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.

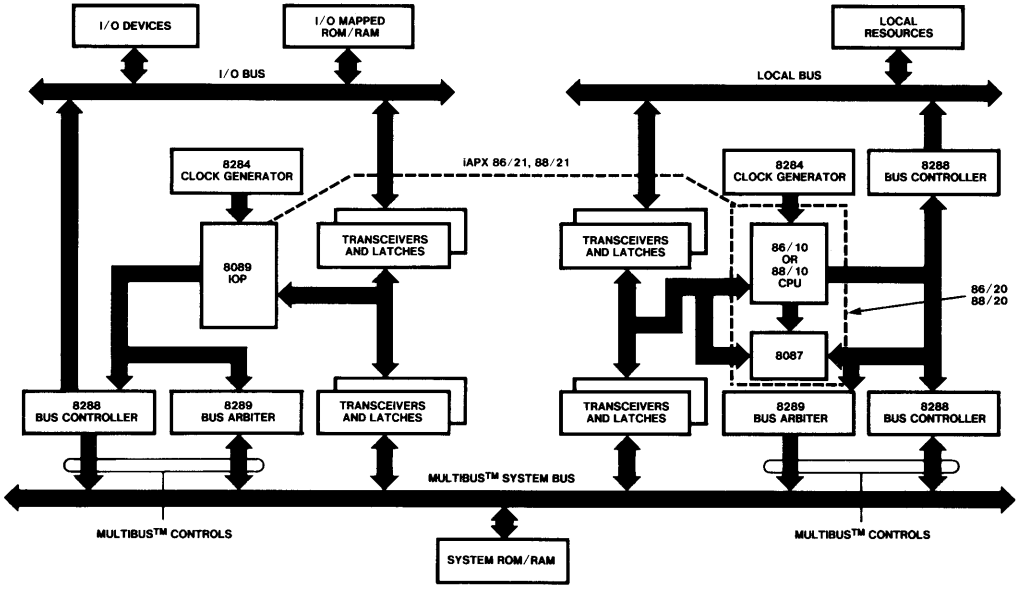


Figure i. iAPX 86, 88 Multiprocessing System

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.

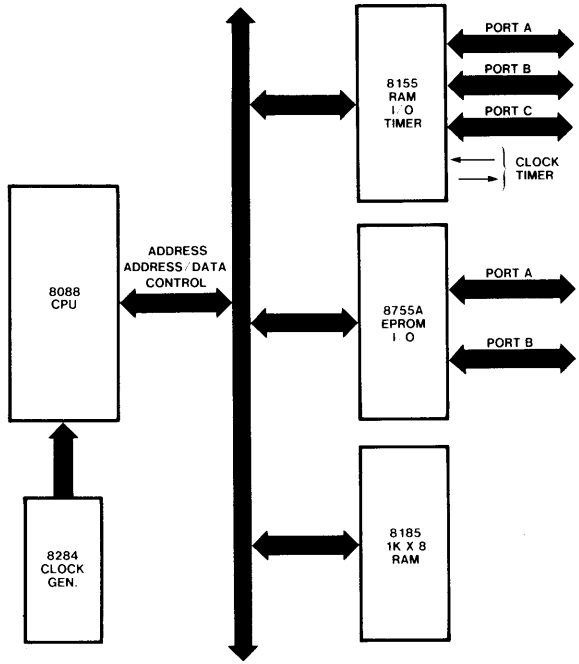


Figure ii. iAPX 88 Allows for a Highly Integrated, High Performance System Using 8085 Family of Components

