming level.

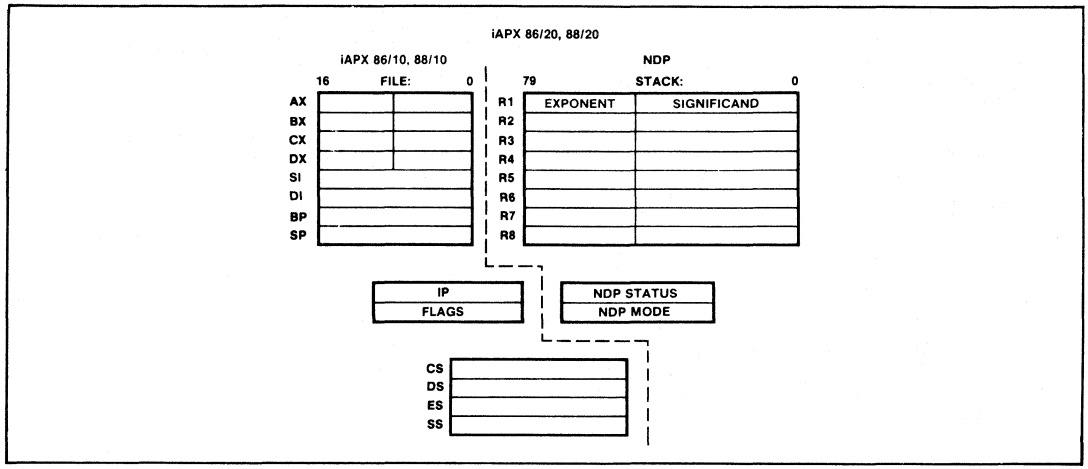


Figure 3. iAPX 86/20 Architecture

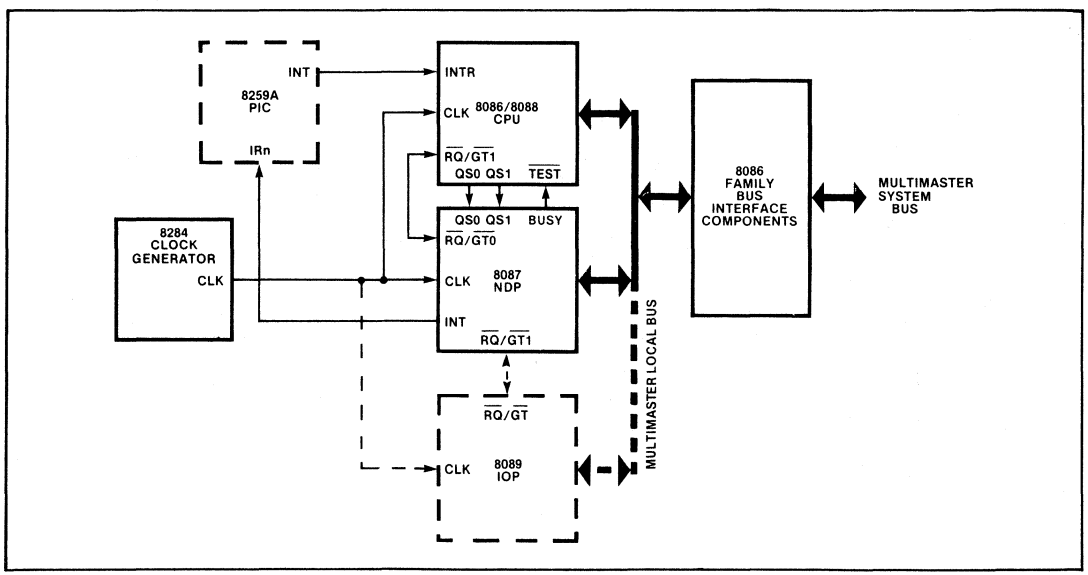


Figure 4. 8087 System Configuration

System Configuration

As a coprocessor to an 8086 or 8088, the 8087 is wired in parallel with the CPU as shown in Figure 4. The CPU's status ($\overline{S0}$-$\overline{S2}$) and queue status lines (QS0-QS1) enable the NDP to monitor and decode instructions in synchronization with the CPU and without any CPU overhead. Once started the 8087 can process in parallel with and independent of the host CPU. For resynchronization, the NDP's BUSY signal informs the CPU that the NDP is executing an instruction and the CPU WAIT instruction tests this signal to insure that the NDP is ready to execute subsequent instructions. The NDP can interrupt the CPU when it detects an error or exception. The NDP's interrupt request line is typically routed to the CPU through an 8259A Programmable Interrupt Controller. (See Figure 2 for 8087 pinout information.)

The NDP uses one of the request/grant lines of the iAPX 86, 88 architecture (typically $\overline{RQ}/\overline{GT1}$) to obtain control of the local bus for data transfers. The other request/grant line is available for general system use (for instance by an I/O processor in LOCAL mode). A bus master can also be connected to the 8087's $\overline{RQ}/\overline{GT1}$ line. In this configuration the 8087 will pass the request/grant handshake signals between the CPU and the attached master when the 8087 is not in control of the bus and will relinquish the bus to the master directly when the 8087 is in control. In this way two additional masters can be configured in an 8086/8087 system; one will share the 8086 bus with the 8087 on a first come first served basis, and the second will be guaranteed to be higher in priority than the 8087.

As Figure 4 shows, all processors utilize the same clock generator and system bus interface components (bus controller, latches, transceivers and bus arbiter).

Bus Operation

The 8087 bus structure, operation and timing are identical to all other processors in the iAPX 86, 88 series (maximum mode configuration). The address is time multiplexed with the data on the first 16/8 lines of the address/data bus. A16 through A19 are time multiplexed

Table 1. 8087 Datatypes

Data Formats	Range	Precision	Most Significant Byte (7 07 07 07 07 07 07 07 07 07 0)
Word Integer	10^4	16 Bits	I_{15} ... I_0 — Two's Complement
Short Integer	10^9	32 Bits	I_{31} ... I_0 — Two's Complement
Long Integer	10^{19}	64 Bits	I_{63} ... I_0 — Two's Complement
Packed BCD	10^{18}	18 Digits	S — D_{17} D_{16} ... D_1 D_0
Short Real	$10^{\pm 38}$	24 Bits	S E_7 E_0 F_1 ... F_{23} — F_0 Implicit
Long Real	$10^{\pm 308}$	53 Bits	S E_{10} E_0 F_1 ... F_{52} — F_0 Implicit
Temporary Real	$10^{\pm 4932}$	64 Bits	S E_{14} E_0 F_0 ... F_{63}

Integer: I

Packed BCD: $(-1)^S(D_{17} \ldots D_0)$

Real: $(-1)^S(2^{E-BIAS})(F_0 . F_1 \ldots)$

Bias = 127 for Short Real
1023 for Long Real
16383 for Temp Real

with four status lines S3-S6. S3, S4 and S6 are always one (high) for 8087 driven bus cycles while S5 is always zero (low). When the 8087 is monitoring CPU bus cycles (passive mode) S6 is also monitored by the 8087 to differentiate 8086/8088 activity from that of a local I/O processor or any other local bus master. (The 8086/8088 must be the only processor on the local bus to drive S6 low.) S7 is multiplexed with and has the same value as $\overline{BHE}$ for all 8087 bus cycles.

The first three status lines, $\overline{S0}$-$\overline{S2}$, are used with an 8288 bus controller to determine the type of bus cycle being run:

$\overline{S2}$	$\overline{S1}$	$\overline{S0}$	
0	X	X	Unused
1	0	0	Unused
1	0	1	Memory Data Read
1	1	0	Memory Data Write
1	1	1	Passive (no bus cycle)

Programming Interface

Table 1 lists the seven data types the 8087 supports and presents the format for each type. Internally, the 8087 holds all numbers in the temporary real format. Load and store instructions automatically convert operands represented in memory as 16-, 32-, or 64-bit integers, 32- or 64-bit floating point numbers or 18-digit packed BCD numbers into temporary real format and vice versa.

Computations in the 8087 use the processor's register stack. These eight 80-bit registers provide the equivalent capacity of 40 16-bit registers. The 8087 register set can be accessed as a stack, with instructions operating on the top one or two stack elements, or as a fixed register set, with instructions operating on explicitly designated registers.

Table 5 lists the 8087's instructions by class. Assembly language programs are written in ASM-86, the iAPX 86, 88 assembly language. Table 2 gives the execution times of some typical numeric instructions and their equivalent time on a 5 MHz 8086.

Table 2. Execution Time for Selected 8087 Actual and Emulated Instructions

Floating Point Instruction	Approximate Execution Time (μs)	
	8087 (5 MHz Clock)	8086 Emulation
Add/Subtract Magnitude	14/18	1,600
Multiply (single precision)	19	1,600
Multiply (extended precision)	27	2,100
Divide	39	3,200
Compare	9	1,300
Load (double precision)	10	1,700
Store (double precision)	21	1,200
Square Root	36	19,600
Tangent	90	13,000
Exponentiation	100	17,100

PROCESSOR ARCHITECTURE

As shown in Figure 1, the NDP is internally divided into two processing elements, the control unit (CU) and the numeric execution unit (NEU). The NEU executes all numeric instructions, while the CU receives and decodes instructions, reads and writes memory operands and executes processor control instructions. The two elements are able to operate independently of one another, allowing the CU to maintain synchronization with the CPU while the NEU is busy processing a numeric instruction.

