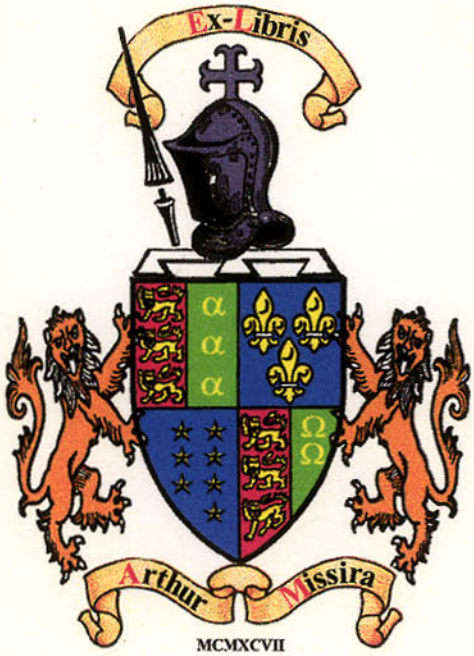
APPLICATIONS HANDBOOK



8 BIT MICROCOMPUTERS AND SUPPORT PRODUCT OVERVIEW

Table of Contents

Introduction 2

8-Bit Microprocessors 6

Peripherals 12

Memory Solutions 18

Intellec™ Development Systems 24

OEM Microcomputer Systems 30
Single Board Computers

The 8-bit universe is expanding.

The application of 8-bit microprocessor technology has barely begun. Despite the allure and intrigue of ever newer, more exotic 16- and 32-bit microcomputer or micromainframe products, the 8-bit solution will remain an industry workhorse over the next decade, with *significant* market growth and new applications.

Why?

A variety of good reasons. The technical potential of the 8-bit microprocessor has yet to be fully exploited. Meanwhile, 8-bit processors and associated peripherals continue to offer lowest design and component costs with performance well matched to thousands of product requirements.

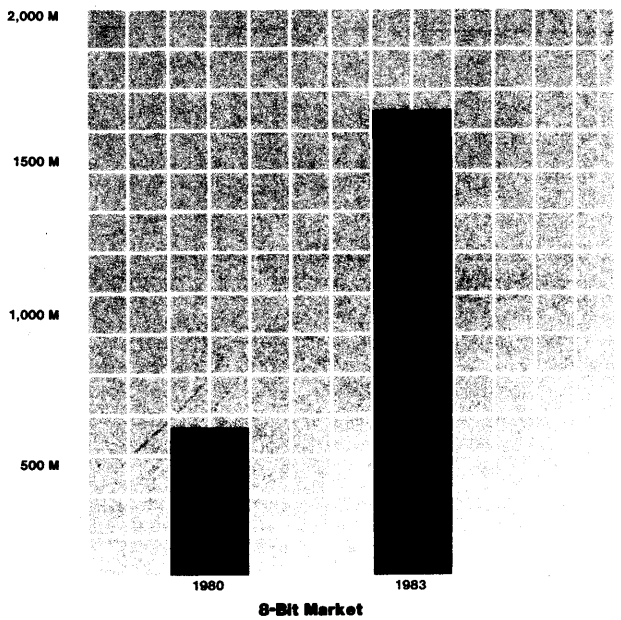
Then there is a huge existing base of 8-bit design technology already incorporated in functioning, established products—from terminals to toys, autos to automated assembly and process control. This base is supported by a development investment that grows best with orderly, evolutionary improvements in technology rather than abrupt shifts in technologies with their new, steep, expensive learning curves.

Intel's 8-bit total solution.

The Intel solution to 8-bit design success is based on total support, the widest array of microprocessor functionalities and capabilities, and orderly, evolutionary growth—both through development of new products, and through application of new technologies to existing products.

Total support means peripheral components, design aids, memory devices, evaluation tools, development equipment, software, languages, operating systems, single board computers, debug support, field engineer support and training. Everything you need to design, produce, and bring to market quickly successful products using 8-bit microprocessor technology. This brochure highlights Intel's vast array of 8-bit products providing an integrated, time saving solution to your design requirements.

The widest array of microprocessor functionalities and capabilities means you select the right performance and power for the job, all the while working with a unified design arsenal of development tools and development software.



The Market for Single- and Multi-chip 8-bit Processors, Peripherals and 8-bit Single Board Computers Will Almost Triple in Volume and New Applications by 1983. Intel Will Contribute to and Drive This Growth by Applying Technology Enhancements to Existing 8-Bit Products to Reduce Cost and Increase Availability, and by Introducing New 8-bit Products to Improve the Total Design and Development Solution.

Growth. New technologies in existing products. And in new products.

There is a growth path, designed to preserve your design investment and minimize your development overhead while providing state-of-the-art computer power. Part of it is the evolutionary application of new Intel MOS technologies to existing products—for example, the reduction of semiconductor die sizes. This reduction of die size means more dice per wafer, for more product availability and lower cost. Smaller circuits also mean lower power requirements and higher performance.

A second evolutionary application of new technologies is the use of new HMOS technologies in place of NMOS, as an upgrade for existing products. HMOS densities and reduced power requirements are constantly improving the price/performance ratio of Intel products; new CMOS versions will lead to further enhancements for existing and future Intel products.

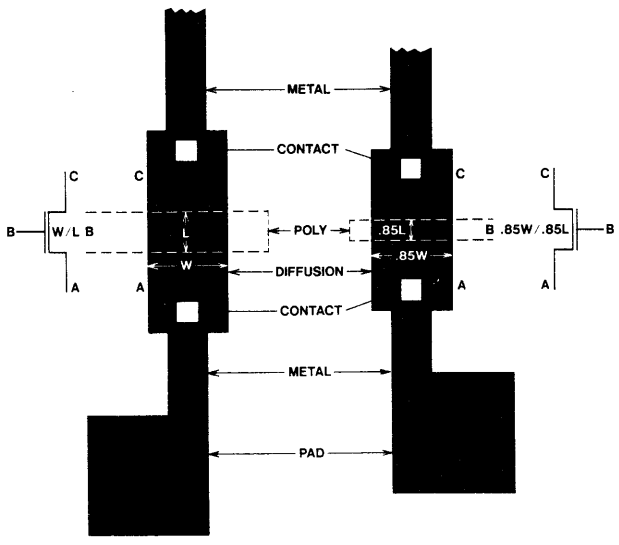
This die size reduction and transfer to new HMOS technologies are not trivial tasks. Many elements (for example, the bonding wire pads on the die) cannot shrink. Retaining the architecture of the product while reducing the die size is not, therefore, a 1:1 reduction process, but is a complex project which requires careful planning in the design of the original products. Intel products are planned to accommodate future HMOS advances and die size reductions—to maintain a long product life with consistently *rising* performance, *falling* power requirements, and *falling* prices.

For example:

The 8048 8-bit microcontroller you designed into your application has become the 8048H. And it is 80% faster, requires 40% less power, and has become 64% less expensive over the last three years.

The Intel 8085A you designed into your application has become the 8085AH, with up to 100% greater performance, 30% less power, and 50% less cost.

The 2114 RAM you designed into your application three years ago has become 40% faster and 80% less expensive. Similar improvements are true for your EPROMS and bipolar PROMS.



Built-in Cost Reductions and Performance Increases

The Layouts to Allow These Shrinkages are Designed Into Intel 8-bit Products Today. This Means Built-in Future Cost Reductions, Performance Increases, Reduced Power Requirements, and a Longer Product Life For The User, Because There are No Logic or Software Changes Required to Absorb These Improvements.

The 8-bit universe is expanding.

This product evolution through technology revolution goes beyond die size reductions and NMOS to HMOS conversions.

The iAPX 88 microprocessor software you've written is upgradeable to the 16-bit iAPX 86, and later to the iAPX 186 and iAPX 286.

The Intellec development system you bought as early as 1975 can be upgraded, through currently available options, to 1980 performance standards, the Series III.

The iSBC 80/24 and 80/10B single board computers represent similar 8-bit upgrades with enhancements to performance and memory capacity as well as improvements to your design flexibility with iSBX MULTIMODULES. The iRMX 80 operating system that you wrote your application around will be upgradeable to the new iRMX 88 operating system for iAPX 88,86 systems.

For many Intel customers, this technology evolution has lowered costs and increased performance *in their applications* without expensive design modification or product specification changes. It has enabled customers to take advantage of the newest, most powerful microcomputer advances using their existing investment in development tools, training, and knowledge.

That's more than our history. It's our strategy. It's how we're managing technology in the 80's. And there's a lot more than evolution of existing product lines, too.

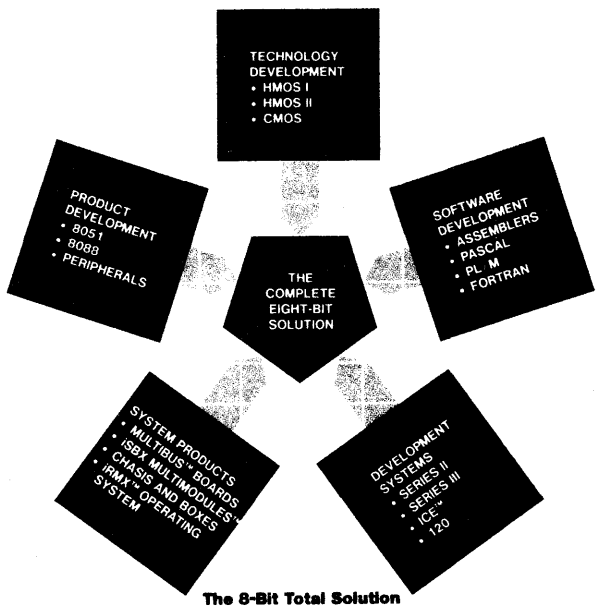
The 8-bit universe is expanding.

There's a continuing rush of new developments. Electrically programmable, electrically erasable ROM memory. Complete bubble memory systems. New floppy disk controllers, peripheral controllers, RAM controllers. New, faster, more powerful microcontrollers. New development systems and development languages. New operating system software—from small, fast real-time executives to full-scale, modular operating systems. New MULTIMODULE on-board expansion products with the new iSBX bus and new iSBC board level computers for the industry-standard MULTIBUS.

The universe of 8-bit applications is expanding and Intel's 8-bit total solution is growing with it.

The entire Intel 8-bit product line—from processors to design aids to peripherals to single board computers to software—is expanding. Providing more design choices, more performance, and more success.

Product and Technology Advances in All Product Segments Add Up to a Total, Integrated 8-bit Solution.



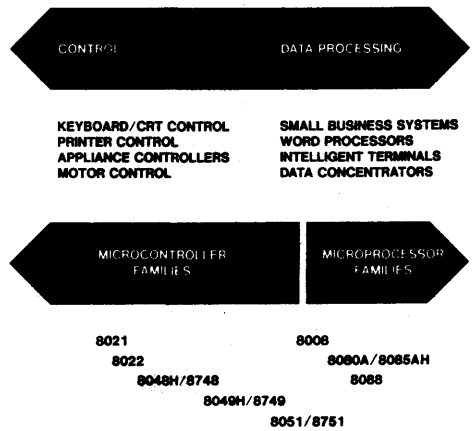
The 8-Bit Total Solution

The Intel 8-bit microprocessor family offers a spectrum of capabilities and performance characteristics, to allow the designer to closely match the functions of the microprocessor to the requirements of the application.

For control oriented applications, Intel's microcontroller family delivers high speed, bit oriented instruction sets, and high integration of memory and peripheral functions on-chip, to reduce system hardware requirements.

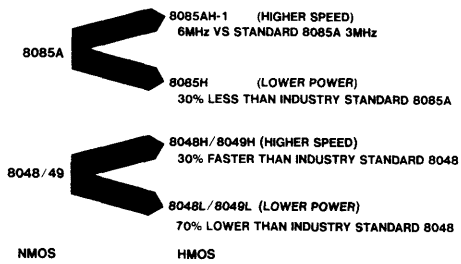
For data processing applications, Intel's microcomputer family offers higher data transfer capabilities, more powerful numerics instructions, and extended memory addressing. These microprocessors are oriented toward reduction of your design costs through architectures suited to time-saving high-level languages, and through built-in interfaces to Intel's debug tools, MULTIBUS system bus, and peripheral devices.

Whether your application requires the low-end 8021 controller or the high-end iAPX 88 microcomputer, you work with a unified design methodology to help bring your product to market quickly. The investment in training and research you make designing any Intel microprocessor into an application carries over to your next application, even though you select a different Intel microprocessor or microcontroller. As important, your investment in development tools is preserved—the Intellec development systems support the entire spectrum of Intel microcontrollers and microprocessors with languages and debug tools.



8-Bit Application Spectrum

8-Bit Applications Span a Wide Spectrum from Simple Control Functions to Complex Data Processing Functions. Intel's Wide Range of 8-Bit Microprocessors Allow the Selection of the Best One for the Application Needs.



More Speed and Power Options

Intel's New HMOS Technology Makes Possible Speed and Power Options Suited to Specific Design Requirements.

HMOS technology is providing new levels of performance in new products—the highly integrated iAPX 88 or the 12MHz MCS 51 microcontroller, for example. HMOS manufacturing process conversions are also increasing performance, reducing power requirements, reducing costs, and allowing you to extend the lives of products based on 8048, 8049, and 8085A microprocessors. You get better system margins, cooler more reliable operation and performance upgrades without any redesign effort. Additional performance also extends these mature products into new applications not previously addressed. As an added bonus, HMOS also provides better availability of product through die size reduction (more dice per wafer).

