intel

Introduction to the iAPX 286

ORDER NUMBER: 210308-001
AFN-02177A

intel®

Introduction to the iAPX 286

February 1982

Intel Corporation makes no warranty for the use of its products and assumes no responsibility for any errors which may appear in this document nor does it make a commitment to update the information contained herein.

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

BXP, CREDIT, i, ICE, iCS, i_m, iMMX, Insite, Intel, int$_e$l, Intelevision, Intellec, iOSP, iRMX, iSBC, iSBX, Library Manager, MCS, Megachassis, Micromainframe, Micromap, Multimodule, Plug-A-Bubble, PROMPT, RMX/80, System 2000 and UPI.

MDS is an ordering code only and is not used as a product name or trademark. MDS® is a registered trademark of Mohawk Data Sciences Corporation.

*MULTIBUS is a patented Intel bus.

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

Intel Corporation
Literature Department SV3-3
3065 Bowers Avenue
Santa Clara, CA 95051

TABLE OF CONTENTS

APPENDIX A
iAPX 286 INSTRUCTIONS

LIST OF FIGURES AND TABLES

Section 1
Introduction

1

SECTION 1
INTRODUCTION

The iAPX 286 is a new VLSI microprocessor system with exceptional capabilities for supporting large-system applications. Based on a new-generation CPU (the Intel 80286), this powerful microsystem is designed to support multi-user reprogrammable and real-time multitasking applications. Its dedicated system support circuits simplify system hardware; sophisticated hardware and software tools reduce both the time and the cost of product development.

The iAPX 286 microsystem is a total-solution approach that enables you to develop high-speed, interactive, multi-user, multitasking—and multiprocessor—systems more rapidly and at costs lower than ever before.

- Reliability and system "up-time" are becoming increasingly important in all applications. Information must be protected from misuse or accidental loss. The iAPX 286 includes a protection mechanism that continually monitors application and operating system programs to maintain a high degree of system integrity.

- The iAPX 286 provides a large address space to support today's application requirements. Real memory consists of up to 16 megabytes (2^{24} bytes) of RAM or ROM. This large physical memory enables the iAPX 286 to keep many large programs and data structures in memory simultaneously for high-speed access.

- For applications with dynamically changing memory requirements such as multi-user business systems, the iAPX 286 CPU provides on-chip memory management and even virtual memory support. On an iAPX 286-based system, each user can have up to a gigabyte (2^{30} bytes) of virtual address space. This large address space virtually eliminates restrictions on the number or size of programs which may be part of the system.

- Large multi-user or real-time multitasking applications can use the advanced features and high-speed operation of the iAPX 286 to reduce response time for system users. Microcomputer systems servicing four or five simultaneous users may now be expanded to support more than three times that number, and real-time systems can respond in one-sixth the time or less.

The iAPX 286 is instruction-set-compatible with the iAPX 86 and the iAPX 88, ensuring that your investments in existing iAPX 86 software are well protected. This software, as well as an extensive base of third party software written for the iAPX 86 and 88, may be executed as-is on the iAPX 286 to take advantage of its much higher performance.

Upgrading iAPX 86 and 88 application programs to use the new memory management and protection features of the iAPX 286 usually requires only reassembly or recompilation. (Some programs may require minor modification.) This compatibility, along with strong system support from Intel, reduces both the time and the cost of software development. You can bring new products to market quickly.

This Introduction to the iAPX 286 describes the features and capabilities of the iAPX 286 microsystem. Section 2, directed to management, gives an overview of iAPX 286 capabilities. Section 3 is directed to the system designer. It discusses the features of the iAPX 286 in greater detail to provide a working knowledge of its operation. A glossary is included at the end of the manual to define terminology describing the operation of the iAPX 286.

Section 2
Management Overview of the iAPX 286

2

SECTION 2
MANAGEMENT OVERVIEW OF THE iAPX 286

Intel's iAPX 286 microsystem provides the most capable and cost-effective solution for your products. Memory protection, virtual memory, and high throughput rates promote the design of reliable multi-user and multitasking system products.

The iAPX 286 is also software-compatible with the world-standard iAPX 86 and 88 microprocessors—you can take advantage of a rapidly growing base of applications software to support your products.

The complete family of support circuits simplifies creation of hardware systems. Development support tools from Intel also aid the designer in creating system hardware and software.

The iAPX 286 includes many features which provide greater capability, reliability, and throughput than previously available with microprocessors. These features address the needs of your markets in three key ways:

1. iAPX 286 memory protection provides superior system reliability—a requirement for multi-user and multitasking systems.
2. Built-in support for memory management and virtual memory allows you to build flexible and cost-effective systems, and build them faster at less cost. The iAPX 286 enables a system designer to use these features in a variety of ways to best fit the target application.
3. Applications software already written for the iAPX 86 and 88 can be used to put iAPX 286 systems to work immediately.

SUMMARY OF iAPX 286 BENEFITS

Greater reliability—An innovative total-protection system not only isolates the operating system from application programs, but also protects application programs and data from other programs. This protection gives unparalleled software integrity and up-time in a microprocessor-based system.

Higher performance—The iAPX 286 can execute instructions up to six times faster than the iAPX 86. As a result, new features and capabilities can be included in your product.

Protection of software investment—Operating in Real Address Mode, the iAPX 286 can use unmodified programs from the iAPX 86 and 88, while providing up to a six-fold increase in performance over these processors. This compatibility lets you use existing software to develop more powerful iAPX 286-based products. Newer software may then take advantage of the advanced iAPX 286 features available in Protected Virtual Address Mode. A major iAPX 286 advantage is that the designer can add these enhanced features at any time to an iAPX 286-based product without hardware modification.

More software flexibility—Users may customize an operating system without compromising its integrity. Additionally, the architecture and instruction set can be customized to the system needs using CPU extension devices which add new capabilities like high-speed floating-point arithmetic.

Increased data access—The iAPX 286 supports a 16-megabyte real memory space able to contain many programs and data files simultaneously for immediate access. The iAPX 286 can also provide as much as a gigabyte (2^{30} bytes) of virtual memory space per user for even greater programming flexibility.

Decreased development costs—The features of the iAPX 286 and the system development tools from Intel reduce the time required to develop new products. Intel provides a complete set of

hardware and software development tools including high-level languages, single-board computers, in-circuit emulators, and networked development systems to drastically reduce the time and cost of product development. The extensive iAPX 286 family of support circuits, memories, and peripheral control circuits reduces the complexity of hardware design.

iAPX 286 ADDRESSES APPLICATION REQUIREMENTS

The iAPX 286 extends the capabilities of the iAPX 86 and 88 processors to increase the power and simplify the development of multi-user and multitasking systems. The iAPX 286 directly addresses the needs of these systems by increasing address space, reducing system response time, increasing system reliability, and reducing software development costs.

System Reliability and Data Integrity—Never before has there been a microprocessor with such extensive and flexible software protection features to keep systems up and running and to keep data protected. For any application, a system designer can structure the protection mechanism to match the requirements of the product.

- Task isolation—All tasks are completely isolated from interference by other tasks. As a result, users of an iAPX 286-based protected computer system do not have to worry about another user crashing the system or violating the integrity of their programs and data.
- Operating system protection—The iAPX 286 protection mechanism provides up to four privilege levels within each task (see Figure 2-1). The highest privilege level is reserved for the operating system kernel, the most trusted program in the system. Below the kernel level, systems can be configured to include a system service level, an applications service level, and an application program level. These hierarchical privilege levels encourage structured operating system design which makes operating systems simpler to implement and easier to maintain.
- Controlled operating system access—The transparent operation of the iAPX 286 protection mechanism saves the programmer from writing special instructions to request operating system services. Quick but well-controlled access from an application program to operating system service routines preserves the integrity of the operating system as well as the high-speed attributes of the CPU.

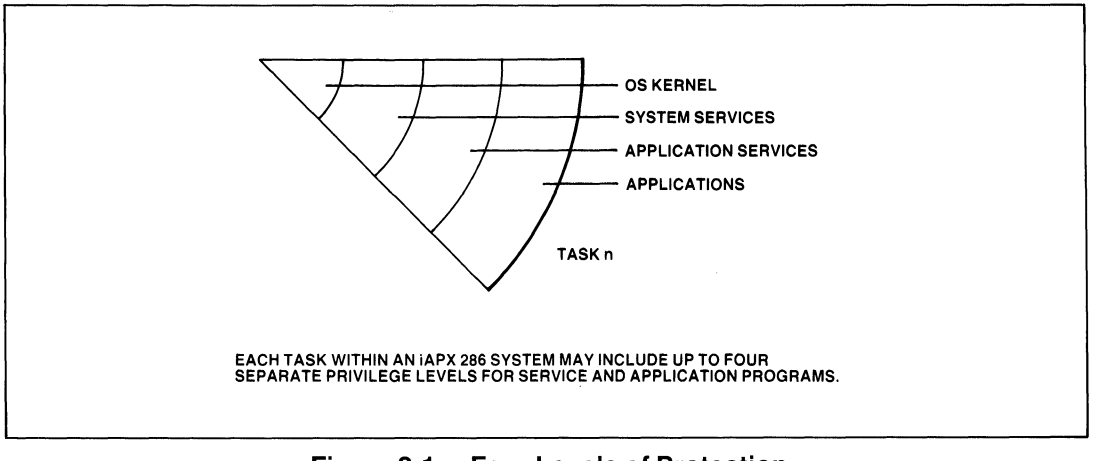

Figure 2-1. Four Levels of Protection

Protection of Software Investments

Software is rapidly becoming the largest investment in any computer system. The iAPX 286 protects both current and future investments in software. It offers two modes of operation which are fully upward compatible with software developed for the iAPX 86 and 88 processors. This feature not only benefits current iAPX 86 and 88 users, but it also allows customers to select from the large base of existing third-party software. Compatibility also extends to include iAPX 86/20, 88/20 Numeric Data Processor systems using the 80287 Numeric Processor extension.

Real Address Mode makes the iAPX 286 appear exactly like an iAPX 86, while executing as much as six times faster. This mode can provide a major savings in software development costs by enabling you to use programs developed for the iAPX 86 and 88 without any modification.

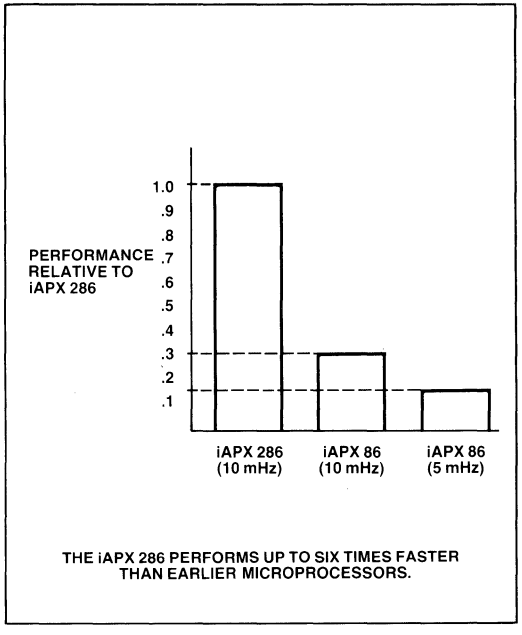

Figure 2.2 iAPX 286 Performance Comparison

Protected Virtual Address Mode (Protected Mode) lets you take advantage of memory management and protection features easily. Protected Mode is used for the highest product reliability, data integrity, and largest address space. This mode also makes it efficient to use the existing iAPX 86 and 88 software.

Application programs developed for the iAPX 86 require little or no modification to use Protected Mode. High-level language application programs only need to be recompiled, while some assembly language programs may require minimal modifications. Because most application programs do not require modification to run on the iAPX 286, the user is assured that this microsystem preserves the original integrity of proven software.

The iAPX 286 completely supports the iAPX 86 instruction set in Protected Mode as well as in Real Address Mode. Intel's commitment to protect software investments continues with the iAPX 286.

Increased Throughput

State-of-the-art Very Large Scale Integration (VLSI) techniques enable the iAPX 286 to execute programs up to six times faster than previous-generation 16-bit microprocessors (see Figure 2-2).

Four processing units—A major performance innovation in the iAPX 286 design is the implementation of four independent processing units within the Central Processing Unit (CPU) (see Figure 2-3). Each independent unit operates to minimize bus requirements and to maximize CPU throughput. The iAPX 286 will be available in both 8-megaHertz and 10-megaHertz versions.

High-performance bus—iAPX 286 design combines a new demultiplexed bus structure with pipelining techniques to more than double bus

efficiency. This increased bus efficiency maximizes the information-handling capacity of the system for greater performance. Pipelined bus operation achieves this higher bus throughput without requiring proportional increases in memory speed. The high-performance bus interface and a 10-megaHertz clock rate provide a bus bandwidth of 10 megabytes per second.

Product Flexibility and Upgradability

The iAPX 286 can address up to 16 megabytes of physical memory. Built-in iAPX 286 virtual memory management support hardware can make a much larger address space available to accommodate even more programs and data. This flexible address space allows you to build products which are easy to upgrade to include new features and capabilities.

This on-chip virtual memory management support enables a system to allocate as much as a gigabyte (2^{30} bytes) of virtual address space to each user. Virtual memory eliminates the problems associated with limited address space. The large address space enables programmers to concentrate their efforts on improving program operation, rather than on designing programs around address space restrictions.

The large physical address space and built-in virtual memory support are ideal for implementing multi-user reprogrammable systems. The number of programs, the size of data bases, and the number of users in the system are not constrained by the size of physical memory.

The operation of the on-chip iAPX 286 memory management unit is transparent and requires no special instructions for its use. Because this memory management support is integral to the CPU, systems based on the iAPX 286 operate faster, are easier to program, and require less hardware and software than systems requiring an external memory management unit.

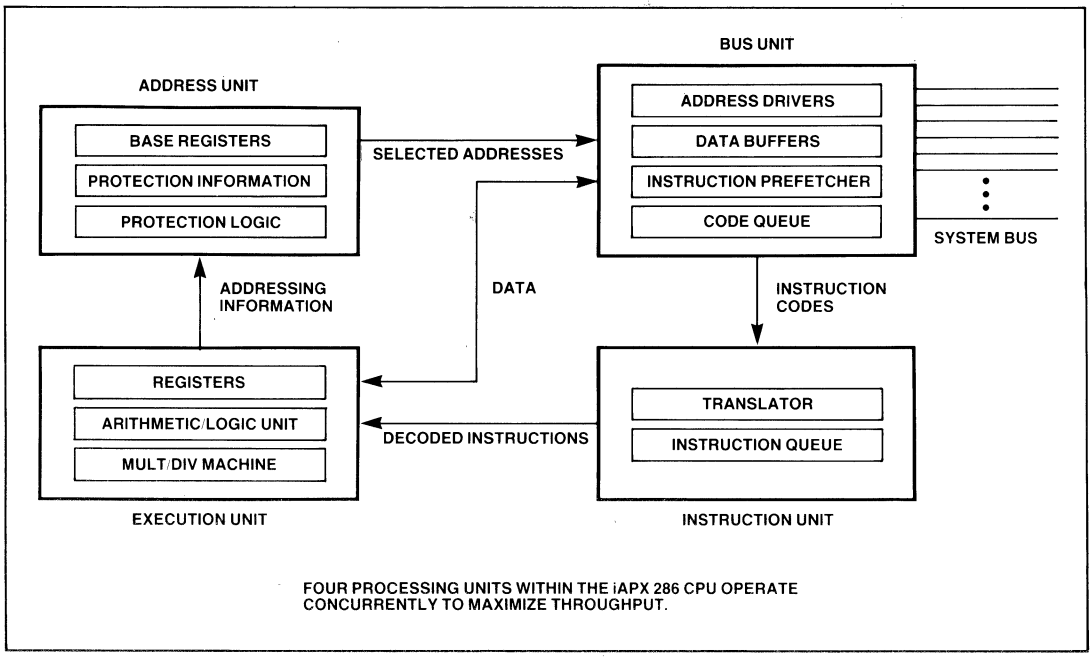

Figure 2-3. Concurrent Processing Enhances Speed

Hardware support for system operations—The iAPX 286 also supports high-speed system operation with specialized on-chip hardware:

- The 16-bit multiply and divide instructions of the iAPX 286 operate more than ten times faster than those of the iAPX 86 and 88. Special instructions for string handling and high-level languages further enhance system performance.

- Multitasking support hardware reduces to less than 18 microseconds the time required to switch processing from one task or user to another. The code requirements for this switch are eliminated because this hardware automatically performs the operations of suspending one program, saving registers, restoring registers, and continuing with another program.

- The processor's high-speed interrupt-handling hardware enables the system to respond quickly to an external event. An iAPX 286 system is capable of responding to an interrupt in less than 3 microseconds.

- Other on-chip hardware simplifies the implementation and speeds the operation of operating system functions such as virtual memory management and memory protection.

Advanced numeric processing—The basic iAPX 286/10 configuration (an 80286 CPU and support circuits) may be expanded to an iAPX 286/20 Numeric Data Processor configuration. This configuration handles number-crunching applications which use up to 64-bit integer or 80-bit floating-point data types. It includes a processor extension that operates concurrently with the 80286 CPU, handling arithmetic calculations up to 100 times faster than comparable software. The iAPX 286/20 configuration conforms to the proposed IEEE Standard P754 for numeric processing.

In summary, the hardware of the 80286 CPU maximizes the efficiency of both the processor's instruction set and the system bus structure. iAPX 286 instructions implement extended functions that typically require many instructions. Additionally, pipelined memory access techniques further enhance processing speed by enabling the CPU to achieve higher bus bandwidth. These features result in the greatest performance while minimizing system cost by reducing the need for higher-speed memories.

New Architecture for Simplified Software Development

The instructions of the iAPX 286 increase code density and execution speed. The iAPX 286 instruction set includes many instructions for powerful data and arithmetic manipulations, as well as enhanced high-level language code efficiency and performance. These instructions and the large address space of the iAPX 286 encourage the use of high-level languages for writing efficient operating systems and application programs.

Hardware integration of both virtual memory support and memory protection significantly reduces the software required to provide these features. Because the iAPX 286 hardware directly controls the most performance-critical functions—task switching and virtual address translation—it reduces the size of an operating system and maximizes system performance. Because the virtual memory support is an integral part of the iAPX 286 architecture, virtual memory address translation is also transparent to software.

The hardware of the iAPX 286 enables the operating system to handle the program and data swapping necessary for virtual memory. When a program requests a segment which is in secondary storage (such as a disk drive), the processor initiates the operations to swap the desired

information into real memory. Swapping operations are completely transparent to the application program.

iAPX 286 instructions are fully restartable to simplify returning to the application program after the completion of program and data swapping. Instruction restartability, combined with on-chip virtual memory support, greatly simplifies the tasks of both hardware and operating system software designers.

The iAPX 286 protection mechanism does even more to simplify both operating system software and applications software development. Application programs do not require special instructions to call the operating system while task switching and parameter passing to the operating system require no software overhead in the operating system.

As a result of its advanced architecture, the iAPX 286 is easy to program. The operation of the address translation and protection mechanisms is handled transparently so that application programs remain uncomplicated. Programs are smaller and cost less to develop. It is this transparency which allows the iAPX 286 to execute programs for the iAPX 86 and 88 with little or no modification, even though they were not originally written for a virtual memory system.

FLEXIBLE HARDWARE CONFIGURATIONS

The iAPX 286 offers extensive system configuration options. The 68-pin CPU provides a full 16-megabyte address bus and a 16-bit data bus. Because the memory management and protection support is incorporated within the CPU, a protected virtual memory system may be implemented without requiring external memory management units (MMUs).

The configuration options range from single-processor dedicated-function systems to multiprocessor, multi-user reprogrammable systems. Specialized support circuits from Intel accommodate dynamic memory interfaces, multiprocessor environments, and CPU extensions such as the 80287 Numeric Processor Extension.

Intel provides a large selection of peripheral devices for interfacing the iAPX 286 with mass storage systems, video displays, and communications networks. You can implement memory systems with a variety of ROM, EPROM, E^2PROM and static or dynamic memories offering a range of speed and density options: The iAPX 286 also supports a Multibus™ system bus interface. Using this interface allows immediate configuration of systems with the extensive base of existing Multibus compatible memory, peripheral control, and CPU boards.

Numeric Data Processor

The iAPX 286/20 Numeric Data Processor configuration provides the high-speed mathematical capabilities many systems demand. Including the 80287 Numeric Processor Extension, this configuration meets the specifications of the proposed IEEE standard P754 for numeric data processing and maintains compatibility with iAPX 86/20 systems. A software emulation package is also available for lower performance systems, to execute in both Real Address and Protected Modes.

Multiprocessor Capabilities

A system designer can easily configure a multiprocessor system using the iAPX 286. The integrated memory management unit, along with special support circuits for system bus interfacing and arbitration, allows well-controlled communications on a multiprocessor bus. Bus control and arbitration support circuits provide flexible solutions for the construction of multiprocessor bus structures like the Intel Multibus (IEEE Standard 796).

System Support Circuits

Support circuits for use in iAPX 286 systems include circuits designed for bus interface, memory, and peripheral system support.

Bus interface support is provided by chips which handle tasks such as system timing, bus communications, and control. These circuits reduce the task of hardware design to the selection of the appropriate component to provide the required function. Examples of this support group are bus controllers, data transceivers, and address latches.

Intel's Advanced Dynamic RAM Controller supports a direct interface between dynamic RAM (DRAM) and the CPU of an iAPX 286 system. It also provides dual-port operation for multiprocessor systems and is compatible with Intel's 8206 Error Detection and Correction Chip (ECC) for systems requiring a high degree of data integrity.

Intel manufactures specialized controllers which provide peripheral support for dedicated system functions.

- To accommodate mass storage needs, Intel manufactures peripheral controllers for bubble memory, floppy diskette, and hard disk systems.
- Video display controllers include a range of simple to sophisticated alphanumeric displays and a graphic display control device.
- Communications controllers can manage simple asynchronous protocols as well as bit-synchronous protocols like SDLC. The communications options for the iAPX 286 will also include advanced techniques like Ethernet* for local area network applications.

*Ethernet is a trademark of the Xerox Corporation. The Ethernet protocol was developed jointly by DEC, Xerox, and Intel.

Board Systems

The iAPX 286 will be available in both component form, and as a preconfigured, pretested Multibus-based single-board computer from Intel for prototyping and for production requirements. The Multibus system bus interface not only provides multiprocessor support, but it also enables you to quickly design complete iAPX 286 systems using available board-level memory and peripheral control products. Using these support products, you can bring your products to market with minimum delay and design effort.

This full complement of system support circuits and board products enables the iAPX 286 to address a wider range of computer applications than any previous microprocessor system. Intel's broad range of modular solutions enables you to develop simple to sophisticated systems quickly and easily.