Control Unit

The CU keeps the 8087 operating in synchronization with its host CPU. 8087 instructions are intermixed with CPU instructions in a single instruction stream. The CPU fetches all instructions from memory; by monitoring the status signals ($\overline{S0}$-$\overline{S2}$, S6) emitted by the CPU, the NDP control unit determines when an 8086 instruction is being fetched. The CU taps the bus in parallel with the CPU and obtains that portion of the data stream.

The CU maintains an instruction queue that is identical to the queue in the host CPU. The CU automatically determines if the CPU is an 8086 or an 8088 immediately after reset (by monitoring the $\overline{BHE}$/S7 line) and matches its queue length accordingly. By monitoring the CPU's queue status lines (QS0, QS1), the CU obtains and decodes instructions from the queue in synchronization with the CPU.

After decoding the instruction, the 8086 executes all opcodes but ESCAPE (ESC), while the 8087 executes only the ESCAPE class instructions. (The first five bits of all ESCAPE instructions are identical.) The CPU does provide addressing for ESC instructions, however.

The CPU distinguishes between ESC instructions that reference memory and those that do not. If the instruction refers to a memory operand, the CPU calculates the operand's address using any one of its available addressing modes, and then performs a "dummy read" of the word at that location. (Any location within the 1M byte address space is allowed.) This is a normal read cycle except that the CPU ignores the data it receives. If the ESC instruction does not contain a memory reference (e.g., an 8087 stack operation), the CPU simply proceeds to the next instruction.

An 8087 instruction either will not reference memory, will require loading one or more operands from memory into the 8087, or will require storing one or more operands from the 8087 into memory. In the first case a non-memory reference escape is used to start 8087 operation. In the last two cases, the CU makes use of the "dummy read" cycle initiated by the CPU. The CU captures and saves the address which the CPU places on the bus. If the instruction is a load, the CU additionally captures the data word when it becomes available on the local data bus. If data required is longer than one word, the CU immediately obtains the bus from the CPU using the request/grant protocol and reads the rest of

the information in consecutive bus cycles. In a store operation, the CU captures and saves the store address as in a load, and ignores the data word that follows in the "dummy read" cycle. When the 8087 is ready to perform the store, the CU obtains the bus from the CPU and writes the operand starting at the specified address.

Numeric Execution Unit

The NEU executes all instructions that involve the register stack; these include arithmetic, logical, transcendental, constant and data transfer instructions. The data path in the NEU is 84 bits wide (68 fraction bits, 15 exponent bits and a sign bit) which allows internal operand transfers to be performed at very high speeds.

When the NEU begins executing an instruction, it activates the 8087 BUSY signal. This signal can be used in conjunction with the CPU WAIT instruction to resynchronize both processors when the NEU has completed its current instruction.

Register Set

The 8087 register set is shown in Figure 5. Each of the eight data registers in the 8087's register stack is 80 bits wide and is divided into "fields" corresponding to the NDP's temporary real data type.

At a given point in time the TOP field in the control word identifies the current top-of-stack register. A "push" operation decrements TOP by 1 and loads a value into the new top register. A "pop" operation stores the value from the current top register and then increments TOP by 1. Like 8086/8088 stacks in memory, the 8087 register stack grows "down" toward lower-addressed registers.

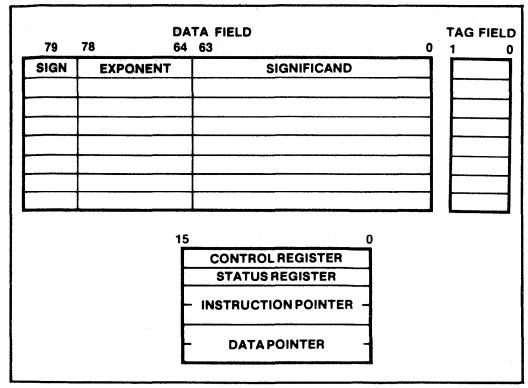


Figure 5. 8087 Register Set

Instructions may address the data registers either implicitly or explicitly. Many instructions operate on the register at the top of the stack. These instructions implicitly address the register pointed to by the TOP. Other instructions allow the programmer to explicitly specify the register which is to be used. Explicit register addressing is "top-relative."

Status Word

The status word shown in Figure 6 reflects the overall state of the 8087; it may be stored in memory and then inspected by CPU code. The status word is a 16-bit register divided into fields as shown in Figure 6. The busy bit (bit 15) indicates whether the *NEU* is executing an instruction (B = 1) or is idle (B = 0). Several instruc-

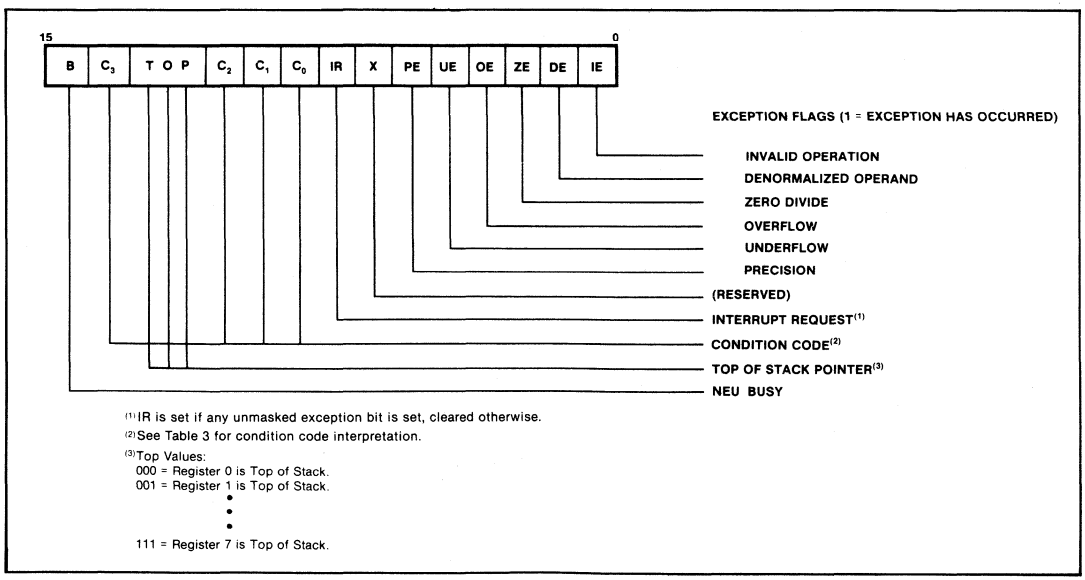


Figure 6. 8087 Status Word

tions which store and manipulate the status word are executed exclusively by the CU, and these do not set the busy bit themselves.

The four numeric condition code bits (C_0-C_3) are similar to the flags in a CPU: various instructions update these bits to reflect the outcome of NDP operations. The effect of these instructions on the condition code bits is summarized in Table 3.

Bits 14-12 of the status word point to the 8087 register that is the current top-of-stack (TOP) as described above.

Bit 7 is the interrupt request bit. This bit is set if any unmasked exception bit is set and cleared otherwise.

Bits 5-0 are set to indicate that the NEU has detected an exception while executing an instruction.

Tag Word

The tag word marks the content of each register as shown in Figure 7. The principal function of the tag word is to optimize the NDP's performance. The tag word can be used, however, to interpret the contents of 8087 registers.

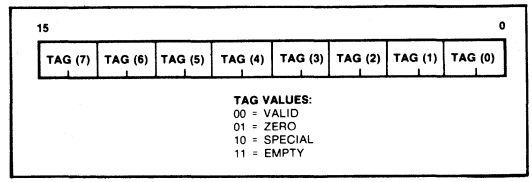


Figure 7. 8087 Tag Word

Instruction and Data Pointers

The instruction and data pointers (see Figure 8) are provided for user-written error handlers. Whenever the 8087 executes an NEU instruction, the CU saves the instruction address, the operand address (if present) and the instruction opcode. 8087 instructions can store this data into memory.

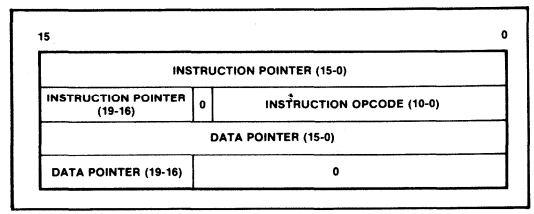


Figure 8. 8087 Instruction and Data Pointers

Table 3. Condition Code Interpretation

Instruction	C_3	C_2	C_1	C_0	Interpretation
Compare, Test	0	X	X	0	A > B
	0	X	X	1	A < B
	1	X	X	0	A = B
	1	X	X	1	A ? B (not comparable)
Remainder	U	0	U	U	Complete reduction
	U	1	U	U	Incomplete reduction
Examine	0	0	0	0	Valid, positive, unnormalized
	0	0	0	1	Invalid, positive, exponent ≠ 0
	0	0	1	0	Valid, negative, unnormalized
	0	0	1	1	Invalid, negative, exponent ≠ 0
	0	1	0	0	Valid, positive, normalized
	0	1	0	1	Infinity, positive
	0	1	1	0	Valid, negative, normalized
	0	1	1	1	Infinity, negative
	1	0	0	0	Zero, positive
	1	0	0	1	Empty
	1	0	1	0	Zero, negative
	1	0	1	1	Empty
	1	1	0	0	Invalid, positive, exponent = 0
	1	1	0	1	Empty
	1	1	1	0	Invalid, negative, exponent = 0
	1	1	1	1	Empty

X = value is not affected by instruction
U = value is undefined following instruction

Control Word

The NDP provides several processing options which are selected by loading a word from memory into the control word. Figure 9 shows the format and encoding of the fields in the control word.