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

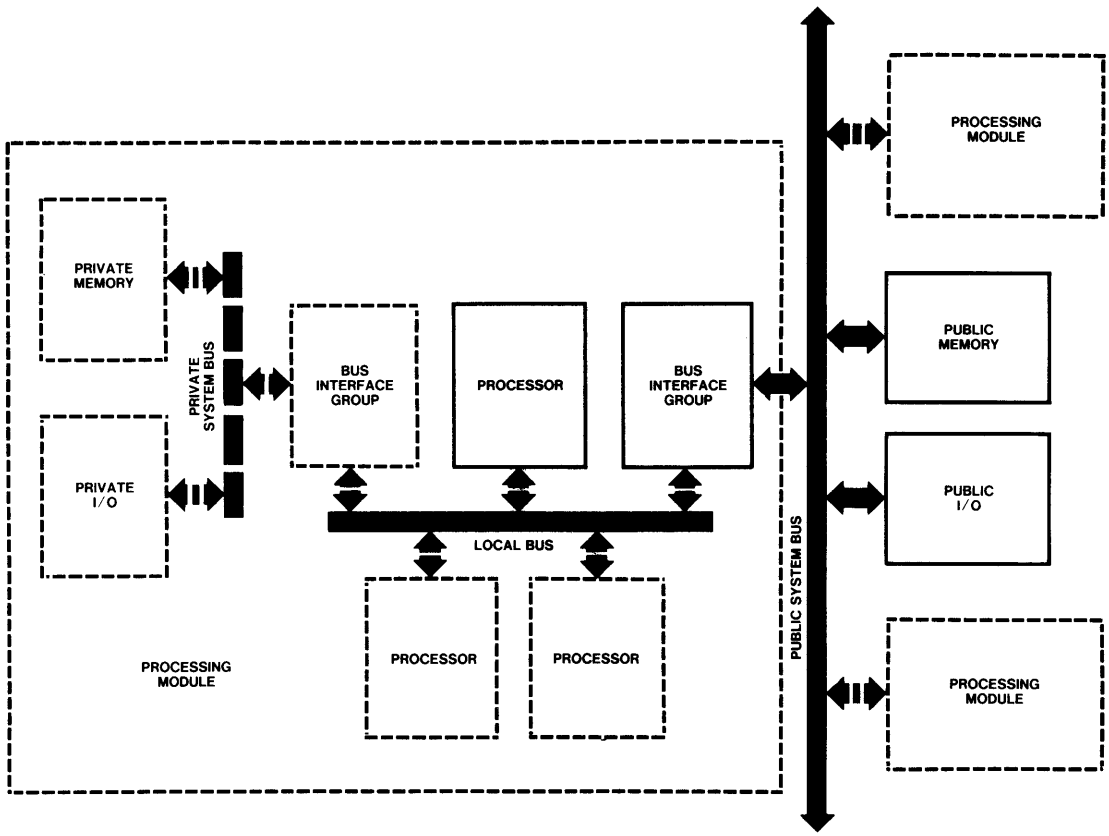


Figure iii. Generalized iAPX 86, 88 Bus Structure

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.