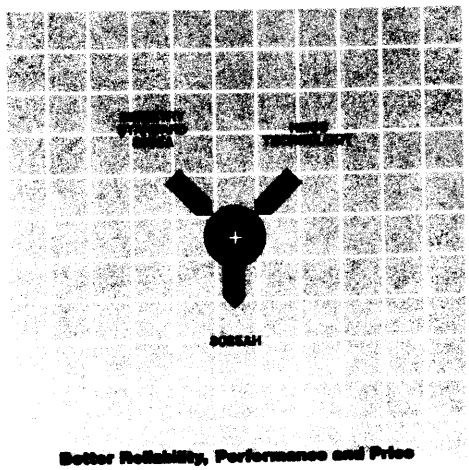
These products deliver the broadest spectrum of design choices, supported by the greatest array of peripheral devices, development tools and languages, memory, systems and bus architectures available.

8085A
8-Bit N-Channel Microprocessor

Single +5V Power Supply	*Cost savings where power supplies represent a significant proportion of system cost.*
100% Software Compatible with 8080A	*Provides easy upgrade from the 8080A. Eliminates the need for programmer retraining and saves costs in development and debug.*
On-Chip Clock Generator and System Controller	*Lowers cost by reducing chip count and board space required.*
3MHz and 5MHz selections available	*The 8085A-2 (5MHz) provides a 60% performance improvement over the standard 8085A.*

Additional Features and Benefits Offered on the New 8085AH

HMOS Processing The 8085AH is ± 10% VCC with 30% less power consumption than the 8085A.	*Improved margins within existing specs* *Cooler running, more reliable systems*
15% reduction in die size	*Improved speed; reduced cost and power consumption.*
3, 5 and 6MHz Selections available. The 8085AH-1 (6MHz) provides a 100% performance improvement over the standard 8085A	*Easy performance upgrades to existing end products*



Better Reliability, Performance and Price

iAPX 88

16 bit architecture	*High performance & software compatibility with 8086 and future extensions. Easy upgrades now and later.*
1 Megabyte address space. Powerful addressing modes	*Capable of managing complex applications and data structures. Enter new markets with your end product.*
High performance with relaxed 8-bit bus interface. Less bus interface and support chips required than in 16-bit systems	*Lowers system cost. Slow memories and peripherals can be used.*
Powerful instruction set with 16-bit math, string handling and support for HLL	*Quicker, less costly software development. Programs easier to understand, write and debug.*
Co-processing and multiprocessing capability. Extra high performance for I/O and numeric intensive applications	*5 X more I/O bandwidth than with single CPU; 100 X more math performance than with iAPX 88 alone.*

iAPX 88 is a family of 8-bit microprocessor products based on the 8088 microprocessor—targeted as a highest performance, most cost effective 8-bit industry standard for the '80's.

A unique design combines a powerful 16-bit internal architecture with a simple, easy to use 8-bit bus interface. This allows current designs utilizing the 8080A, 8085A and other popular 8-bit microprocessors to obtain much higher levels of performance while minimizing the amount and complexity of hardware design or modification.

The 8088's powerful instruction set, 8-bit bus interface and relaxed bus timings can lead to significant system cost savings. The larger, more versatile instruction set simplifies the programming task by directly implementing previously cumbersome code sequences. Software development costs are typically reduced 30-50% due to the reduction in lines of code

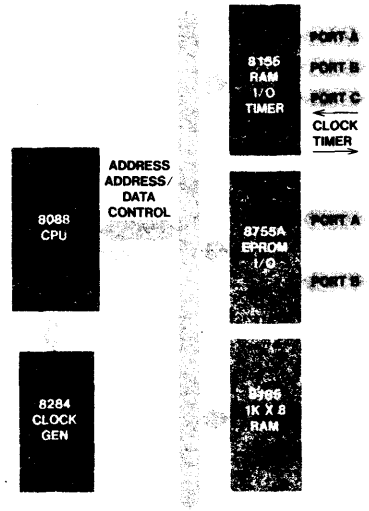
Improved Reliability, Performance, Supply Tolerance, Availability; Reduced Cost, Power Consumption are the Results of HMOS Processing.

necessary. Memory and peripheral speed requirements and system timing margins are improved with 460ns of access time available, far more than slower competitive machines. The amount of memory required for program storage is also reduced typically saving 30% of program memory costs.

For 8-bit systems with key performance requirements in numeric and/or I/O intensive tasks, the 8088 can be combined with specialized numerics (iAPX 88/20) and I/O processors (iAPX 88/11), or both (iAPX 88/21). The 88/20 is the industry's only VLSI implementation of the complete IEEE standard for floating point arithmetic.These systems combine minicomputer performance with microprocessor cost effectiveness and ease of use.

Perhaps most important, of all available 8-bit microprocessors, the 8088 has a clear, well defined growth path for the '80's via the 16-bit iAPX 86 and future architectural extensions into the areas of high performance and high integration CPU's.

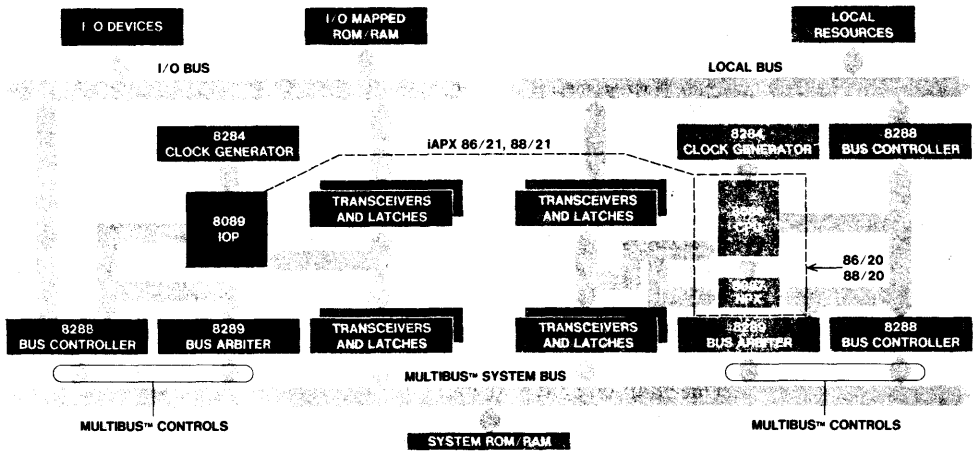
The 8088 is the processor of choice for nearly all byte oriented applications requiring highest performance in which system cost, component count and a growth path are key considerations.



Simple Upgrades Possible With iAPX 88

iAPX 88 Allows For a Highly Integrated, High Performance System Using 8085 Family of Components. 8088's 8-bit Hardware Interface Allows for Easy Incorporation in Upgrades of Existing 8-bit Systems.

The iAPX 88/10 Architecture Expands With the Addition of the 8087 Numeric Processor and the 8089 Input/Output Processor to Include Additional Numerics and Input/Output Instruction and Register Resources.



Expand Math and I/O Resources Easily

8021
Single Component 8-Bit Microcomputer

1K Bytes ROM/64 Bytes RAM, 21 I/O Lines, 1 Timer Counter, on-chip oscillator, zero-cross-detection capability, high current drive capability	*On board memory for system integration to reduce component count. Software compatible with industry standard 8048.*

The 8021 allows low system cost while maintaining 8-bit throughput.

8022
Single Component 8-Bit Microcomputer With On-Chip A/D Converter

2K bytes ROM, 64 Bytes RAM, 28 I/O Lines, 1 Timer Counter, 8-bit A/D, on-chip oscillator, 8 comparator inputs (PORT0), two interrupts—external and timer.	*Low price 8021 CPU with added features offering the lowest system costs in the industry.*

A higher performance version of the 8021. The part provides an 8-bit A/D converter, 2K ROM and 8 comparator inputs to handle more of your application needs.

8048H
Single Component 8-Bit Microcomputer

1K Bytes ROM/64 Bytes RAM, 27 I/O Lines, 1 Timer Counter and 1 external interrupt, microprocessor bus, on-chip oscillator	*On board memory for system integration to reduce component count. Ability to interface to external world, 8080/8085 peripherals and memories*
HMOS Technology New 8MHz operations for 8048H 11 MHz 8048H-1 ICC = 80MA	*Lower power consumption with increased performance.*

Intel HMOS technology has been applied to the industry standard single-chip microcomputer to generate the new 8048H. Capabilities such as a 1.3 μsec cycle time and a typical current of 40MA are now available to provide you with a microcontroller to make your product solution more cost effective.

This additional performance margin coupled with a full line of development support provide you with a leading edge over your competitors in your end product.

8048L

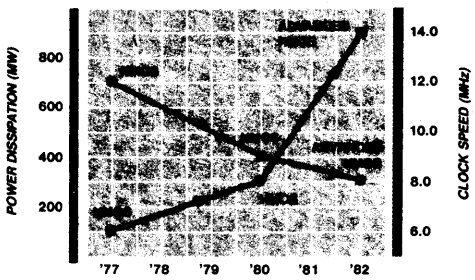
20 ma typical supply current 2 ma typical memory standby current	*Low operating power allows portable and remotely located equipment to be reduced in size and cost*
Minimum voltage for RAM memory retention = 2.2V	*Retain memory during system power down with only two Ni-Cd cells. Reduces battery/charger size and cost.*
Architecture identical to 8748/8048H with 1K bytes ROM, 64 bytes RAM, two interrupts, 1 timer-counter and 27 I/O lines	*More powerful 8048 architecture to provide more system functionality and features with very low current drain.*

A special version of Intel's HMOS technology allows you to use the computing power of an 8-bit machine in applications where power consumption is critical. The 8048L can be quickly and easily designed into any system by using system support tools including a macroassembler, ICE-49 debug system and pin-compatible 8748 member of the 8048 family.

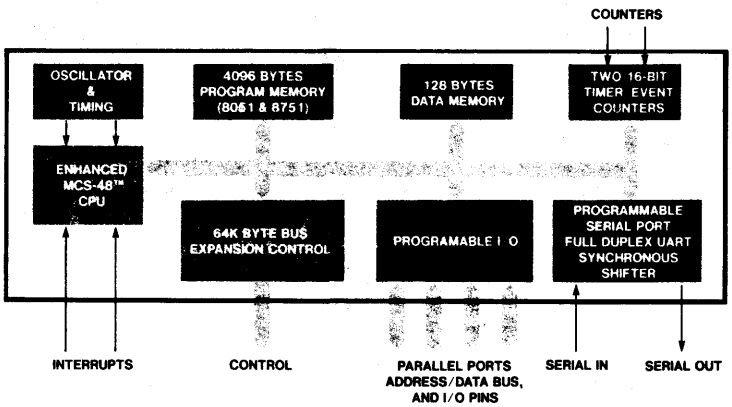
8049H
2K Single Chip 8-Bit Microcontroller

11MHz operation, 2K Bytes ROM, 128 Bytes RAM, 27 I/O lines, Timer Counter, On-Chip Oscillator	*Expanded on-board program storage.*
100% Software compatible to 8048	*Leverage existing software and software expertise in the industry standard 8048.*

A higher performance version of the 8048, the 8049H has a 1.3 μsec instruction cycle. This part gives powerful performance where your application needs more on-chip memory capabilities than an 8048H. Total 8049H support is available from the 8049H assembler and through the powerful Intel debug tool, ICE-49. You can also benefit from the large amount of code written by accessing INSITE, Intel's software library.



HMOS Lowers Power and Increases Performance for 8048



8051 Block Diagram

The 8051 Microcontroller is a Boolean Processor with 4k Bytes EPROM/ ROM, 5 Source-Two Priority Interrupt Structure, Serial I/O Port, On-Chip Oscillator and 32 I/O Lines

MCS-51™ FAMILY

3 Family Members 8751 8051 8031 EPROM ROM No internal Program store	*Gives price flexibility based on application need.*
4K bytes EPROM/ROM, 5 source-two priority interrupt structure, serial I/O port, on-chip oscillator, 32 I/O lines	*EPROM provides fast prototyping. High integration reduces system costs, increases ease of design.*
Boolean Processor	*Excels in bit handling for control applications*
Multiply, divide, subtract, compare	*Instruction set has powerful instructions to relieve software overhead.*
ASM-51, SDK-51, ICE-51	*Complete support package from code preparation, prototyping and debug to speed development. (Up to 10X the 8048 performance)*

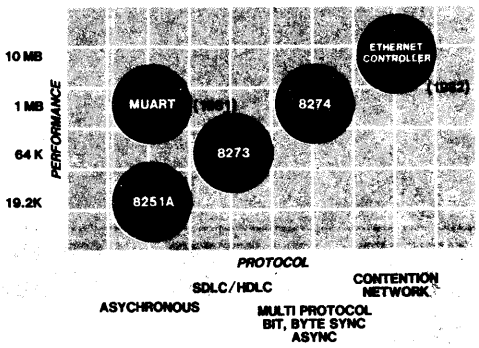
The 8051 is a stand-alone high performance single-chip computer intended for use in sophisticated real-time applications such as instrumentation, industrial control and intelligent computer peripherals. It provides the hardware features, architectural enhancements and new instructions that make it a powerful and cost effective controller for applications requiring up to 64K-bytes of program memory and/or up to 64K bytes of data storage.

The 8051 microcomputer, like its 8048 predecessor, is efficient both as a controller and as an arithmetic processor. The 8051 has extensive facilities for binary and BCD arithmetic and excels in bit-handling capabilities. Efficient use of program memory results from an instruction set consisting of 45% one-byte, 41% two-byte, and 14% three-byte instructions. With a 12 MHz crystal, 58% of the instructions execute in 1 μs, 40% in 2 μs and multiply and divide require only 4 μs. Among the many instructions added to the standard 8048 instructions set are multiply, divide, subtract and compare.