The low order byte of this control word configures 8087 interrupts and exception masking. Bits 5-0 of the control word contain individual masks for each of the six exceptions that the 8087 recognizes and bit 7 contains a general mask bit for all 8087 interrupts. The high order byte of the control word configures the 8087 operating mode including precision, rounding, and infinity control. The precision control bits (bits 9-8) can be used to set the 8087 internal operating precision at less than the default of temporary real precision. This can be useful in providing compatibility with earlier generation arithmetic processors of smaller precision than the 8087. The rounding control bits (bits 11-10) provide for directed rounding and true chop as well as the unbiased round to nearest mode specified in the proposed IEEE standard. Control over closure of the number space at infinity is also provided (either affine closure, $\pm\infty$, or projective closure, ∞, is treated as unsigned, may be specified).

Exception Handling

The 8087 detects six different exception conditions that can occur during instruction execution. Any or all exceptions will cause an interrupt if unmasked and interrupts are enabled.

If interrupts are disabled the 8087 will simply suspend execution until the host clears the exception. If a specific exception class is masked and that exception occurs, however, the 8087 will post the exception in the status register and perform an on-chip default exception handling procedure, thereby allowing processing to continue. The exceptions that the 8087 detects are the following:

1. INVALID OPERATION: Stack overflow, stack underflow, indeterminate form (0/0, $\infty - \infty$, etc.) or the use of a Non-Number (NAN) as an operand. An exponent value is reserved and any bit pattern with this value in the exponent field is termed a Non-Number and causes this exception. If this exception is masked, the 8087's default response is to generate a specific NAN called INDEFINITE, or to propagate already existing NANs as the calculation result.
2. OVERFLOW: The result is too large in magnitude to fit the specified format. The 8087 will generate an encoding for infinity if this exception is masked.
3. ZERO DIVISOR: The divisor is zero while the dividend is a non-infinite, non-zero number. Again, the 8087 will generate an encoding for infinity if this exception is masked.
4. UNDERFLOW: The result is non-zero but too small in magnitude to fit in the specified format. If this exception is masked the 8087 will denormalize (shift right) the fraction until the exponent is in range. This process is called gradual underflow.
5. DENORMALIZED OPERAND: At least one of the operands or the result is denormalized; it has the smallest exponent but a non-zero significand. Normal processing continues if this exception is masked off.
6. INEXACT RESULT: If the true result is not exactly representable in the specified format, the result is rounded according to the rounding mode, and this flag is set. If this exception is masked, processing will simply continue.

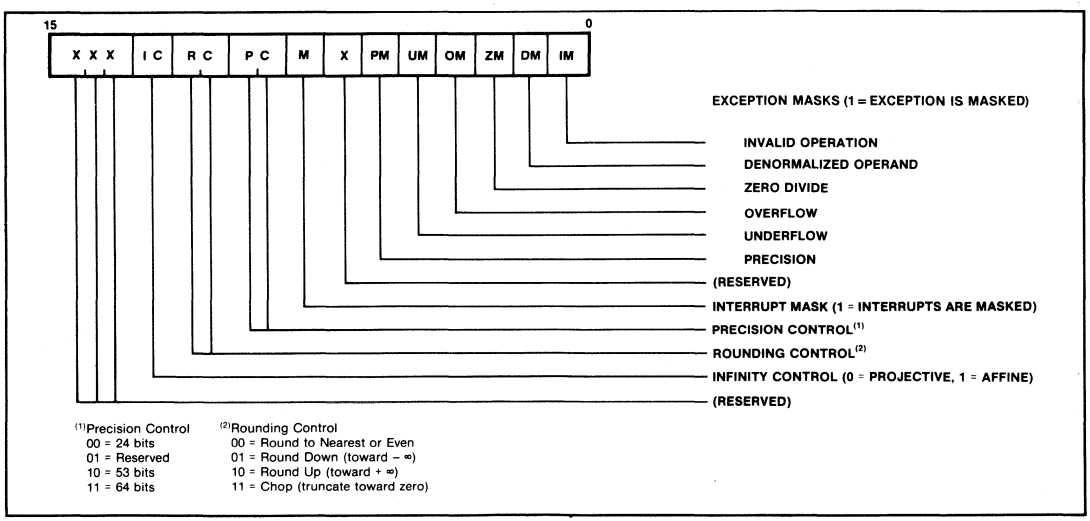


Figure 9. 8087 Control Word

Table 4. Pin Description

Pin(s)	I/O	Function
AD15-AD0	I/O	These lines constitute the time multiplexed memory address (T_1) and data (T_2, T_3, T_W, T_4) bus. A0 is analogous to $\overline{BHE}$ for the lower byte of the data bus, pins D7-D0. It is LOW during T_1 when a byte is to be transferred on the lower portion of the bus in memory operations. Eight-bit oriented devices tied to the lower half of the bus would normally use A0 to condition chip select functions. These lines are active HIGH. They are input/output lines for 8087 driven bus cycles and are inputs which the 8087 monitors when the 8086/8088 is in control of the bus.
A19/S6, A18/S5, A17/S4, A16/S3	I/O	During T_1 these are the four most significant address lines for memory operations. During memory operations, status information is available on these lines during T_2, T_3, T_W, and T_4. For 8087 controlled bus cycles, S6, S4, and S3 are reserved and currently one (HIGH), while S5 is always LOW. These lines are inputs which the 8087 monitors when the 8086/8088 is in control of the bus.
$\overline{BHE}$/S7	I/O	During T_1 the bus high enable signal ($\overline{BHE}$) should be used to enable data onto the most significant half of the data bus, pins D15-D8. Eight-bit oriented devices tied to the upper half of the bus would normally use $\overline{BHE}$ to condition chip select functions. $\overline{BHE}$ is LOW during T_1 for read and write cycles when a byte is to be transferred on the high portion of the bus. The S7 status information is available during T_2, T_3, T_W, and T_4. The signal is active LOW. S7 is an input which the 8087 monitors during 8086/8088 controlled bus cycles.
$\overline{S2}$, $\overline{S1}$, $\overline{S0}$	I/O	For 8087 driven bus cycles, these status lines are encoded as follows: $\overline{S2}$ $\overline{S1}$ $\overline{S0}$ 0 (LOW) X X Unused 1 (HIGH) 0 0 Unused 1 0 1 Read Memory 1 1 0 Write Memory 1 1 1 Passive Status is driven active during T_4, remains valid during T_1 and T_2, and is returned to the passive state (1, 1, 1) during T_3 or during T_W when READY is HIGH. This status is used by the 8288 Bus Controller to generate all memory access control signals. Any change in $\overline{S2}$, $\overline{S1}$, or $\overline{S0}$ during T_4 is used to indicate the beginning of a bus cycle, and the return to the passive state in T_3 or T_W is used to indicate the end of a bus cycle. These signals are monitored by the 8087 when the 8086/8088 is in control of the bus.

Pin(s)	I/O	Function
$\overline{RQ}$/$\overline{GT0}$	I/O	This request/grant pin is used by the NDP to gain control of the local bus from the CPU for operand transfers or on behalf of another bus master. It must be connected to one of the two processor request/grant pins. The request grant sequence on this pin is as follows: 1. A pulse one clock wide is passed to the CPU to indicate a local bus request by either the 8087 or the master connected to the 8087 $\overline{RQ}$/$\overline{GT1}$ pin. 2. The NDP waits for the grant pulse and when it is received will either initiate bus transfer activity in the clock cycle following the grant or pass the grant out on the $\overline{RQ}$/$\overline{GT1}$ pin in this clock if the initial request was for another bus master. 3. The 8087 will generate a release pulse to the CPU one clock cycle after the completion of the last NDP bus cycle or on receipt of the release pulse from the bus master on $\overline{RQ}$/$\overline{GT1}$.
$\overline{RQ}$/$\overline{GT1}$	I/O	This request/grant pin is used by another local bus master to force the NDP to release the local bus at the end of the processor's current bus cycle. If the NDP is not in control of the bus when the request is made the request/grant sequence is passed through the NDP on the $\overline{RQ}$/$\overline{GT0}$ pin one cycle later. Subsequent grant and release pulses are also passed through the NDP with a two and one clock delay, respectively, for resynchronization. $\overline{RQ}$/$\overline{GT1}$ has an internal pull-up resistor, and so may be left unconnected. If the NDP has control of the bus the request/grant sequence is as follows: 1. A pulse 1 CLK wide from another local bus master indicates a local bus request to the 8087 (pulse 1). 2. During the NDP's next T_4 or T_I a pulse 1 CLK wide from the 8087 to the requesting master (pulse 2) indicates that the 8087 has allowed the local bus to float and that it will enter the "RQ/GT acknowledge" state at the next CLK. The NDP's control unit is disconnected logically from the local bus during "RQ/GT acknowledge." 3. A pulse 1 CLK wide from the requesting master indicates to the 8087 (pulse 3) that the "RQ/GT" request is about to end and that the 8087 can reclaim the local bus at the next CLK. Each master-master exchange of the local bus is a sequence of 3 pulses. There must be one dead CLK cycle after each bus exchange. Pulses are active LOW.

Table 4. Pin Description (cont.)