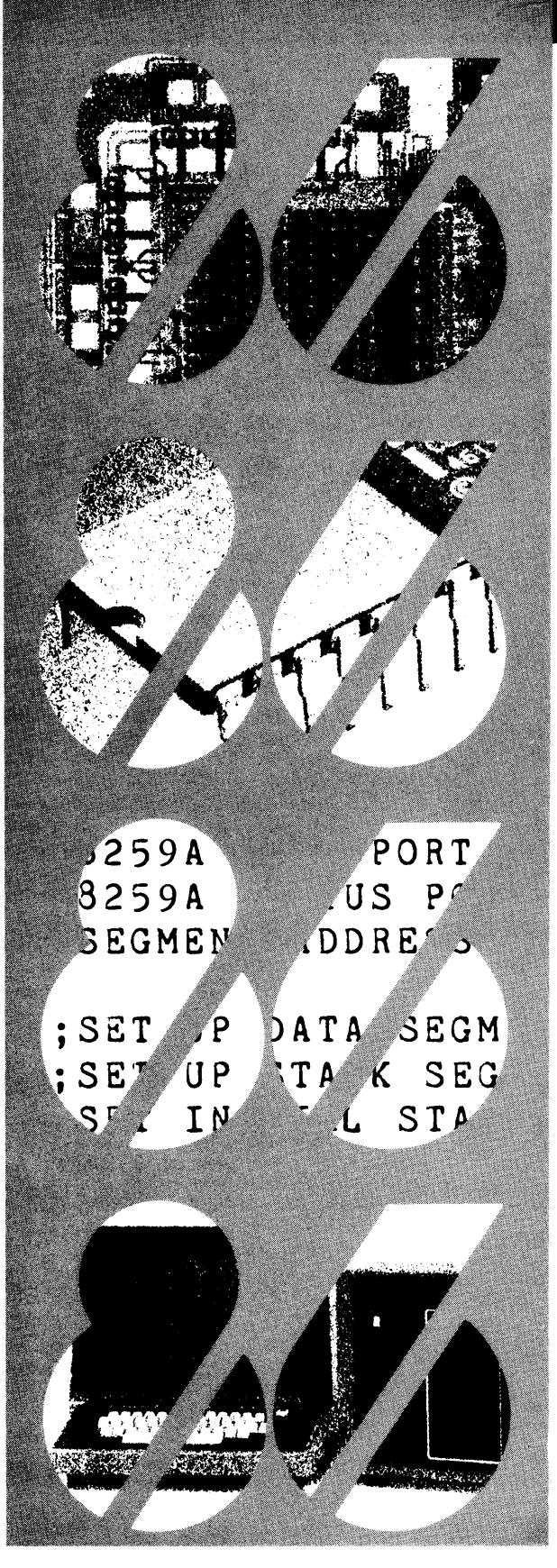
A second bus interface group may be connected to a processing module's local bus, generating a demultiplexed bus. This bus can provide the processing module with a private address space that is not accessible to other processing modules. Distributing memory and I/O resources in this manner can improve 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

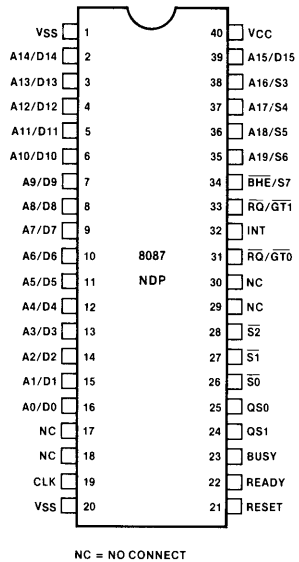


Figure S-1. 8087 Numeric Data Processor Pin Diagram

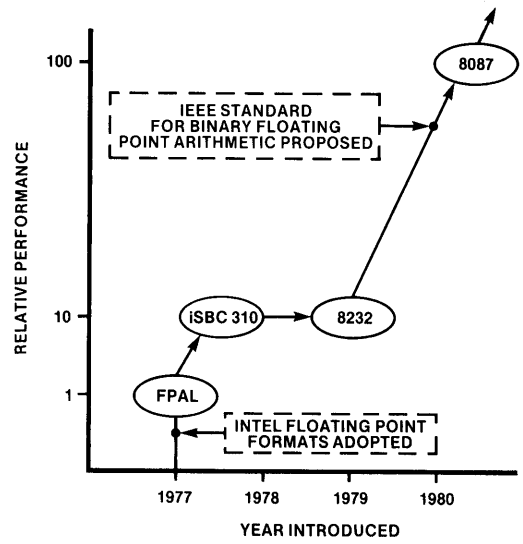


Figure S-2. 8087 Evolution and Relative Performance

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, $\text{Log}_2 10$, $\text{Log}_2 e$
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 recompilation 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