SAMPLE APPLICATIONS

The iAPX 286 is ideal for multi-user, real-time multitasking, and multiprocessor systems. It can support high-performance, real-time applications which have been traditionally beyond the capabilities of a microprocessor-based system. This section briefly describes a few of the potential applications for this advanced microprocessor system.

iAPX 86 System Upgrade

The iAPX 286 Real Address Mode enables an iAPX 286-based hardware system to use existing iAPX 86 programs without modification. The major benefits of this system migration path are:

- Total preservation of software investment
- As much as a six-fold increase in throughput

Later, as end-product requirements grow, the system software may be modified in stages to add software protection, without requiring hardware changes.

Multi-user Systems

The high execution speeds, total task isolation, and operating system protection make the iAPX 286 ideal for multi-user systems. The high execution speeds enable a system to handle more users with improved response time for each user. Intertask isolation protects one user's code and data from another's. Operating system protection provides total system integrity.

The iAPX 286 is ideal for multi-user business systems. The protection features can separate private financial data from word-processing or order-entry data even as the data is being processed. The virtual memory support gives each user high system availability without restricting simultaneous use of programs. Separate tasks can be created to handle slow I/O devices such as printers or modems; these tasks can run simultaneously with user service routines without degrading user response.

The iAPX 286 provides the flexibility to swap programs and data in and out of memory for virtual memory systems. This flexibility reduces the amount of real memory required to operate a large system which, in turn, reduces system cost.

Real-time Multitasking Systems

The iAPX 286 hardware task-switching support considerably reduces the time and code required to save the state of one task and load another. The multitasking capability of the iAPX 286 handles complex applications such as a large PABX system controlling thousands of lines. Distributed process control is another application which can take advantage of the multitasking and protection capabilities of the iAPX 286.

Multiprocessor Systems

Many applications can benefit from the concurrent operation of multiple processors. A typical application for multiple processors is a system with dedicated I/O processors which assume the entire burden of driving a complex I/O device. The iAPX 286 is an ideal master processor for these systems.

The iAPX 286 simplifies the design of multiprocessor configurations with hardware that supports the industry-standard Multibus structure. The iAPX 286 protection mechanism also provides support for multiprocessor communications to simplify the task of programming concurrently operating CPUs.

This interprocessor compatibility enables a system designer to construct a multiprocessor system using any members of the family. For example, an iAPX 286 CPU can serve as a master controller for a series of I/O processors (such as the Intel 8089) which, in turn, control the I/O for the system. In this type of a system, the dedicated processors remove the I/O overhead from the master CPU to increase system throughput.

Real-time High-performance Systems

The high-speed operation of the iAPX 286 opens new doors to applications which have been neither feasible with slower microprocessor systems nor cost-effective on minicomputer systems. Higher speed operation allows you to add more features and more users without degrading system response time. Image processing, graphics, and educational computing systems are examples of real-time applications for the iAPX 286's high-speed, economical hardware.

DEVELOPMENT SYSTEMS AND SOFTWARE

Intel assists the system designer with assemblers, compilers, development utilities, and an in-circuit emulator for debugging which decrease both the time and the costs of producing software. From single-board computers to full

hardware development systems, Intel provides an environment for the creation of extremely sophisticated microcomputer products.

Intellec™ Microcomputer Development Systems

Intel produces the Intellec Series III Microcomputer Development System as a complete center for the development of microcomputer-based products. This system includes an iAPX 86 CPU, a hard-disk system interface with operating software, and a CRT with a 2000-character display and detachable keyboard.

In-circuit emulation (ICE™) modules and software will be available for the Series III to simplify the development of iAPX 286 hardware systems. This option enables a designer to individually develop and test software modules and then integrate them to completely isolate various system functions for testing and analysis.

Intel also manufactures the Intellec Model 675 Network Development System. This development system provides the tools to develop and test communications software and applications using Ethernet for office automation systems. An Ethernet communications controller incorporates the Ethernet Data Link and Physical Link Control to provide a 10-megabit-per-second data transmission rate over coaxial cable.

These development systems are compatible with the wide range of hardware designed for the Intel Multibus. Software compatibility among the Intel development systems enables the use of the programs described below for the rapid development of microcomputer-based products.

Evaluation Software

The Intellec Series III Development System supports an iAPX 286 Evaluation Package. This is a package of programs that enables system designers to evaluate the operation and capabilities of the iAPX 286.

The Evaluation Package includes all of the tools a programmer needs to become familiar with the operation of the iAPX 286 protection mechanism and other new features. The simulator enables a programmer to gain experience by developing software modules which can later be used directly on an iAPX 286 system.

Development Software

Intel's wide range of development programs and high-level languages attests to its tradition of strong support for the development of hardware and software products. The Development Software Release for the iAPX 286 has the right mix of tools to simplify the development of all levels of software from application programs to operating systems. These tools include high-level languages, an assembler, and utilities for combining and structuring software.

The Development Software Release includes utility programs that can eliminate much of the time, effort, and expense involved in creating system software. These utilities include a binder that links modules written in different languages into a single program. This capability allows a programmer to write each module in the most efficient language, rather than having to write the entire system in a single language.

A system-building utility simplifies the generation and maintenance of operating systems. It also provides an easy means of specifying the operation of the iAPX 286 software protection mechanism. Other utilities provide run-time support for high-level languages such as Pascal and FORTRAN.

High-level Languages

High-level languages are most valuable for reducing the time and cost of program development and maintenance. Intel's efficient compilers, combined with the large address space and

the high performance of the iAPX 286, encourage the use of high-level languages for the development of software. Intel's structured system language, PL/M, will be the first high-level language available for the iAPX 286. Following PL/M will be Pascal and FORTRAN.

SUMMARY

The iAPX 286 is a microsystem for the '80s. On increased throughput alone, the iAPX 286 is a major advancement in microprocessor technology. In addition, its extraordinary on-chip memory management support, complemented by the powerful protection mechanism, gives the iAPX 286 the ability to address applications never before served by microprocessors.

Recognizing that supplying a complete microsystem solution can reduce software costs, Intel produces software and hardware development systems that significantly simplify the program development cycle.

Versatile support components from Intel simplify the creation of systems around the iAPX 286; iAPX 86-family compatibility ensures volumes of compatible software. The high performance level encourages the use of high-level languages for efficient program development. The four-level protection mechanism allows the high degree of operating system and user program integrity required of new applications. The large virtual address space and virtual memory management supports the dynamic memory demands of multi-user systems while maximizing the efficient use of real memory.

The iAPX 286 microsystem provides the framework for the development of extremely powerful products. This capability—with strong support from Intel—makes the iAPX 286 the ideal choice for applications ranging from intelligent reprogrammable workstations to complete multi-user, real-time multitasking, interactive systems.

Section 3
Technical Overview of the iAPX 286

3

SECTION 3
TECHNICAL OVERVIEW OF THE iAPX 286

This technical overview presents the concepts governing the operation of the iAPX 286 microsystem. The discussion covers all aspects of the iAPX 286 microsystem, including the 80286 CPU as well as support devices and optional extensions.

The material in the technical overview is organized by levels of increasing detail. Readers familiar with the basic operation of the iAPX 86 or 88 may choose to begin directly with the later sections which discuss the unique attributes of the iAPX 286.

The first section describes the iAPX 286 basic programming model. This section explains the registers, data types, addressing modes, and instructions which handle most of the processing functions within the system. These features are representative of the base iAPX 286 architecture without extensions.

The second section of the technical overview describes the extended capabilities which are available with the addition of a Numeric Processor Extension to an iAPX 286 system. These capabilities include new floating-point register extensions for high-speed extended-precision arithmetic calculations.

The third section of the technical overview expands the description of the basic programming model to include features which provide special capabilities. These advanced features include special data types, enhanced instructions, special addressing modes, and protection of software.

The iAPX 286 provides two modes of operation, Real Address Mode and Protected Virtual Address Mode (or Protected Mode). Both modes are compatible with the iAPX 86 and 88 CPU's so that most existing application programs can execute in either mode with little or no modifications.

On power-up, the iAPX 286 begins execution in Real Address Mode and can address up to a megabyte (2^{20} bytes) of physical memory. The software may switch the CPU from Real Address Mode to Protected Mode, expanding the address space up to 16 megabytes (2^{24} bytes) of real memory within the system and up to a gigabyte (2^{30} bytes) of virtual memory per user (see Figure 3-1). This expansion in programmer visible address space is the primary difference in programs that migrate from one mode to the other.

BASIC PROGRAMMING MODEL

The basic programming model described here extends across the range of 16-bit processors from Intel. The architecture of the iAPX 86, 88, 186, and 286 simplifies programming by supporting the most commonly used data types as operands. It also provides a wide range of flexible addressing modes and a powerful instruction set.

The features described in this section apply to both modes of iAPX 286 operation. An application program written for Real Address Mode is also compatible with Protected Mode operation.

Basic Register Model

The 80286 basic register model includes the ten registers shown in Figure 3-2. The eight 16-bit general-purpose registers (AX to SP) contain arithmetic and logical operands; the two special-purpose registers reflect certain aspects of the 80286 processor state. The register set is symmetrically addressable for all basic move, arithmetic, and logical operations.

Four of the general data registers may be divided into high and low sections for manipulating byte variables. These are the AX, BX, CX, and DX

registers. The corresponding eight 8-bit registers are AH, AL, BH, BL, CH, CL, DH, and DL. This allows efficient manipulation of individual bytes within words such as character manipulation within strings of data.

Not only are arithmetic and logical operations available with all general data registers, but each of these registers also handles special functions for certain instructions. To provide code efficiency, the iAPX 286 includes compact instruction encodings with implied use of general registers for these instructions. These instructions extend the base instruction set which operates on the general register set. As examples, the AX and DX registers contain operands for Multiply and Divide instructions or I/O addresses and operands for I/O instructions. Additionally, the CX register can maintain a count value for automatic iteration control during the execution of software loops.

The SP register in Figure 3-2 is the stack pointer. The stack pointer normally indicates the offset address for the current top of stack. Push, Pop, and other instructions implicitly refer to the stack. These instructions use SP as a destination or source operand address and automatically increment or decrement the value of SP as part of their operation.

Four of the general data registers may also serve as base and index registers for flexible memory addressing. The two base registers are the base register (BX) for addressing data and the stack frame register base (BP) for addressing stack information. The two index registers are the SI (source index) and the DI (destination index) registers. A programmer uses the index registers for general data addressing and to simultaneously specify two operands for memory-to-memory string operations.

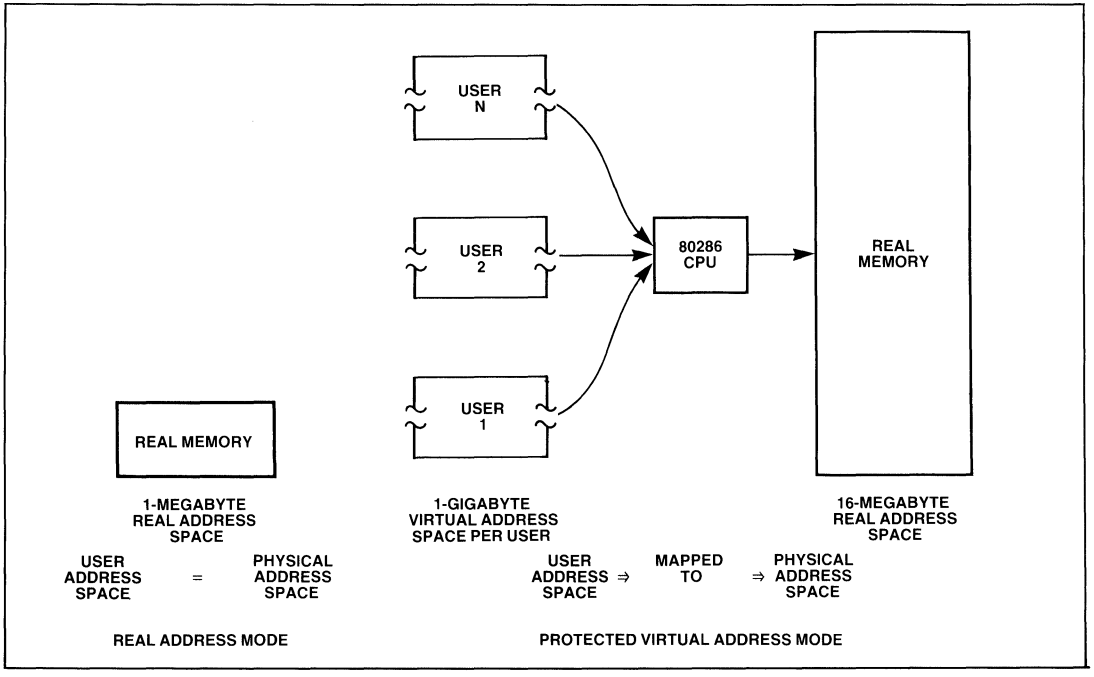

Figure 3-1. Address Space of Real Address Mode and Protected Virtual Address Mode

The two special-purpose registers included in the base register model are the instruction pointer (IP) and the flag register. The instruction pointer contains the offset address of the next sequential instruction for execution. This offset value is automatically incremented to point to the next instruction unless the program jumps, calls, or returns to a different location within that code section. The 80286 flag register contains 12 status bits of which 9 are part of the base register model discussed in this section.

Basic Data Types

The base architecture includes five types of data representations for arithmetic operations: unsigned binary (ordinal), signed binary (integers), unsigned packed decimal (BCD), unsigned unpacked decimal (BCD), and ASCII (using unpacked BCD). Multiprecision instructions can extend a numeric data type to any desired length. Also supported are a string data type for character manipulation and a pointer data type to represent addresses.

Binary numbers may be either 8 or 16 bits in length. Unsigned 8-bit binary numbers have a range of 0 to 255 and signed 8-bit binary numbers have a range of -128 to $+127$. Unsigned 16-bit binary numbers have a range of 0 to 65,535 while signed 16-bit binary numbers have a range of $-32{,}768$ to $+32{,}767$.

Decimal numbers are a succession of 8-bit quantities. Packed decimal numbers are stored as unsigned byte quantities each with a range of 0 to 99. Unpacked decimal numbers are stored as byte quantities each with a range of 0 to 9. The unpacked BCD data type is commonly used for arithmetic operations involving ASCII numbers.

The base architecture supports string manipulation with special string instructions. These instructions treat any contiguous block of memory up to 64K bytes as a single unit which may be input, output, moved, compared, or scanned without resorting to a multi-instruction loop. Similarly the Compare instruction enables the programmer to use two separate blocks of memory for sources, while the Move instruction supports the use of one memory block as a source and a second memory block as a destination.

Basic Addressing Modes

The instruction set enables operands to be immediate, register, or memory based. All instructions, except string manipulation instructions, may use combinations of immediate, register, and memory-based operands. The combinations available to all classes of instructions are:

- Register to register
- Immediate to register

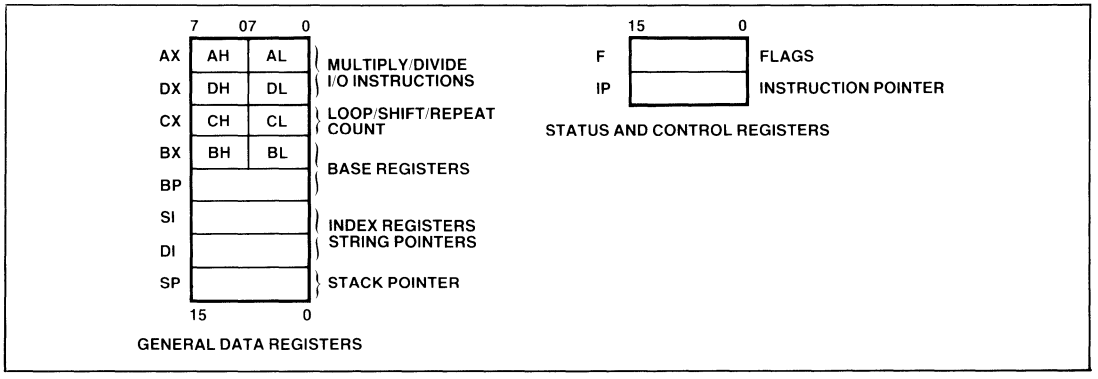

Figure 3-2. Program Registers of the Base Architecture

- Register to memory
- Memory to register
- Immediate to memory

The string instructions and stack manipulation instructions (such as Push and Pop) also allow memory-to-memory operations.

An immediate operand is a constant which is stored immediately following the instruction which uses it (see Figure 3-3). Immediate operands may be 8-bit or 16-bit values, and 8-bit operands are sign-extended for compact representation of 16 bit operands. Register operands reside within the registers of the iAPX 286 (see Figure 3-4). These registers are symmetrically accessible to the instruction set so that an instruction may use as operands any of the eight byte registers (AH to DL) or the eight word registers (AX to SP).

To access data in memory, an effective address provides an offset into the section of memory designated for data storage. A programmer can generate an effective address in a variety of ways to allow the program to address a given data structure in the most efficient manner. An 8-megaHertz iAPX 286 forms a complete address, for all addressing modes, in no more than 125 nanoseconds.

Each instruction that addresses memory contains an address mode field and an optional displacement. Five basic methods of generating effective addresses are described below.

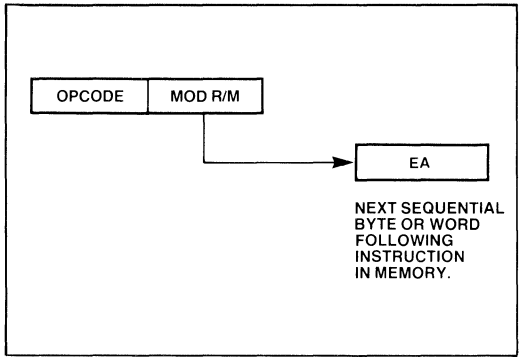

Figure 3-3. Immediate Addressing

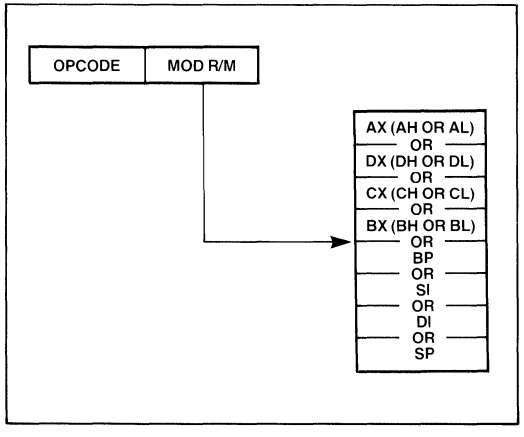

Figure 3-4. Register Addressing

DIRECT ADDRESSING

A program can use direct addressing to access all data types. The term direct addressing refers to an operation that uses a value specified in the instruction as a direct offset to the desired location. A programmer normally uses direct addressing to access constants or static variables stored in memory (see Figure 3-5).

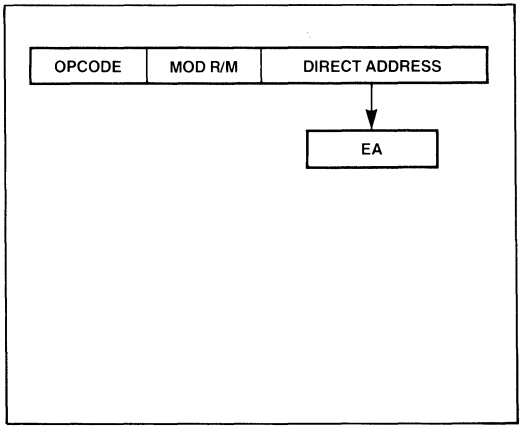

Figure 3-5. Direct Addressing

INDIRECT ADDRESSING

An indirect address expression may include a base register, an index register, and a displacement value, or it may consist of various combinations of these three elements. The displacement values may be either 16-bit or 8-bit sign-extended for efficient instruction encoding. A program may easily address a wide variety of data structures using the different indirect addressing modes available with the iAPX 286, described below.

Register Indirect Addressing

For register indirect addressing, an instruction takes the effective address of a memory operand directly from one of the index or base registers

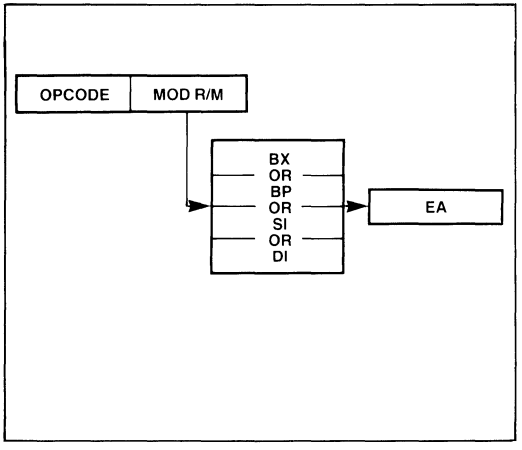

Figure 3-6. Register Indirect Addressing

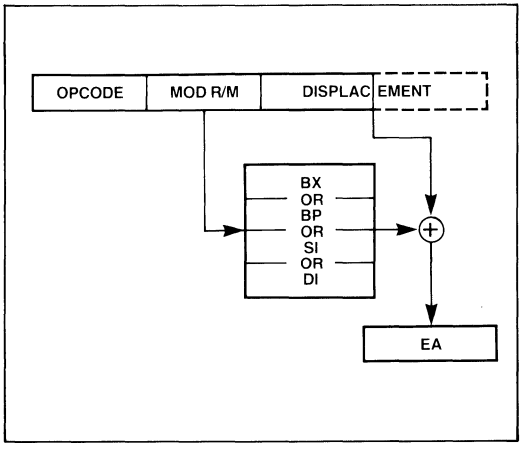

Figure 3-7. Based Addressing

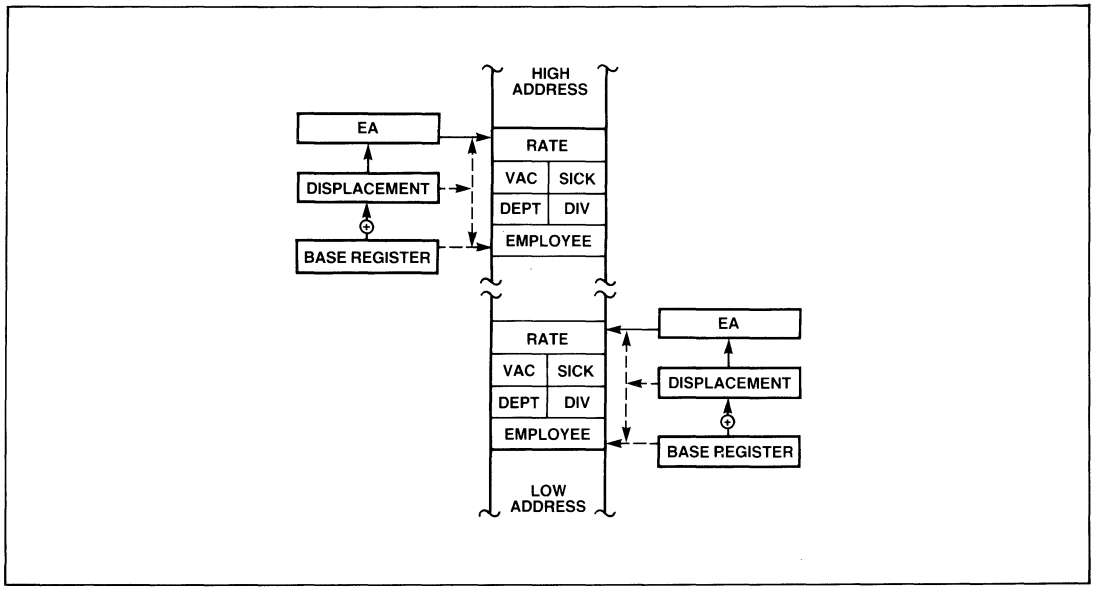

Figure 3-8. Accessing a Structure with Based Addressing

(see Figure 3-6). Using this addressing method, an instruction can operate on different memory locations according to the current value of a register.