Pin(s)	I/O	Function
QS1, QS0	I	QS1 and QS0 provide the 8087 with status to allow tracking of the CPU instruction queue. QS1 QS0 0 (LOW) 0 No Operation 0 1 First Byte of Op Code from Queue 1 (HIGH) 0 Empty the Queue 1 1 Subsequent Byte from Queue
INT	O	This line is used to indicate that an unmasked exception has occurred during numeric instruction execution when 8087 interrupts are enabled (or deadlock has been detected). This signal is typically routed to an 8259A. INT is active HIGH.
BUSY	O	This signal indicates that the 8087 NEU is executing a numeric instruction. It is connected to the CPU's $\overline{TEST}$ pin to provide CPU-NDP synchronization. In the case of an unmasked exception BUSY remains active until the exception is cleared. BUSY is active HIGH.
Ready	I	READY is the acknowledgement from the addressed memory device that it will complete the data transfer. The RDY signal from memory is synchronized by the 8284A Clock Generator to form READY. This signal is active HIGH.
Reset	I	RESET causes the processor to immediately terminate its present activity. The signal must be active HIGH for at least four clock cycles. RESET is internally synchronized.
CLK	I	The clock provides the basic timing for the processor and bus controller. It is asymmetric with a 33% duty cycle to provide optimized internal timing.
V_{CC}		V_{CC} is the +5V power supply pin.
GND		GND are the ground pins.

ABSOLUTE MAXIMUM RATINGS*

Ambient Temperature Under Bias ... 0°C to 70°C
Storage Temperature ..... −65°C to + 150°C
Voltage on Any Pin with
Respect to Ground ..... −1.0 to +7V
Power Dissipation ..... 3.0 Watt

*Notice: Stresses above those listed under Absolute Maximum Ratings may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

D.C. CHARACTERISTICS T_A = 0°C to 70°C, V_{CC} = 5V ± 10%

Symbol	Parameter	Min.	Max.	Units	Test Conditions
V_{IL}	Input Low Voltage	−0.5	+0.8	V	
V_{IH}	Input High Voltage	2.0	V_{CC} + 0.5	V	
V_{OL}	Output Low Voltage		0.45	V	I_{OL} = 2.0 mA
V_{OH}	Output High Voltage	2.4		V	I_{OH} = −400 μA
I_{CC}	Power Supply Current		475	mA	T_A = 25°C
I_{LI}	Input Leakage Current		±10	μA	0V ≤ V_{IN} ≤ V_{CC}
I_{LO}	Output Leakage Current		±10	μA	0.45V ≤ V_{OUT} ≤ V_{CC}
V_{CL}	Clock Input Low Voltage	−0.5	+0.6	V	
V_{CH}	Clock Input High Voltage	3.9	V_{CC} + 1.0	V	
C_{IN}	Capacitance of Input & Output Buffers (all except I/O Buffer and CLK)		10	pF	fc = 1 MHz
C_{IO}	Capacitance of I/O Buffer (AD0-15, A_{16}–A_{19}, $\overline{BHE}$, $\overline{S2}$-$\overline{S0}$, $\overline{RQ}/\overline{GT}$) and CLK		15	pF	fc = 1MHz

A.C. CHARACTERISTICS T_A = 0°C to 70°C, V_{CC} = 5V ± 10%

Timing Requirements

Symbol	Parameter	Min.	Max.	Units	Test Conditions
TCLCL	CLK Cycle Period	200	500	ns	
TCLCH	CLK Low Time	(2/3TCLCL) − 15		ns	
TCHCL	CLK High Time	(1/3 TCLCL) + 2		ns	
TCH1CH2	CLK Rise Time		10	ns	From 1.0V to 3.5V
TCL2CL1	CLK Fall Time		10	ns	From 3.5V to 1.0V
TDVCL	Data In Setup Time	30		ns	
TCLDX	Data in Hold Time	10		ns	
TRYHCH	READY Setup Time	(2/3TCLCL) − 15		ns	
TCHRYX	READY Hold Time	30		ns	
TRYLCL	READY Inactive to CLK (See Note 3)	−8		ns	
TGVCH	$\overline{RQ}/\overline{GT}$ Setup Time	30		ns	
TCHGX	$\overline{RQ}/\overline{GT}$ Hold Time	40		ns	
TQVCL	QS0-1 Set up Time	30		ns	
TCLQX	QS0-1 Hold Time	10		ns	
TSACH	Status Active Set up Time	30		ns	
TSNCL	Status Inactive Set up Time	30		ns	

Timing Responses

Symbol	Parameter	Min.	Max.	Units	Test Conditions
TCLML	Command Active Delay (See Note 1)	10	35	ns	
TCLMH	Command Inactive Delay (See Note 1)	10	35	ns	
TRYHSH	Ready Active to Status Passive (See Note 2)		110	ns	
TCHSV	Status Active Delay	10	110	ns	
TCLSH	Status Inactive Delay	10	130	ns	
TCLAV	Address Valid Delay	10	110	ns	
TCLAX	Address Hold Time	10		ns	
TCLAZ	Address Float Delay	TCLAX	80	ns	
TSVLH	Status Valid to ALE High (See Note 1)		15	ns	
TCLLH	CLK Low to ALE Valid (See Note 1)		15	ns	
TCHLL	ALE Inactive Delay (See Note 1)		15	ns	CL = 20–100 pF for all
TCLDV	Data Valid Delay	10	110	ns	8087 Outputs (in addition
TCHDX	Data Hold Time	10		ns	to 8087 self-load)
TCVNV	Control Active Delay (See Note 1)	5	45	ns	
TCVNX	Control Inactive Delay (See Note 1)	10	45	ns	
TCHBV	BUSY and INT Valid Delay	10	150	ns	
TCHDTL	Direction Control Active Delay (See Note 1)		50	ns	
TCHDTH	Direction Control Inactive Delay (See Note 1)		30	ns	
TCLGL	$\overline{RQ}/\overline{GT}$ Active Delay	0	85	ns	
TCLGH	$\overline{RQ}/\overline{GT}$ Inactive Delay	0	85	ns	

Notes:

1. Signal at 8284A or 8288 shown for reference only.
2. Applies only to T_3 and wait states.
3. Applies only to T_2 state (8 ns into T_3).

WAVEFORMS

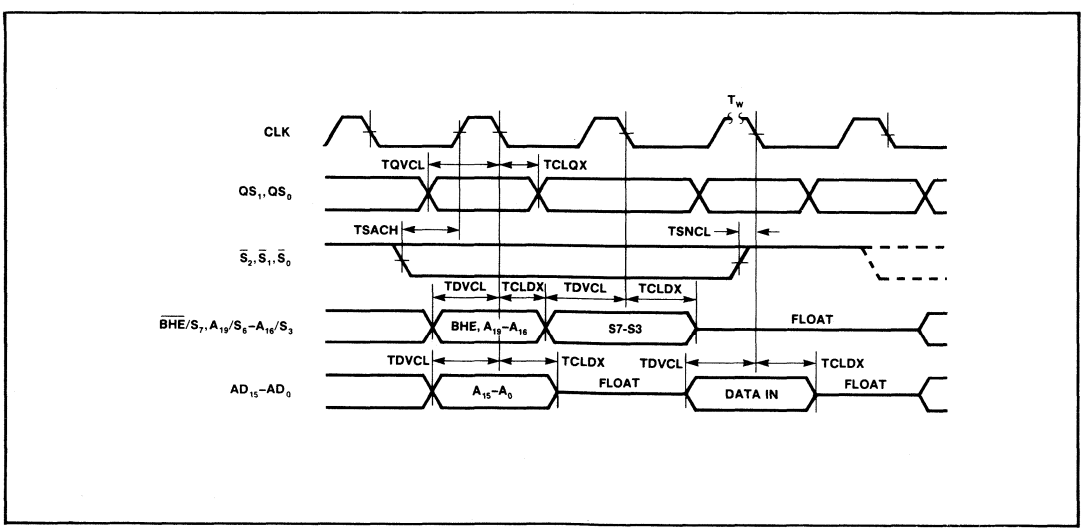
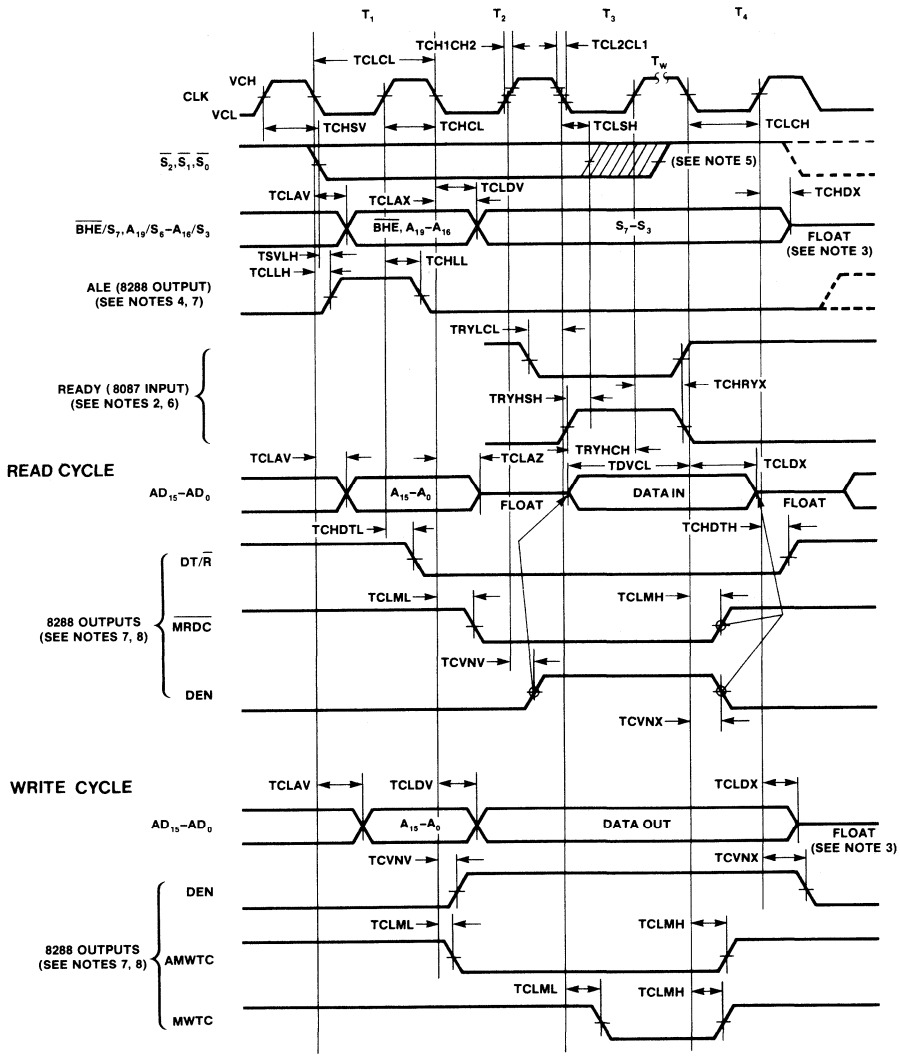