Commitment to the 8-bit market means having the broadest line of peripheral controllers. It also means having peripheral components that are designed to work with the processor line to efficiently address specific application needs while easing interfacing tasks.

With Intel peripherals, the interface to the processor follows a format where bus timings and proper control signals are planned and standardized to ensure compatibility. Each peripheral component is compatible with each new processor, thus preserving and leveraging your design investment in existing peripherals for future processor upgrades.

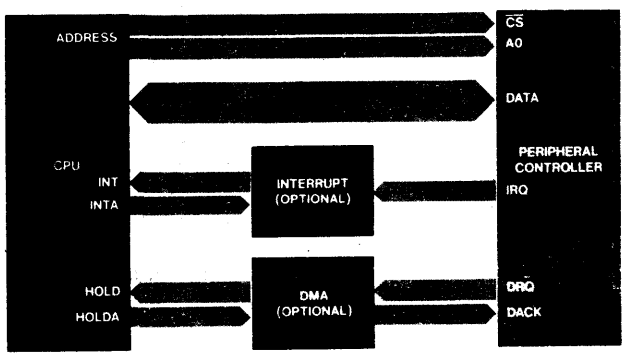
Peripheral support is divided into 3 major areas: data communications, slave processing and controllers. The first, data communications, represents a spectrum of needs from high speed contention networks to low cost asynchronous interfaces. Because different applications require different protocols, various controllers are provided for asynchronous, bit synchronous and byte synchronous support. These controllers handle the data transfer while off-loading the low level tasks from the host CPU and increasing system performance. Data transfer for two types of parallel communication is also supported: Bus arbiters and controllers (8288, 8289 for iAPX 88, 8218 for MCS 80 and 8219 for MCS 85) support MULTIBUS, while transceivers, controllers and talker/listeners (8293, 92, 91) implement the complete IEEE-488 protocol.



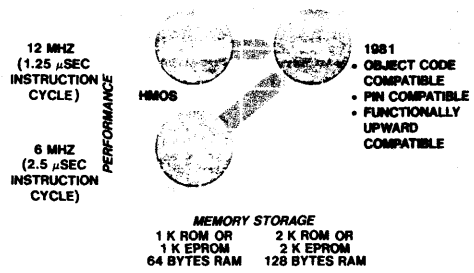
A Spectrum of Data Communications Solutions

Intel's Data Communication Peripherals Address a Spectrum of Needs From Low Cost Asynchronous Interfaces to High Speed Contention Networks.

Intel's Standard Peripheral Interface Enhances System Integrity. It Also Allows Use of the Broadest Available Line of Peripherals with the Intel Processor Line.



Standard Peripheral Interface



Intel's Slave Processing:
Performance and More Performance

Intel's HMOS Upgrades the UPI-41AH Slave Processor to a Performance Level Twice Its Current Operating Speed. Both the UPI-41A and the UPI-41AH Will Be Object Code and Pin Compatible with the UPI-42 To Be Introduced in 1981.

I/O port	*Memory*	*Floppy Disk*	*CRT*	*Timer/Counter*
8255	7220 Bubble	8271 Single	8275 CRT	8253
	8202A DRAM	8272 Double	8276 Small System CRT	8254 10 MHz
	8237 DMA			

Intel Offers Solutions to a Wide Range of Peripheral Device Control Problems.

The second area, slave processing, involves the use of a highly flexible general purpose microcontroller optimized for use in a distributed processing configuration. This approach is useful where no specialized controller exists for the end application. Slave processors also benefit the host CPU by off-loading detailed hardware interfacing and software overhead for ease of design and lower cost. Slave processing is an example where HMOS technology can benefit an existing peripheral. For example, the 8041A will become the 8041AH in the first quarter of 1981 with twice the performance of the existing 8041A.

Slave processing also includes math processing. The 8231A and 8232 components offer floating point capabilities to supplement the general control and data processing software of a CPU.

The third peripheral area is controllers. Here there are solutions for a wide range of peripheral device control: CRT's, dynamic RAM's, floppy disks, and bubble memories. In each case the high degree of LSI integration allow the replacement of 20 to 100 TTL packages. These lines will be expanded with error correction, advance dynamic RAM and CRT control in 1981.

Intel's commitment to peripheral support provides an unsurpassed vehicle for design modularity to reduce development time and increase performance and reliability while reducing part count.

8274
Multi-Protocol Serial I/O Controller (MPSC)

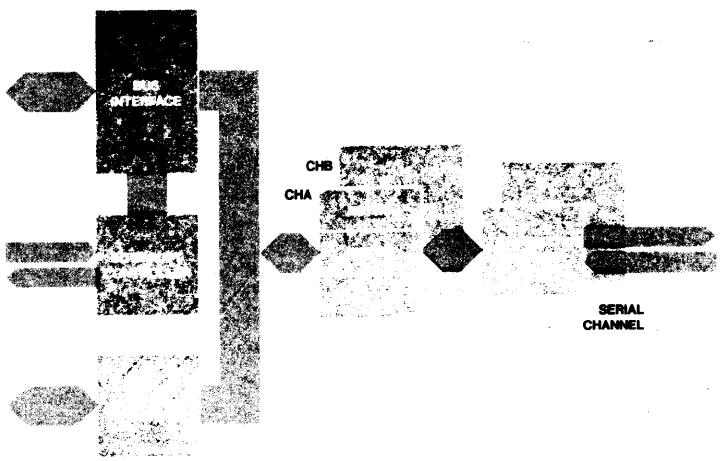
2 Independent, full duplex serial I/O channels	*Provides increased integration (higher reliability, reduced cost) reduces CPU overhead; simpler hardware interfacing*
Multi-Protocols supported: async, bisync, bitsync, X.25, HDLC/SDLC compatible	*Extends end product flexibility and supports applications where multiple protocols are needed.*
1Mb/sec transfer rate	*Improves system performance*
Supports FLAG framing, CRC generation, address recognition, zero bit insertion/deletion	*Offloads host CPU of software overhead for simpler to design, higher performance system.*
iAPX 88, 86, MCS 85, 80, 51 Bus compatible	*Preserves design investment and simplifies hardware interfacing.*

The 8274 combined with an Intel processor exploits Intel technology to provide a very versatile, high performance communication sub-system. It can be used in applications such as data concentrators where multiple protocols and high performance are often needed, and general communication requirements where multiple protocols can enhance product applicability. The 8274 handles the frame level communications while the 8-bit CPU provides higher level functions such as ASCII translation, packing and unpacking data, link establishment, etc.

8273
Programmable HDLC/SDLC Protocol Controller

X.25 HDLC/SDLC compatible, up to 64K baud	*Easiest to use HDLC/SDLC controller for frame-level data communication applications.*
Automatic CRC, programmable NRZI Encode/decode, zero bit insertion/deletion. 2 modem control ports	*Additional integration—simplifies hardware interface to modems, simplifies hardware/software design, removes low level protocol over-head from CPU.*
Full duplex, half duplex, or SDLC loop operation	*Provides compatibility with all commonly used network topologies.*

The 8273 illustrates the use of technology to provide an optimum solution for a specific function, i.e. HDLC/SDLC communications. It provides the simplest HDLC/SDLC solution via a frame level instruction set and by handling low level tasks associated with frame assembly/disassembly and data integrity. The 8273 is ideal for CCITT X.25 applications.



Dual Channel, Multi-protocol Control

Intel's 8274 Multi-Protocol Serial I/O Controller Provides a Very Versatile, High Performance Communication Sub-System for Multi-Protocol Dual Channel Applications. DMA Control, If Desired, Can Be Provided by an 8237 DMA Controller or 8089 I/O Channel Processor.

8041A/8741A
Universal Peripheral Interface

8-bit CPU, 1K x 8 memory, 64 Bytes RAM, 8-bit timer/counter, 18 programmable I/O lines	*Highly integrated, versatile peripheral interface to offload detailed hardware control/interfacing and reduce software overhead to command level.*
Slave peripheral interface	*Slave interface provides higher system performance and modular design for faster development time*
HMOS processing	*8041AH (Q1/2 81) allows for 12MHz performance—1.25 μsec/instruction cycle*
EPROM compatibility & ICE Support	*Allows for fast prototyping and product line flexiblity to support multiple market needs (EPROM)*

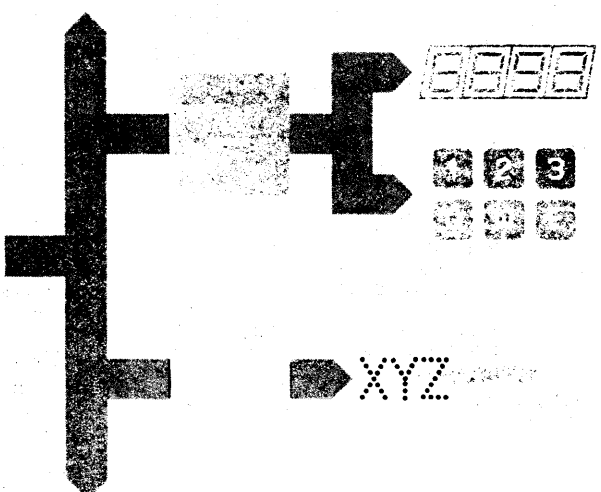
A universal peripheral interface, or slave processor, offloads detailed hardware interfacing and reduces software overhead for the host CPU. This will improve system performance, and simplify design. A slave processor does not become a bus master on the host CPU bus, thus eliminating contention and arbitration problems.

The slave processing concept will be expanded by applying HMOS to the present product line to reduce costs, increase performance and availability. Also, the technology will allow doubling the program/data memory capabilities (8042/8742), the other critical parameter in distributed slave processing designs.

8291A/8292/8293
General Purpose Interface Bus Chip Set

8291A talker/listener for IEEE 488 Bus (GPIB)	*Provides product compatibility to IEEE instrumentation bus standard.*
8292 controller-synchronous control for service requests	*Simplifies state control of IEEE 488 bus, configures the system for multiple talker/listeners in a network.*
8293 bus interface transceivers	*Provides high drive capability for expandibility.*

The 8291/8292/8293 are the highest performance GPIB controllers available (1 Megabyte/sec data transfer). This component set provides a VLSI solution for interfacing microprocessor based instruments and other peripheral devices such as floppy disks, printers, etc., to the IEEE 488 Bus standard.

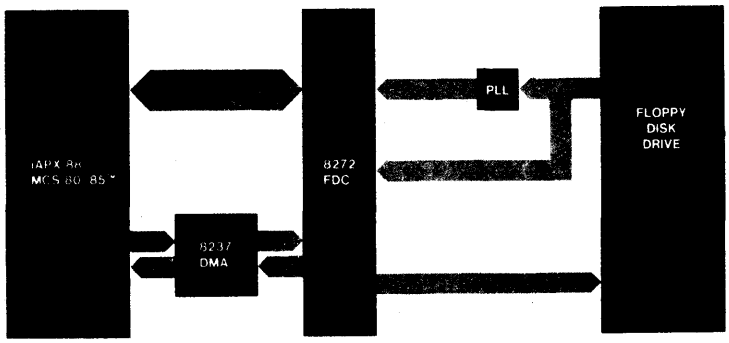


Typical Applications of Intel's 8041A Universal Peripheral Interface

Typical Applications Include:
- Key Board Encoding Control
- Daisy Wheel Printer Control
- Dot Matrix Printer Control
- Communication Control
- Paper Tape Interfacing
- High Speed Fly Reader
- Cassette Controllers
- Analog/Digital Conversion Control
- 7 Segment Displays
- Vacuum Fluorescent Display Interface
- Stepper Motors
- Data Encryption/Decryption
- VCR Edit Controller
- Sensor Matrix Control

Function	Performance 8231A 4MHz	(In μsec) 8232 4MHz
SP Mul	40	50
DP Mul	-	437
SP Div	42	57
DP Div	-	1140
Square Root	205	-
Tangent	1471	-

Execution Speeds for Performance Of Various Math Functions By Intel's 8231A and 8232 Floating Point Math Units.



Simplified Hardware Interfacing of Intel's 8272 Floppy Disk Controller

The 8272 Reduces Software Overhead Read Data Example:
CPU Sends Read Command to the 8272.
8272 Performs:

- **Loads the Head**
- **Automatically Waits for Head Settling**
- **Begins Reading ID Marks and Fields**
- **Compares Disk Sector Number with Target Value**
- **Automatically Begins Reading Data Byte by Byte**
- **Automatically Continues to Read Across Sector Boundaries**
- **Sequence Continues Until Terminal Count is Reached**
- **The 8272 Informs the CPU That the Task is Completed**

8231A/32
Arithmetic and Floating Point Processing Units

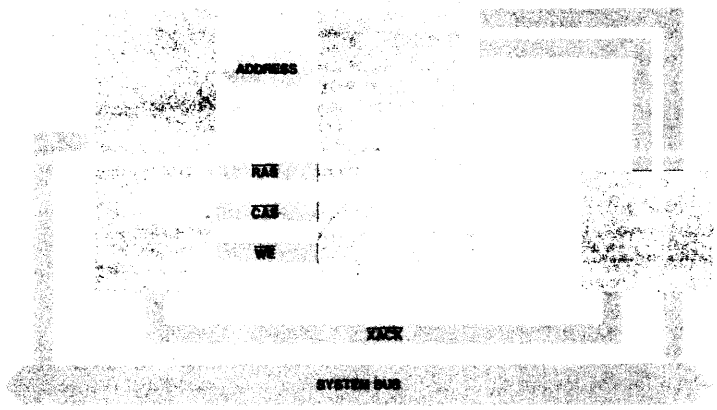
8231A/32—Peripheral interfacing-single precision add, mult., sub., div., 4 mHz operation, 8 x 16 stack oriented storage.	*Ease of use (iAPX 88, MCS 85, 80, 51 compatible). High performance, permits chained calculations without additional external data references.*
8232 Supports REAL MATH®-compatible with the proposed IEEE standard, and double precision add, mult., sub., div., (64 Bits)	*Additional precision and compatibility/portability of results across Intel line of boards/software/components.*
8231A Additionally supports trig., inverse trig., log (16/32 bit)	*Additional commands giving higher performance on these directly supported functions over what software subroutines can provide.*

Applying HMOS technology to the 8231A/32 has given them 33% more performance at the same cost as comparable peripheral math components. The 8231A/32 components provide significant performance (50x in many cases) over that of CPU software routines. Also, the computation can be done in parallel with host CPU processing, further increasing system performance.