Based or Indexed Addressing

Based addressing enables a program to form an effective address by specifying a register (BX, SI, DI, or BP) and adding the displacement value within the instruction to the value within the specified register (see Figure 3-7). This form of addressing is a convenient way to address stack data (using BP) as well as to address data structures which may be located at different places in memory using BX, SI, or DI (see Figure 3-8).

Based addressing enables a programmer to write a routine to access a block of memory which does not have a known location until run time. An example of this situation is a procedure which reads an array from the routine that called it. By using a register to indicate the base address of an array, a programmer can use the same procedure to process arrays at different locations.

A programmer may also use either of the two index registers, SI and DI, or the two base registers, BX and BP, for indexed addressing (see Figure 3-9). The instruction specifies a register and supplies a displacement value. The processor adds the value within the index register to the value of the displacement to form an effective address. By placing the location of the beginning of an array in the displacement field of an instruction, a programmer may address the entire array using indexed addressing methods.

The iAPX 286 indexed addressing enables a programmer to easily read or write information

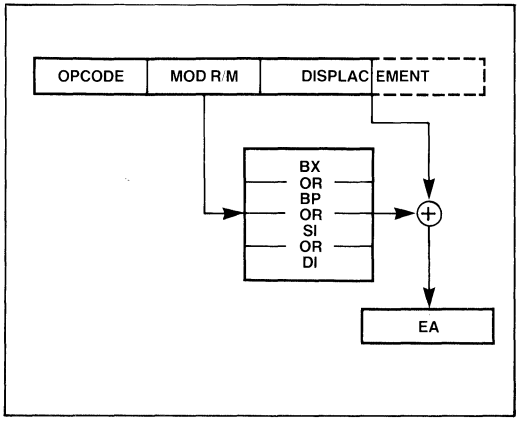

Figure 3-9. Indexed Addressing

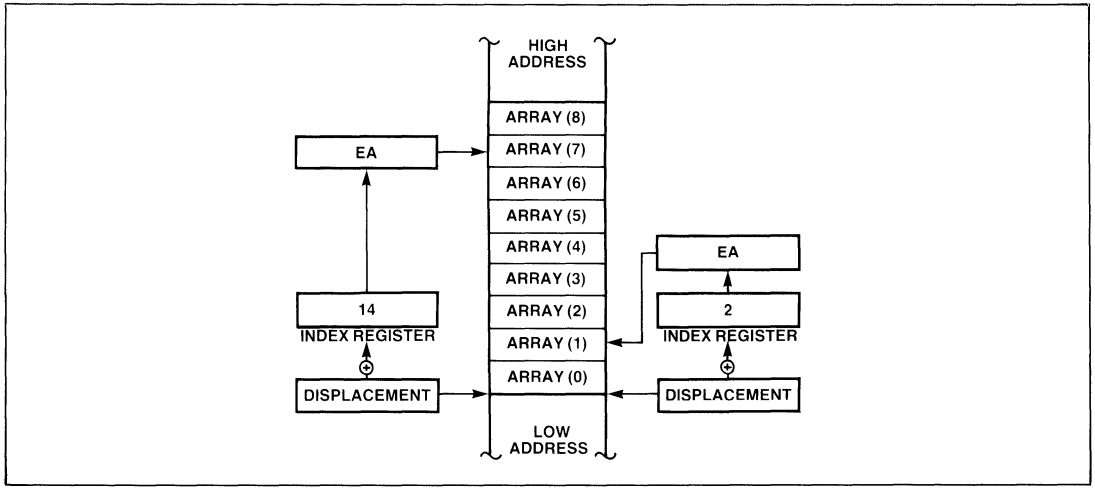

Figure 3-10. Accessing an Array with Indexed Addressing

stored in array form. A program accesses fixed arrays stored in memory using the source index and destination index. A displacement value within a read or write instruction indicates the start of the array; the program increments or decrements the value in the register to refer to different elements within the array (see Figure 3-10).

Based Indexed Addressing

Based indexed addressing generates an effective address that is the sum of a base register, an index register, and the displacement within the instruction (see Figure 3-11). Based indexed addressing is very flexible because two address components can be varied at execution time.

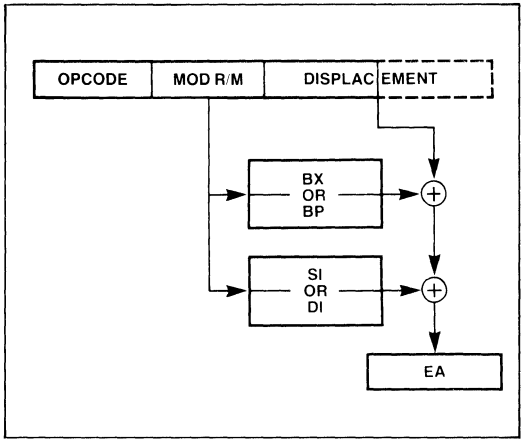

Figure 3-11. Based Indexed Addressing

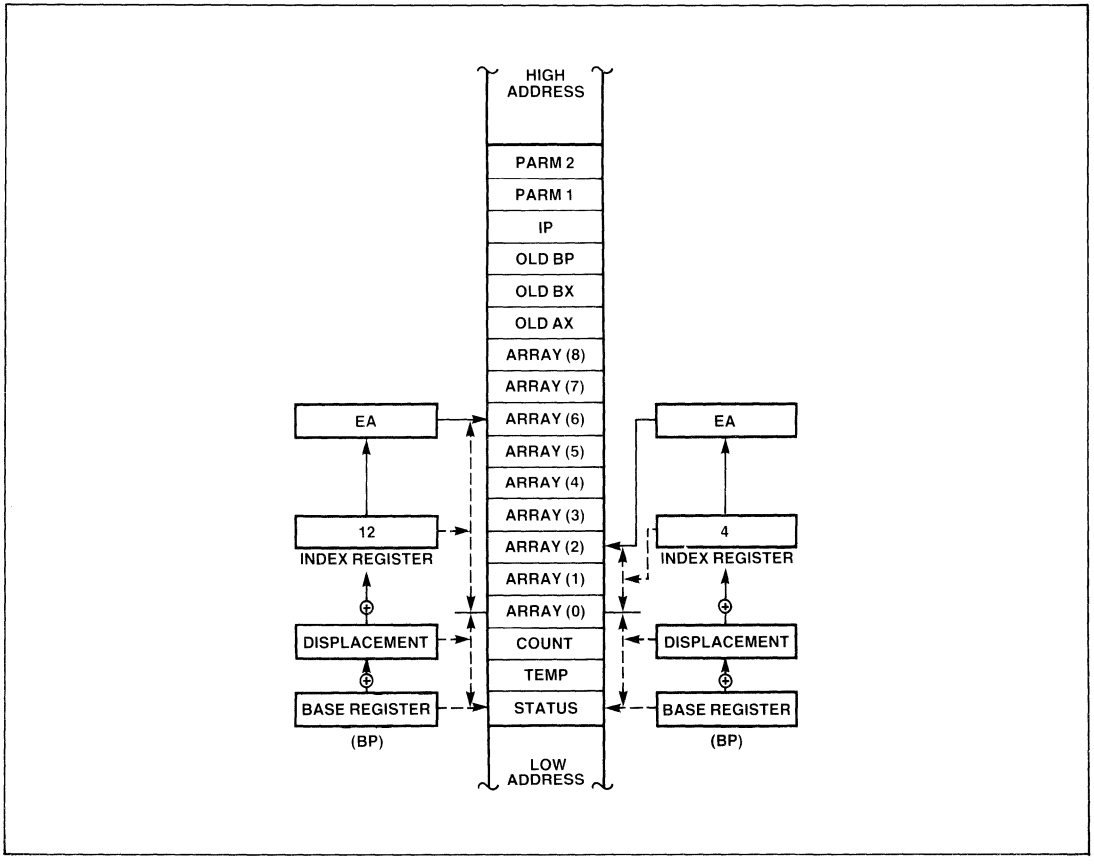

Figure 3-12. Accessing a Stack Array with Based Indexed Addressing

In the case where a program must address record elements within an array of records, the formation of an address may require two levels of indirection as well as a displacement value. For this application, two registers supply base and index values and the optional displacement within the read (or write) instruction specifies the exact location of the record element within the selected record.

To access parameters passed on the stack or dynamic work space allocated on the stack, a program uses the BP register as a base pointer to the stack (see Figure 3-12). Using the BP register as a base pointer automatically refers an instruction to the section of memory containing stack data. This capability supports recursive procedures and the dynamic variables of block-structured high-level languages.

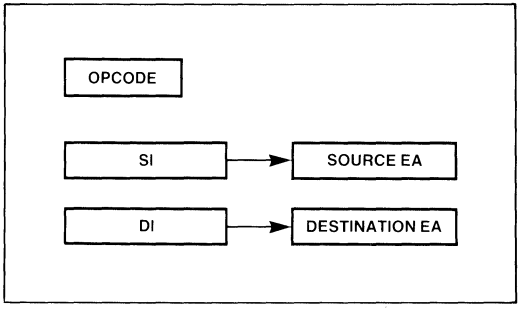

Figure 3-13. String Addressing

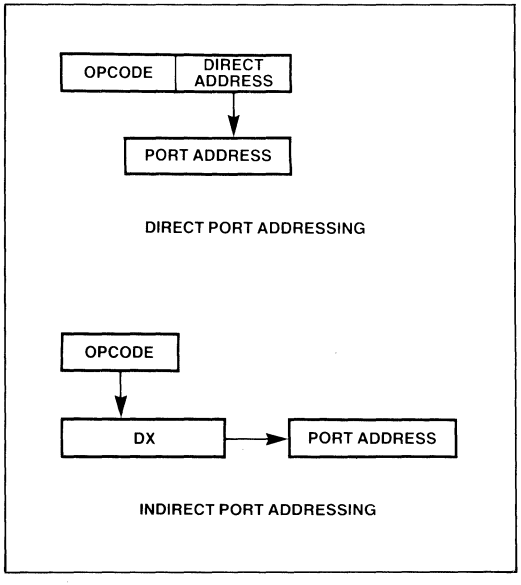

Figure 3-14. I/O Port Addressing

STRING ADDRESSING

Instructions that manipulate string data types use the SI and DI registers to supply the effective address of the source operand and the effective address of the destination operand, respectively (see Figure 3-13). When the processor executes a string instruction, it assumes that SI points to the first byte or word of a source string and that DI points to the first byte or word in a destination string. This technique of addressing string data allows high-speed manipulation of character information for text-processing applications.

I/O PORT ADDRESSING

If an I/O port is memory mapped, a programmer may use the full instruction set with any of the memory operand addressing modes to access the port. For example, a group of terminals can be accessed as an "array".

Additionally, direct, indirect, and string addressing modes can provide access to ports located in a separate I/O space using special I/O instructions (see Figure 3-14). Direct port addressing operations supply an 8-bit direct address for access to 256 byte ports or 128 word ports. A program can use the DX register to indirectly address up to 64K byte ports or 32K word ports.

ADDRESS MODE SUMMARY

To summarize, a comprehensive set of addressing modes enables the instruction set of the iAPX 286 to satisfy the data storage requirements of high-level languages and dynamic memory allocation. The three elements of an indirect address expression—a base register, an index register, and a constant 8-bit or 16-bit displacement—may be used in any combination to address many different data structures.

Flag Operations

Arithmetic instructions post specific characteristics of the result of an operation to the six arithmetic flags. Most of these flags can be tested with conditional jump instructions following the arithmetic instruction.

The condition codes within the flag register reflect certain characteristics of the results of arithmetic or logical operations. The carry flag (CF) is set by a carry-out-of or a borrow-into the high-order bit of a result. CF is typically used for unsigned integer comparisons and multiprecision arithmetic. The auxiliary carry flag (AF) provides carry and borrow information for decimal data operations.

The CPU sets the sign flag (SF) equal to the high-order bit (bit 7 or 15) of a result. If the result of an arithmetic or logical operation is zero, then the CPU sets the zero flag (ZF). If the low-order eight bits of an arithmetic or logical result contain an even number of 1-bits, then the CPU sets the parity flag (PF). The overflow flag (OF) reflects a signed operation which results in a positive number which is too large, or a negative number which is too small, to fit in the destination operand.

Basic Instruction Set

The iAPX 286 includes eight basic functional groups of instructions. These groups are data transfer, arithmetic, flag, logic, shift, rotate, string manipulation, and control transfer. The instruction set treats the basic addressing modes and the three types of data operands (immediate, register, and memory) symmetrically to provide the greatest number of programming options.

All instructions are byte-aligned to eliminate the space wasted by implementations that require instructions to be aligned on integral word boundaries. The eight instruction groups are explained below.

GENERAL-PURPOSE DATA TRANSFERS

The base architecture of the iAPX 286 includes Move, Exchange, Push, and Pop instructions to perform data transfer operations within the system. These data transfer instructions can operate on both byte and word operands.

The Move instruction is the most general data transfer instruction; it can perform transfers between the iAPX 286 registers as well as between the registers and memory. The Exchange instruction performs a swapping operation between two registers or between a register and a memory operand. Memory-to-memory data transfers are also supported by the base architecture with instructions that operate on string data.

Push and Pop instructions perform operations with the processor stack. Push instructions can store both register and memory operands on the stack for later retrieval by a Pop instruction. These instructions automatically increment and decrement the stack pointer to indicate the current top of stack. The instruction set includes special forms of Push and Pop for saving or restoring all general purpose registers with a single instruction.

ARITHMETIC INSTRUCTIONS

The arithmetic instructions provide a means of manipulating numerical data. The four basic arithmetic operations include addition, subtraction, multiplication, and division. They may operate on signed and unsigned binary integers as well as on unpacked decimal numbers. Addition and subtraction instructions can also operate on packed decimal numbers.

The iAPX 286 microsystem supports high-speed execution of arithmetic instructions to improve performance for number-crunching applications. Examples of this performance are the 16-bit Multiply and Divide instructions which require only 2.1 microseconds and 2.4 microseconds,

respectively, with an 80286 operating at 10 megaHertz. The basic memory to register add/subtract time is only .7 microseconds.

Arithmetic operations on decimal operands occur in two steps. The first step is the arithmetic operation and the second step is an adjustment to create a valid decimal result. The order of these instructions is reversed for divide instructions to adjust the dividend before performing the operation to produce a valid decimal result.

ASCII adjust instructions produce valid decimal results in unpacked form following addition, subtraction, multiplication, and division operations. Decimal adjust instructions correct the results of addition and subtraction operations on packed decimal numbers.

To provide multiple-precision addition and subtraction, the Add With Carry and Subtract With Borrow instructions allow a series of calculations to produce a single multiple-precision result. Conditional jump instructions can assist in handling multiple-precision operations and detecting overflow conditions.

Division instructions use a full-word or doubleword dividend. A shorter dividend must, therefore, expand to 16 or 32 bits in length while preserving its original sign. The Convert Byte To Word and Convert Word To Doubleword instructions simplify division by extending the sign of the dividend to the required 16-bit or 32-bit value.

A Compare instruction provides a way to test an operand without destroying the operand (as would occur with a subtract instruction). Compare performs a subtract operation which updates the arithmetic flags but does not return a result.

Increment, Decrement, and Negate instructions allow a program to alter either register or memory operands in a single instruction. The Increment and Decrement instructions are convenient for maintaining count values outside of registers. The Negate instruction provides a binary two's complement for any register or memory operand.

The instructions just described comprise the group of arithmetic instructions that are part of the base architecture. Multiple-precision data types and transcendental functions are available with a system that includes a Numeric Data Processor (see "Extended Capabilities" later in this section).

FLAG INSTRUCTIONS

To control the operation of the flags, the base architecture includes instructions to manipulate the carry flag, the interrupt-enable flag, and the direction flag.

The program can set, clear, or complement the carry flag to control arithmetic calculations. The carry flag instructions are also useful in conjunction with Rotate With Carry instructions.

The direction flag (DF) selects the auto-decrement mode or the auto-increment mode for string instructions. Setting this flag causes the CPU to process strings from high addresses to low addresses, or from "right to left", by decrementing a register. Clearing this flag designates "left to right" string processing by incrementing the register which points to locations within the string.

The interrupt-enable flag controls the execution of maskable interrupts. It enables (= 1) or disables (= 0) the CPU to respond to maskable hardware interrupts. The flag is automatically reset on response to an interrupt and restored to its previous value on return from an interrupt service routine.

LOGIC INSTRUCTIONS

The logic instructions include the Boolean operators "not", "and", "inclusive or", and "exclusive or". These operators may be applied to

both register and memory operands. The logic instructions set the flags to reflect the results of the specified Boolean operation.

In addition to the Boolean operators, the logic instructions include a Test instruction that sets the flags but does not alter either of its operands. This Test instruction performs a logical "and" operation and is the logical equivalent of the Compare instruction for arithmetic operands.

SHIFT INSTRUCTIONS

The bits in register or memory operands may be shifted arithmetically or logically. A shift instruction may specify the number of bit shifts using a constant immediate value or a dynamic value contained in the CL register. Using the CL register to specify the shift count enables the program to specify this value at execution time.

Arithmetic shifts provide a fast way to multiply and divide binary numbers by powers of two. Because the carry flag (CF) reflects the last bit shifted out of an operand, logical shifts are useful for isolating individual bits in bytes or words.

The Arithmetic Shift Left and Logical Shift Left instructions pad the shifted value on the right with zeros. The Arithmetic Shift Left instruction sets overflow if the result is too big to represent. In a similar fashion, the Logical Shift Right pads the shifted value on the left with zeros. The Arithmetic Shift Right, however, preserves the sign of an operand by copying the sign bit into the vacant positions on the left created by the shift to the right.

ROTATE INSTRUCTIONS

Bits in register or memory operands may be rotated right or left. Bits rotated out of an operand are not lost as in a shift, but are "circled" back into the other "end" of the operand.

Additionally, a rotate instruction may specify the carry flag as a high-order one-bit extension of the operand to allow multiword rotate or shift operations or bit testing. As with the shift instructions, a rotate instruction may specify the number of bit positions to rotate using an immediate value or a dynamic value contained in the CL register.

The basic register shift and rotate performance is 250 nanoseconds for single-bit operations and 625 nanoseconds plus 125 nanoseconds per bit for multibit operations at 8 megaHertz.

STRING OPERATIONS

The base architecture provides five basic string operations called string primitives. The string primitives process strings of bytes or words one element at a time. Strings may consist of up to 64K bytes or 32K words. A programmer typically combines the string primitives with a Repeat prefix and other instructions to construct customized string processing instructions.

The five string primitives are Move, Compare, Scan, Load, and Store. The Move primitive transfers information from one memory location to another. The Compare primitive instruction checks two blocks of memory for the first instance of nonidentical contents.

The Scan primitive searches a block of memory for the first occurrence of, or departure from, a specified word or byte value. The Load and Store string primitives move string elements to and from the AX or AL registers (AX for word strings and AL for byte strings).

The base architecture supplements the string primitives with three different Repeat prefixes to process blocks of string data stored in memory. Each Repeat prefix terminates on a different set of conditions. A Repeat prefix causes the hardware to repeat a string primitive operation until the processor detects a terminating condition specified by that particular Repeat prefix.

The iteration count for a repeated string operation is programmable through the CX register. The program places an iteration count in the CX

register and the CPU automatically decrements this count after each repeated string operation. The programmer selects the Repeat prefix that terminates on the appropriate zero or nonzero result.

The direction flag (DF) in the flag register controls the "left to right" or "right to left" direction of repeated string processing operations. If the DF flag is set, the CPU automatically decrements the operand pointers after each operation. Clearing this flag causes the CPU to automatically increment these pointers.

Since the prefix and string primitive are not fetched and executed for each operand processed, a system can process strings much faster than would be possible with a normal software loop. The CPU performs memory-to-memory block transfers at a bus bandwidth of 8 megabytes per second for an 8 megaHertz iAPX 286 system.

A special Jump If CX Is Zero instruction allows a program to bypass string operations if the iteration count specifies zero executions of the routine. The execution of repeated string operations may be interrupted and resumed without consequence.

CONTROL TRANSFER INSTRUCTIONS

The iAPX 286 provides both conditional and unconditional program transfer instructions to direct the flow of execution. Unlike unconditional program transfers, conditional program transfers depend on the results of operations which affect the flag register.

Conditional Transfer Instructions

The conditional transfer instructions are jumps that may or may not transfer control depending on the state of the CPU flags at the time the instruction executes. The target for all conditional jumps must be within -128 to $+127$ bytes of the first byte of the next instruction within the current procedure. Conditional transfer instructions may be divided into two testing categories: signed and unsigned.

Signed conditional transfer instructions test the flags for "greater-than", "less-than", or "equal" results. The unsigned instructions, on the other hand, test the flags for "above", "below", or "equal" results which depend only on the relative magnitude of the operands.

The signed conditional transfer instructions test the sign flag (SF) and the overflow flag (OF) to make transfer decisions. The unsigned conditional transfer instructions test the carry flag (CF) and the zero flag (ZF) for the conditions specified in the instruction.

Conditional transfer instructions are also provided to test individual status flags such as carry, zero, parity, and overflow.

Unconditional Transfer Instructions

The group of unconditional transfer instructions includes the Call, Return, and unconditional Jump instructions. The Call and Return instructions form a pair which enables a program to execute an out-of-line procedure and then return to the original routine. The unconditional Jump instruction, on the other hand, provides a one-way transfer of control to a new location.

Jump, Call, and Return instructions allow control-transfer operations of various scope and implementation. Short forms of these instructions use only a 16-bit direct address referring to locations within the current program module and provide byte efficiency in instruction encoding. Long forms refer to locations outside the current module and use a full 32-bit address pointer. Call and Jump operations allow for direct, register indirect (short form only), and indirect-through-memory control transfers for relocatable programs. Return operations are always indirect through the stack.