Figure 10. 8087 Bus Timing — Passive Mode



NOTES:

1. ALL SIGNALS SWITCH BETWEEN V_{OL} AND V_{OH} UNLESS OTHERWISE SPECIFIED.
2. RDY IS SAMPLED NEAR THE END OF T_2, T_3 AND T_W TO DETERMINE IF T_W MACHINE STATES ARE TO BE INSERTED.
3. THE LOCAL BUS FLOATS ONLY IF THE 8087 IS RETURNING CONTROL TO THE 8086/8088.
4. ALE RISES AT LATER OF (TSVLH, TCLLH).
5. STATUS INACTIVE IN STATE JUST PRIOR TO T_4.
6. READY SHOULD REMAIN ACTIVE UNTIL S_{0-2} BECOME INACTIVE.
7. SIGNALS AT 8284A OR 8288 ARE SHOWN FOR REFERENCE ONLY.
8. THE ISSUANCE OF 8288 COMMAND AND CONTROL SIGNALS ($\overline{MRDC}$, $\overline{MWTC}$, $\overline{AMWC}$ AND DEN) LAGS THE ACTIVE HIGH 8288 CEN.
9. ALL TIMING MEASUREMENTS ARE MADE AT 1.5V UNLESS OTHERWISE NOTED.

Figure 10. 8087 Bus Timings — Master Mode (cont.)

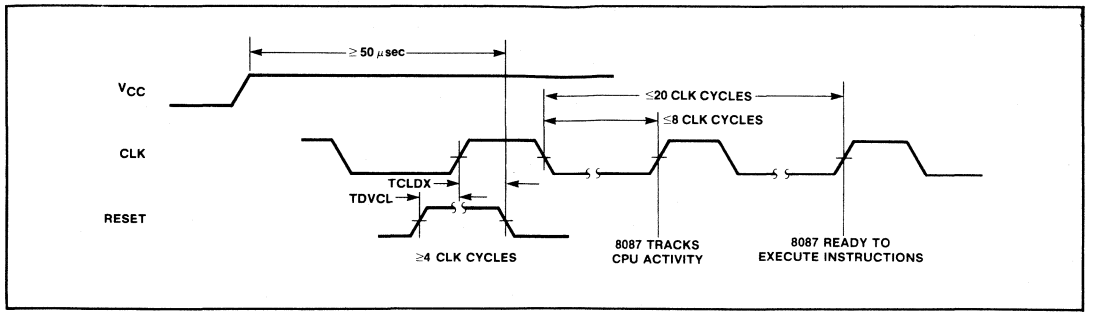


Figure 11. Reset Timing

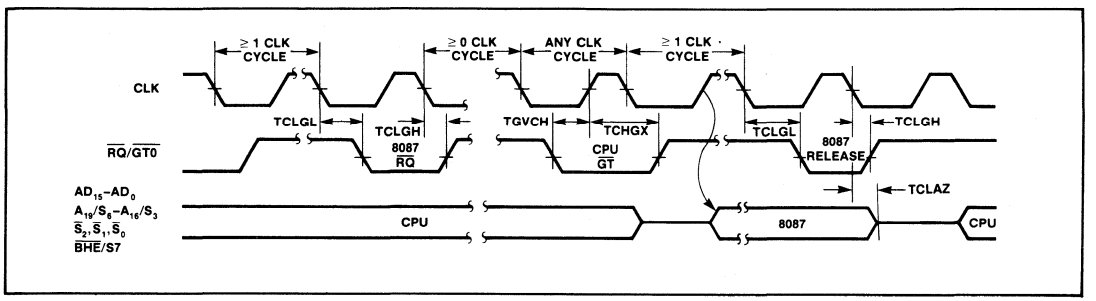
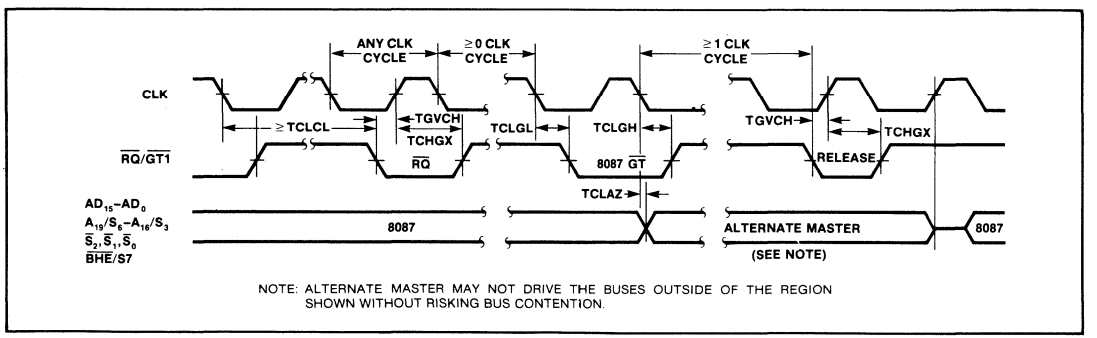


Figure 12. Request/Grant$_0$ Timing



NOTE: ALTERNATE MASTER MAY NOT DRIVE THE BUSES OUTSIDE OF THE REGION SHOWN WITHOUT RISKING BUS CONTENTION.

Figure 13. Request/Grant$_1$ Timing

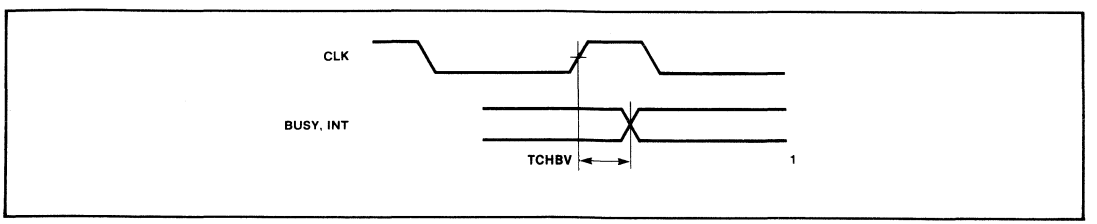


Figure 14. Busy and Interrupt Timing

Table 5. 8087 Extensions to the 8086/8088 Instruction Set

	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Data Transfer				
FLD = LOAD				
Integer/Real Memory to ST(0)	ESCAPE MF 1	MOD 0 0 0 R/M	(DISP-LO)	(DISP-HI)
Long Integer Memory to ST(0)	ESCAPE 1 1 1	MOD 1 0 1 R/M	(DISP-LO)	(DISP-HI)
Temporary Real Memory to ST(0)	ESCAPE 0 1 1	MOD 1 0 1 R/M	(DISP-LO)	(DISP-HI)
BCD Memory to ST(0)	ESCAPE 1 1 1	MOD 1 0 0 R/M	(DISP-LO)	(DISP-HI)
ST(i) to ST(0)	ESCAPE 0 0 1	1 1 0 0 0 ST(i)		
FST = STORE				
ST(0) to Integer/Real Memory	ESCAPE MF 1	MOD 0 1 0 R/M	(DISP-LO)	(DISP-HI)
ST(0) to ST(i)	ESCAPE 1 0 1	1 1 0 1 0 ST(i)		
FSTP = STORE AND POP				
ST(0) to Integer/Real Memory	ESCAPE MF 1	MOD 0 1 1 R/M	(DISP-LO)	(DISP-HI)
ST(0) to Long Integer Memory	ESCAPE 1 1 1	MOD 1 1 1 R/M	(DISP-LO)	(DISP-HI)
ST(0) to Temporary Real Memory	ESCAPE 0 1 1	MOD 1 1 1 R/M	(DISP-LO)	(DISP-HI)
ST(0) to BCD Memory	ESCAPE 1 1 1	MOD 1 1 0 R/M	(DISP-LO)	(DISP-HI)
ST(0) to ST(i)	ESCAPE 1 0 1	1 1 0 1 1 ST(i)		
FXCH = Exchange ST(i) and ST(0)	ESCAPE 0 0 1	1 1 0 0 1 ST(i)		
Comparison				
FCOM = Compare				
Integer/Real Memory to ST(0)	ESCAPE MF 0	MOD 0 1 0 R/M	(DISP-LO)	(DISP-HI)
ST(i) to ST(0)	ESCAPE 0 0 0	1 1 0 1 0 ST(i)		
FCOMP = Compare and Pop				
Integer/Real Memory to ST(0)	ESCAPE MF 0	MOD 0 1 1 R/M	(DISP-LO)	(DISP-HI)
ST(i) to ST(0)	ESCAPE 0 0 0	1 1 0 1 1 ST(i)		
FCOMPP = Compare ST(1) to ST(0) and Pop Twice	ESCAPE 1 1 0	1 1 0 1 1 0 0 1		
FTST = Test ST(0)	ESCAPE 0 0 1	1 1 1 0 0 1 0 0		
FXAM = Examine ST(0)	ESCAPE 0 0 1	1 1 1 0 0 1 0 1		

Table 5. 8087 Extensions to the 8086/8088 Instruction Set (cont.)