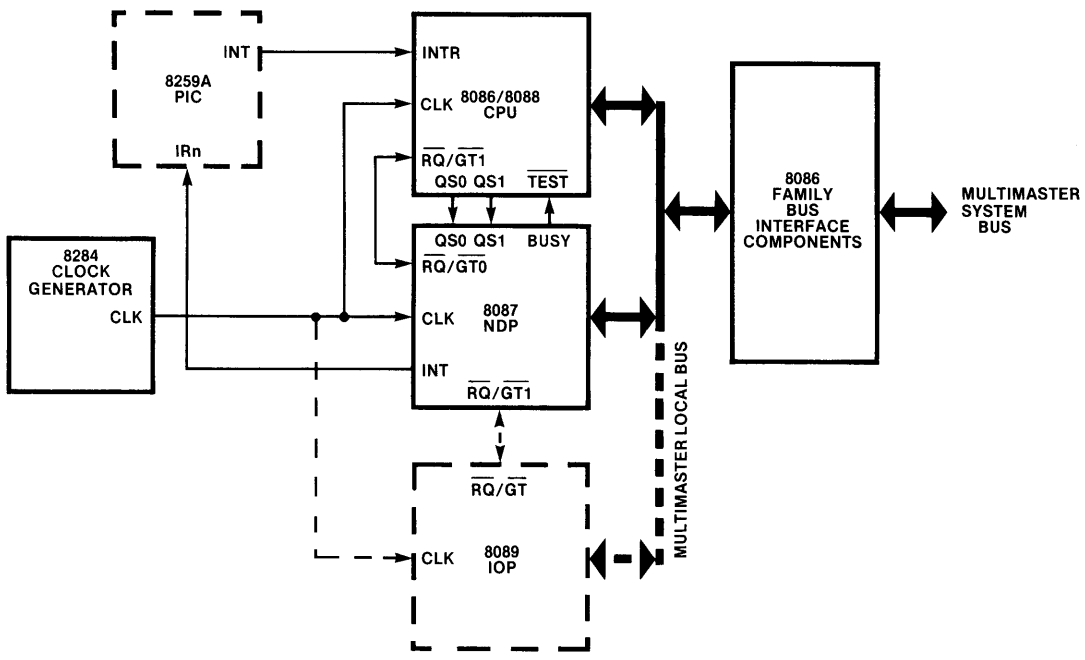


Figure S-3. NDP Interconnect

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

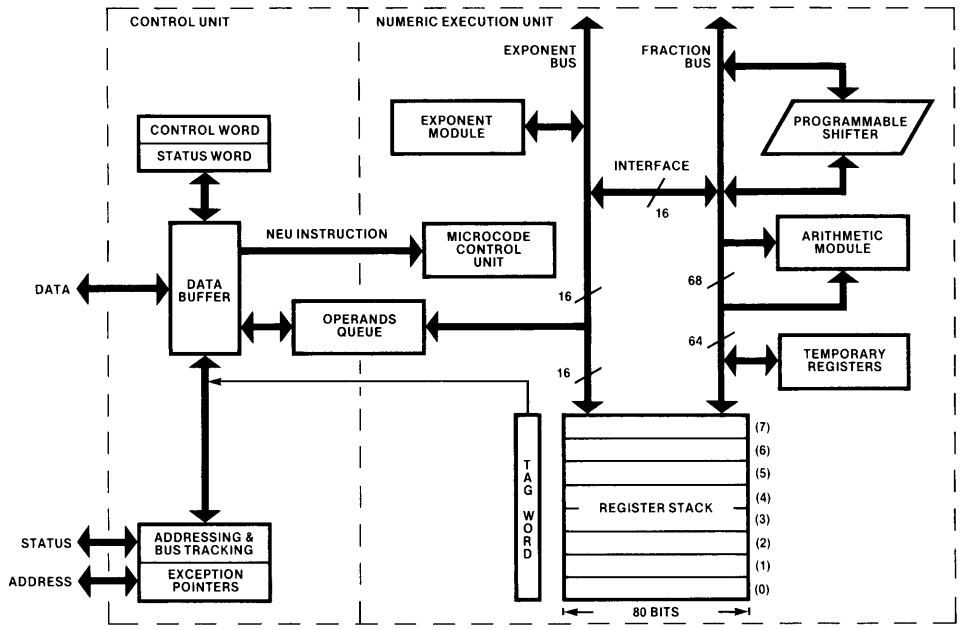


Figure S-4. 8087 Block Diagram

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.