High-level Instructions

The basic instruction set described earlier in this section provides all of the functions normally required for data processing. To create a more powerful instruction set and allow greater code density, Intel also provides a group of high-level instructions. Each high-level instruction replaces a series of instructions from the basic instruction set to automatically handle a multi-operation programming function.

BLOCK I/O INSTRUCTIONS

The iAPX 286 provides two block I/O instructions, In String and Out String, to accomplish block input and output of information between memory and an I/O port. The Repeat prefix enables a block I/O operation to specify an iteration count in the CX register just as with the other string operations. After each byte or word transfer, the CPU automatically decrements the iteration count and tests for the terminating condition specified in the Repeat prefix.

The program uses the DX register to specify an I/O port address for a block I/O operation. This indirect method of selecting an I/O port allows a single device-handling procedure to service multiple devices. The direction flag controls the automatic adjustment of the source (SI) or destination (DI) memory address just as it does with other string instructions.

The block I/O operations provide a fast way to transfer blocks of information without resorting to the additional complications of Direct Memory Access (DMA). An iAPX 286 system operating at a clock rate of 10 megaHertz can transfer 10 megabytes per second through an I/O port without DMA.

ITERATION CONTROL INSTRUCTIONS

Loop instructions combine the functions of the Decrement and conditional Jump instructions to simplify the implementation of software loops. When the CPU executes a Loop instruction, it first decrements the iteration count contained in the CX register. Then it tests for the terminating condition specified in the Loop instruction.

TRANSLATE INSTRUCTION

The Translate instruction performs a conversion function which would otherwise require many instructions from the basic instruction set. The Translate instruction is useful for converting byte values from one coding system to another such as from ASCII to EBCDIC.

The CPU executes a Translate instruction by using the value contained in the AL register as an index into a user-defined memory-based translation table. The table may be up to 256 bytes long. The CPU completes the execution of a Translate instruction by replacing the original value in AL with the indicated value from the translation table. The base address of the translation table is specified by the BX register to allow multiple translation tables in the system.

ARRAY BOUNDS CHECK INSTRUCTION

The Array Bounds Check instruction provides information which can allow a program to test the limits of an array before it attempts to read or write elements within the array. This instruction is also useful in high-level languages such as Pascal to test and enforce the subranges of variables.

The Array Bounds Check instruction specifies a register which contains an index value to an element within an array. The CPU executes this instruction by comparing this index value with the memory-based limit values for that array.

If the index value refers to a location outside the array, the instruction causes an exception reserved for the Array Bounds Check instruction. The system may use this exception to invoke an interrupt procedure that either expands the size of the array or gracefully terminates the program.

ENTER AND LEAVE INSTRUCTIONS

The Enter and Leave instructions implement the stack frame used by block-structured high-level languages for nested procedures.

Each time a program calls a procedure which begins with the Enter instruction, the CPU sets up a new stack frame display and specifies the amount of dynamic stack storage required for the called procedure. The stack frame display allows the inner levels of a nested procedure to access variables defined on the stack at outer procedure levels. After execution of the procedure is complete, the Leave instruction reverses the action of the Enter instruction. Leave releases all of a procedure's dynamic stack space and restores the stack frame of the calling routine.

EXTENDED CAPABILITIES

The previous section explains the registers, data types, addressing modes, and instructions common to the base architecture of the Intel 16-bit

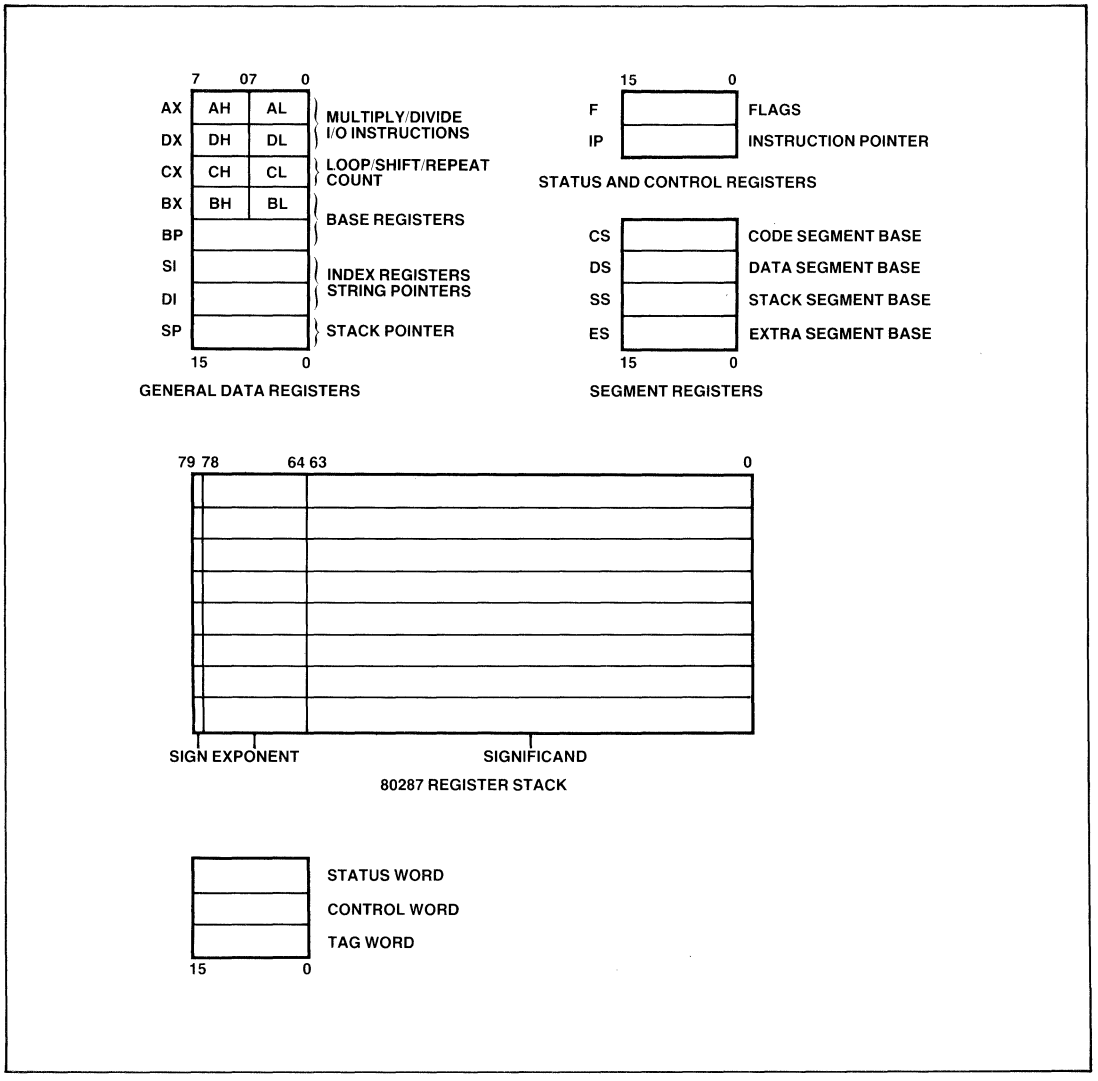

Figure 3-15. Extended Register Set

microsystems. To extend this base architecture, Intel offers the 80287 Numeric Processor Extension (NPX) which supports floating point registers, multiple-precision numeric data types, and new arithmetic instructions.

An iAPX 286 system may include different components depending on the intended application. The iAPX 286/10 configuration is based on the 80286 CPU and includes clock and interface support circuits. The iAPX 286/20 Numeric Data Processor (NDP) configuration adds an 80287 NPX to the iAPX 286/10 system. The iAPX 286/20 supports the IEEE Microprocessor Floating Point Standard P754.

The iAPX 286 microsystem also supports a software emulation of the Numeric Processor Extension complete with the registers, data types, and instructions of the iAPX 286/20 configuration. Programs written to use the NPX can execute on any iAPX 286 using the software emulator.

Extended Register Model

The extended register set includes 8 80-bit floating-point registers which provide the equivalent capacity of 40 16-bit registers (see Figure 3-15). These 8 floating-point registers are stack-oriented to simplify arithmetic programming. Two 16-bit registers control and report the results of numeric instructions. The control word register defines the rounding, infinity, precision, and error-mask controls required by the IEEE standard. The status word register reports any errors detected in numeric operations; it also contains a condition code to control conditional branching.

Extended Data Types

The extended data types described below and shown in Figure 3-16 are of different lengths and domains (real and integer). These convenient data types allow number representation most suitable for the application, with the required precision.

INTEGER DATA TYPES

The NDP supports 16-bit, 32-bit, and 64-bit integer operands for integer computations. These integers use the two's complement format for negative numbers, making the high-order bit a sign bit.

The 16-bit word integer format is common to both processors to ensure fast transfers of data between them. The 32-bit and 64-bit integer formats enable high-precision calculations with large integers. The 64-bit data type allows integers as large as 9×10^{18}.

FLOATING-POINT DATA TYPES

For extended-precision floating-point calculations, the NPX supports 32-bit, 64-bit, and 80-bit real values. These normalized floating-point data types include a sign bit, an exponent field, and a mantissa field.

The 32-bit short real values provide 24-bit precision and a range of values from 10^{-38} to 10^{+38}. The 64-bit long real values provide 53-bit precision and a range of values from 10^{-308} to 10^{+308}. This 64-bit real data type is sufficient to handle most high-precision calculations.

The temporary 80-bit real values provide extremely high precision for numeric calculations. Temporary 80-bit real values have a precision of 64 bits and a range of values from 10^{-4932} to 10^{+4932}. To prevent problems resulting from overflow or underflow during intermediate calculations the NPX uses an internal 80-bit numeric format. This format ensures a high degree of accuracy for the results of chained calculations.

PACKED BCD DATA TYPE

In addition to the extended integer and floating-point data types, the NDP also supports signed 18-digit packed BCD values. The packed BCD data type has a range of 0 to 10^{18}. These decimal

values satisfy the requirements of most commercial programs by providing high precision for decimal arithmetic.

Extended Addressing Modes

The Numeric Data Processor supports three techniques for operand addressing. The first implicitly uses the top element on the register stack and, optionally, the next element on the register stack for operands. Special-purpose numeric instructions can use this mode to save memory space because they do not require full addressing flexibility.

The second addressing technique enables an instruction to use any other register stack element along with the top element. Most two-operand instructions support this addressing mode to allow shorter and simpler numeric programs.

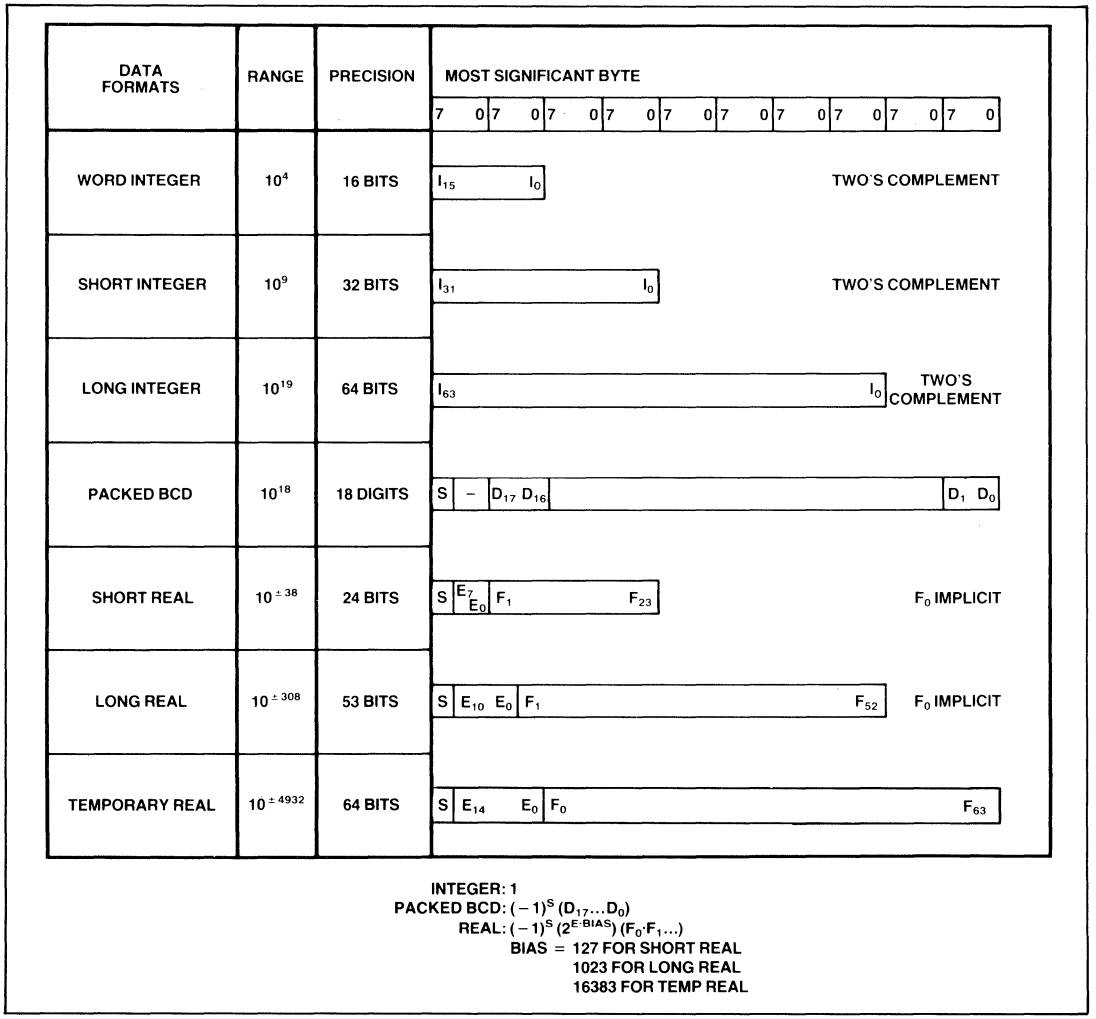

Figure 3-16. Extended Data Types

The third technique uses any of the existing iAPX 286 memory addressing modes to refer to memory operands. Instructions which move operands between memory and the internal stack use this technique.

Extended Numeric Instruction Set

The Numeric Processor Extension (or a software emulation of the NPX) provides an extension to the instruction set of the CPU's base architecture. A programmer places instructions for the NPX in line with the instructions for the CPU. The system executes these instructions in the same order as they appear in the instruction stream. The NPX operates concurrently with the CPU to provide maximum throughput for numeric calculations.

The Numeric Processor Extension extends the instruction set of the CPU to support high-precision integer and floating-point calculations. To process the extended data types, the extended instruction set includes arithmetic, comparison, transcendental, and data transfer instructions. The NPX also contains a set of useful constants to enhance the speed of numeric calculations.

ARITHMETIC INSTRUCTIONS

The extended instruction set includes not only the four arithmetic operations (Add, Subtract, Multiply, and Divide), but it also includes Subtract Reversed and Divide Reversed instructions. The arithmetic functions include Square Root, Modulus, Absolute Value, Integer Part, Change Sign, Scale Exponent, and Extract Exponent instructions.

COMPARISON INSTRUCTIONS

The comparison instructions are Compare, Examine, and Test. Special forms of the Compare operation can optimize algorithms by allowing comparisons of binary integers with real numbers in memory.

TRANSCENDENTAL INSTRUCTIONS

The instructions in this group perform the otherwise time-consuming core calculations for all common trigonometric, inverse trigonometric, hyperbolic, inverse hyperbolic, logarithmic, and exponential functions. The transcendental instructions include Tangent, Arctangent, $2^X - 1$, $Y.\log_2 X$, and $Y.\log_2 (X + 1)$.

DATA TRANSFER INSTRUCTIONS

The data transfer instructions move operands among the registers and between a register and memory. This group includes Load, Store, and Exchange instructions.

CONSTANT INSTRUCTIONS

Each of the constant instructions loads a commonly used constant into an NPX register. The values have a real precision of 64 bits and are accurate to approximately 19 decimal places. The constants loaded by these instructions include 0, 1, Pi, $\log_2 10$, $\log_2 e$, $\log_{10} 2$, and $\log_e 2$.

ADVANCED PROGRAMMING MODEL

The previous sections describe the registers, data types, addressing modes, and instructions that apply to a single program module with a single source for data. This section on the advanced programming model expands the view of the base architecture to include the segmentation of memory that enables the CPU to address a megabyte or more of physical memory.

Segmentation of Memory

As defined by the base architecture, the code, data, and stack information for a program occupy separate sections of the system memory (either separate or overlapping). These sections of memory are called segments. The base address of an addressable segment is stored in a segment register.

The iAPX 286 addressing mechanism operates on variable-length segments. Each segment may be as short as a single byte or as long as 64K bytes. A variable-length segment makes better use of memory and provides better granularity over the traditional fixed-length page. Because a program is structured into units of varying size, a variable-length segment can be arranged to reflect the structure of the program rather than the structure of the machine.

Pointer Data Types

A pointer data type is a value that represents a complete memory address. Part of the pointer specifies a segment and the remaining part specifies an offset into that segment (see Figure 3-17). Through the manipulation of pointers, a system can control access to common data areas or code sections.

For greater code density, the base architecture also provides short pointers consisting of only an offset into the current segment. The use of segment registers and short pointers allows most instructions to use 16-bit short pointers rather than 32-bit full pointers. This technique produces compact programs for faster execution.

An intersegment Call or Jump requires a full pointer to completely specify the new point at which the program resumes execution. A Call or Jump within the same code segment requires only a short pointer to transfer control. Call and unconditional Jump instructions may use either a full pointer or a short pointer. Conditional Jumps and instructions which read or write memory use only short pointers.

Segment Registers

The four segment registers of the iAPX 286 define the portion of address space available to a program at a given instant (see Figure 3-18). Each segment register contains the base address of a memory segment. A program switches to a different section of the address space by loading a segment register with the base address of the desired section. The iAPX 286 CPU forms an address within a segment by adding an effective address value to the base address in the segment register.

The code segment (CS) and stack segment (SS) registers provide access to segments containing instructions and stack data, respectively. The data segment (DS) register refers to a segment that stores data. The extra segment (ES) register refers to an alternate data segment. The extra segment register normally provides access to a storage area external to the segment indicated by the DS register, such as a system data segment (see Figure 3-19).

All addressing within the iAPX 286 uses a segment register as a base and the effective address value specified in the instruction as an offset within the segment. The effective addresses discussed in the section on "Basic Addressing Modes" do not refer to absolute memory locations; they are actually offsets from a segment base address.

Just as instructions include an offset into the current data segment to specify the effective address, the instruction pointer (IP) provides an offset into the current code segment for the location of the next instruction. The stack pointer (SP) similarly contains an offset into the current stack segment for the location of the top of stack.

In addition to SP, the iAPX 286 includes the stack frame base (BP) for convenient addressing of parameters and dynamic work space allocated on the stack by a procedure call.

The operation of the segment registers follows the way structured programs are written. The CPU implicitly assumes that instructions which address program data refer to the segment indicated by the DS register. Likewise, the CPU

assumes that stack operations refer to the stack segment and that instructions reside in the code segment.

These implicit segment assignments allow more compact instruction code by eliminating the need to specify segment registers, segment base addresses, or full 32-bit pointers in every instruction. These assignments may be altered, however, to handle special situations.

For the case where a program must access data from the code segment, stack segment, or extra segment, the base architecture provides segment-override prefix codes. This flexible addressing capability enables a program to read immediate operands within the code segment and to address parameter information within the stack segment.

Because program modules are normally smaller than 64K bytes, they seldom need to refer to locations outside the segments currently selected by the four segment registers. However, when it becomes necessary to establish addressability within a new segment, the program needs only to load a new base address into the appropriate segment register. The programmer normally uses a Move instruction to change the contents of the DS, ES, and SS registers. The CPU uses a full 32-bit address pointer to alter the contents of the CS register when the program specifies a transfer of control to an address outside of the current code segment.

A program can use a Move instruction to specify addressability within a new segment. Other instructions can enable the program to specify a location within the new segment, as well as the segment itself. The Load Pointer into DS and Load Pointer into ES instructions include the base address of a segment and an offset value. The CPU executes a Load DS or Load ES instruction by placing the base address in the appropriate segment register and the offset value in one of the general data registers (typically a base or index register). These instructions enable a program to quickly access data in another segment using a single instruction.

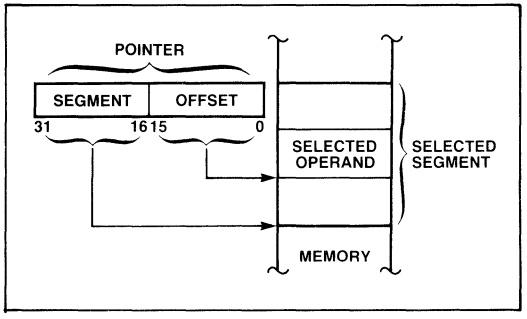

Figure 3-17. Address Pointer

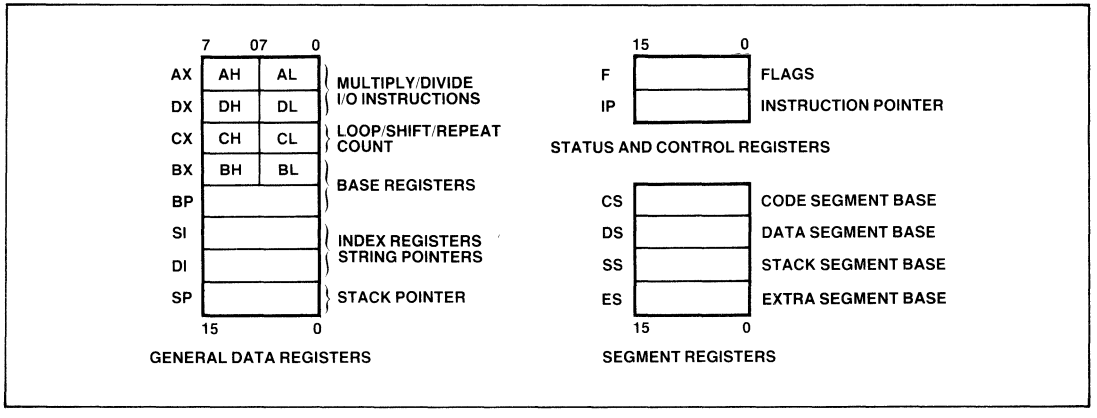

Figure 3-18. Advanced Register Set

Interrupts

The base architecture supports a versatile interrupt system. An interrupt vector table contains pointers to as many as 256 separate interrupt service routines. Program interrupts can occur in response to external devices and software interrupt instructions, as well as program exceptions such as an undefined opcode or a divide error.

Most external interrupts are "maskable". A program allows the execution of maskable interrupts only when the interrupt-enable flag (IF) is set.