Arithmetic	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
FADD = Addition																																
Integer/Real Memory with ST(0)	ESCAPE					MF		0	MOD		0	0	0	R/M			(DISP-LO)								(DISP-HI)							
ST(i) and ST(0)	ESCAPE					d	P	0	1	1	0	0	0	ST(i)																		
FSUB = Subtraction																																
Integer/Real Memory with ST(0)	ESCAPE					MF		0	MOD		1	0	R	R/M			(DISP-LO)								(DISP-HI)							
ST(i) and ST(0)	ESCAPE					d	P	0	1	1	1	0	R	R/M																		
FMUL = Multiplication																																
Integer/Real Memory with ST(0)	ESCAPE					MF		0	MOD		0	0	1	R/M			(DISP-LO)								(DISP-HI)							
ST(i) and ST(0)	ESCAPE					d	P	0	1	1	0	0	1	R/M																		
FDIV = Division																																
Integer/Real Memory with ST(0)	ESCAPE					MF		0	MOD		1	1	R	R/M			(DISP-LO)								(DISP-HI)							
ST(i) and ST(0)	ESCAPE					d	P	0	1	1	1	1	R	R/M																		
FSQRT = Square Root of ST(0)	ESCAPE			0	0	1	1	1	1	1	1	0	1	0																		
FSCALE = Scale ST(0) by ST(1)	ESCAPE			0	0	1	1	1	1	1	1	1	0	1																		
FPREM = Partial Remainder of ST(0) ÷ ST(1)	ESCAPE			0	0	1	1	1	1	1	1	0	0	0																		
FRNDINT = Round ST(0) to Integer	ESCAPE			0	0	1	1	1	1	1	1	1	0	0																		
FXTRACT = Extract Components of ST(0)	ESCAPE			0	0	1	1	1	1	1	0	1	0	0																		
FABS = Absolute Value of ST(0)	ESCAPE			0	0	1	1	1	1	0	0	0	0	1																		
FCHS = Change Sign of ST(0)	ESCAPE			0	0	1	1	1	1	0	0	0	0	0																		
Transcendental																																
FPTAN = Partial Tangent of ST(0)	ESCAPE			0	0	1	1	1	1	1	0	0	1	0																		
FPATAN = Partial Arctangent of ST(0) ÷ ST(1)	ESCAPE			0	0	1	1	1	1	1	0	0	1	1																		
F2XM1 = $2^{ST(0)}-1$	ESCAPE			0	0	1	1	1	1	1	0	0	0	0																		
FYL2X = ST(1) · $\log_2$ [ST(0)]	ESCAPE			0	0	1	1	1	1	1	0	0	0	1																		
FYL2XP1 = ST(1) · $\log_2$ [ST(0) + 1]	ESCAPE			0	0	1	1	1	1	1	1	0	0	1																		
Constants																																
FLDZ = LOAD + 0.0 into ST(0)	ESCAPE			0	0	1	1	1	1	0	1	1	1	0																		
FLD1 = LOAD + 1.0 into ST(0)	ESCAPE			0	0	1	1	1	1	0	1	0	0	0																		
FLDPI = LOAD π into ST(0)	ESCAPE			0	0	1	1	1	1	0	1	0	1	1																		
FLDL2T = LOAD $\log_2 10$ into ST(0)	ESCAPE			0	0	1	1	1	1	0	1	0	0	1																		
FLDL2E = LOAD $\log_2 e$ into ST(0)	ESCAPE			0	0	1	1	1	1	0	1	0	1	0																		
FLDLG2 = LOAD $\log_{10} 2$ into ST(0)	ESCAPE			0	0	1	1	1	1	0	1	1	0	0																		
FLDLN2 = LOAD $\log_e 2$ into ST(0)	ESCAPE			0	0	1	1	1	1	0	1	1	0	1																		

Table 5. 8087 Extensions to the 8086/8088 Instruction Set (cont.)

	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Processor Control				
FINIT = Initialize NDP	ESCAPE 0 1 1	1 1 1 0 0 0 1 1		
FENI = Enable Interrupts	ESCAPE 0 1 1	1 1 1 0 0 0 0 0		
FDISI = Disable Interrupts	ESCAPE 0 1 1	1 1 1 0 0 0 0 1		
FLDCW = Load Control Word	ESCAPE 0 0 1	MOD 1 0 1 R/M	(DISP-LO)	(DISP-HI)
FSTCW = Store Control Word	ESCAPE 0 0 1	MOD 1 1 1 R/M	(DISP-LO)	(DISP-HI)
FSTSW = Store Status Word	ESCAPE 1 0 1	MOD 1 1 1 R/M	(DISP-LO)	(DISP-HI)
FCLEX = Clear Exceptions	ESCAPE 0 1 1	1 1 1 0 0 0 1 0		
FSTENV = Store Environment	ESCAPE 0 0 1	MOD 1 1 0 R/M	(DISP-LO)	(DISP-HI)
FLDENV = Load Environment	ESCAPE 0 0 1	MOD 1 0 0 R/M	(DISP-LO)	(DISP-HI)
FSAVE = Save State	ESCAPE 1 0 1	MOD 1 1 0 R/M	(DISP-LO)	(DISP-HI)
FRSTOR = Restore State	ESCAPE 1 0 1	MOD 1 0 0 R/M	(DISP-LO)	(DISP-HI)
FINCSTP = Increment Stack Pointer	ESCAPE 0 0 1	1 1 1 1 0 1 1 1		
FDECSTP = Decrement Stack Pointer	ESCAPE 0 0 1	1 1 1 1 0 1 1 0		
FFREE = Free ST(i)	ESCAPE 1 0 1	1 1 0 0 0 ST(i)		
FNOP = No Operation	ESCAPE 0 0 1	1 1 0 1 0 0 0 0		
FWAIT = CPU Wait for NDP	1 0 0 1 1 0 1 1			

FOOTNOTES:

if mod = 00 then DISP = 0*, disp-low and disp-high are absent
if mod = 01 then DISP = disp-low sign-extended to 16-bits, disp-high is absent
if mod = 10 then DISP = disp-high; disp-low
if mod = 11 then r/m is treated as an ST(i) field

if r/m = 000 then EA = (BX) + (SI) + DISP
if r/m = 001 then EA = (BX) + (DI) + DISP
if r/m = 010 then EA = (BP) + (SI) + DISP
if r/m = 011 then EA = (BP) + (DI) + DISP
if r/m = 100 then EA = (SI) + DISP
if r/m = 101 then EA = (DI) + DISP
if r/m = 110 then EA = (BP) + DISP*
if r/m = 111 then EA = (BX) + DISP

*except if mod = 000 and r/m = 110 then EA = disp-high: disp-low.

MF = Memory Format
00 — 32-bit Real
01 — 32-bit Integer
10 — 64-bit Real
11 — 16-bit Integer

ST(0) = Current stack top
ST(i) = i^{th} register below stack top

d = Destination
0 — Destination is ST(0)
1 — Destination is ST(i)

P = Pop
0 — No pop
1 — Pop ST(0)

R = Reverse
0 — Destination (op) Source
1 — Source (op) Destination

For **FSQRT:** $-0 \leq ST(0) \leq +\infty$
For **FSCALE:** $-2^{15} \leq ST(1) < +2^{15}$ and ST(1) integer
For **F2XM1:** $0 \leq ST(0) \leq 2^{-1}$
For **FYL2X:** $0 < ST(0) < \infty$
$-\infty < ST(1) < +\infty$
For **FYL2XP1:** $0 < |ST(0)| < (2 - \sqrt{2})/2$
$-\infty < ST(1) < \infty$
For **FPTAN:** $0 \leqslant ST(0) < \pi/4$
For **FPATAN:** $0 \leqslant ST(0) < ST(1) < +\infty$

U.S. AND CANADIAN SALES OFFICES

3065 Bowers Avenue
Santa Clara, California 95051
Tel: (408) 987-8080
TWX: 910-338-0026
TELEX: 34-6372

May 1980

ALABAMA

Intel Corp.
303 Williams Avenue, S.W.
Suite 1422
Huntsville 35801
Tel: (205) 533-9353