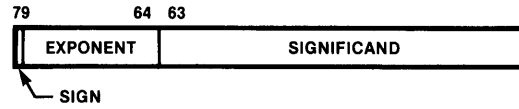


Figure S-5. Register Structure

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 ($0 \leq i \leq 7$). 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.

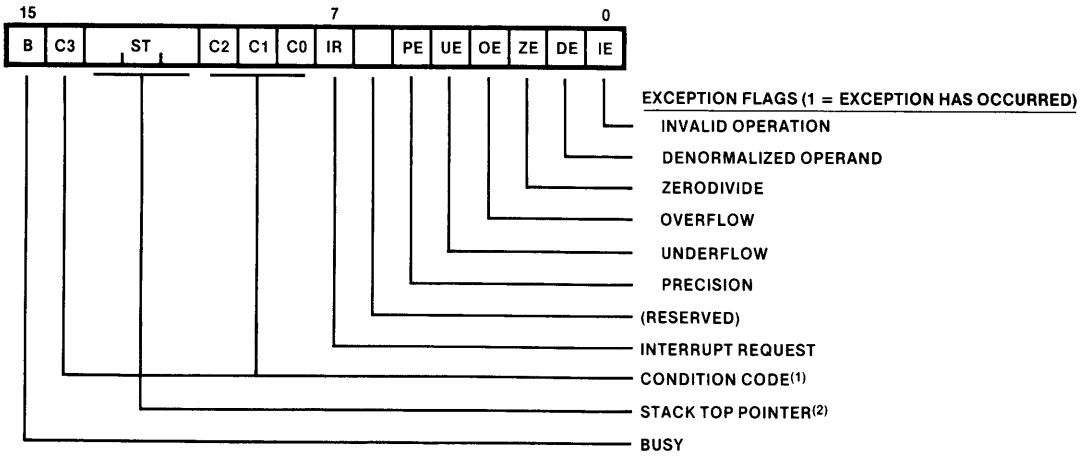
Bits 5-0 are set to indicate that the NEU has detected an exception while executing an instruction. Section S.3 explains these exceptions.

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

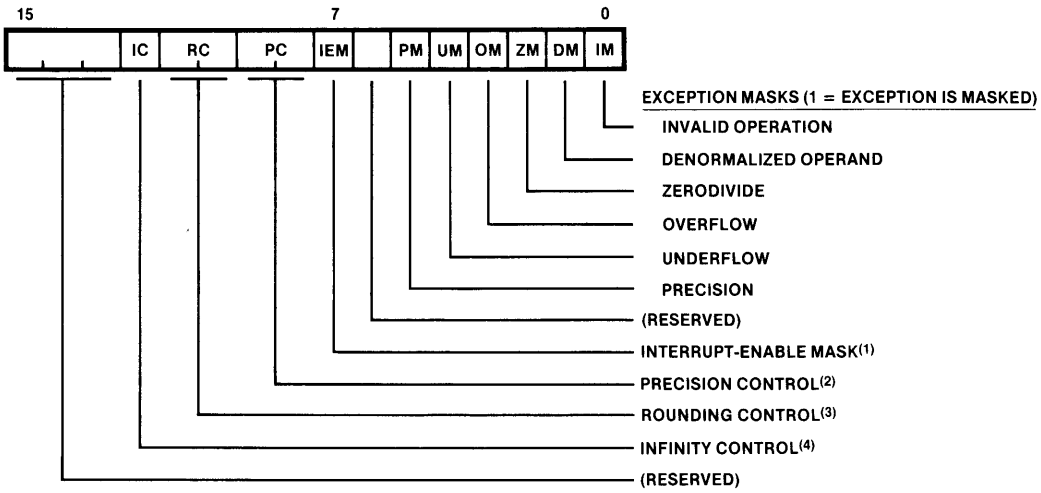
The tag word marks the content of each register as shown in figure S-8. The principal function