Internally generated interrupts and exceptions are not maskable. The base architecture also provides a single externally triggered nonmaskable interrupt to protect the system from catastrophic failures such as a power loss.

Two Modes: Real Address and Protected Virtual Address

The previous sections discuss iAPX 286 operation in either Real Address Mode or Protected Virtual Address Mode (Protected Mode). In Real Address Mode, the iAPX 286 executes unmodified programs from the iAPX 86 and iAPX 88 while providing additional instructions and allowing up to six times the throughput. In Protected Mode, the iAPX 286 executes application

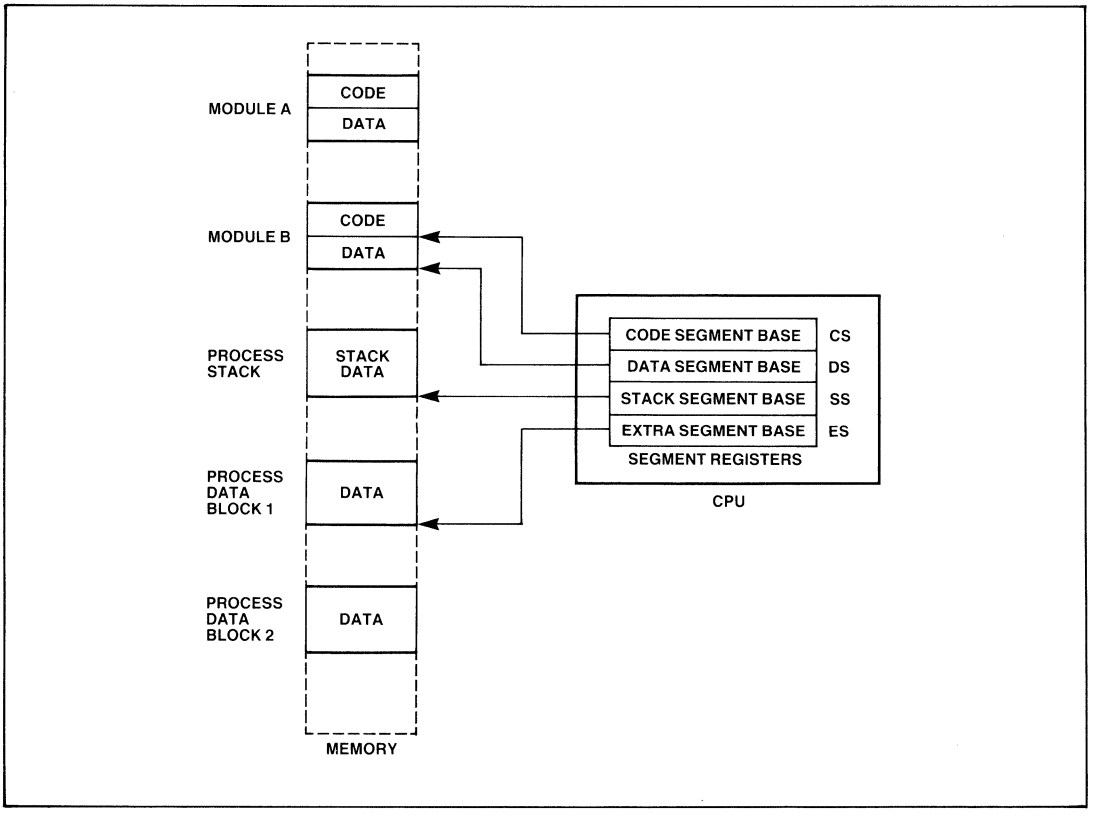

Figure 3-19. Segmented Memory Helps Structure Software

programs from the iAPX 86 and iAPX 88 within a protected virtual address space.

Protected Mode offers three advantages:

- Larger address space
- Protection of resources within the system
- Advanced operations

Following power-up or a system reset, the iAPX 286 CPU is in Real Address Mode, supporting a 1-megabyte real address space. All addresses in Real Address Mode are real addresses; they are formed by combining the real base address from a segment register and the offset value provided by the instruction (see Figure 3-20). The CPU shifts the 16-bit base address value in the segment register left four bits before adding the 16-bit effective address value to form a 20-bit real address.

Protected Virtual Address Mode

The iAPX 286 in Protected Virtual Address Mode (Protected Mode) integrates the capabilities of Real Address Mode with memory management, virtual memory support, and address space protection. This integration enables the iAPX 286 to support reliable multi-user systems which provide new levels of performance, yet cost far less to develop and manufacture than other systems of equivalent capabilities.

After power-up in Real Address Mode, the system can either continue to execute in that mode or switch to Protected Mode by setting a bit in a status register. Once set, only a system reset can clear the Protected Mode bit. This restriction protects the system from programs attempting to circumvent the protection mechanism by returning to Real Address Mode.

In Protected Mode, the iAPX 286 supports a 16-megabyte real address space and a virtual address space per task (per user) of up to 1 gigabyte (2^{30} bytes). Programs do not deal with physical addresses as in Real Address Mode. Instead, the segment registers now specify one of up to 16K 64K-byte segments of virtual address space. The effective address specifies the offset of the desired operand within the segment. The translation of virtual to physical memory address is automatically performed by the iAPX 286 on-chip memory management facility. This eliminates the need for special instructions or routines to manage an external memory management facility and provides significantly higher system throughput.

The segmentation of the virtual address space in Protected Mode of the iAPX 286 parallels the segmentation of the real address space in Real Address Mode. This design compatibility simplifies the migration of programs from Real Address Mode to Protected Mode. An application programmer perceives no difference between the memory models of the Real Address and Protected Modes except the amount of addressable memory and the information contained within a segment register.

To manage the 16-megabyte real address space available in Protected Mode, the iAPX 286 supports a series of memory-based address mapping

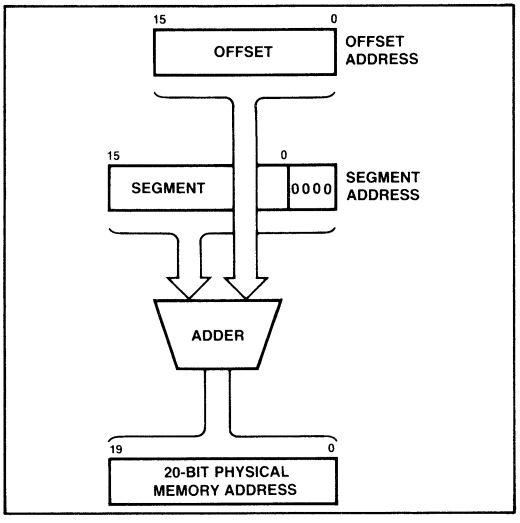

Figure 3-20. iAPX 86 Real Address Mode Address Calculation

(descriptor) tables. These tables define the virtual address space for every task in the system and greatly simplify relocation of programs and data.

The addresses used in Protected Mode are natural extensions of the real addresses of Real Address Mode, providing compatibility for application programs. The iAPX 286 supports a local address space for each task's private code and data, and a global address space for shared code and data available to all tasks in the system.

The iAPX 286 greatly reduces the complexity of an operating system (OS) for a virtual memory system by handling in hardware many of the operations required for virtual addressing. The 80286 CPU contains all of the necessary logic to perform virtual address translation using the memory-based descriptor tables defined by Protected Mode operation. The information stored in these tables includes the base address, length, access rights, and status of every segment in the system. Handling virtual address translation in hardware relieves the OS of this major duty and provides faster operation.

The hardware-based address space protection of the 80286 CPU makes large multi-user reprogrammable systems easier to develop. System and application software is easier to write and debug.

This protection ensures reliable systems by:

- Isolating the OS from applications programs
- Isolating users from users
- Protection checking which occurs concurrently with other CPU functions so that it does not affect system performance.

These Protected Mode features prevent a system failure due to the unauthorized modification of the OS by application programs. Because the protection mechanism detects errors before they can cause damage, the coding problems are often much easier to trace.

FULL REGISTER SET

To provide the hardware support for multitasking, virtual memory, and protection, the register set of the 80286 CPU includes new flag bits, new registers, and extensions in addition to previously described registers (see Figures 3-21a and 21b).

New flag bits control task transitions and the execution of certain instructions. New registers and extensions support virtual addressing, multitasking, and protection. The operation of these registers is optimized to simplify the task of writing both system and application programs.

SEGMENT REGISTERS IN PROTECTED VIRTUAL ADDRESS MODE

As described above under "Segmentation of Memory", the four segment registers (CS, DS, ES, and SS) point to the four currently addressable segments. A segment register performs the same function in Protected Mode except that it points to an entry in a table which, in turn, points to the target segment, rather than pointing directly to the segment. This added level of indirection enables the operating system to place a segment anywhere in real memory without having to adjust the address constants of a program.

In other words, Protected Mode alters the interpretation of the value in a segment register from a 16-bit real address to a table index. In addition, Protected Mode extends the segment registers to 64 bits to hold the addressing information copied from the table. This additional information includes the upper limit of the segment and the parameters governing the operation of the iAPX 286 protection mechanism.

Selector loading in Protected Mode parallels the loading of a segment base in Real Address Mode. By retaining the basic addressing procedures of Real Address Mode in Protected Mode, the iAPX 286 eliminates the need to rewrite application programs to use virtual memory.

DESCRIPTOR DATA TYPE

The 48-bit extension of a segment register contains a "segment descriptor". When a segment is in use, the descriptor which defines that segment occupies the 48-bit extension of the appropriate segment register. A descriptor is a data type which specifies the base address, limit, and protection parameters for a single segment of memory. Each virtual address translation table (or descriptor table) of the iAPX 286 consists of a list of descriptors for the segments defined for that program.

The iAPX 286 CPU uses the descriptor information in a segment register to translate a program's virtual addresses to real addresses. As long as a

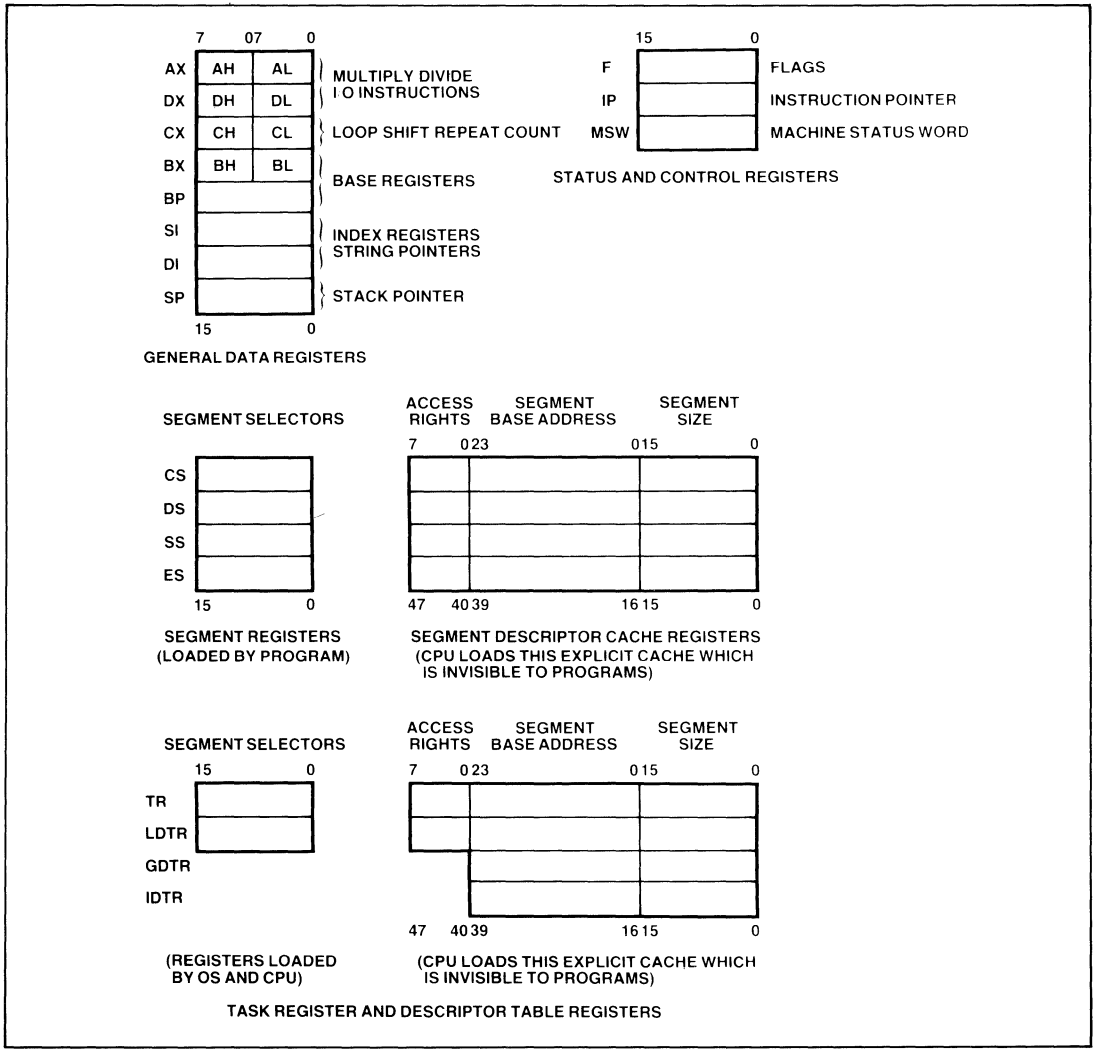

Figure 3-21a. Full Register Set

program remains within the boundaries of a single segment, the processor obtains all address translation information from the descriptor field in the segment register.

The iAPX 286 virtual addressing mechanism uses an *explicit cache* that speeds operations by eliminating the need to refer to a descriptor table for every memory reference instruction. This explicit cache consists of the segment register extensions which hold descriptor information.

Whenever a program loads a selector into a segment register, the CPU automatically copies the descriptor for that segment into the explicit cache for that segment register (see Figure 3-22).

After the hardware of the iAPX 286 copies the segment addressing information from a descriptor to a segment register, the CPU does not need to refer to a descriptor table again until the program requires access to another segment.

The iAPX 286 includes many different types of descriptors in addition to the segment descriptor. The system uses these other types of descriptors to define segments which store virtual addressing information and to regulate transfers of control within the system.

DESCRIPTOR TABLES

A descriptor table is an array of descriptors which the hardware interprets for virtual address translation. These descriptor tables form the interface between the OS software and the iAPX 286 virtual addressing hardware. This hardware supports one global descriptor table (GDT), one interrupt descriptor table (IDT), and multiple

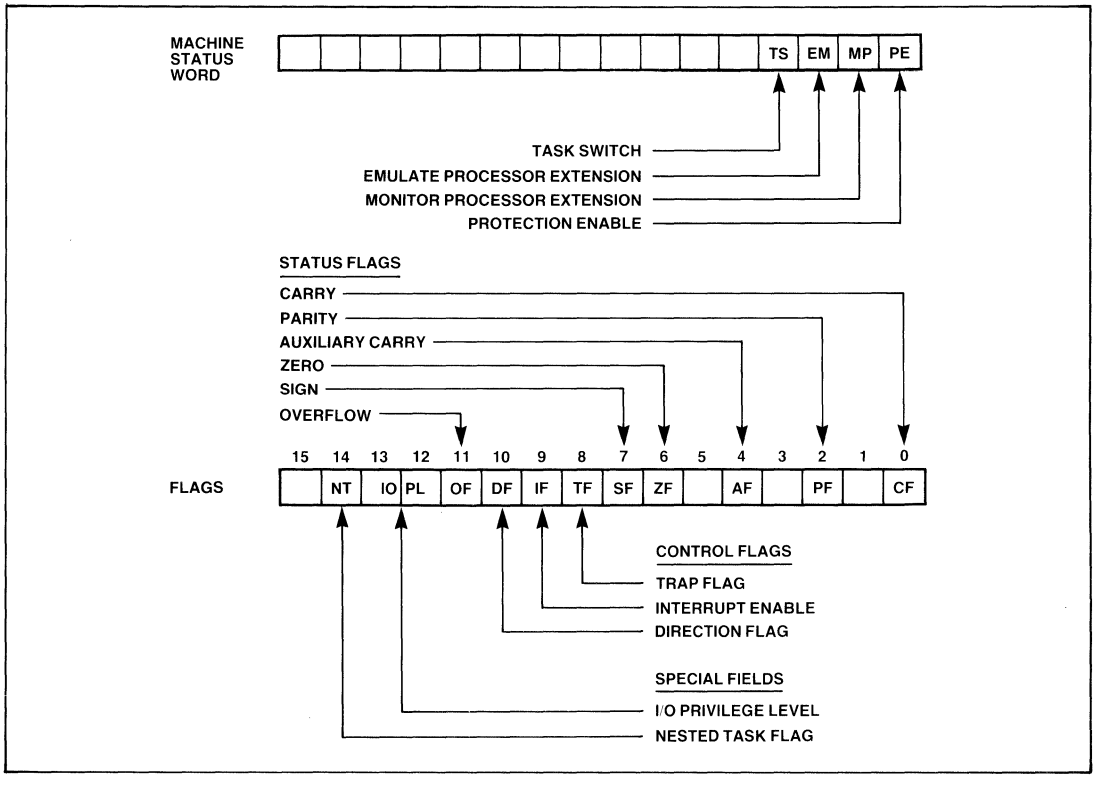

Figure 3-21b. Flags and Machine Status Word Bit Functions

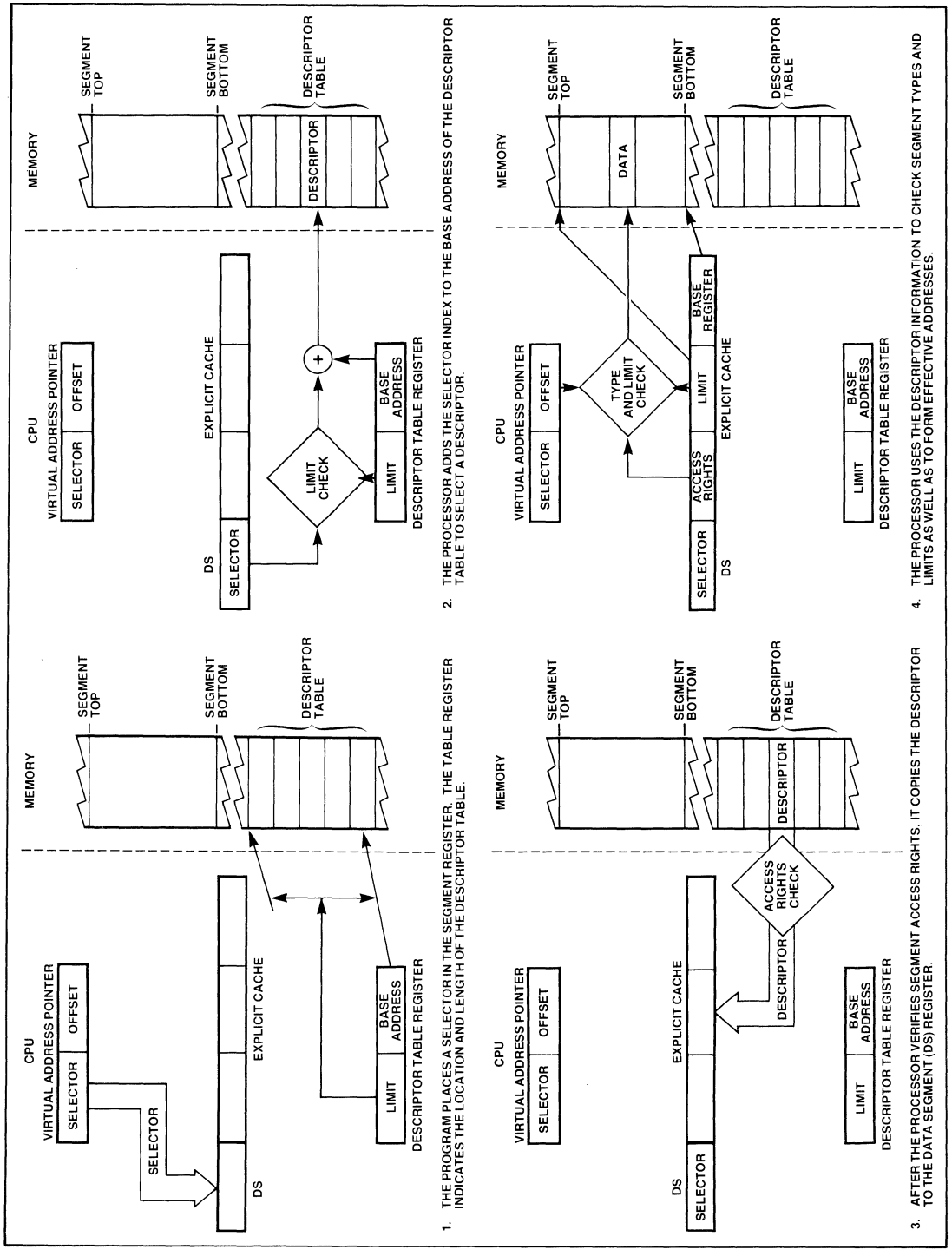

Figure 3-22. Loading the Explicit Cache

local descriptor tables (LDTs). The system programmer locates these tables (see Figure 3-23) anywhere in memory by using the three descriptor table registers: GDTR, IDTR, and LDTR (see Figure 3-24).

The operation of an application program may require a private local memory space to ensure reliable operation. An OS program, on the other hand, must be system-wide in scope to provide services to multiple application programs within a system. The iAPX 286 uses descriptor tables to separate the addressable memory space into a global portion for the OS programs and shared data, and a separate local portion for each task within the system. Switching from one user's local address space to another's only requires changing the LDTR register.

The descriptor tables consolidate all address translation information into easily manageable areas. Each time the OS brings a segment from secondary storage into real memory, it also updates the addressing information contained in the descriptor for that segment. If the address translation information were not collected into descriptor tables, the OS would have to locate and modify every memory operand which refers to a segment every time the OS relocates that segment.

The IDT contains an enhanced version of the interrupt vector table discussed under "Interrupts". The interrupt types of Protected Mode operation are identical to those of Real Address Mode to provide mode independence for application programs.