Pen-Tech Associates, Inc.
Holiday Office Center
3322 Memorial Pkwy., S.W.
Huntsville 35801
Tel: (205) 881-9298

ARIZONA

Intel Corp.
10210 N. 25th Avenue, Suite 11
Phoenix 85021
Tel: (602) 997-9695

BFA
4426 North Saddle Bag Trail
Scottsdale 85251
Tel: (602) 994-5400

CALIFORNIA

Intel Corp.
7670 Opportunity Rd.
Suite 135
San Diego 92111
Tel: (714) 268-3563

Intel Corp.*
2000 East 4th Street
Suite 100
Santa Ana 92705
Tel: (714) 835-9642
TWX: 910-595-1114

Intel Corp.*
15335 Morrison
Suite 345
Sherman Oaks 91403
Tel: (213) 986-9510
TWX: 910-495-2045

Intel Corp.*
3375 Scott Blvd.
Santa Clara 95051
Tel: (408) 987-8086
TWX: 910-339-9279
910-338-0255

Earle Associates, Inc.
4617 Ruffner Street
Suite 202
San Diego 92111
Tel: (714) 278-5441

Mac-l
2576 Shattuck Ave.
Suite 4B
Berkeley 94704
Tel: (415) 843-7625

Mac-l
P.O. Box 1420
Cupertino 95014
Tel: (408) 257-9880

Mac-l
558 Valley Way
Calaveras Business Park
Milpitas 95035
Tel: (408) 946-8885

Mac-l
P.O. Box 8763
Fountain Valley 92708
Tel: (714) 839-3341

Mac-l
1321 Centinela Avenue
Suite 1
Santa Monica 90404
Tel: (213) 829-4797

Mac-l
20121 Ventura Blvd., Suite 240E
Woodland Hills 91364
Tel: (213) 347-5900

COLORADO

Intel Corp.*
650 S. Cherry Street
Suite 720
Denver 80222
Tel: (303) 321-8066
TWX: 910-931-2289

Westek Data Products, Inc.
25921 Fern Gulch
P.O. Box 1355
Evergreen 80439
Tel: (303) 674-5255

Westek Data Products, Inc.
1322 Arapahoe
Boulder 80302
Tel: (303) 449-2620

Westek Data Products, Inc.
1228 W. Hinsdale Dr.
Littleton 80120
Tel: (303) 797-0482

CONNECTICUT

Intel Corp.
Peacock Alley
1 Padanaram Road, Suite 146
Danbury 06810
Tel: (203) 792-8366
TWX: 710-456-1199

FLORIDA

Intel Corp.
1001 N.W. 62nd Street, Suite 406
Ft. Lauderdale 33309
Tel: (305) 771-0600
TWX: 510-956-9407

Intel Corp.
5151 Adanson Street, Suite 203
Orlando 32804
Tel: (305) 628-2393
TWX: 810-853-9219

Pen-Tech Associates, Inc.
201 S.E. 15th Terrace, Suite K
Deerfield Beach 33441
Tel: (305) 421-4989

Pen-Tech Associates, Inc.
111 So. Maitland Ave., Suite 202
P.O. Box 1475
Maitland 32751
Tel: (305) 645-3444

GEORGIA

Pen Tech Associates, Inc.
Cherokee Center, Suite 21
627 Cherokee Street
Marietta 30060
Tel: (404) 424-1931

ILLINOIS

Intel Corp.*
2550 Golf Road, Suite 815
Rolling Meadows 60008
Tel: (312) 981-7200
TWX: 910-651-5881

Technical Representatives
1502 North Lee Street
Bloomington 61701
Tel: (309) 829-8080

INDIANA

Intel Corp.
9101 Wesleyan Road
Suite 204
Indianapolis 46268
Tel: (317) 299-0623

IOWA

Technical Representatives, Inc.
St. Andrews Building
1930 St. Andrews Drive N.E.
Cedar Rapids 52405
Tel: (319) 393-5510

KANSAS

Intel Corp.
9393 W. 110th St., Ste. 265
Overland Park 66210
Tel: (913) 642-8080

Technical Representatives, Inc.
8245 Nieman Road, Suite 100
Lenexa 66214
Tel: (913) 888-0212, 3, & 4
TWX: 910-749-6412

Technical Representatives, Inc.
360 N. Rock Road
Suite 4
Wichita 67206
Tel: (316) 681-0242

MARYLAND

Intel Corp.*
7257 Parkway Drive
Hanover 21076
Tel: (301) 796-7500
TWX: 710-862-1944

Mesa Inc.
11900 Parklawn Drive
Rockville 20852
Tel: Washington (301) 881-8430
Baltimore (301) 792-0021

MASSACHUSETTS

Intel Corp.*
27 Industrial Ave.
Chelmsford 01824
Tel: (617) 667-8126
TWX: 710-343-6333

EMC Corp.
381 Elliot Street
Newton 02164
Tel: (617) 244-4740
TWX: 922531

MICHIGAN

Intel Corp.*
26500 Northwestern Hwy.
Suite 401
Southfield 48075
Tel: (313) 353-0920
TWX: 810-244-4915

Lowry & Associates, Inc.
135 W. North Street
Suite 4
Brighton 48116
Tel: (313) 227-7067

Lowry & Associates, Inc.
3902 Costa NE
Grand Rapids 49505
Tel: (616) 363-9839

MINNESOTA

Intel Corp.
7401 Metro Blvd.
Suite 355
Edina 55435
Tel: (612) 835-6722
TWX: 910-576-2867

MISSOURI

Intel Corp.
502 Earth City Plaza
Suite 121
Earth City 63045
Tel: (314) 291-1990

Technical Representatives, Inc.
320 Brookes Drive, Suite 104
Hazelwood 63042
Tel: (314) 731-5200
TWX: 910-762-0618

NEW JERSEY

Intel Corp.*
Raritan Plaza
2nd Floor
Raritan Center
Edison 08817
Tel: (201) 225-3000
TWX: 710-480-6238

NEW MEXICO

BFA Corporation
P.O. Box 1237
Las Cruces 88001
Tel: (505) 523-0601
TWX: 910-983-0543

BFA Corporation
3705 Westerfield, N.E.
Albuquerque 87111
Tel: (505) 292-1212
TWX: 910-989-1157

NEW YORK

Intel Corp.*
350 Vanderbilt Motor Pkwy.
Suite 402
Hauppauge 11787
Tel: (516) 231-3300
TWX: 510-227-6236

Intel Corp.
80 Washington St.
Poughkeepsie 12601
Tel: (914) 473-2303
TWX: 510-248-0060

Intel Corp.*
2255 Lyell Avenue
Lower Floor East Suite
Rochester 14606
Tel: (716) 254-6120
TWX: 510-253-7391

Measurement Technology, Inc.
159 Northern Boulevard
Great Neck 11021
Tel: (516) 482-3500

T-Squared
4054 Newcourt Avenue
Syracuse 13206
Tel: (315) 463-8592
TWX: 710-541-0554

T-Squared
2 E. Main
Victor 14564
Tel: (716) 924-9101
TWX: 510-254-8542

NORTH CAROLINA

Intel Corp.
154 Huffman Mill Rd.
Burlington 27215
Tel: (919) 584-3631

Pen-Tech Associates, Inc.
1202 Eastchester Dr.
Highpoint 27260
Tel: (919) 883-9125

OHIO

Intel Corp.*
6500 Poe Avenue
Dayton 45415
Tel: (513) 890-5350
TWX: 810-450-2528

Intel Corp.*
Chagrin-Brainard Bldg., No. 210
28001 Chagrin Blvd.
Cleveland 44122
Tel: (216) 464-2736
TWX: 810-427-9298

OREGON

Intel Corp.
10700 S.W. Beaverton
Hillsdale Highway
Suite 324
Beaverton 97005
Tel: (503) 641-8086
TWX: 910-467-8741

PENNSYLVANIA

Intel Corp.*
275 Commerce Dr.
200 Office Center
Suite 300
Fort Washington 19034
Tel: (215) 542-9444
TWX: 510-661-2077

Q.E.D. Electronics
300 N. York Road
Hatboro 19040
Tel: (215) 674-9600

TEXAS

Intel Corp.*
2925 L.B.J. Freeway
Suite 175
Dallas 75234
Tel: (214) 241-9521
TWX: 910-860-5617

Intel Corp.*
6420 Richmond Ave.
Suite 280
Houston 77057
Tel: (713) 784-3400
TWX: 910-881-2490

Industrial Digital Systems Corp.
5925 Sovereign
Suite 101
Houston 77036
Tel: (713) 988-9421

Intel Corp.
313 E. Anderson Lane
Suite 314
Austin 78752
Tel: (512) 454-3628

WASHINGTON

Intel Corp.
Suite 114, Bldg. 3
1603 116th Ave. N.E.
Bellevue 98005
Tel: (206) 453-8086
TWX: 910-443-3002

WISCONSIN

Intel Corp.
150 S. Sunnyslope Rd.
Brookfield 53005
Tel: (414) 784-9060

CANADA

Intel Semiconductor Corp.*
Suite 233, Bell Mews
39 Highway 7, Bells Corners
Ottawa, Ontario K2H 8R2
Tel: (613) 829-9714
TELEX: 053-4115

Intel Semiconductor Corp.
50 Galaxy Blvd.
Unit 12
Rexdale, Ontario
M9W 4Y5
Tel: (416) 675-2105
TELEX: 06983574