8272
Double Density Floppy Disk Controller

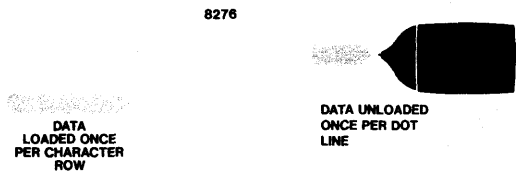
Handles up to 4 double density, double sided drives.	*High integration minimizes external components needed in the disk interface*
Programmable interface functions: Data record length, gap length, head load/unload time, track stepping rate.	*Allows flexible disk interfacing.*
15 High level commands including multi track transfer and data scan capability	*Provides increased performance through reduction of CPU software overhead.*
On-chip address mark detection	*Simplifies data separation logic design.*

The 8272 provides compatibility with IBM recording standards while reducing CPU/software overhead with a powerful command set for high performance. Simple DMA interfacing is supported with 8237 DMA controller or 8089 I/O channel processor.



Dynamic RAM Subsystem

The 8202A Handles the Complete Hardware Problem of Interfacing a Microprocessor Bus to Dynamic Memory. The 8202A Also Handles the Arbitration of Refresh Cycles Versus Memory Cycles.



Efficient CRT Control with Dual Row Buffers

Dual Row Buffer Architecture Reduces CPU Overhead and Memory Speed Requirements.

8202A
Dynamic RAM Controller

Provides address multiplexing strobes for 2104A, 2117, 2118 DRAM memories	*Maps processor control/address/data bus (iAPX 88, MCS 85, 80) into complex signals for memories—saves 15-20 TTL packages.*
Provides transparent refresh capability (Internal) or supports external requests	*Solves the difficult arbitration problem of memory requests and refresh requests for designer—minimizes design problems.*

The 8202A solves the classic hardware design problem of interfacing to dynamic RAM and reliably arbitrating memory and refresh requests. Significantly simplifying the interface, the 8202A provides an effective one-chip solution over delay line or multipackage TTL approaches.

8276
Small System CRT Controller

Interfaces CRT raster scan display with microcomputer bus.	*Permits low component count (<20 packages) CRT designs*
Programmable screen and character format, field attributes, and cursor control	*Allows flexibility, more features, and ease of design.*
Dual row buffers on-chip	*Allows CPU control over display data for linked list and pointer table structures.*

The low-cost Intel 8276 small system CRT controller interfaces CRT Raster scan displays with Intel microcomputer systems. Its primary function is to refresh the display portion of the screen. The flexibility designed into the 8276 will allow simple interfacing to almost any raster scan CRT display with a minimum system IC count.

8254
Programmable Timer/Counter

3 Independent 16-bit counters, Pin compatible with industry-standard 8253. 10 mHz operation.	*Allows simplified upgrade Higher speed for faster processor clocks*

The 8254 is a programmable timer/counter that is pin compatible with the 8253. Its 10 mHz count rate allows it to be used in systems with high-speed processor clocks. Like the 8253, all modes are software programmable.

Technology is constantly applied to reduce the cost of memory per bit and to provide additional functionality, performance and ease of use to the designer and end user. Technology has also allowed Intel to make inroads into exciting new approaches to memories to open up new application areas with pseudostatic byte wide RAMS, mass bubble storage and non-volatile erase and write memories (E^2PROMs).

RAMS

Technology is being applied to random access memories in three directions: cost reductions and increased availability of existing products; CMOS technology for low power dissipation; and higher density/functionality to simplify microprocessor system design.

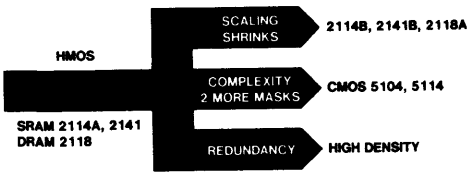
HMOS scaling applied to the popular 2114 static RAM has produced a smaller, less expensive version. Scaling existing and mature product lines has delivered similar results in the 2141 and 2118.

Redundancy on denser RAMs helps improve availability by increasing yields. Redundancy also helps improve quality by allowing larger, more conservative designs to be mass produced. Redundancy will be implemented on nearly all of Intel's future RAMs.

High functionality byte-wide pseudostatic RAMs relieve the system designer of refresh timing, arbitration and control tasks, simplifying memory-to-microprocessor interfacing. Byte-wide RAMS will offer standard configuration RAM for lower density applications.

Bubbles

Based on the principles of magnetic domain storage, Bubble Storage is an important technology which has matured rapidly over the past decade. Intel has developed an affordable, commercially available *complete* Bubble Storage System—one which allows the system designer to concentrate on immediate applications rather than the internal operation of the magnetic bubble memory itself.

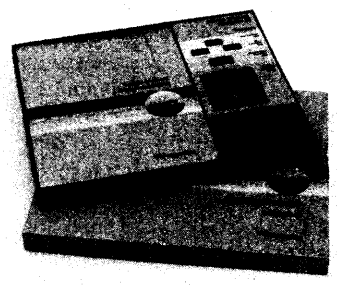


Microprocessor RAM Commitment

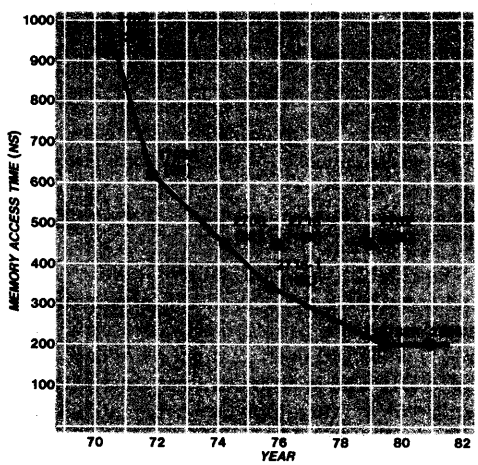
Byte wides
Muxed
4K x 8 PSRAM
Non-Muxed
2K x 8 SRAM
4K x 8 SRAM
8K x 8 PSRAM

DRAM + Controllers
2164 64K x 1

New Processes are Yielding Cost Reductions, Increased Availability, Lower Power Dissipation Characteristics and Higher Density/Functionality in Intel's RAM's



BPK-70 Bubble Storage System
Intel Makes Commercially a Complete Bubble Storage System, Allowing System Designers to Concentrate on the Application Rather than the Internal Operations of the Magnetic Bubble.



EPROM Access Time Trends

Intel's New 2732A and Soon to be Introduced 2764 Have Access Times of 200 ns, 1½ to 2x Faster Than Predecessors of Just a Year or Two Ago.

Components: At the heart of the Intel Bubble Storage System is a VLSI 7220 controller which interfaces easily to Intel microprocessors. The 7220 controller handles up to eight (8) BPK70 one megabit Bubble Storage subsystems.

All of the intricate timing and control functions are totally transparent to the designer, which greatly facilitates design. A 7220 with a BPK70 forms a complete 128K byte Bubble Storage System which occupies less than 100 square centimeters of printed circuit board space.

Additional Reliability: In order to build high reliability into the Bubble Storage System, Intel has utilized advanced VLSI and LSI technology to bring the support chip count down to six. Error detection and correction circuitry is used to give the system additional reliability. A power fail reset circuit is built-in to ensure data integrity in the event of a power failure.

System Programming: The complete Bubble Storage System acts as a peripheral device to the host processor which simplifies system programming. In addition, software routines that demonstrate the basic programming techniques such as intialize, read, seek, write, et cetera, are supplied for reference to help shorten your software development time.

Non-Volatile PROMs

Just as FAMOS technology heralded the beginning of the EPROM world in the 1970's, HMOS-E will begin the new EPROM era in the 1980's. The results? Higher density and higher performance EPROM's (ultraviolet erasable programmable memories) and E²PROM's (electrically erasable programmable memories). Both devices offer non-volatility and the flexibility for production cycle (UV-erasable) or field (electrically erasable) programming capability. And, these devices are now available with up to 8K bytes per chip compatible with the highest speed microprocessors, in the convenient byte format.

2K x 8
2048 x 8-Bit Static RAM

Two line control, CE controls power-down, OE controls output buffers	*Eliminates output data bus contention*
Auto power-down	*Maximize system standby power requirements*
150 ns access/125 ma ICC	*Compatible with high performance CPUs*
25-pin standard pinout	*Industry standard/EPROM compatible*

The Intel 2K x 8 is a 16,384 bit static RAM organized as 2048 words by 8-bits. It employs fully static circuitry which eliminates the need for clocks, refresh or address setup and hold times. Auto power-down cuts power consumption substantially when the device is deselected.

The 24 pin industry standard pinout allows upgrades to 4K x 8 static RAMs and compatibility to the 2732 4K x 8 and 2764 8K x 8 EPROMs. The two line control simplifies decoding and eliminates any possibility of bus contention.

CMOS Static RAMs

1K x 1 and 4K x 1 organizations	*Pin compatible with standard pinouts*
2V data retention/microwatt standby	*Battery operation/backup*
100-150 ns access	*Compatible with high performance CPUs*

The Intel 4096-bit CMOS RAMs employ fully static circuitry which eliminates the need for clocks, address setup and hold times, and reduced data rates due to cycle times that are longer than the access times.

Using Intel's high performance technology, Intel's CMOS RAMs achieve both true microwatt standby, and access times to match present and next generation microprocessors. Intel CMOS RAMs are ideally suited for battery operation and battery backup applications.

21D1
4096 x 8 Bit Pseudostatic RAM

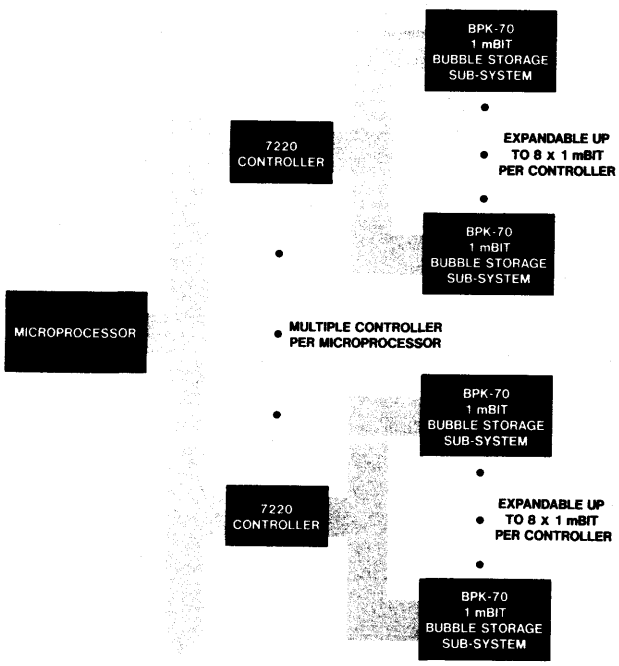
Multiplexed address and data buses	*Direct interface to iAPX processor families*
8 or 16-bit system capability	*Flexible system designs*
Fully integrated refresh	*No external overhead required*
HMOS-D2 technology	*Proven reliability*

The Intel 21D1 is a 4096 word by 8-bit pseudostatic random access memory—PSRAM. Integrating all refresh control circuitry at the chip level allows the system designer to take advantage of dynamic RAM density, performance and price without the added cost of designing the refresh control interface. The 21D1 is intended for use with multiplexed address/data bus microprocessors such as iAPX 88 and MCS 85. All operating parameters have been optimized for high performance, no-wait state operation without TTL interface components. Output drive capabilities and access times are consistent with all present and future microprocessors.

2164 Family
65,536 x 1 Bit Dynamic RAM

Industry standard 16-pin DIP	*High packaging density*
HMOS technology	*High reliability and proven manufacturability*
Pin 1 is No-connect	*Allows for future system upgrade*
Page mode and hidden refresh capability	*High speed data transfers*

The Intel 2164 is a 65,536 by 1-bit N-channel MOS dynamic RAM fabricated in Intel, production proven HMOS process. The 2164 is packaged in the industry standard 16-pin DIP and is designed to operate with a single +5V power supply, with +/− 10% tolerances. The use of a single transistor cell and advanced dynamic circuitry enable the 2164 to achieve high speed at low power dissipation.