INSTRUCTIONS IN PROTECTED MODE

The instruction set of the Protected Mode iAPX 286 is identical (object code compatible) to the iAPX 86,88 and Real Address Mode iAPX 286 to enable application programs to migrate easily

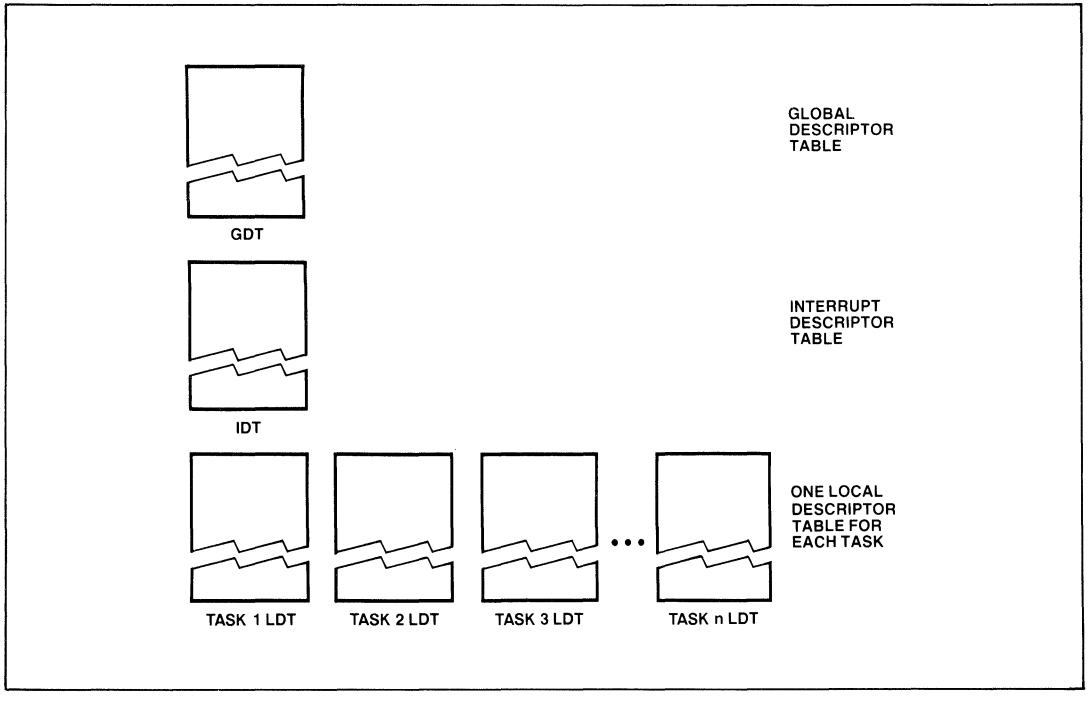

Figure 3-23. Descriptor Tables

from Real Address Mode to Protected Mode operation. Any instruction that modifies the contents of a segment register in Real Address Mode has the same net effect in Protected Mode but will load a different value (the operand loaded into the segment register). Because descriptor loading is handled automatically by hardware, it is completely transparent to the program. No additional instructions are required to initiate descriptor loading operations.

The only difference to the program between Real Address Mode and Protected Mode operation is the processor's interpretation of the 16-bit value

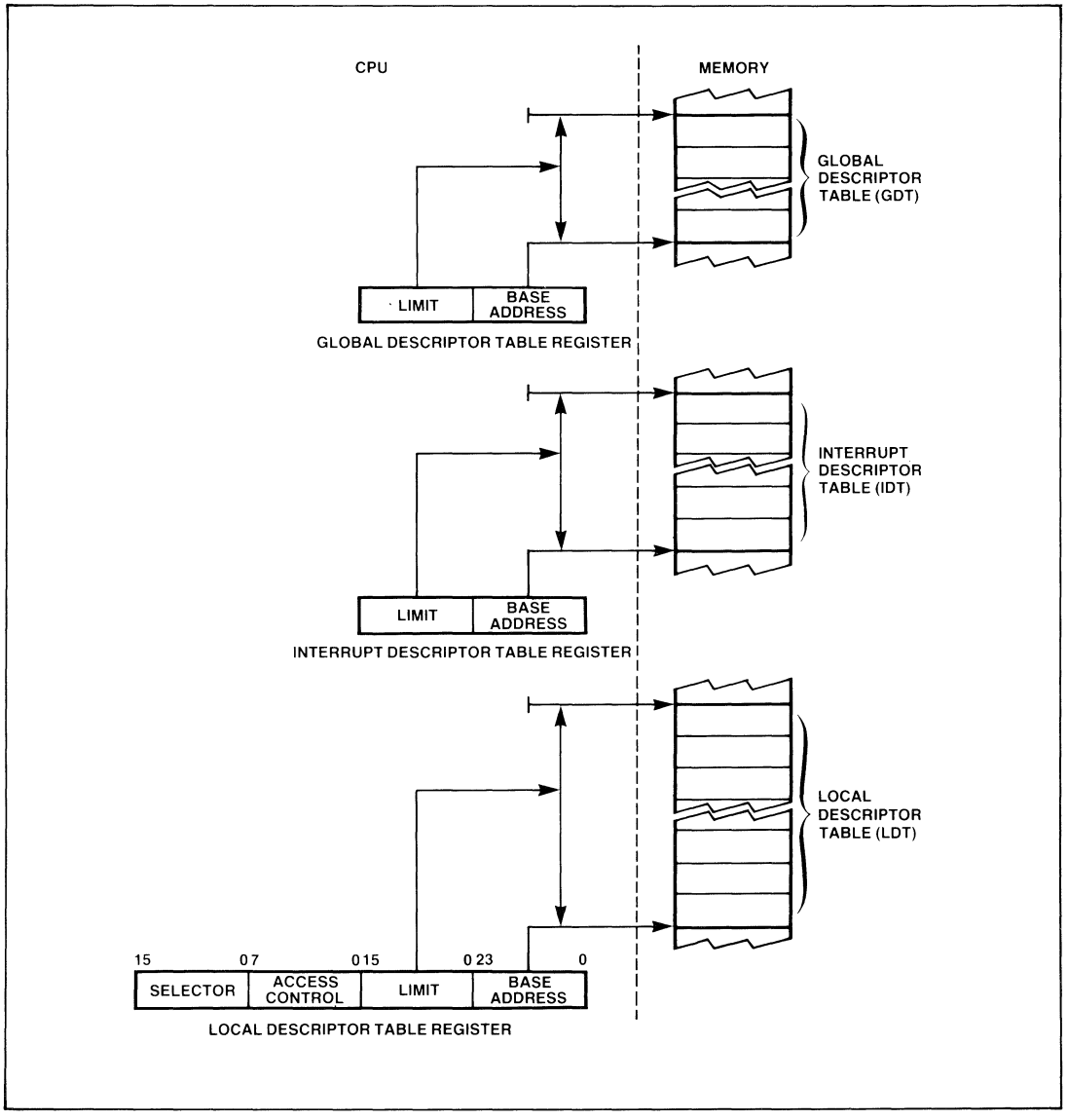

Figure 3-24. Descriptor Table Registers

that it places in the segment register. Real Address Mode installs the actual base address of a segment in the segment register, while Protected Mode specifies this base address indirectly through a segment selector.

Application programs written for the iAPX 86 and iAPX 88 may execute in Protected Mode by simply replacing segment base values with segment selectors. An assembler can automatically handle this conversion, because the syntax for segment addressing remains the same in both modes.

To summarize, Protected Mode operation is transparent to application programs and simplifies OS programs. The application programmer's view of the address space expands from a megabyte to a gigabyte in size. Otherwise, addressing remains consistent to allow the use of application programs from the iAPX 86 and 88.

The memory-based descriptor tables integrate the key system functions of memory management, virtual address translation, and memory protection to relieve the OS of these duties. This extensive OS support in hardware simplifies the job of the system programmer and shortens the product development cycle.

Protection Concepts

As computer system designs grow more complex and memories grow larger, protection of system resources becomes more important. To ensure reliable operation, a protection mechanism must prevent unauthorized modification to the code and data sections of every program in the system.

THE NEED FOR PROTECTION

A protection mechanism must guard a system's OS programs and data to prevent application programs from making illegal or improper modifications to the OS which could result in system failure. Each task within the system must also be isolated from undesirable actions of other tasks.

Transfer of control between system modules must be precisely controlled to achieve total reliability. A system must be able to honor requests for OS services without granting control over the OS to the calling program.

A protection mechanism must assign different priorities to different programs within the system to impart greater privilege to more important programs (such as an OS program). The OS programs are always assigned to a higher privilege level than application programs.

If it were possible for an application program to define the actions of the protection mechanism, that program could alter the protection parameters to override the authority of the OS. To prevent this problem, the hardware must enable only programs at the highest privilege level to control the protection mechanism.

A complete protection mechanism must also be able to detect addressing errors before they cause damage. To prevent improperly coded instructions from overwriting programs and executing data, each instruction must be checked to verify that it is performing the intended operation.

A protection mechanism reduces the cost of developing software by providing a quick means of diagnosing program "bugs". The protection mechanism can prevent an erroneous instruction from executing before it causes a problem. The operating system stays unaffected, and the program remains intact for examination. Continuous instruction checking can also reduce the cost of software maintenance, because fewer problems make it through the development process.

iAPX 286 PROTECTION MODEL

The hardware protection and memory management mechanisms of the iAPX 286 allow the OS to be part of every application program's virtual

address space. The use of a global address space enables an application program to access the OS with a simple high-speed Call instruction rather than the traditional and time-consuming context switch to a separate protected address space. To provide protection of the OS from an application program, the iAPX 286 provides four hierarchical privilege levels within each user's virtual address space. This approach allows complete protection of the OS from applications programs even though the OS is within the application program's address space.

The advantage for the programmer of making the OS a global resource is that the OS appears simply as a set of procedures which an applications program may call as a subroutine. This feature not only improves performance, but it also greatly simplifies the development of multi-user systems.

SEGMENT PROTECTION

Because each program and data structure occupies its own variable-length segment in an iAPX 286 system, these segments constitute ideal units for protection. The iAPX 286 provides four segment types for protection purposes. These four segment types are execute-only, execute and read, read-only, and read and write. The iAPX 286 protection mechanism checks both segment register loading and each instruction that uses the segment to verify that it does not violate the segment usage defined by the system designer for each segment.

Segment register loading verifies the segment is present and is the correct type; for example, executable for loading into CS, and read/write for loading in SS (see Figure 3-22). Usage checks include limit checks and access rights, such as read, write, and execute only (see Figure 3-22).

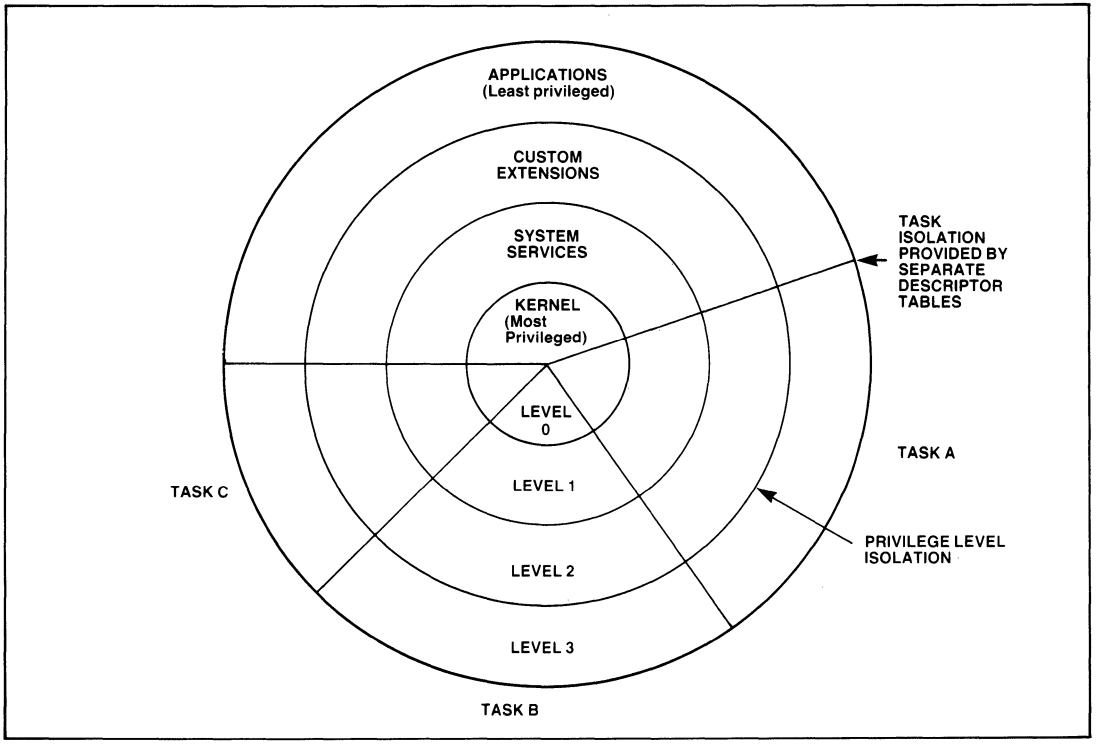

Figure 3-25. Four Privilege Levels

In addition to segment typing, the iAPX 286 provides up to four levels of privilege for different segments in the system (see Figure 3-25). The privilege level of a segment is defined within the descriptor for that segment.

The hardware-enforced privilege levels of the iAPX 286 provide total control over the use of code and data segments. The protection mechanism enforces firewalls which prevent the effects of bugs from propagating to more privileged levels. A program may access data at only the same or a less-privileged level; it may call services at only the same or a more-privileged level.

For total isolation between levels, the iAPX 286 maintains a separate stack and stack pointer for each privilege level. The use of separate stacks prevents a program from corrupting a service procedure through unauthorized manipulation of the procedure's stack.

When an application program makes a call to the OS, it typically passes a series of parameters to the called procedure. The iAPX 286 automatically copies parameters from the caller's stack to the stack of the procedure to provide a transparent call interface between protection levels.

The availability of multiple privilege levels enables a system designer to divide an operating system into separate hierarchical sections. These separate sections are easier to write because they are smaller. They are also easier to debug due to the error-trapping capabilities of the protection mechanism. The following is an example of the way a system designer might configure these multiple protection levels.

Ideally, an OS kernel, which occupies the most-privileged level, is small and fast so that it may serve as a user-customized extension to the iAPX 286 CPU. A system designer specifies the use of the iAPX 286 protection mechanism within this OS kernel.

The second and third privilege levels commonly support system services and custom extensions, respectively, while application programs occupy the least-privileged level. A user can also use these less-privileged levels to support custom OS extensions without affecting the integrity of the original OS. This hierarchical structuring of privilege levels promotes structured system design and reduces the time required to develop and maintain system software.

CONTROL TRANSFER WITHIN A TASK

The traditional implementation of a virtual memory system uses one type of call instruction to access local procedures and a special supervisor call instruction to request OS services. The

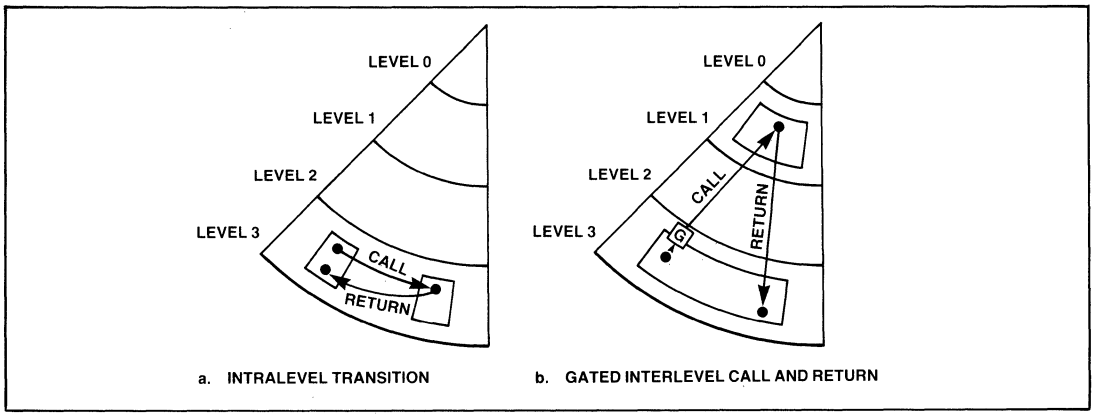

Figure 3-26. Control Transfers Within a Task

iAPX 286 simplifies programming by using the same instruction to call both regular procedures and OS services from the same virtual address space (see Figure 3-26).

Within the same iAPX 286 privilege level, a program may call to any procedure as long as that procedure resides within an executable segment (as opposed to a data segment). The same Call instruction which calls a procedure within the current privilege level also can call to a procedure at a more-privileged level using a call gate.

A call gate provides an additional level of indirection which enables the protection mechanism of the iAPX 286 to verify the privilege level of the calling program before granting access to a procedure. Each call gate controls the visibility of a single procedure, so that a program may not call the procedure if it does not satisfy the gate-defined access requirements. This restriction protects the integrity of an OS by preventing access to OS code by programs of insufficient privilege. For example, procedures at levels 2 and 3 may both access a procedure at level 0 to allocate more memory, however level 2 may also wish to call level 0 to disable interrupts without giving level 3 the same ability. The existence of gates within the privilege level of the caller controls (restricts) the caller's ability to access a more-privileged procedure.

External interrupts, software interrupts, and program exceptions can also cause privilege-level transitions. Because the iAPX 286 protection mechanism does not allow a transfer to a less-privileged level within a task, an interrupt routine serviced within the task normally resides at the most-privileged level to be accessible at all times. Interrupt service routines which execute as a separate task may execute at any level.

Advanced Architectural Features

The features of the iAPX 286 described in this section support specific system functions which would be difficult to implement and would limit the performance of other microprocessor systems. The benefits of these new iAPX 286 features can be added to upgraded products using existing iAPX 86 and iAPX 88 application programs.

INTERRUPT AND TRAP OPERATIONS

A real-time interactive computer system must handle frequent interrupts. The iAPX 286 provides a high-speed interrupt response mechanism to allow maximum throughput while still handling a large number of interrupts. The iAPX 286 can respond to an interrupt in less than 4 microseconds to provide prompt execution of

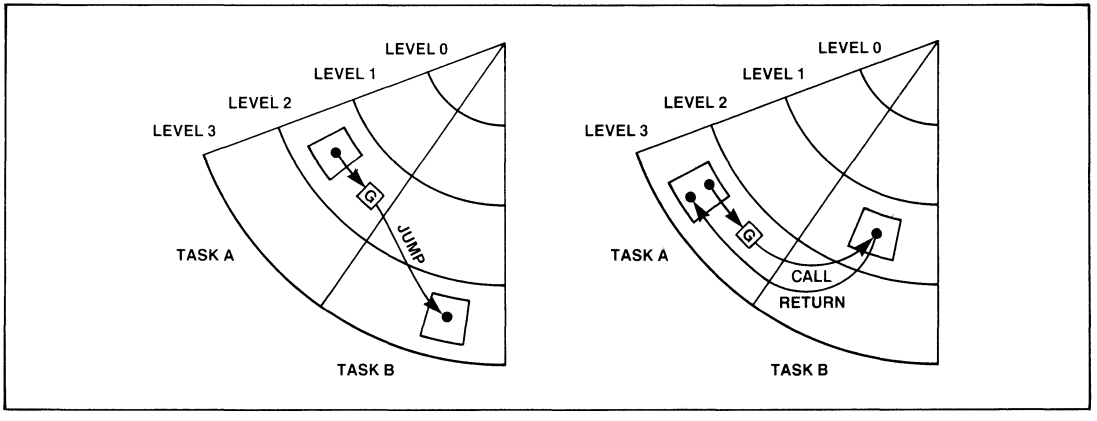

Figure 3-27. Control Transfers Between Tasks

an interrupt handler. This extremely low interrupt latency results in a system which can support many more real-time events.

The interrupt system of the Protected Mode includes two classes of transfers to interrupt service routines. The first class supports interrupt service routines which reside in the address space of the interrupted task. These routines are normally part of the OS in the global address space.

The second class of interrupt transfer automatically performs a high-speed task switch from the interrupted task to a special interrupt service task which is completely isolated from other tasks. The second class of interrupts provides a convenient way to add new features or custom I/O drivers to an existing system.

These interrupt operations are transparent to the software. Both classes of interrupt transfer use the same Return From Interrupt instruction to return control to the interrupted task.

The base architecture also includes a special interrupt type to simplify program development. The trap flag (TF) provides an interrupt following every instruction for single-step execution or program tracing. Single-stepping is a valuable debugging tool. A single-step procedure can act as a "window" into the system through which a programmer observes the instruction-by-instruction operation of a program.

MULTITASKING SUPPORT

Task dispatching is one of the most important functions of an OS in a multi-user or multitasking system. To allow concurrent execution of multiple programs, the OS must be able to support multitasking.

Because task switching occurs so frequently in real-time multitasking systems, the iAPX 286 CPU incorporates special high-speed hardware to perform this operation (see Figure 3-27). A 10-megaHertz iAPX 286 can save the state of one task (all registers), load the state of another task (all registers and the four segment descriptors), and resume execution all in less than 17 microseconds. For greater performance, the iAPX 286 also enables interrupts to cause a task switch directly, without OS intervention.

To support task switching in hardware, the Protected Mode of the iAPX 286 defines a special task state segment for each task in the system. A task state segment is a segment which retains the state of a task. The format of a task state segment includes the contents of the general registers, the selector for the task's local descriptor table, the stack pointers for that task, and a back link to a previously suspended task. The processor uses the back link to return from a nested chain of suspended tasks if such a chain exists.

Interrupts, exceptions, and traps can specify an inter-task transfer. To maintain low interrupt latency, interrupts can directly invoke a task switch without requiring a call to the OS. The same Call, Jump, and software Interrupt instructions which perform transfers within a task can also take advantage of this high-performance hardware to change tasks. This instruction compatibility allows a high degree of protection without introducing greater program complexity.

RESTARTABLE INSTRUCTIONS

All segment loading and stack reference instructions are restartable. This restartability enables an iAPX 286 system to handle program exceptions without requiring an elaborate program restart routine following the exception. The restartable instructions not only reduce programming complexity, but they also provide high-speed handling of program exceptions. Restartable stack faults allow an OS to provide growable stacks to accommodate varying stack requirements. Because a virtual memory system must frequently handle segment not-present exceptions, instruction restartability can significantly simplify the implementation of a virtual memory system.

VIRTUAL MEMORY

Any computer system which supports a virtual address space must effectively manage the swapping of programs (and data) between the system's real memory and secondary storage (usually a disk). If an instruction refers to a virtual address that is not present in real memory, the system must have a way to notify the OS to bring the requested entity into real memory.

The previous discussion of descriptor loading assumes that the target segment is present in real memory. When the OS runs out of real memory space, it creates free space by moving one or more segments from real memory to secondary storage. To allow the OS to easily identify the most likely candidate for swapping, the iAPX 286 marks a segment as "used" each time it accesses that segment.

After the OS determines which segment is used the least, it can move the segment into secondary storage and mark the segment descriptor "not present". A segment which has not been accessed at all may simply be overwritten because the version in secondary storage is still current.