Multilek, Inc.*
15 Grenfell Crescent
Ottawa, Ontario K2G 0G3
Tel: (613) 226-2365
TELEX: 053-4585

Multilek, Inc.
Toronto
Tel: (416) 245-4622

Multilek, Inc.
Montreal
Tel: (514) 481-1350

*Field Application Location

INTERNATIONAL SALES AND MARKETING OFFICES

3065 Bowers Avenue
Santa Clara, California 95051
Tel: (408) 987-8080
TWX: 910-338-0026
TELEX: 34-6372

INTERNATIONAL DISTRIBUTORS/REPRESENTATIVES

May 1980

ARGENTINA

Micro Sistemas S.A.
9 De Julio 561
Cordoba
Tel: 54-51-32-880
TELEX: 51837 BICCO

AUSTRALIA

A.J.F. Systems & Components Pty. Ltd.
310 Queen Street
Melbourne
Victoria 3000
Tel:
TELEX:

Warburton Franki
Corporate Headquarters
372 Eastern Valley Way
Chatswood, New South Wales 2067
Tel: 407-3261
TELEX: AA 21299

AUSTRIA

Bacher Elektronische Geraete GmbH
Rotenmulgasse 26
A 1120 Vienna
Tel: (0222) 83 63 96
TELEX: (01) 1532

Rekirsch Elektronik Geraete GmbH
Lichtensteinstrasse 97
A1000 Vienna
Tel: (222) 347646
TELEX: 74759

BELGIUM

Inelco Belgium S.A.
Ave. des Croix de Guerre 94
B1120 Brussels
Tel: (02) 216 01 60
TELEX: 25441

BRAZIL

Icotron S.A.
0511-Av. Mutinga 3650
6 Andar
Pirituba-Sao Paulo
Tel: 261-0211
TELEX: (011) 222 ICO BR

CHILE

DIN
Av. Vic. Mc kenna 204
Casilla 6055
Santiago
Tel: 227 564
TELEX: 3520003

CHINA

C.M. Technologies
525 University Avenue
Suite A-40
Palo Alto, CA 94301

COLOMBIA

International Computer Machines
Adpo. Aereo 19403
Bogota 1
Tel: 232-6635
TELEX: 43439

CYPRUS

Cyprus Eltrom Electronics
P.O. Box 5393
Nicosia
Tel: 21-27982

DENMARK

STL-Lyngso Komponent A/S
Ostmarken 4
DK-2860 Soborg
Tel: (01) 67 00 77
TELEX: 22990

Scandinavian Semiconductor
Supply A/S
Nannasgade 18
DK-2200 Copenhagen
Tel: (01) 83 50 90
TELEX: 19037

FINLAND

Oy Fintronic AB
Melkonkatu 24 A
SF-00210
Helsinki 21
Tel: 0-692 6022
TELEX: 124 224 Ftron SF

FRANCE

Celdis S.A.*
53, Rue Charles Frerot
F-94250 Gentilly
Tel: (1) 581 00 20
TELEX: 200 485

Feutrier
Rue des Trois Glorieuses
F-42270 St. Priest-en-Jarez
Tel: (77) 74 67 33
TELEX: 300 0 21

Metrologie*
La Tour d'Asnieres
4, Avenue Laurent Cely
92606-Asnieres
Tel: 791 44 44
TELEX: 611 448

Tekelec Airtronic*
Cite des Bruyeres
Rue Carle Vernet
F-92310 Sevres
Tel: (1) 534 75 35
TELEX: 204552

GERMANY

Electronic 2000 Vertriebs GmbH
Neumarkter Strasse 75
D-8000 Munich 80
Tel: (089) 434061
TELEX: 522561

Jermyn GmbH
Postfach 1180
D-6077 Camberg
Tel: (06434) 231
TELEX: 484426

Kontron Elektronik GmbH
Breslauerstrasse 2
8057 Eching B
D-8000 Munich
Tel: (89) 319.011
TELEX: 522122

Neye Enatechnik GmbH
Schillerstrasse 14
D-2085 Quickborn-Hamburg
Tel: (04106) 6121
TELEX: 02-13590

GREECE

American Technical Enterprises
P.O. Box 156
Athens
Tel:* 30-1-8811271
30-1-8219470

HONG KONG

Schmidt & Co.
28/F Wing on Center
Connaught Road
Hong Kong
Tel: 5-455-644
TELEX: 74766 Schmc Hx

INDIA

Micronic Devices
104/109C, Nirmal Industrial Estate
Sion (E)
Bombay 400022, India
Tel: 486-170
TELEX: 011-5947 MDEV IN

ISRAEL

Eastronics Ltd.*
11 Rozanis Street
P.O. Box 39300
Tel Aviv 61390
Tel: 475151
TELEX: 33638

ITALY

Eledra 3S S.P.A.*
Viale Elvezia, 18
I 20154 Milan
Tel: (02) 34.93.041-31.85.441
TELEX: 332332

JAPAN

Asahi Electronics Co. Ltd.
KMM Bldg. Room 407
2-14-1 Asano, Kokura
Kita-Ku, Kitokyushu City 802
Tel: (093) 511-6471
TELEX: AECKY 7126-16

Hamilton-Avnet Electronics Japan Ltd.
YU and YOU Bldg. 1-4 Horidome-Cho
Nihonbashi
Tel: (03) 662-9911
TELEX: 2523774

Nippon Micro Computer Co. Ltd.
Mutsumi Bldg. 4-5-21 Kojimachi
Chiyoda-ku, Tokyo 102
Tel: (03) 230-0041

Ryoyo Electric Corp.
Konwa Bldg.
1-12-22, Tsukiji, 1-Chome
Chuo-Ku, Tokyo 104
Tel: (03) 543-7711

Tokyo Electron Ltd.
No. 1 Higashikata-Machi
Midori-Ku, Yokohama 226
Tel: (045) 471-8811
TELEX: 781-4473

KOREA

Koram Digital
Room 411 Ahil Bldg.
49-4 2-GA Hoehyun-Dong
Chung-Ku Seoul
Tel: 23-8123
TELEX: K23542 HANSINT

Leewood International, Inc.
C.P.O. Box 4046
112-25, Sokong-Dong
Chung-Ku, Seoul 100
Tel: 28-5927
CABLE: "LEEWOOD" Seoul

NETHERLANDS

Inelco Nether. Comp. Sys. BV
Turfstekerstraat 63
Aalsmeer 1431 D
Tel: (2977) 28855
TELEX: 14693

Koning & Hartman
Koperwerf 30
2544 EN Den Haag
Tel: (70) 210.101
TELEX: 31528

NEW ZEALAND

W. K. McLean Ltd.
P.O. Box 18-065
Glenn Innes, Auckland, 6
Tel: 587-037
TELEX: NZ2763 KOSFY

NORWAY

Nordisk Elektronik (Norge) A/S
Postoffice Box 122
Smedsvingen 4
1364 Hvalstad
Tel: 02 78 62 10
TELEX: 17546

PORTUGAL

Ditram
Componentes E Electronica LDA
Av. Miguel Bombarda, 133
Lisboa 1
Tel: (19) 545313
TELEX: 14347 GESPIC

SINGAPORE

General Engineers Associates
Blk 3, 1003-1008, 10th Floor
P.S.A. Multi-Storey Complex
Telok Blangah/Pasir Panjang
Singapore 5
Tel: 271-3163
TELEX: RS23987 GENERCO

SOUTH AFRICA

Electronic Building Elements
Pine Square
18th Street
Hazelwood, Pretoria 0001
Tel: 789 221
TELEX: 30181SA

SPAIN

Interface
Av. Generalisimo 51 9°
E-Madrid 16
Tel: 456 3151

ITT SESA
Miguel Angel 16
Madrid 10
Tel: (1) 4190957
TELEX: 27707/27461

SWEDEN

AB Gosta Backstrom
Box 12009
10221 Stockholm
Tel: (08) 541 080
TELEX: 10135

Nordisk Electronik AB
Box 27301
S-10254 Stockholm
Tel: (08) 635040
TELEX: 10547

SWITZERLAND

Industrade AG
Gemsenstrasse 2
Postcheck 80 - 21190
CH-8021 Zurich
Tel: (01) 60 22 30
TELEX: 56788

TAIWAN

Taiwan Automation Co.*
3d Floor #75, Section 4
Nanking East Road
Taipei
Tel: 771-0940
TELEX: 11942 TAIAUTO

TURKEY

Turkelek Electronics
Apapurk Boulevard 169
Ankara
Tel: 189483

UNITED KINGDOM

Comway Microsystems Ltd.
Market Street
68-Bracknell, Berkshire
Tel: (344) 51654
TELEX: 847201

G.E.C. Semiconductors Ltd.
East Lane
North Wembley
Middlesex HA9 7PP
Tel: (01) 904-9303/908-4111
TELEX: 28817

Jermyn Industries
Vestry Estate
Sevenoaks, Kent
Tel: (0732) 501.44
TELEX: 95142

Rapid Recall, Ltd.
6 Soho Mills Ind. Park
Wooburn Green
Bucks, England
Tel: (6285) 24961
TELEX: 849439

Sintrom Electronics Ltd.*
Arkwright Road 2
Reading, Berkshire RG2 0LS
Tel: (0734) 85464
TELEX: 847395

VENEZUELA

Componentes y Circuitos
Electronicos TTLCA C.A.
Apartado 3223
Caracas 101
Tel: 718-100
TELEX: 21795 TELETIPOS

*Field Application Location