Modular System Approach Allows Expansion at 1 mBit (128K Byte) Increments

Using a Modular System Approach, Intel's BPK-70 Allows Expansion at 1MBit (128K BYTE) Increments

MEMORY REQUIREMENTS	SOLUTION	
	MULTIPLEXED	NON-MULTIPLEXED
8K BYTES	PSEUDOSTATIC 4K x 8	STATIC RAM 2K x 8, UPGRADES
8K–64K BYTES		PSEUDOSTATIC 8K x 8
64K BYTES	DYNAMIC x 1 2118 2164 8202A	DYNAMIC x 1 2118 2164 8202A

Intel Byte-Wide RAMs

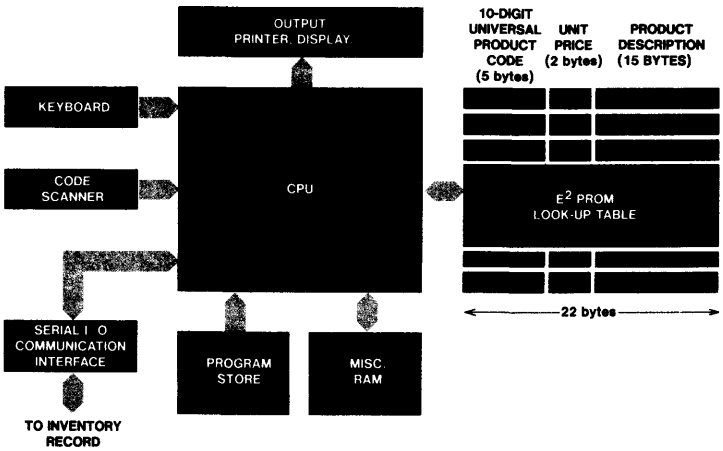
Bubble Storage System

Complete System Approach—7220 controller, up to 8 BPK-70's per controller	*Ease of interfacing to microprocessors; Modular expansion*
"Peripheral" like interfacing	*Simplifies system programming, Directly compatible with Intel microprocessors*
Non-volatility, built-in detection/correction, low component count	*Improves data integrity, system availability and reliability*
Compact and solid-state design	*Microprocessor, LSI circuits and bubble storage can reside on common printed circuit board; This simplifies packaging and reduces cost*

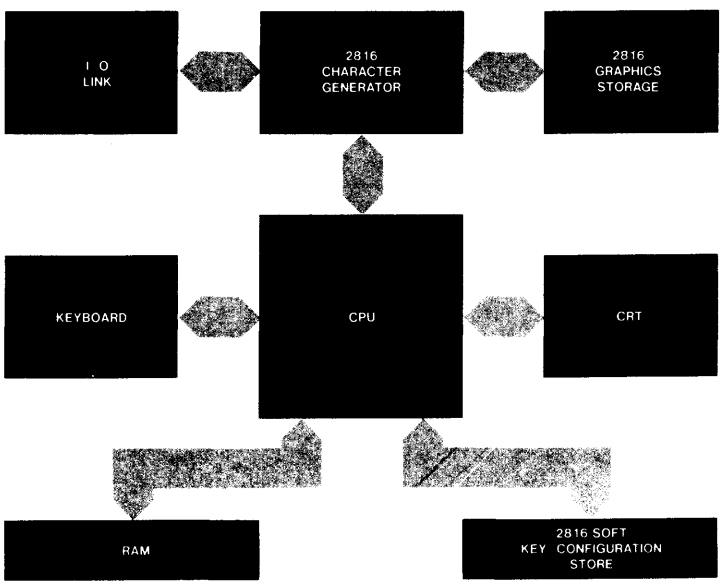
Bubble Storage expands the memory application spectrum. With its unique non-volatility, block orientation, large capacity, and solid-state nature, Bubble Storage is a natural mass storage system for many current and forthcoming microprocessor based applications. Bubble systems can enhance the capabilities of many products, adding significant competitive values.

In the intelligent terminal market place, for example, Bubble Storage can permanently "remember" all the infrequently changed information such as terminal ID codes, display format, look-up tables and soft key definitions. Bubble Storage, in this case, adds accuracy, economy of time and convenience to the product by eliminating the need to key in this information every time the system is used.

In data communications systems, crucial information such as switching tables can now be securely stored and quickly accessed in Bubble Storage. Bubble Storage can be used as a staging device to hold information temporarily until it can be forwarded to the file system or transmitted on a network path.



A POS Terminal Using E² PROM



A CRT Terminal Using E² PROM

2816
16K (2K x 8) Electrically Erasable PROM

Electrically Erasable PROM	*Field Reprogrammable*
Read voltage: +5 Write voltage: +5, +21	*Simplified interface with 8-bit systems*
Byte Alterable	*Optimum for byte or chip alter applications*
Fast read access—200 ns	*Performance compatible with high speed processors.*
Flotox Technology	*Built with HMOS-E Technology.*
High endurance—10^4 writes minimum	*Reliable for frequently changed data/program.*

The Intel 2816 is a 16, 384 bit electrically erasable programmable read-only memory (E^2PROM). The 2816 can be easily erased and reprogrammed on an individual byte basis. A chip erase function is also provided. The device operates from a 5-volt power supply in the read mode; writing and erasing are accomplished by providing a single 21-volt pulse.

The 2816, with its very fast read access speed, is compatible with high performance microprocessors. Using the fast access speed allows zero WAIT operation in large system configurations.

Because of these device parameters, the device is ideal for new and future designs as well as a replacement for existing ROM devices. Some potential uses are:

1. Calibration constant storage (continuous calibration).
2. Software alterable control stores (dynamic reconfiguration).
3. Remote communications programming in a field location.
   - –CRT terminal configuration and custom graphic and font sets
   - –Military replacements for core memory and fuse-link PROMs
   - –Point of sale terminals
   - –Remote alterable look-up tables
   - –Printer and communication controllers
   - –Remote data gathering

2732A—32K (4K x 8) UV Erasable PROM

200 ns access	*Provides end system with additional performance, Compatible with High Speed Microprocessors*
Output enable provides two line control	*Simplifies design, eliminates bus contention*
Power Down Capability	*Reduces power usage*

The Intel 2732A is a +5V only 32,768 bit ultraviolet erasable and electrically programmable read-only memory (EPROM). It is pin compatible to Intel's 450 ns 2732. A 200ns device is available, and the standard 2732A is tested for 250ns performance.

The access time is compatible with high performance microprocessors. This provides for an efficient memory subsystem for iAPX 88 program store that can be operated without WAIT states.

In addition to performance compatibility, the 2732A features an output enable which alleviates bus contention problems associated with single control line architectures.

3636 Bipolar PROMs

Fast Access Time—65ns for 3636-1	*Provides speed for high performance systems*
Three chip select inputs	*Simplified memory decoding*
± 10% Power Supply Tolerance	*More immune to system power interference*
Polycrystalline silicon fuses	*Higher programming yield*

The 3636 is a high performance 16K fuse programmable PROM. The device represents the "state-of-the-art" in bipolar PROMs for the industry. The sub-100ns access makes this device ideal for memory applications where extensive buffering or decoding is needed.

2764 EPROM

8K x 8 format	*Address compatible with 8-bit CPUs*
JEDEC pinout	*Easily upgradable from 24 pin package*
Lowest power per bit EPROM	*Stand-by power reduces power by 75%*
Two line control	*Avoids bus contention*
Fast access—200ns	*Performance compatible with highest speed CPUs.*

The Intel 2764 is a 5-volt only, 65,536-bit ultraviolet erasable and electrically programmable read-only memory (EPROM). The standard 2764 access time is 250ns with speed selection available at 200ns. The access time is compatible with high performance microprocessors. In these systems, the 2764 allows the microprocessor to operate without the addition of WAIT states.

The 2764 has a standby mode which reduces the power dissipation without increasing access time. The active current is 150ma, while the standby current is only 35ma, a 75% savings. The standby mode is achieved by applying TTL-high signal to the CE input.

The 2764 is fabricated with HMOS-E technology, Intel's high-speed N-channel MOS Silicon Gate Technology.

Intel 8-Bit Development Systems

The Intellec Development System offers a broad range of system configurations from low-cost to high performance models and from stand-alone systems to multi-station networks.

When your application is defined and you are ready to develop 8-bit systems, Intellec Development Systems take over the support role and provide all the hardware and software development tools you will need.

Intel 8-Bit Software

The Intellec-based 8-bit comprehensive software family is designed to increase programmer productivity and reduce total software cost. The selection includes five high-level languages (PL/M, PASCAL, FORTRAN, BASIC, COBOL). Macroassemblers are available for every Intel 8-bit processor.

The high-level languages and assemblers produce linkable and relocatable code, allowing you to match the language to the task. They reduce programming costs, maintenance costs, and development time. In addition, Intel's language design allows you to run Intel languages on your final product.

Intel's language family will save implementation time and free resources to work on the value-added portion of your product.

Real-Time In-Circuit Emulation

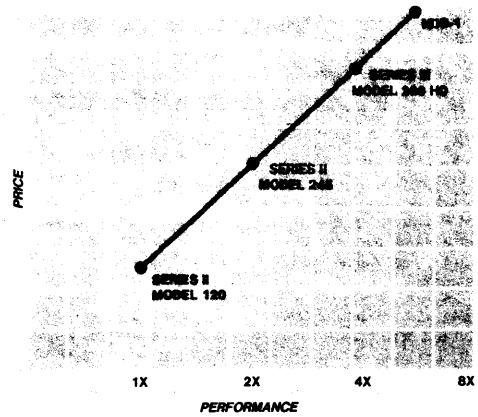
An ICE In-Circuit Emulator extends the Intellec Development System's powerful software execution and system debugging capabilities into your prototype and production systems.

ICE modules are available for all Intel 8-bit processors. ICE emulation provides debugging and test capabilities for the entire development project period. Hardware engineers can use ICE in developing the prototype. Software engineers can use ICE to debug and test programs even before a hardware prototype is available. The earlier you start integrating and debugging your hardware and software, the more control you have over your project. ICE gives you this capability.

Intel's in-circuit emulation capability has expanded with its microcomputer technology. For example: ICE 88 accommodates the full 1 megabyte address capacity of the iAPX 88 processor. ICE 51 utilizes special expanded pin-out versions of the 8051 to deliver an unprecedented level of precise, full function, real-time emulation.

The Powerful Intellec™ Microcomputer Development Systems Support All Intel 8-Bit Products

microprocessors:	8080A, 8085AH, 8088 . . . .
microcontrollers:	8048H, 8049H, 8748, 8021, 8022, 8041AH, 8741A, 8051, 8751 . . . .
single-board computers:	iSBC 80/10, 80/20, 80/04, 80/05, 80/30, 88/40 . . . .
real-time systems software:	iRMX 80, iRMX 88 . . . .



8-Bit Investment Protection With Performance Upgrade Path

Note: 1X Means PL/M 88 Compilation (Statement/Min) Based on Intel Benchmarks.

Successful 8-Bit Development Means Complete Design Cycle Support

Features	*Model 120*	*Series II*	*Series III*
12" CRT/Keyboard	Yes	Yes	Yes
CPU Board	8080 Based	8085A Based	8086/8085A Based
Dynamic RAM	32K bytes expandable to 64K bytes	64K bytes	192K bytes expandable to 1Mbytes
Operating System	ISIS	ISIS	ISIS (Super Set)
Disk Capacity	250K bytes (All expandable to 2.5Mbytes)	1.2Mbytes	1.2Mbytes
Hard Disk Option	7.3 Mbytes	7.3 Mbytes	7.3Mbytes
Assemblers and ICE In-Circuit Emulators	8080A, 8048H/49H 8041AH, 8022	8080,8085AH 8048H/49H, 8088 8041AH, 8051 8022	8080,8088 8048H/49H, 8085AH 8041AH, 8051 8022
High-Level Languages	NONE	PL/M 80/85/88 BASIC 80/85 PASCAL 80/85 FORTRAN 80/85/88	PASCAL 80/85/88 PL/M 80/85/88 FORTRAN 80/85/88 BASIC 80/85/88

8-Bit Total Support

Intel's total 8-bit support goes beyond software and in circuit emulation. It includes the INSITE user's program library, worldwide training workshops, field application engineer support, and on-site customer service.

8-Bit Investment Protection

As Intel's 8-bit product lines evolve and advance in technology, your development system and support tool investment becomes more valuable. Each system can be expanded to handle both your current and future 8-bit requirements.

Intel's Development System Family Series III

The Series III is a self-contained dual-processor (8085A/8086) development system. It is designed to provide optimal software development and debug tools for the iAPX 88,86. Additionally, it provides tools for support of all Intel 8-bit processors. iAPX 88 code may be written either in resident macroassembler or the PASCAL, FORTRAN and PL/M compilers. Using the hard disk, 8088 languages compile at least four times faster on the Series III than on previous systems.

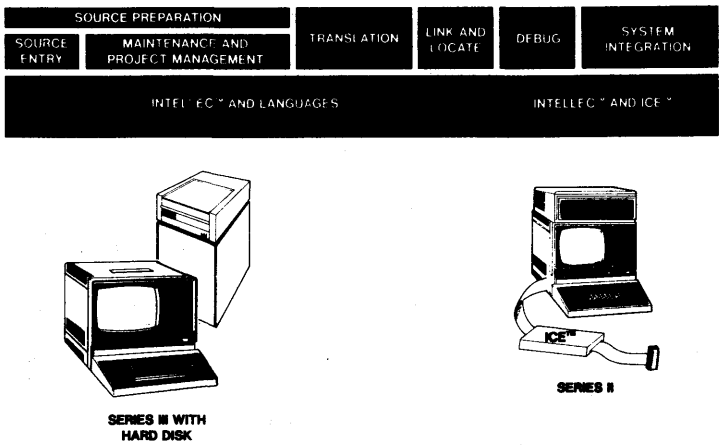
All previous Intellec systems, Series II and MDS-800, may be upgraded to Series III performance.