If an instruction requests access to a segment that is not present in real memory, the not-present bit

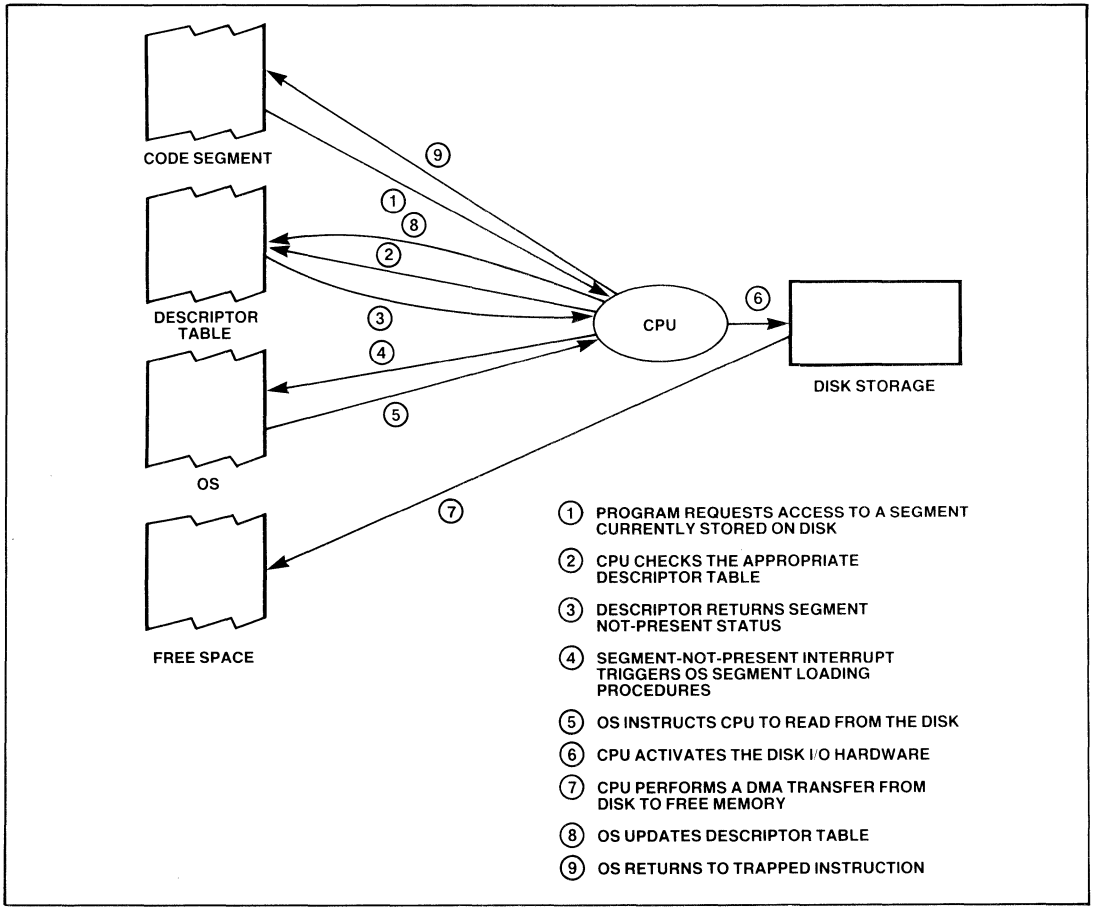

Figure 3-28. Virtual Memory Operation

in the descriptor for that segment triggers a segment not-present exception. The OS handles this type of interrupt by freeing the required amount of real address space and then loading the target segment into real memory from disk storage (see Figure 3-28).

After the segment is loaded into real memory, the OS marks the descriptor "present" and updates the address information for that segment in the corresponding descriptor. At this point, the OS may restart the instruction that originally requested access to the segment.

Because the iAPX 286 detects any exception before executing the instruction that caused the exception, an instruction restart is extremely simple. After the exception-handling routine completes its execution, the processor simply places the address of the interrupted instruction back in the instruction pointer and continues executing the program.

RECOVERABLE STACK FAULTS

A common problem in systems which support a wide variety of programs is stack overflow. This condition results from extensive use of recursive procedures or subroutines which require more dynamic storage on the stack than provided. The result of a stack overflow in most systems is either the destruction of information or a system crash. The protection mechanism of the iAPX 286 prevents these disasters from occurring.

As mentioned above, the iAPX 286 CPU actually halts any instruction which causes stack overflow before execution. Because the instruction does not execute, it is simple to restart after the exception handler corrects the problem.

The iAPX 286 can detect any instruction which attempts to write beyond the limit of the current stack segment before it causes a problem. The resulting program exception can then call a procedure which increases the size of the stack segment. Following the operation of the exception handler, the processor restarts the interrupted instruction. The original program continues executing with a larger stack. The iAPX 286 can also detect a potential stack underflow.

POINTER VERIFICATION

One of the more complex problems in protected systems is pointer verification to make sure that a user's program does not pass an erroneous pointer to the operating system. If undetected, such an error could cause the OS to modify data not belonging to the caller and violate the integrity of other programs in the system or even the OS itself.

To assist the programmer with pointer verification, the iAPX 286 provides special instructions which allow the software to restrict pointers from an unknown caller to the caller's address space and privilege level. Other instructions check pointer characteristics such as access rights and segment limits against the predefined values for each segment in the system.

MULTIPROCESSOR SUPPORT

Multiprocessing systems offer the advantage of even higher performance while maintaining modularity and reliability. The base architecture simplifies the development of systems using multiple independent processors by providing facilities for coordinating the interaction of processors.

The iAPX 286 provides support for multiprocessor coordination through an efficient bus interface, an 82289 Bus Arbiter circuit, and a special bus LOCK signal. Asserting the LOCK signal causes the Bus Arbiter to prevent other

CPUs in the system from using the common system bus. In this way, the LOCK signal provides integrity for data transfers between multiple processors using a common address space.

The programmer may invoke the LOCK signal by adding a Lock prefix to any memory access instruction. No other processor in the system may alter the contents of a shared memory location while the LOCK signal is in effect.

Certain operations automatically invoke the LOCK signal. These automatically LOCKed operations include the Exchange instruction and the descriptor loading procedures of the 80286. The Exchange instruction is LOCKed to allow the implementation of semaphores indicating the available or busy status of various system resources. The descriptor loading process is LOCKed to maintain total integrity for the definition of a processor's address space.

Interprocessor communication is required whenever one processor in a multiprocessor system alters the descriptor for a common resource in memory. An iAPX 286 system can force all processors in the system to reload a descriptor with a simple interrupt routine. This procedure allows each processor to access memory using only current descriptor information.

SYSTEM CONFIGURATIONS

The iAPX 286 may be used in a wide variety of configurations ranging from single processor to multiprocessor systems. The family of devices associated with the iAPX 286 microsystem include the 80286 CPU, support devices (82284 clock chip, 82288 bus controller, and 82289 bus arbiter), latches and transceivers (8282 and 8283 latches, and 8286 and 8287 transceivers), and processor extensions like the 80287 Numeric Processor Extension (NPX).

Figure 3-29 is an example of a single-processor iAPX 286/20 Numeric Data Processor system which includes an 80287 NPX for high-speed calculations with extended numeric data types. A simple high-performance data processor system, the iAPX 286/10, is obtained by simply removing the Numeric Processor Extension from the system. Because the iAPX 286 integrates memory management and protection within the CPU, this configuration provides full memory management and protection without additional circuitry.

Adding a 82289 Bus Arbiter to the previous configuration allows it to perform as an independent processing module within a multiple processor system (see Figure 3-30). This configuration provides a multimaster system bus interface as well as a local bus for private memory and I/O .

Other system configurations include interrupt systems, dynamic RAM systems, and dual-port memory configurations. The 8259A Interrupt Controller supports interrupt systems with up to 65 separate hardware interrupts for each CPU in an iAPX 286 system.

To simplify the implementation of dynamic memory systems, Intel manufactures the 8207 Dynamic RAM Controller and the 8206 Error Detection and Correction unit. The Advanced DRAM Controller supports a high-speed synchronous interface between the CPU and DRAM as well as dual-port operation for multiprocessor systems. The 8206 further enhances system reliability by providing single-bit error corrections and multi-bit error detection in DRAM systems.

The combination of CPU processor extensions, support devices, interrupt controllers, and memory system controllers provides the core elements of a simple yet flexible high-performance system architecture. Combining this architecture with Intel's full line of peripherals and memory components results in a comprehensive set of system solutions for almost any application.

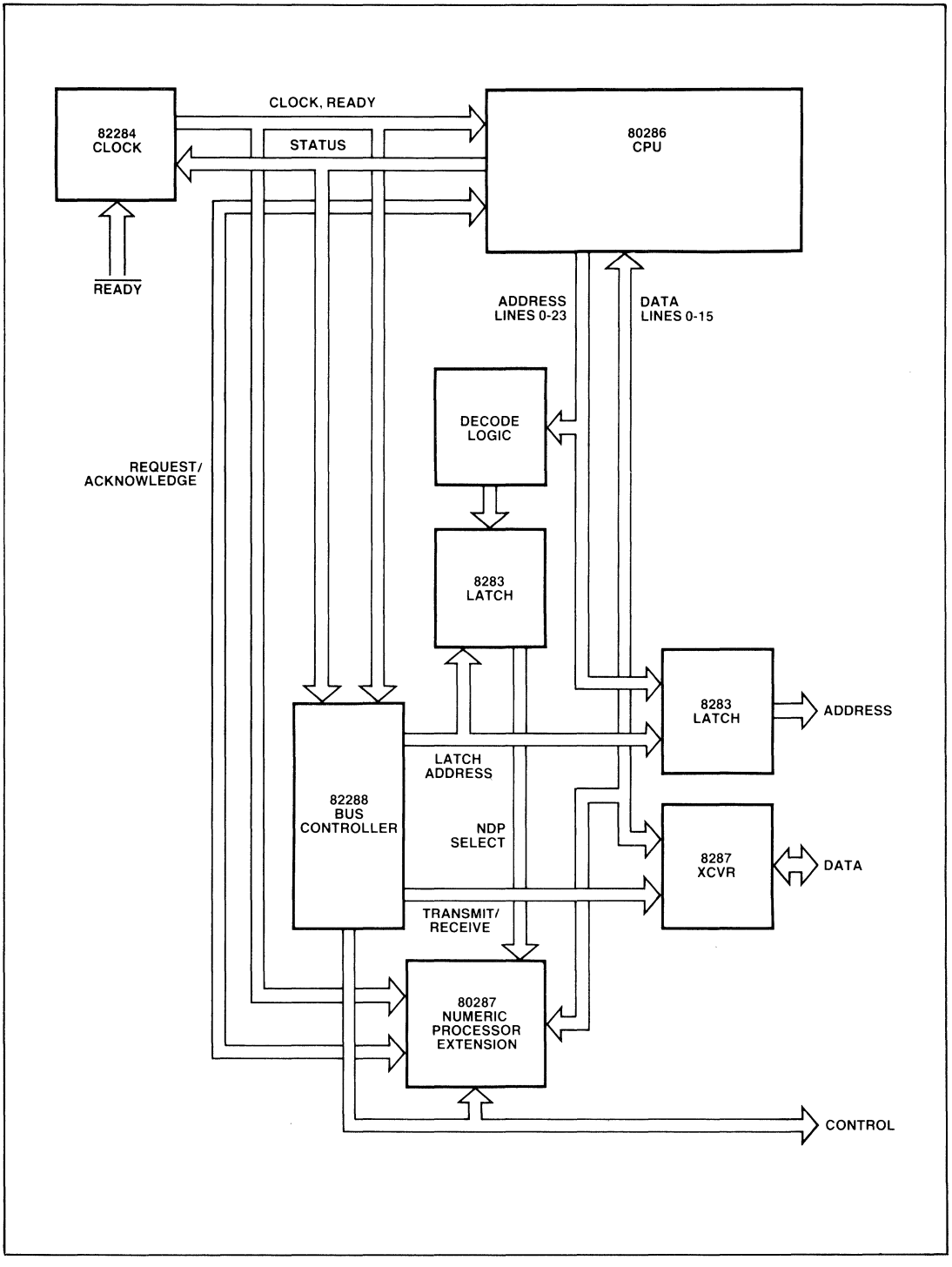

Figure 3-29. Numeric Data Processor Configuration

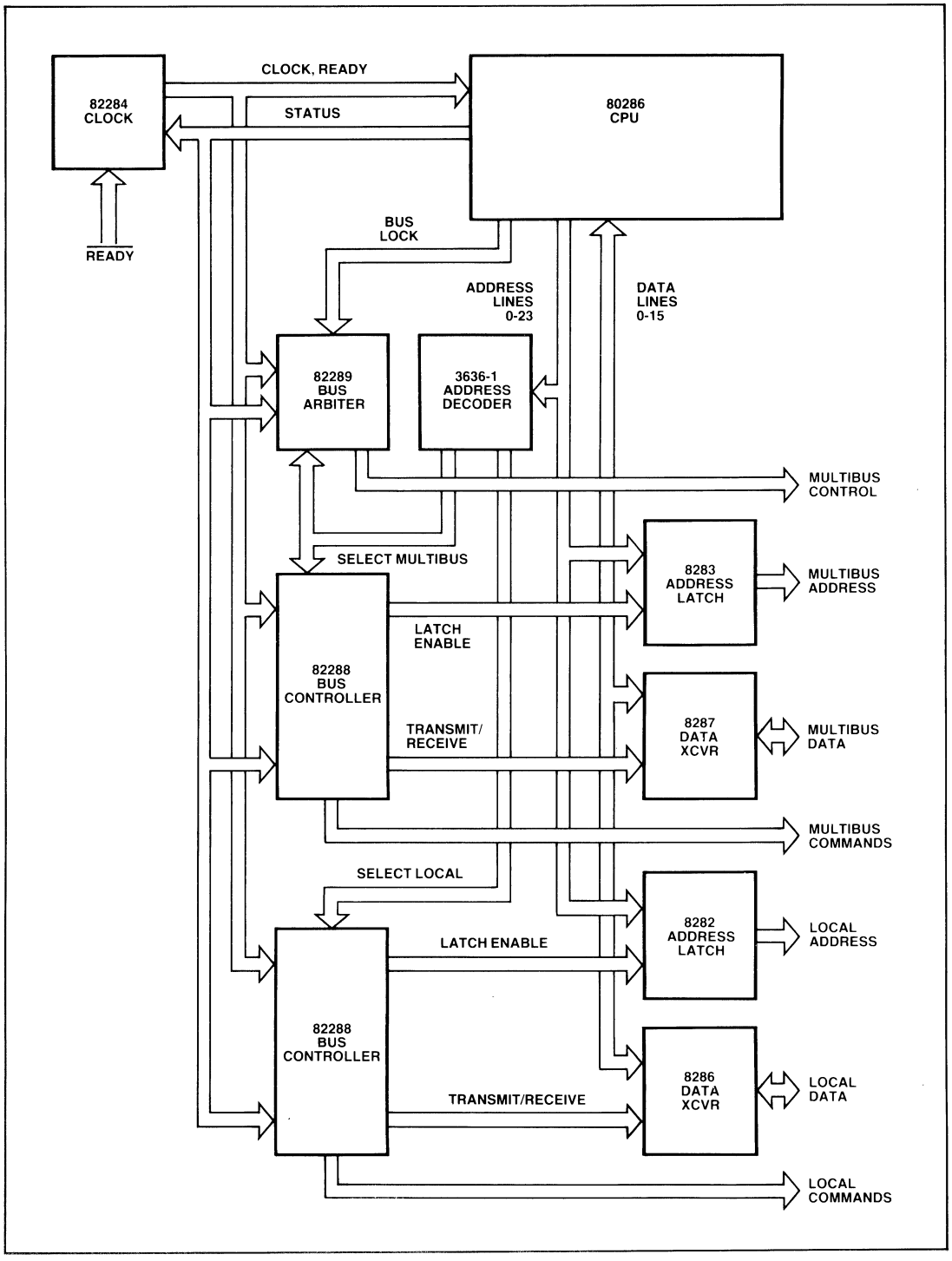

Figure 3-30. Multibus/Multiprocessor Configuration

APPENDIX A
iAPX 286 INSTRUCTIONS

This appendix describes the operation of each iAPX 286 instruction. The list of instructions is organized by type:

Data transfer
Arithmetic
Logic
String
Control transfer
Procedure implementation
Processor control
Protected Virtual Address Mode

For more detailed information about these instructions, refer to the iAPX 286 Programmer's Reference Manual.

DATA TRANSFER INSTRUCTIONS

General-purpose Data Transfers

MOV (Move)

MOV transfers a byte or a word from a source operand to a destination operand.

PUSH*

PUSH decrements the stack pointer (SP) by two and then transfers a word from the source operand (register, memory, immediate) to the top of stack indicated by SP. PUSH is often used to place parameters on the stack before calling a procedure; it is also the basic means of storing temporary variables on the stack.

POP*

POP transfers the word at the current top of stack (indicated by SP) to the destination operand (register or memory), and then increments SP by two to point to the new top of stack. POP moves information from the stack to either a register or memory.

XCHG (Exchange)

XCHG exchanges the byte or word source operand with the destination operand. The Exchange instruction automatically invokes a bus LOCK signal to support the use of semaphores in a multiprocessor system.

Accumulator-specific Transfers

IN

IN transfers a byte (or word) from an input port to the AL register (or AX register for a word). The port is specified either with an in-line data byte, allowing fixed access to ports 0 through 255, or with a port number in the DX register, allowing variable access to 64K input ports.

OUT

OUT is similar to IN except that the transfer is from AX or AL to the output port.

XLAT (Translate)

XLAT performs a table-lookup byte translation. The AL register is used as an index into a 256-byte table whose base is addressed by the BX register. The byte operand so selected is transferred to AL.

Address Transfers

LEA (Load Effective Address)

LEA transfers the offset of the source operand (rather than its value) to the destination operand. The source operand must be a memory operand, and the destination operand must be a 16-bit general register.

---

* Special forms of these instructions are also available for saving and restoring all general-purpose registers. (See the Procedure Implementation Instructions later in this appendix.)

LDS (Load Pointer Into DS)

LDS transfers a pointer (that is, a 32-bit value containing an offset and a segment selector) from the source operand, which must be a memory operand, to a pair of destination registers. The segment selector is transferred to the DS segment register. The offset must be transferred to a 16-bit register.

LES (Load Pointer Into ES)

LES performs the same function as LDS except that the segment selector is transferred to the ES segment register.

Flag Register Transfers

LAHF (Load AH With Flags)

LAHF transfers the flag registers SF, ZF, AF, PF, and CF into specific bits of the AH register.

SAHF (Store AH Into Flags)

SAHF transfers specific bits of the AH register to the flag registers SF, ZF, AF, PF, and CF.

PUSHF (Push Flags)

PUSHF decrements the SP register by two and transfers all of the flag registers into specific bits of the stack element addressed by SP.

POPF (Pop Flags)

POPF transfers specific bits of the stack element addressed by the SP register to the flag registers and then increments SP by two.

ARITHMETIC INSTRUCTIONS

Addition Instructions

ADD

ADD replaces the destination operand with the sum of a source and a destination operand.

ADC (Add With Carry)

ADC sums the operands, which may be bytes or words, adds one if CF is set, and replaces the destination operand with the result. ADC can be used to add numbers longer than 16 bits.

INC (Increment)

INC adds one to the destination operand. The operand may be either a byte or a word and is treated as an unsigned binary number.

AAA (ASCII Adjust For Addition)

AAA changes the contents of register AL to a valid unpacked decimal number, and zeroes the top four bits.

DAA (Decimal Adjust For Addition)

DAA corrects the result of adding two valid packed decimal operands. DAA changes this result to a pair of valid packed decimal digits.

Subtraction Instructions

SUB (Subtract)

SUB subtracts the source operand from the destination operand, and replaces the destination operand with the result. The operands may be signed or unsigned bytes or words.

SBB (Subtract With Borrow)

SBB subtracts the source operand from the destination operand, subtracts one if CF is set, and returns the result to the destination operand. The operands may be signed or unsigned bytes or words. SBB may be used to subtract numbers longer than 16 bits.

DEC (Decrement)

DEC subtracts one from the destination operand, which may contain a byte or a word.

NEG (Negate)

NEG subtracts the byte or the word in the destination operand from zero and returns the result to the destination. This instruction forms the two's complement of the number, effectively reversing the sign of an integer.

CMP (Compare)

CMP subtracts the source operand from the destination operand, which may be bytes or words, but does not return the result. The operands remain unchanged, but the flags are updated and can be tested by a subsequent conditional Jump instruction.

AAS (ASCII Adjust For Subtraction)

AAS changes the contents of register AL to a valid unpacked decimal number, and zeroes the top four bits.

DAS (Decimal Adjust For Subtraction)

DAS corrects the result of subtracting two valid packed decimal operands. DAS changes this result to a pair of valid packed decimal digits.

Multiplication Instructions

MUL (Multiply)

MUL performs an unsigned multiplication of the source operand and the AX or AL register. If the source operand is a byte, then it is multiplied by register AL, and the two-byte result is returned with the most-significant byte in AH and the least-significant byte in AL. If the source operand is a word, then it is multiplied by register AX, and the two-word result is returned in registers DX and AX with DX containing the most-significant word.

IMUL (Integer Multiply)

IMUL performs a signed multiplication of the source operand and the accumulator. IMUL uses the same registers as the MUL instruction.

AAM (ASCII Adjust For Multiplication)

AAM corrects the result of a multiplication of two valid unpacked decimal numbers.

Division Instructions

DIV (Divide)

DIV performs an unsigned division of the AX or AX and DX registers by the source operand. If the source operand is a byte, it is divided into the two-byte dividend assumed to be in registers AH and AL with the most-significant byte in AH. The single-byte quotient is returned in AL, and the single-byte remainder is returned in AH.

If the source operand is a word, it is divided into the two-word dividend in registers AX and DX. The single-word quotient is returned in AX, and the single-word remainder is returned in DX. Nonintegral quotients are truncated to integers.

IDIV (Integer Divide)

IDIV performs a signed division of the accumulator by the source operand. IDIV uses the same registers as the DIV instruction. For byte integer division, the maximum positive quotient is +127 and the minimum negative quotient is −128. For word integer division, the maximum positive quotient is +32,767 and the minimum negative quotient is −32,768.

AAD (ASCII Adjust For Division)

AAD modifies the numerator in AL before dividing two valid unpacked decimal operands so that the quotient produced by the division will be a valid unpacked decimal number. AH must be zero for the subsequent DIV to produce the correct result. The quotient is returned in AL, and the remainder is returned in AH; both high-order half-bytes are zeroed.

CBW (Convert Byte To Word)

CBW extends the sign of the byte in register AL throughout register AH. CBW does not affect

any flags. CBW can be used to produce a double-length (word) dividend from a byte before a byte division.

CWD (Convert Word To Doubleword)

CWD extends the sign of the word in register AX throughout register DX. CWD does not affect any flags. CWD can be used to produce a double-length (double word) dividend from a word before a word division.

LOGIC INSTRUCTIONS

All of the logic instructions, except NOT, affect the sign flag (SF), zero flag (ZF), and parity flag (PF).