(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



(1) Interrupt-Enable Mask:
0 = Interrupts Enabled
1 = Interrupts Disabled (Masked)

(2) Precision Control:
00 = 24 bits
01 = (reserved)
10 = 53 bits
11 = 64 bits

(3) Rounding Control:
00 = Round to Nearest or Even
01 = Round Down (toward $-\infty$)
10 = Round Up (toward $+\infty$)
11 = Chop (Truncate Toward Zero)

(4) Infinity Control:
0 = Projective
1 = Affine

Figure S-7. Control Word Format

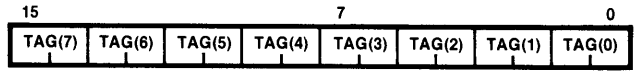
of the tag word is to optimize the NDP's performance under certain circumstances and programmers ordinarily need not be concerned with it.

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.

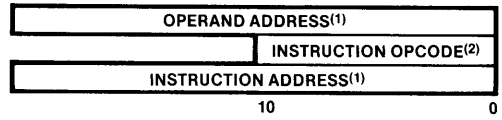
S.3 Computation Fundamentals

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

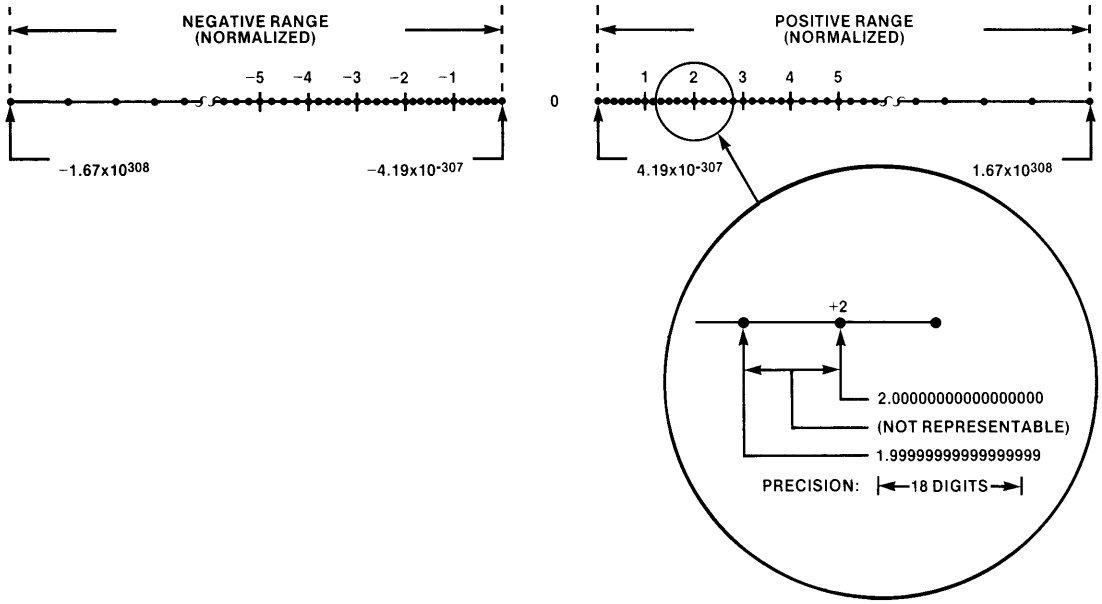


Figure S-10. 8087 Number System