Series II

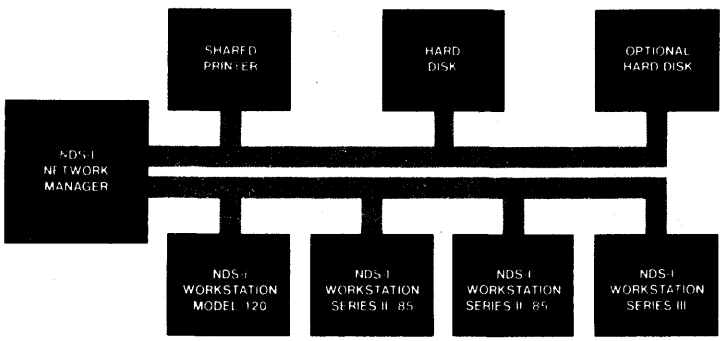
The Series II supports resident 8080/8085AH high-level language compilation and debugging. In addition, the Series II offers compilers and assemblers to generate 8048H/49H, 8041AH, 8022, 8051, and 8088 object code. For ICE debug requirements, the Series II supports every single 8-bit ICE product.

Model 120

The low cost Model 120 provides the minimum system required for the 8080A, 8048H/49H, 8041AH, and 8022 development while allowing you the option of easily upgrading to the Series II or Series III as your performance needs and budget allow.



Successful 8-Bit Development Means Complete Design Cycle Support



Intel Network Development System NDS-I

NDS-I

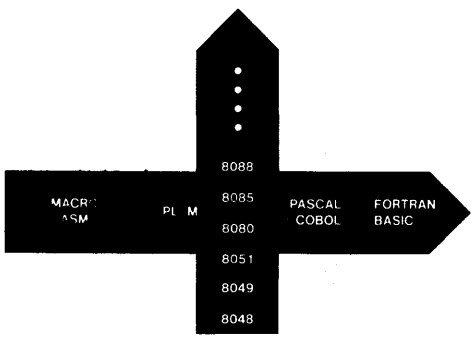
Supports up to 8 Intellec Standalone Systems.	*Allows 8 stations to share central files for coordination of larger projects.*
Up to 14.6 Mbytes of hard disk storage.	*Ample data and program storage for large projects; high-speed compilation, assembly and linkage.*
50% performance improvement over standalone floppy disk-based system.	*Minimum waiting, more efficient use of engineering manpower.*
Shared background printer and hard disk.	*Sharing expensive peripherals means lower cost per user.*
Configurable from existing Intellec Systems Model 120, Series II, Model 800 and Series III.	*Maximum investment protection for Intellec users.*

The NDS-I distributed development system provides support for 8-bit applications in multiple projects and large program environments. The NDS-I delivers tools for cost-effective management of large projects, including shared access to both the central hard disk and a central line printer. The NDS-I allows high-speed compilation stations, debugging stations, and editing stations to reside on a common network providing optimum price/performance tailored to the user's needs.

PL/M Language

Block structure with English-like statements	*PL/M programs can be virtually self-documenting*
Built-in string handling, data structure facilities	*You reduce the expense—and shorten the time—involved in developing software for your product.*
Relocatable and linkable object code	*Program modules are compatible and combinable with modules written in PASCAL, FORTRAN, and assemblers.*
Debugging options allow inclusion of symbol table information in object modules	*Use of this by ICE in-circuit emulators allows symbolic rather than absolute references to labels and variables, leading to faster and simpler debugging.*

PL/M, first introduced by Intel for the 8080 microprocessor in 1973, is a systems implementation language specifically for performing software development for the Intel 8-bit processors. It has evolved into the most powerful and popular microprocessor systems language.



Intel 8-Bit Languages

PASCAL Language

ISO PASCAL standard	*ISO standard aids portability of application programs*
Extensions for application microprocessor Interrupt handling Direct Port I/O Structured constants	*Powerful extensions increase programmer productivity and program functionality.*
Separate compilation extensions allow: Modular decomposition of large programs Type-checking at link-time	*Extends the user's ability to write structured and modular programs; easier to maintain.*
Macro capability compatible with assembler	*Facilitates conditional compilation and implementation of in-line subroutines.*
PASCAL compiler runs on Series III, iRMX, or a user-defined operating system.	*User has a variety of easy-to-implement run-time options.*
PASCAL compiler generates symbol record information	*Supports symbolic debugging for faster debugging using ICE.*

PASCAL is a highly structured, block-oriented programming language which has become very popular as an 8-bit microcomputer application language. Its rigid and readable structure enforces good programming techniques which help produce more reliable and maintainable software.

FORTRAN Language

Extensive floating-point arithmetic capability.	*Earlier project completion and faster program execution with comprehensive arithmetic and data management support.*
Provides reentrant, interrupt, and error handling procedures.	*Increase programming productivity and system performance.*
Direct byte- and word-oriented port I/O.	*Easy access to microprocessor hardware simplifies programming tasks.*
Well-defined interface for user-supplied I/O device driver or operating systems.	*Facilitates user replacement of development operating system by tailored interfaces in the final product.*

Intel's 8-bit FORTRAN compilers, FORTRAN 80 and FORTRAN 88, meet the ANS 77 standard subset language that translates FORTRAN statements into relocatable object modules. The object modules are linkable with PL/M , PASCAL, and macroassembler object modules. The Intel FORTRAN compilers have a well-defined operating system interface to support Intellec Series II, Series III, iRMX and user defined operating systems.

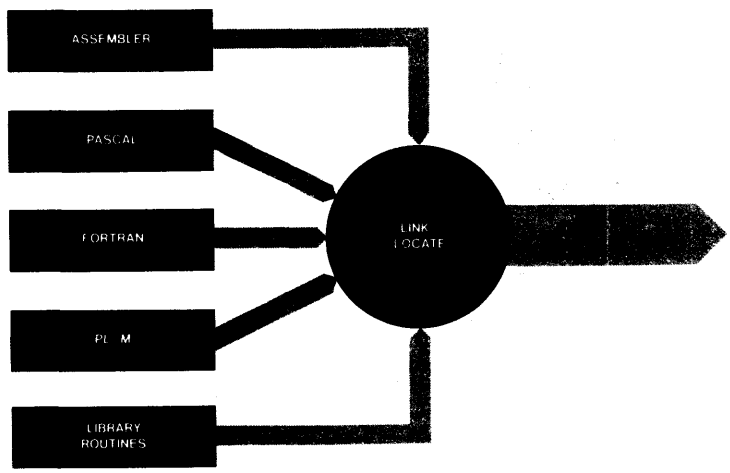
Both FORTRAN compilers take advantage of the Intel iSBC math board or the 8087 Intel floating point chip.

8-Bit Macro Assemblers

Macro Facility and built-in functions include: condition, repetition, and string processing.	*Aid debugging and increase programming efficiency.*
Generates relocatable and linkable object code.	*Permits program development in small and manageable objects; linkable with high level language object modules.*
Generates Symbol Table, supporting ICE symbolic debugging.	*Faster debug and product completion.*

Intel provides a macro assembler for every Intel 8-bit processor. The macro assemblers translate symbolic assembly language mnemonics into machine code with maximum code efficiency.

Each 8-bit macro assemblers offer many features normally found only in high-level languages. For example, the 8051 and 8088 assembly languages are strongly typed. The assemblers perform extensive checks on the usage of variables and labels. Thus, many programming errors will be detected and development time reduced.



Intel Software is Designed to Allow Intermixing of the Output of Different Languages

ICE 88 /51 /41A /49 /80 /85B /22™

Real-time hardware emulation	*Test your application under actual system operation*
Full symbolic debugging	*Communicate with your hardware and software without calculating address values*
Program trace	*Obtain an immediate history of system execution*
Disassembly of object code into instruction mnemonics	*View program traces and code in memory as assembly language source*
Mapped emulation memory	*Run programs without application system memory in place*
ICE 41A/49 MULTI-ICE multiple in-circuit emulation	*Debug a multiple processor system in combination with ICE-85 emulator*
ICE 88 One megabyte addressing	*Integrate entire 8088 application with in-circuit emulation*
ICE 51/22 HELP Displays	*Look up command syntax from the keyboard*
Single-Line assembler	*Patch code in assembly language*
ICE 88/51/MULTI-ICE Command macros	*Store frequently used command sequences as a single name.*
Compound commands	*Automate debugging sessions with conditional commands thereby saving time.*

An ICE in-circuit emulator provides control of the application system design throughout the project. The ICE module is operated by entering commands from the Intellec system keyboard or from a disk file. The ICE module controls and diagnoses your system via a connection to the microcomputer socket.

An ICE emulator allows you to design more advanced features into your product and complete the design faster. The ICE module provides direct hardware access to key points in the system. Emulator software helps automate debugging sessions. The result is more time allocated to design, less time allocated to devising and setting up test equipment and analyzing data.

All ICE emulators feature symbolic debugging, which lets you reference addresses and variables by names or symbols. When you assemble or compile object code on the Intellec system, you can direct the system to generate a symbol table which can be loaded directly into the emulator environment to start your emulation symbol table. Symbolic debugging spares you the tedium of looking up continuously changing values in your listings throughout a debugging session. The result, again, is that you concentrate your efforts on successful system design, not on test procedures or test equipment.

ICE benefits begin the day you start writing programs for your microcomputer. Each ICE module allows operation in a software debugging configuration which provides the capability to execute programs before prototype hardware is ready. Once you have written segments of code to exercise the microcomputer's interface with the prototype system, you can use the emulator's real-time operating capability to develop the hardware by testing each section under actual operating conditions.

The value of maintaining control by developing your project in manageable segments continues as you integrate hardware and software. As later phases of the system hardware become available, ICE allows you to exercise each prototype. ICE provides a cornerstone and frame of reference as the project moves toward final integration of hardware and software.

When your system moves from the lab to the factory, the Intellec system and ICE module continue to pay off in high productivity. Diagnostic routines, stored as Intellec disk files, provide the emulator with commands to exercise the system under test.

A shortcut to success. Intel OEM Microsystems deliver packaged VLSI microcomputer innovation so you can focus on your application.

Modular board and software products break complex designs into manageable pieces configurable off-the-shelf.

A complete family of 8-bit single board computers with on board CPU, RAM, EPROM, and I/O.

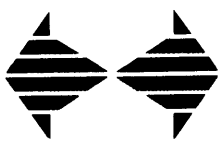
The MULTIBUS architecture to expand system resources from a selection of more than 30 Intel boards.

A new family of MULTIMODULE boards and a new Intel bus standard, the iSBX bus, provide incremental, low-cost, on-board expansion for Single Board Computers.

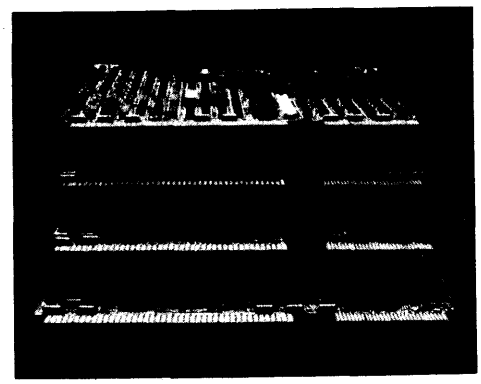
The engineer has two major VLSI design alternatives: components and boards. For many, the best combination of high performance, low risk, minimal investment, and fast turnaround is the board level solution. Intel single board computers have become industry standards due to continued commitment to technology.

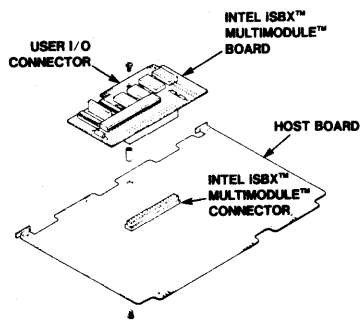
Integrating new technology into the MULTIBUS™ board family.

New microprocessors mean new single board computers. Intel was first to take the 8-bit microprocessor to a single board solution in 1976 announcing the iSBC 80/10 board with the 8080. Soon after, numerous 8085 based boards were developed, tracking the microprocessor evolution. And now, the first 8088 based board is underway—the iSBC 88/40 Measurement and Control Computer. The MULTIBUS board family has currently grown to more than 30 boards including a broad range of Single Board Computers, memory, I/O and peripheral controllers.



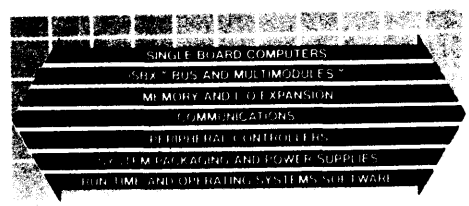
iSBX™ Bus

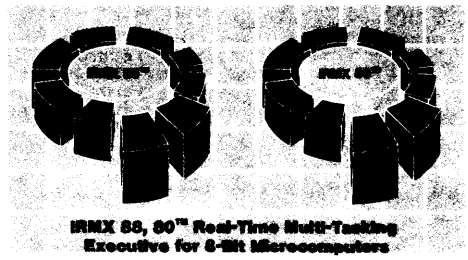




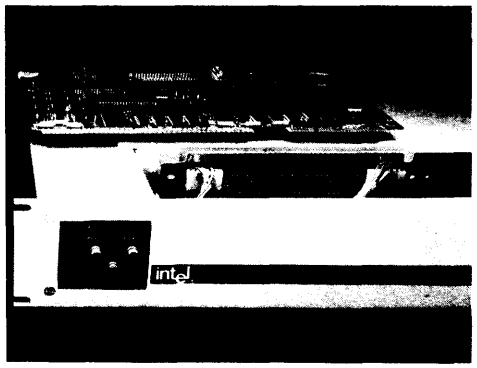
MULTIMODULES™ Provide Incremental, Low-Cost, On-Board Expansion

The new iSBX™ Bus MULTIMODULE™ boards connect computers by means of a special iSBX Bus connector.





iRMX 88, 80™ Real-Time Multi-Tasking Executive for 8-bit Microcomputers



Adapting to technology enhancements. As technology improvements develop, second and third generation components offer more capacity and speed. Intel board products continue to improve by integrating technology into new board generations. Today more than half the MULTIBUS products are a result of these improvements. To cite a few examples:

The iSBC 80/24 Single Board Computer—more than twice the speed and four times the memory capacity of its predecessor.