NOT

NOT inverts the bits to form a one's complement of the byte or word operand. NOT has no effect on the flags.

AND

AND performs the logical "and" of the operands (byte or word) and returns the result to the destination operand.

OR

OR performs the logical "inclusive or" of the two operands (byte or word) and returns the result to the destination operand.

XOR (Exclusive Or)

XOR performs the logical "exclusive or" of the two operands and returns the result to the destination operand.

TEST

TEST performs the logical "and" of the two operands (byte or word), updates the flags, but does not return the result. Neither operand is changed.

SHL (Shift Logical Left)
SAL (Shift Arithmetic Left)

SHL and SAL perform the same function and are physically the same instruction. The destination byte or word is shifted left by the number of bits specified in the count operand. The processor shifts zeroes in from the right.

SHR (Shift Logical Right)

SHR shifts the bits in the destination operand (byte or word) to the right by the number of bits specified in the count operand. The processor shifts zeroes in from the left.

SAR (Shift Arithmetic Right)

SAR shifts the bits in the destination operand (byte or word) to the right by the number of bits specified in the count operand. Bits equal to the original high-order (sign) bit are shifted in from the left, preserving the sign of the original value.

ROL (Rotate Left)
ROR (Rotate Right)

ROL and ROR rotate the destination operand (byte or word) left or right by the number of bits specified in the count operand.

RCL (Rotate Through Carry Left)
RCR (Rotate Through Carry Right)

RCL and RCR rotate bits in the destination operand (word or byte) to the left or right by the number of bits specified in the count operand. These

instructions treat the carry flag (CF) as a high-order one-bit extension of the destination operand and taking part in the bit rotation.

STRING INSTRUCTIONS

MOVS (Move String)

MOVS transfers a byte or word operand from the source operand to the destination operand. As a repeated operation, this instruction moves a string from one location in memory to another.

CMPS (Compare String)

CMPS subtracts the destination byte or word operand from the source operand and affects the flags, but does not return the result. As a repeated operation, this instruction compares two strings. With the appropriate Repeat prefix, it is possible to determine the string element after which the two strings become unequal, thereby establishing an ordering between the strings.

SCAS (Scan String)

SCAS subtracts the destination byte or word operand from (AL or AX) and affects the flags, but does not return the result. As a repeated operation, this instruction scans for the occurrence of, or departure from, a given value in the string.

LODS (Load String)

LODS transfers a byte or word operand from the source operand to AL (or AX). This operation ordinarily is not repeated.

STOS (Store String)

STOS transfers a byte or word operand from AL (or AX) to the destination operand. As a repeated operation, this instruction fills a string with a given value.

REP (Repeat)
REPE (Repeat While Equal)
REPNE (Repeat While Not Equal)
REPZ (Repeat While Zero)
REPNZ (Repeat While Not Zero)

REP, REPE, REPNE, REPZ, and REPNZ repeat the action of the Move String, Store String, and Compare String operations to process a block of memory with a single instruction. The Repeat instruction uses the CX register to count repetitions. Repeat While Equal and Repeat While Zero prefixes terminate their actions when the zero flag (ZF) is cleared. Repeat While Not Equal and Repeat While Not Zero terminate when the zero flag is set.

CONTROL TRANSFER INSTRUCTIONS

Conditional Transfer Instructions

Table A-1 describes the conditional transfer mnemonics and their interpretations. The conditional jumps which are listed as pairs are actually the same instruction. The instruction set provides the alternate mnemonics for greater clarity within a program listing.

Unconditional Transfer Instructions

CALL

CALL activates an out-of-line procedure, saving information on the stack for later use by a Return instruction. The Return instruction in the called procedure uses this information to transfer execution back to the calling program.

Although long calls are provided for intersegment transfers, short intrasegment call instructions require fewer bytes, providing greater code

density for better use of memory. The program uses immediate data to specify a direct call and a register or memory to specify an indirect call.

RET (Return)

RET transfers control from a procedure back to the instruction after the call that activated the procedure. Long and short forms of the Return instruction are provided for intersegment and intrasegment transfer, respectively.

A value which may be coded within the Return instruction is added to the stack pointer after the return address is popped off the stack. This operation releases the stack space which was used for parameter passing.

JMP (Jump)

JMP unconditionally transfers control to the target location. Unlike a Call instruction, Jump is a one-way transfer of execution; it does not save any return information on the stack. As with the Call instruction, the Jump instruction can include the address of the target operand directly or obtain the address indirectly from a register or memory.

Iteration Control Instructions

LOOP

LOOP decrements the CX (''count'') register by one and transfers execution if CX is not zero.

LOOPZ (Loop While Zero)
LOOPE (Loop While Equal)

LOOPZ and LOOPE decrement the CX register by one and transfers execution if CX is not zero and the ZF flag is set.

Table A-1: Interpretation of Conditional Transfers

UNSIGNED CONDITIONAL TRANSFERS

Mnemonic	Condition Tested	"Jump If . . ."
JA/JNBE	(CF or ZF) = 0	above/not below nor equal
JAE/JNB	CF = 0	above or equal/not below
JB/JNAE	CF = 1	below/not above nor equal
JBE/JNA	(CF or ZF) = 1	below or equal/not above
JC	CF = 1	carry
JE/JZ	ZF = 1	equal/zero
JNC	CF = 0	not carry
JNE/JNZ	ZF = 0	not equal/not zero
JNP/JPO	PF = 0	not parity/parity odd
JP/JPE	PF = 1	parity/parity even

SIGNED CONDITIONAL TRANSFERS

Mnemonic	Condition Tested	"Jump If . . ."
JG/JNLE	((SF xor OF) or ZF) = 0	greater/not less nor equal
JGE/JNL	(SF xor OF) = 0	greater or equal/not less
JL/JNGE	(SF xor OF) = 1	less/not greater nor equal
JLE/JNG	((SF xor OF) or ZF) = 1	less or equal/not greater
JNO	OF = 0	not overflow
JNS	SF = 0	not sign
JO	OF = 1	overflow
JS	SF = 1	sign

LOOPNZ (Loop While Not Zero)
LOOPNE (Loop While Not Equal)

LOOPNZ and LOOPNE decrement the CX register by one and transfers if CX is not zero and the ZF flag is cleared.

JCXZ (Jump If CX Zero)

JCXZ transfers execution if the CX register is zero. This instruction is useful for jumping over a loop to execute it zero times.

Flag Operations

CF (Carry Flag)

If an addition results in a carry out of the high-order bit of the result, then CF is set; otherwise CF is cleared. If a subtraction results in a borrow into the high-order bit of the result, then CF is set; otherwise CF is cleared. Add With Carry (ADC) and Subtract With Borrow (SBB) instructions may use the carry flag to perform multibyte addition and subtraction.

AF (Auxiliary Carry Flag)

This flag detects a carry or borrow across the boundary between the two halves of the AL register. AF is provided for the decimal-adjust instructions and is not ordinarily used for any other purpose.

SF (Sign Flag)

Arithmetic and logic instructions set the sign flag equal to the high-order bit (bit 7 or 15) of the result. Standard two's complement notation is used to indicate a 1 for negative results.

ZF (Zero Flag)

If the result of an arithmetic or logical operation is zero, then ZF is set; otherwise ZF is cleared.

PF (Parity Flag)

If the low-order eight bits of an arithmetic or logical result contain an even number of 1-bits, then the parity flag is set; otherwise it is cleared.

OF (Overflow Flag)

If the result of an operation is a too-large positive number or a too-small negative number to fit in the destination operand (excluding the sign bit), then OF is set; otherwise OF is cleared.

Interrupt Instructions

INT (Interrupt)

INT pushes the flag registers (as in PUSHF), clears the TF, and transfers control by an indirect call through any one of the 256 interrupt vector elements. The instruction may or may not clear IF depending on the contents of the interrupt vector table. A one-byte form of this instruction uses an implied interrupt vector type 3 and is used for a breakpoint interrupt.

Table A-2: Predefined Interrupt Vectors

0 Divide error exception
1 Single-step interrupt
2 Nonmaskable interrupt
3 Breakpoint interrupt
4 INTO detected overflow exception
5 Bound RANGE exceeded exception
6 Invalid opcode exception
7 Processor extension not-present trap
8 Double protection exception
9 Processor extension segment overrun exception
10 Task segment format exception
11 Segment not-present exception
12 Stack under/overflow exception
13 General protection exception
14-15 Reserved
16 Processor extension error interrupt
17-31 Reserved

INTO (Interrupt On Overflow)

INTO pushes the flag registers (as in PUSHF), clears the TF, and transfers control to an interrupt service routine for this instruction if OF is set. The instruction may or may not clear IF depending on the contents of the interrupt vector table.

IRET (Return From Interrupt)

IRET transfers control to the return address saved by a previous interrupt operation and restores the saved flag register (as in POPF).

PROCEDURE IMPLEMENTATION INSTRUCTIONS

ENTER (Enter Procedure)

ENTER creates the stack frame required by most block-structured high-level languages. ENTER specifies how many bytes of dynamic storage to allocate on the stack for the routine being entered. It also specifies the lexical nesting level of the routine and determines how many stack frame pointers the CPU will copy into the new stack frame from the preceding frame. This list of stack frame pointers is called the display. BP always contains the current stack frame pointer.

LEAVE (Leave Procedure)

LEAVE reverses the effects of the Enter instruction by releasing all the stack space used by a routine for its dynamic storage and stack frame pointers (display).

PUSHA (Push All)

PUSHA saves all registers on the stack using a single instruction prior to the execution of a procedure.

POPA (Pop All)

POPA restores the registers saved by the Push All instruction, except that SP remains unchanged as the CPU ignores the SP value saved by Push All.

PROCESSOR CONTROL INSTRUCTIONS

Various instructions and mechanisms control the 80286 processor and its interaction with its environment.

Flag Instructions

STC (Set Carry Flag)

STC sets the carry flag (CF) to 1.

CLC (Clear Carry Flag)

CLC zeroes the carry flag (CF).

CMC (Complement Carry Flag)

CMC reverses the current status of the carry flag (CF).

STD (Set Direction Flag)

STD sets the direction flag (DF), causing the string instructions to auto-decrement the operand pointer(s).

CLD (Clear Direction Flag)

CLD clears the direction flag (DF), causing the string instructions to auto-increment the operand pointer(s).

STI (Set Interrupt-enable Flag)

STI sets the interrupt-enable flag (IF), allowing the processing of maskable interrupts.

CLI (Clear Interrupt-enable Flag)

CLI clears the interrupt-enable flag (IF), preventing the processing of maskable interrupts.

HLT (Halt)

HLT causes the 80286 processor to halt processing. The HALT state is cleared by RESET or an enabled external interrupt.

WAIT (Wait)

WAIT causes the processor to enter a WAIT (idle) state until the CPU receives a signal to continue processing. The CPU includes a special input line called TEST which is reserved for this operation. The CPU will exit WAIT to service interrupts and resume WAIT on return from the interrupt service routine.

ESC (Escape)

ESC provides a mechanism by which other processors (such as the Numeric Processor Extension) may receive their instructions from the 80286 instruction stream and make use of the 80286 addressing modes.

LOCK (Bus Lock Prefix)

LOCK precedes any instruction to cause the processor to assert its bus LOCK signal for the duration of the operation caused by that instruction. This prefix code has use in multiprocessing applications.

PROTECTED VIRTUAL ADDRESS MODE INSTRUCTIONS

Protection Control Instructions

LGDT
(Load Global Descriptor Table Register)

LGDT loads the GDT register with base and limit information for the system's global descriptor table. The CPU obtains the required six bytes from memory beginning at the location indicated within the instruction.

LIDT
(Load Interrupt Descriptor Table Register)

LIDT loads the IDT register with six bytes of base and limit information for the system's interrupt descriptor table.

LLDT
(Load Local Descriptor Table Register)

LLDT loads the LDT register with a 16-bit selector value to choose one of the local descriptor tables listed within the GDT. The source may be either a 16-bit register or memory operand.

LMSW
(Load Machine Status Word Register)

LMSW loads the MSW register from either a 16-bit register or memory operand. This instruction allows a system to enter Protected Mode after power-up or a system reset. A power-down or a system reset is the only way to return to Real Address Mode.

LTR (Load Task Register)

LTR loads the task register with a 16-bit selector value to choose one of the task state segments listed within the GDT. The source may be either a 16-bit register or memory operand.

SGDT (Store Global Descriptor Table)

SGDT stores the contents of the GDT register in six consecutive bytes beginning at the location indicated within the instruction.

SIDT (Store Interrupt Descriptor Table)

SIDT stores the contents of the IDT register in six consecutive bytes beginning at the location indicated within the instruction.

SLDT
(Store Local Descriptor Table Register)

SLDT stores the selector field of the LDT in the indicated 16-bit register or memory operand.

SMSW
(Store Machine Status Word Register)

SMSW stores the Machine Status Word in the indicated 16-bit register or memory operand.

STR (Store Task Register)

STR stores the selector field of the task register in the indicated 16-bit register or in the memory operand.

ARPL (Adjust Requested Privilege Level)

ARPL sets the RPL field of the operand to the maximum of its original value and a value contained within the instruction. This action ensures that a selector passed to a procedure does not request more privilege than the caller was entitled to. The instruction sets ZF (zero flag) if it alters the selector passed to the procedure. The RPL field may be within either a 16-bit register or memory operand.

CTS (Clear Task Switched Flag)

CTS clears the TS flag which the hardware sets each time it performs a task switching operation. If a task switch occurs during the operation of the Numeric Data Processor, the TS flag can activate a routine which saves the context of the co-processor and clears TS to prepare for the next task switching operation.

Protection Parameter Verification Instructions

Bound (Detect Value Out Of Range)

Bound verifies that the value contained in the indicated register lies within specified limits. These limit values reside in a two-word block of memory beginning at a location specified within the instruction. If the instruction detects that the value is not within the specified range, the special interrupt for this instruction occurs. The Bound instruction can serve as an array bounds check to prevent accidental overwriting of data outside an array.

LAR (Load Access Rights)

LAR loads the high eight bits of a 16-bit register with the access rights field of a descriptor. The instruction indicates the 8-byte descriptor indirectly through its associated 16-bit selector. The selector value may be either a register or memory operand. The instruction sets ZF to indicate that the descriptor was visible and that the access rights are present within the specified register.

LSL (Load Segment Limit)

LSL operates in the same way as the LAR instruction, except that it places the 16-bit segment limit value from a descriptor into the specified register.

VERR (Verify Read Access)

VERR sets ZF if the segment indicated within the instruction is readable or clears ZF if it is read-protected. The instruction indicates the segment to be verified with a selector value contained in a 16-bit register or memory operand. This instruction allows a segment to be tested without causing a protection fault.

VERW (Verify Write Access)

VERW operates the same as the VERR instruction except that it verifies write-protect status.

APPENDIX B
GLOSSARY

Call gate. Each call gate represents the protected entry point of a procedure. A call gate is the only way to increase the current privilege level within a task. Call gates are represented by a gate descriptor within a descriptor table.

Code segment. The code segments in the system contain the instructions and immediate data for the programs in the system. A code segment is also known as an executable segment.

Code segment (CS) register. The CS register holds the selector and descriptor for the currently selected code segment. The selector chooses the descriptor which, in turn, provides the base address and protection information for the system.

Control descriptor. System segment descriptors and gate descriptors are control descriptors. System segment descriptors enable a program to define the protection parameters for a system, and gate descriptors provide pathways between protected areas of memory. The descriptors for code and data segments, on the other hand, are called segment descriptors.

Data segment. The data segments in the system contain the data (other than immediate data) for the programs in the system.

Data segment (DS) register. The DS register holds the selector and the descriptor for the currently selected data segment. The selector chooses the descriptor which, in turn, provides the base address and protection information for the system.

Descriptor. Descriptors define the attributes of all memory segments and control gates.

Descriptor table. The iAPX 286 supports three types of memory-based tables which contain the descriptors for the system. These tables are the interrupt descriptor table (IDT), the global descriptor table (GDT), and the local descriptor table (LDT).

Descriptor table descriptor. The location of all local descriptor tables within the system memory is defined by descriptor table descriptors within the global descriptor table (GDT).

Explicit cache. The descriptor portions of the four segment registers comprise the explicit cache. It is used for high-speed addressing and the verification of access rights.

Extra segment. An extra segment is simply a data segment which the processor may use as an alternate data source or destination by placing the segment's selector and descriptor in the extra segment register.

Extra segment (ES) register. The ES register holds the selector and descriptor for an alternate data segment. The selector chooses the descriptor which, in turn, provides the base address and protection information for the system.

Gate descriptor. The gate descriptors include the call, trap, task, and interrupt gates. Unlike a memory segment, a gate descriptor defines the protected entry point of a code segment.

Global descriptor table (GDT). The global descriptor table contains descriptors which are available to every task in the system.

Global descriptor table register. This register contains the base address and the limit of the global descriptor table. The contents of this register normally remain constant during system operation.

Interrupt descriptor table (IDT). The interrupt descriptor table contains descriptors which vector interrupts, traps, and protection exceptions to their respective handling routines.

Interrupt descriptor table register. This register contains the base address and limit of the interrupt descriptor table. The contents of this register normally remain constant during system operation.

I/O. The I in I/O represents the input of information from a device such as a keyboard or a card reader, while the O represents output to a device such as a printer or a video display.

Local descriptor table (LDT). A local descriptor table contains the descriptors for segments which are generally private to a single task. The local descriptor tables enforce the isolation of tasks from one another.

Machine status word (MSW). The Machine Status Word contains three flags which control system configuration and one flag which assists the operating system with Numeric Data Processor control in multitasking systems.

Microsystem. A microsystem is a group of integrated circuits designed to implement microprocessor-based computer systems. The microsystem includes a CPU and special support circuits. Each support circuit performs a specific function which enables the CPU to communicate with the outside world or to operate more efficiently.

Memory-mapped I/O. Memory-mapped I/O differs from port-oriented I/O in that it is performed through a normal memory access instruction rather than a special I/O instruction which addresses one of the 64K ports of the iAPX 286.

Multibus™. The Multibus is Intel's general-purpose multiprocessing bus for 8- and 16-bit computer systems.

Numeric Data Processor. The iAPX 286 CPU, with the 80287 processor extension, comprises the iAPX 286/20 Numeric Data Processor configuration. The iAPX 286/20 handles long data types and complex arithmetic operations in parallel with normal processing operations.

Physical memory. The physical memory of a system is made up of random-access (read/write) memory (RAM) and read-only memory (ROM) which are part of a system's hardware. The physical memory is manipulated by an operating system to support virtual memory.

Pipelining. Instruction pipelining enables the processor to fetch an instruction while still processing the previous instruction. This technique maximizes bus efficiency.

Pointer. A full pointer is a complete virtual address. It includes a selector which indirectly chooses a segment base address and an offset value which points to a specific address within the segment. The offset by itself is called a short pointer.

Port. The iAPX 286 may address a maximum of 64K I/O ports using protected I/O instructions. These I/O ports are not part of the virtual address space of the iAPX 286.

Privilege level. The protection mechanism of the iAPX 286 provides four hierarchical protection levels to ensure program reliability. The most trusted service procedures occupy the higher levels (levels 0, 1, and 2), while the less-trusted application programs are placed at the lowest level of privilege (level 3).

Privileged instruction. The operating system kernel, which operates at the highest level of privilege (level 0), may execute privileged instructions. These instructions control system setup and operation.

Real memory. See physical memory.

Segment. The virtual address space of the iAPX 286 is made up of variable-length regions called segments. A segment has a minimum length of 1 byte and a maximum length of 64K bytes. The system includes code, data, task state, and descriptor table segments.

Segment register. The four segment registers point to the four segments of memory which are currently available for immediate access by the processor. The segment registers are the code, data, extra, and stack segment registers.

Selector. Each selector specifies a descriptor within either the global descriptor table or the

local descriptor table. The program uses a selector as part of a full pointer to gain access to a new segment.

Stack segment. The processor uses a stack segment to hold return addresses, dynamic data, temporary data, and parameters. An instruction addresses a stack segment usually with the stack pointer or the base pointer. Each privilege level has its own stack segment for total isolation. Stack segments usually expand downward.

Stack segment (SS) register. The SS register holds the selector and the descriptor for the currently selected stack segment. The selector chooses the descriptor which, in turn, provides the base address and protection information for the system.

System segment descriptor. The CPU uses the system segment descriptors to define task state segments and local descriptor tables within the global descriptor table.

Task. A task is a single sequential thread of execution which has an associated processor state and a well-defined address space with specific access rights. The processor state is defined by the contents of the task-variable registers while the address space and access rights are defined by the descriptor tables within the system.

Task register. The task register points to the task state segment of the currently active task. The task switch hardware of the iAPX 286 saves the state of the processor in this task state segment before it loads the state of the incoming task.

Task state segment. The task state segment contains the contents of the task-variable program registers, the stack segment selectors and pointers for the three highest privilege levels, the selector for the task's local descriptor table, and a back link which may point to another task in a chain of nested task invocations.

Trusted instruction. The trusted instructions which are controlled by the I/O privilege level flag include Block I/O, In, Out, Enable Interrupts, Disable Interrupts, Pop Flags (to change the interrupt-enable flag), and Halt instructions as well as the Lock prefix code.

Virtual address. A complete virtual address consists of a selector and an offset value. The selector chooses a descriptor for a code or data segment. The offset provides an index into the selected segment.

Virtual address space. The virtual address space of a task consists of all possible addresses which the task can access as defined by the system's global descriptor table and the task's own local descriptor table.

Virtual addressing. Virtual addressing is a system for providing uniform names (addresses) for areas of memory and control structures. The entities described by virtual addresses normally have constant attributes for size and protection and also variable attributes for presence and location. The operating system may make an entity present by swapping unneeded entities to secondary storage (usually a disk) and swapping the desired entity to primary storage (RAM). Virtual addressing presents the user with an address space much larger than the actual physical memory. Any I/O operation required to access programs or data in virtual memory is transparent to the user.

APPENDIX C
SAMPLE INSTRUCTION TIMINGS

The following list provides the execution timing for some of the instructions which enable the iAPX 286 to execute up to 6 times faster than its predecessors:

Operation	8 MHz	10 MHz
16-bit add	0.25 usec	0.20 usec
16-bit multiply	2.60	2.00
Complete task switch	21.00	16.80
Block I/O (12 bytes moved)	8.60	6.80
String Move (16 bytes moved)	8.60	6.80
Double-precision multiply*	18.00	14.40

*Using 80287 Numeric Processor Extension

Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

Intel Corporation S.A.
Parc Seny
Rue du Moulin à Papier 51
Boite 1
B-1160 Brussels
Belgium

Intel Japan K.K.
5-6 Tokodai Toyosato-machi
Tsukuba-gun, Ibaraki-ken 300-26
Japan

Printed in U.S.A./T-596/30K/1082/RRD JL