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

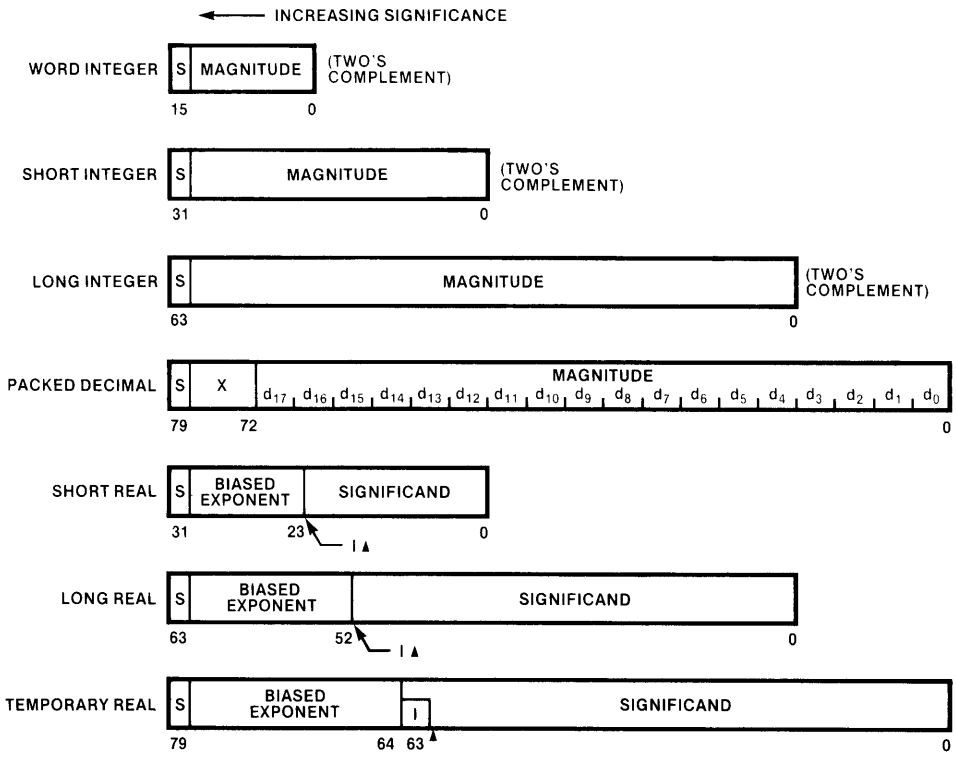
In its internal operations, the 8087 actually employs a number system that is a substantial superset of that shown in figure S-10. The internal format (called temporary real) extends the 8087's range to about $\pm 3.4 \times 10^{-4932}$ to $\pm 1.2 \times 10^{4932}$, and its precision to about 19 (equivalent 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.

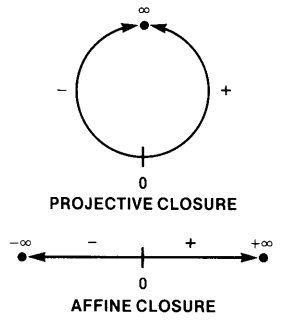


Figure S-12. Projective Versus Affine Closure

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.

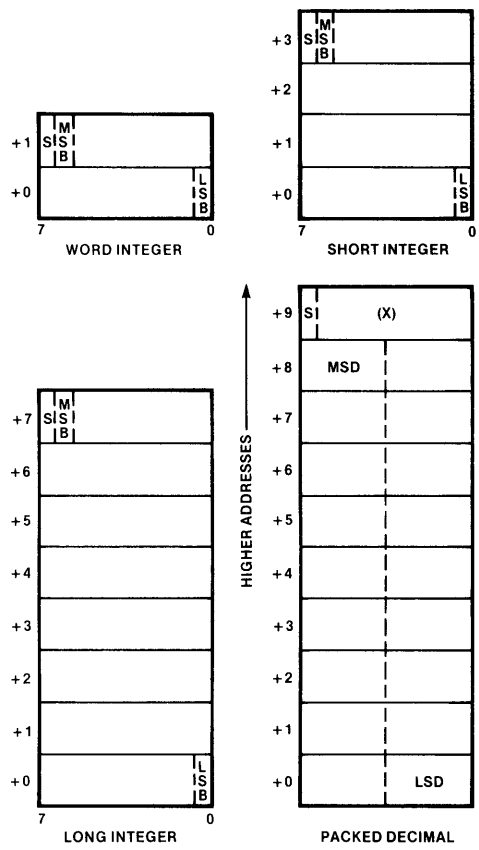
Notice that an exception may or may not constitute an error depending on the application. For example, an invalid operation caused by a stack overflow could signal an ambitious exception handler to extend the register stack to memory and continue running.