The iSBC 464 64K byte EPROM Expansion Board—a third generation product compatible with today's high technology EPROM components.

The iSBC 208 Flexible Disk Controller—a third generation product integrating features previously requiring three boards onto a single board.

Offering flexible system expansion. iSBX MULTIMODULE boards (announced in 1980) provide a new kind of system expansion complementing the MULTIBUS architecture. The MULTIMODULE concept offers on-board incremental expansion, to tailor-fit the system functions to the application. The designer selects the appropriate MULTIBUS board, then adds a low cost MULTIMODULE board to precisely fit the expansion need.

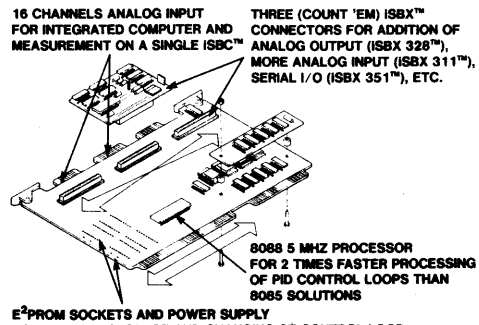
A total solution

Intel systems—hardware and software are available providing total solutions. The iRMX Real-Time Operating System provides an easy-to-use, structured environment for application programs. The operating system together with language support for FORTRAN and BASIC greatly simplify the application design and reduce development time and risk.

Systems packaging and power supplies take the final step to a complete solution. System packaging from Intel includes chassis systems which provide power supply, card cage, backplane control panel and 19-inch rackmount structure. There are also modular cardcage/backplane assemblies and power supplies.

iSBC 88/40™ Measurement and Control Computer

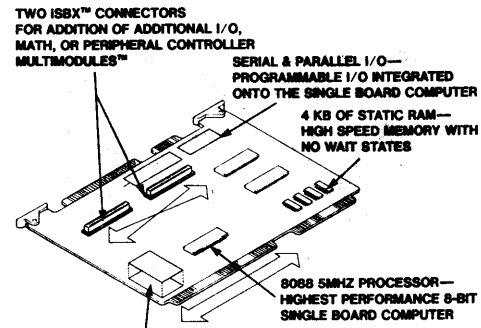
The Intel iSBC 88/40 Measurement and Control Computer is a member of Intel's large family of Single Board Computers that takes full advantage of Intel's VLSI technology to provide an economical self-contained computer based solution for applications in the areas of process control and data acquisition. The on-board iAPX 88/10 processor with its powerful instruction set allows users of the iSBC 88/40 board to update process loops as much as 5-10 times faster than previously possible with other 8-bit microprocessors. For example, the high performance iSBC 88/40 can concurrently process and update 16 control loops in less than 200 milliseconds using a traditional PID (Proportional-Integral-Derivative) control algorithm. The iSBC 88/40 board is capable of functioning by itself in a stand-alone system or as a multimaster or intelligent slave in a large MULTIBUS system.



iSBC 88/40™ Measurement and Control Computer

iSBC 88/25™ Single Board Computer Available 2nd Half 1981

The Intel iSBC 88/25 Single Board Computer is a member of Intel's complete line of OEM microcomputer systems which take full advantage of Intel's LSI technology to provide economical, self-contained computer-based solutions for OEM applications. Full MULTIBUS interface logic is included to offer compatibility with the Intel OEM Microcomputer Systems family of Single Board Computers, expansion memory options, digital and analog I/O expansion boards, and peripheral and communications controllers.



iSBC 88/25™ Single Board Computer

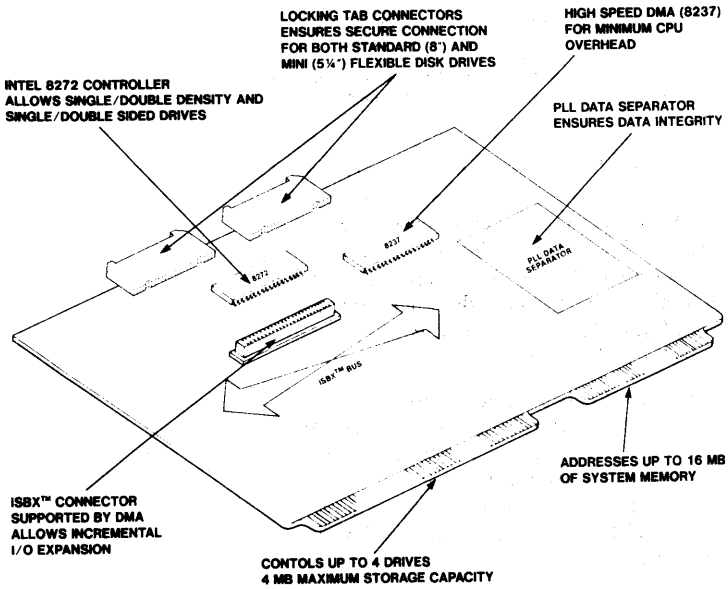
iSBC 254™ Bubble Memory Storage

Capacity up to 512K Bytes of non-volatile bubble memory storage—expandable to 1 megabyte.	*High density memory subsystem that maintains data once power goes off*
DMA capability—up to 50K bytes/sec	*Increases system performance through burst transfer capability*
Automatic error correction powerfail data protection	*Insures data integrity*
Average access time of 48ms	*Low latency access time for performance*
Low power consumption (32 watts per 1/2 megabyte)	*Reduces power consumption costs, supply requirements*

The iSBC 254 is a completely assembled and tested non-volatile memory utilizing the Intel Magnetics 7110 1Mbit bubble memory, 7220 bubble memory controller and 8257 DMA controller. It provides up to 512K bytes of bubble memory and is designed for use with MULTIBUS systems. The iSBC 254 is supported by iRMX 80.

The iSBC 254 is designed to exceed the reliability of mechanical disk and tape systems and will operate in harsh environments of shock and vibration where they can't. The bubble and its support components have an estimated MTTF (mean time to failure) of twenty years.

Data transfers can take place via one of three methods: DMA mode (8257), polled mode or DRQ mode (FIFO Buffer on 7220 BMC). Handshaking with the host CPU for status information can be either interrupt driven or polled.



iSBC 208 Flexible Disk Controller

The iSBC 208 Flexible Disk Controller Controls a Variety of Single and Double Density Drives, Reducing Costs, Backplane and Power Requirements.

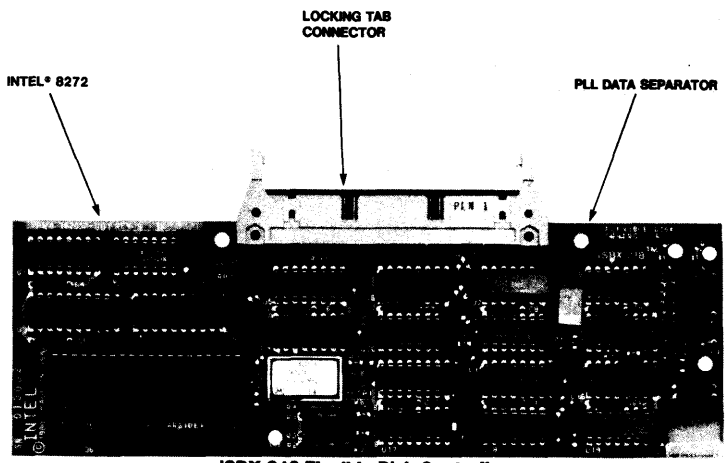
iSBC 208™ Flexible Disk Controller—Available Early 1981

Intel 8272 Floppy Disk Controller Controls Single/Double Density Single/Double Sided Industry Standards FM (Single Density) MFM (Double Density) 4 Drives Standard (8") Mini (5¼")	*Single MULTIBUS system card controls a variety of single and double density drives, reducing cost, backplane and power requirements.*
Locking Tab Cable Connector	*Maximum Connector Security Cable*
Phase Lock Loop (PLL) Data Separator	*Maximum data integrity*

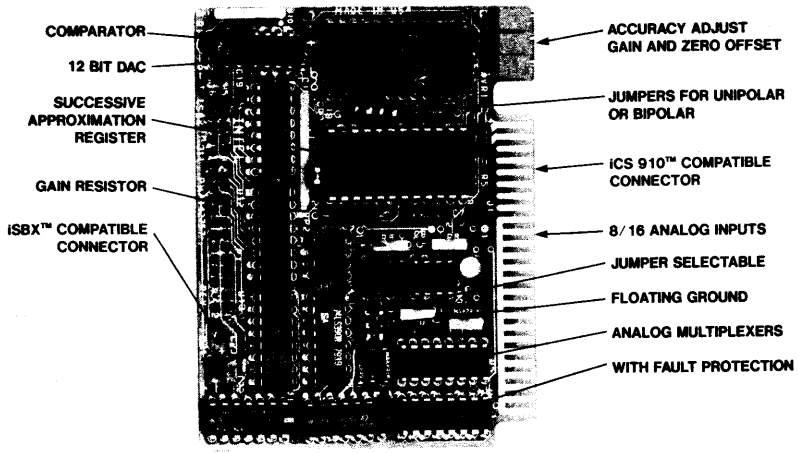
The Intel iSBC 208 Flexible Disk Controller is a diskette controller capable of supporting virtually any soft-sectored, double density or single density diskette drive. The standard controller can control up to four drives with up to eight surfaces. In addition to the standard IBM 3740 formats and IBM System 34 formats, the controller supports sector lengths of up to 8192 bytes. The controller can read, write, verify, and search either single or multiple sectors. Additional capability such as parallel or serial I/O special math functions can be placed on the iSBC 208 board by utilizing the iSBX bus connection. I/O data transfers are made via high speed DMA, freeing the processor for concurrent operation.

iSBX 218™ Flexible Disk Controller

The Intel iSBX 218 Flexible Disk Controller is a double-wide iSBX board diskette controller capable of supporting virtually any soft-sectored, double density or single density diskette drive. The standard controller can control up to four drives with up to eight surfaces. In addition to the standard IBM 3740 formats and IBM System 34 formats, the controller supports sector lengths of up to 8192 bytes. The iSBX 218 board's wide range of drive compatibility is achieved without compromising performance. The operating characteristics are specified under user program control. The controller can read, write, verify, and search either single or multiple sectors.



iSBX 218 Flexible Disk Controller



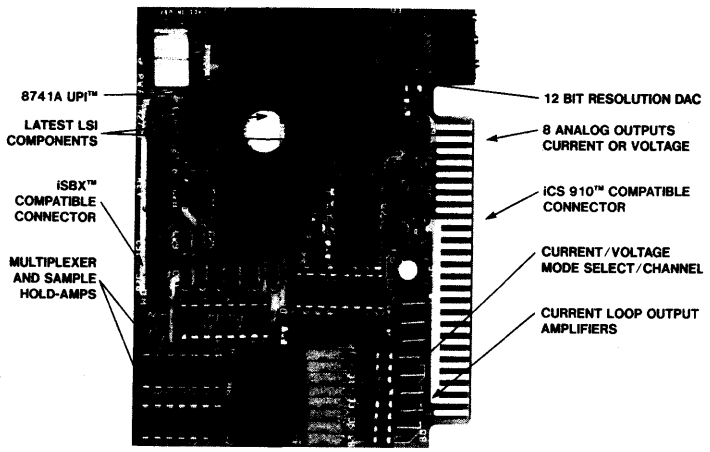
iSBX 311™ Analog Input Board

The iSBX 311 Analog Input Provides Incremental, Low Cost Control of Analog Elements.

iSBX 311™ Analog Input—Available Early 1981

16 channels of analog input on a single iSBX compatible MULTIMODULE board	*iSBC analog inputs now cost one-half as much as before—a low-cost incremental analog interface.*
8 differential or 16 single-ended analog inputs—jumper selectable	*Flexibility for OEM's to mix iSBX's to match their applications. High density for low noise environments (16 channels per iSBX 311) or 8 differential channels to reduce common mode noise.*
Gain selection from 1x to 250x via user supplied gain resistors	*Millivolt level signals from strain gauges or thermocouples can be monitored with the 250x gain (20 mV full scale voltage input). Gain x1 (5V full scale) is useful for most "high level" signals and 4-20 mA inputs from the iCS 910 (see below).*

The Intel iSBX 311 Analog Input MULTIMODULE board provides simple interfacing of non-isolated analog signals to any iSBC board which has an iSBX compatible bus and connectors. The single-wide iSBX 311 plugs directly onto the iSBC board, providing data acquisition of analog signals from eight differential or sixteen single-ended voltage inputs, jumper selectable. Resistor gain selection is provided for both low level (20mV full scale range) and high level (5 volt FSR) signals. Incorporating the latest high quality IC components, the iSBX 311 MULTIMODULE board provides 12 bit resolution, 11 bit accuracy, and a simple programming interface, all on a low cost iSBX MULTIMODULE board.