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.

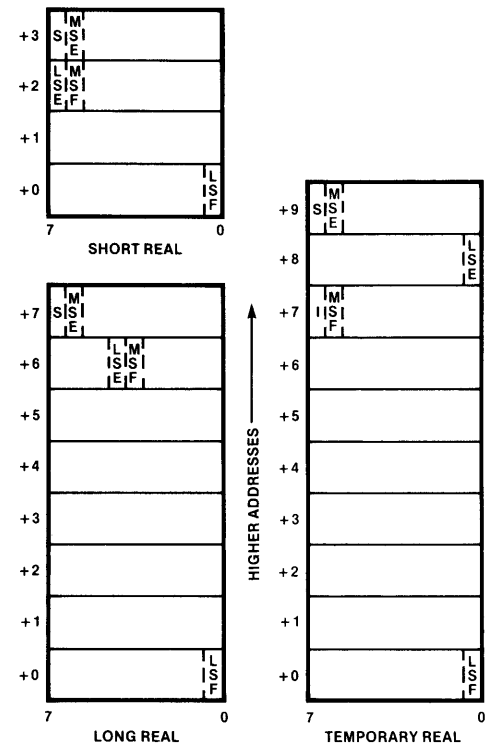
Data Storage

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



S: Sign bit
MSB/LSB: Most/least significant bit
MSD/LSD: Most/least significant decimal digit
(X): Bits have no significance

Figure S-13. Storage of Integer Data Types



S: Sign bit
MSE/LSE: Most/least significant exponent bit
MSF/LSF: Most/least significant fraction bit
I: Integer bit of significand

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
```

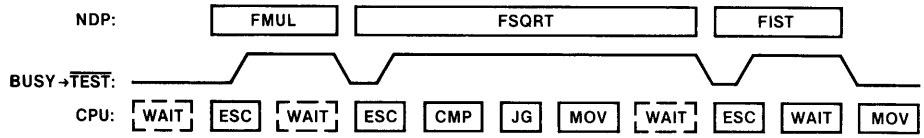
This ensures that the multiply runs to completion before the CPU and the 8087 CU decode the divide.

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

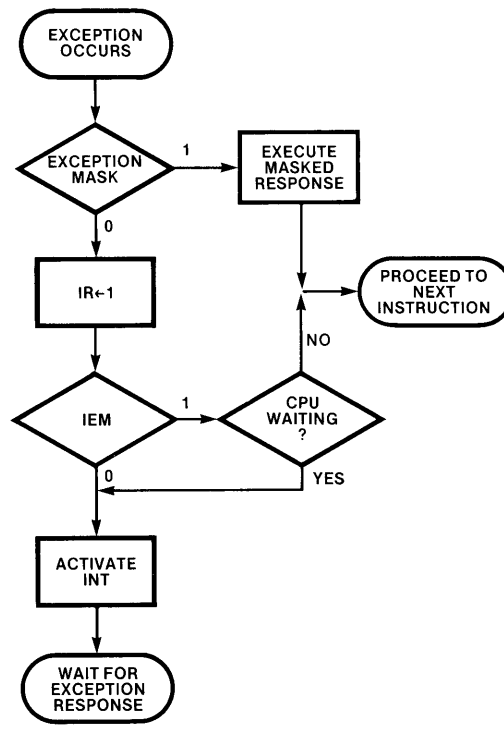


Figure S-16. Interrupt Request Logic

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.