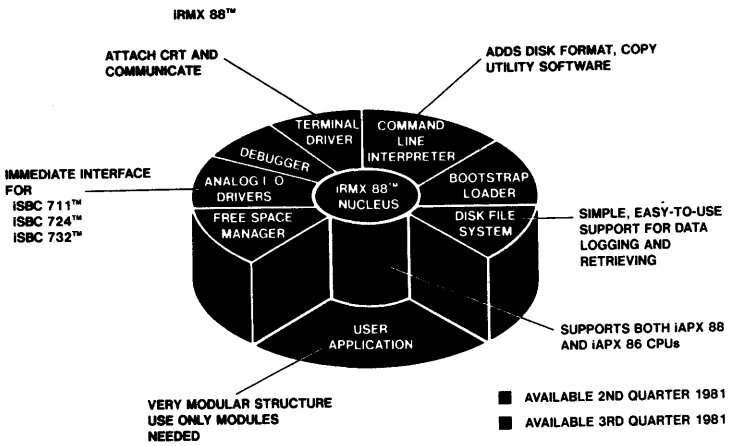


iSBX 328™ Analog Output Board

iSBX 328™ Analog Output—Available Early 1981

8 channels of analog output on a single-wide iSBX compatible MULTIMODULE board. Twice the channels at 1/2 the price of the iSBC 724.	*Low cost control of analog control elements (dampers, valves and linear voltage controlled devices). Up to 24 channels can be mounted on single iSBC (3 times the competition).*
4-20 mA current loop or voltage output, jumper configurable on any mix on 8 channels. Jumper selectable unipolar (0 to 5V) voltage, bipolar (−5V to +5V) voltage, or 4-20 mA current loop output on any channel.	*Matched to the current and voltage ranges of the most popular industrial analog controlled devices. Allows a mix of meters, valves, CRT's and other devices from a single iSBX.*
12 bit resolution Digital-to-Analog converter (DAC) with 11 bits accuracy (0.035%) full scale).	*Sufficient resolution and accuracy for voltage stimulus on test stands and to provide precise control of most industrial control elements.*

The Intel iSBX 328 MULTIMODULE board provides analog signal output for any iSBC board which has an iSBX compatible bus and connectors. The single-wide iSBX 328 plugs directly onto the iSBC board, providing eight independent output channels of analog voltage for meters, CRT control, programmable power supplies, etc. Voltage output can be mixed with current loop output for control of popular 4–20mA industrial control elements. By using an Intel single chip LSI computer (8041A) for refreshing separate sample-hold amplifiers through a single 12-bit DAC, eight channels can be contained on a single MULTIMODULE board, for high density and low cost per channel. High quality analog components provide 12 bit resolution, 11 bit accuracy, and slew rates per channel of 0.1 volt per microsecond. Maximum channel update rates are 5kHz on a single channel to 1 kHz on all eight channels.



iRMX 88™ Real-Time Multitasking Executive

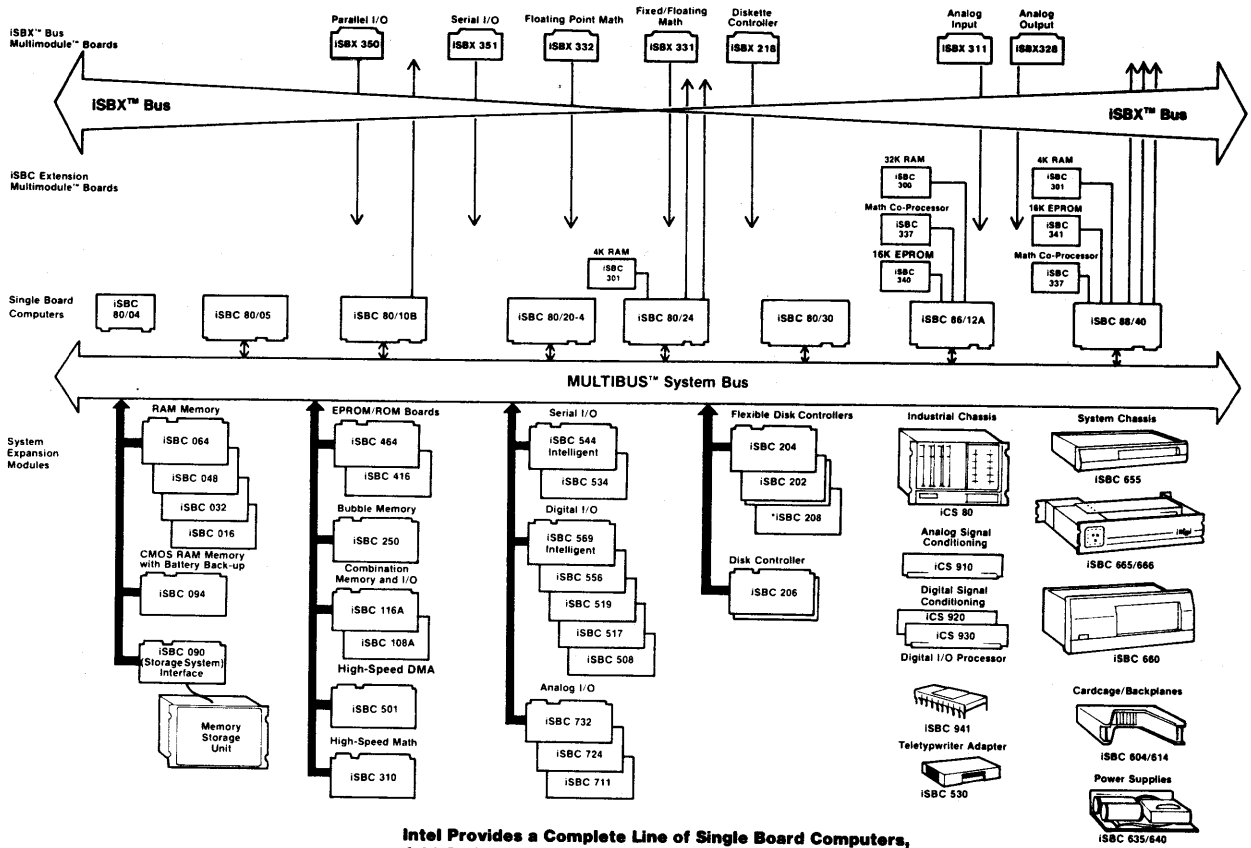
The iRMX 88 Real-Time Multitasking Executive Will Provide Operating System Software Control for the CPU. It Will Support Real-Time Application Requirements for Intertask Communication, Asynchronous I/O Control, Priority Based Resource Allocation and Standard iSBC Disk Controller Interfaces.

iRMX 88™ Real-Time Multitasking Executive—Available Early 1981

Convenient, consistent structure; Standard operating system interface for application development	*Experience can be easily transferred from CPU to CPU reducing program cost*
Small size	*Multitasking nucleus with 1.2 Kbyte minimal memory requirements for economical applications*
Fast, low overhead, operating system	*Faster interrupt handling assures customer can meet I/O requirements, won't lose incoming real-time data.*
Simple, field-proven interface compatible with iRMX 80	*Readily learned for first-time users*
Supports iAPX 88 and iAPX 86 based boards and components	*Both iSBC and component design support allows customer to channel dollars to those areas yielding greatest return. 8/16 compatibility for easy upgrade path.*

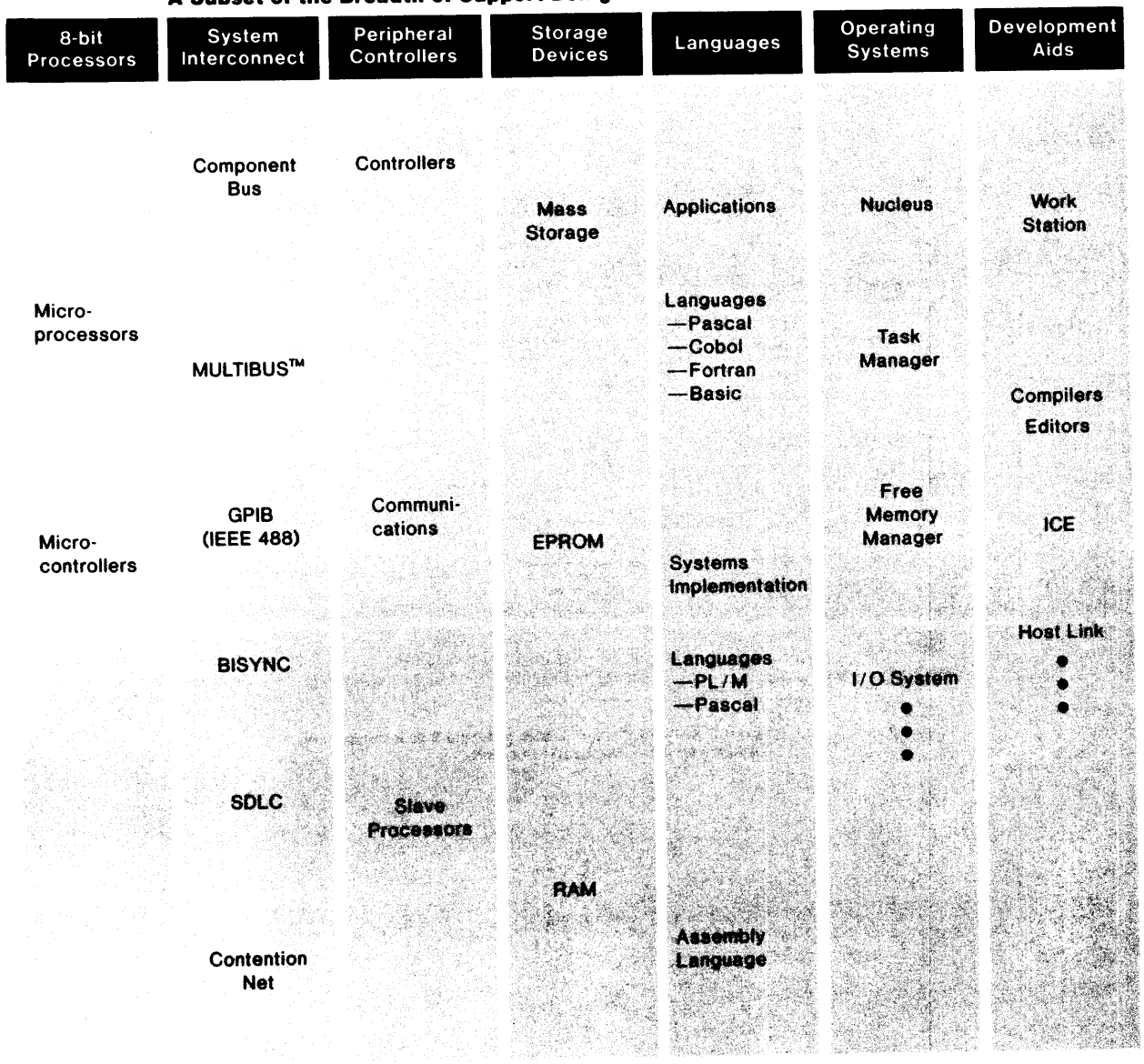
The iRMX 88 Real-Time Multitasking Executive is a small, performance-oriented executive system which can be used on Intel's 8 and 16-bit single board computers, iAPX 88 and iAPX 86-based boards. Based on the iRMX 80 Executive's field-proven and reliable architecture, the iRMX 88 Executive supports upgrading 8-bit based applications and helps the first-time microcomputer customer with simple, easy-to-use interfaces for both the iSBC and component-based applications. The iRMX 88 Executive Software provides the innermost, software control layer for the CPU that supports real-time application requirements for intertask communication, asynchronous I/O control, priority-based resource allocation, and standard iSBC disk controller interfaces.

The iRMX 88 Executive offers features that are suitable for reliable, performance-critical process control applications, production test stand units, sophisticated laboratory analysis, instrumentation, or specialized data acquisition and monitoring stations. Now, previous iRMX 80-based designs can upgrade to small size and high performance solution using the iRMX 88 Executive. The application will be easily tailored using the Interactive Configuration Utility.



Intel Provides a Complete Line of Single Board Computers, Add-On Modules and System Chassis

A Subset of the Breadth of Support Being Provided the 8-bit Product Line.



THE
8051
SINGLE CHIP
MICROCOMPUTER

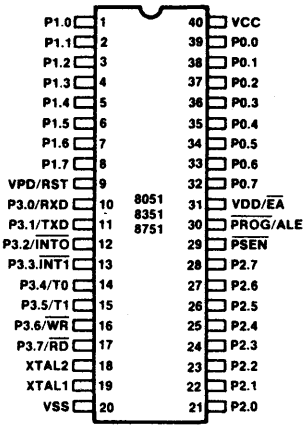


Figure 1a. 8051 Microcomputer Pinout Diagram

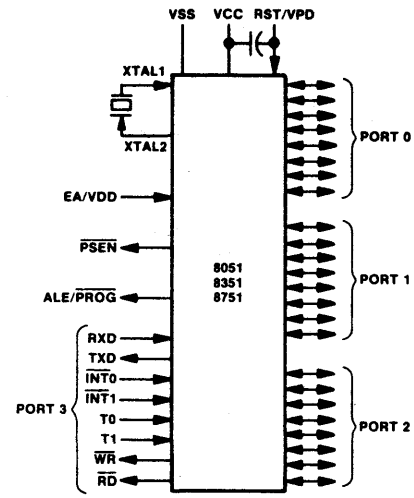


Figure 1b. 8051 Microcomputer Logic Symbol

1. INTRODUCTION

In 1976 Intel introduced the MCS-48™ family, consisting of the 8048, 8748, and 8035 microcomputers. These parts marked the first time a complete microcomputer system, including an eight-bit CPU, 1024 8-bit words of ROM or EPROM program memory, 64 words of data memory, I/O ports and an eight-bit timer/counter could be integrated onto a single silicon chip. Depending only on the program memory contents, one chip could control a limitless variety of products, ranging from appliances or automobile engines to text or data processing equipment. Follow-on products stretched the MCS-48™ architecture in several directions: the 8049 and 8039 doubled the amount of on-chip memory and ran 83% faster; the 8021 reduced costs by executing a subset of the 8048 instructions with a somewhat slower clock; and the 8022 put a unique two-channel 8-bit analog-to-digital converter on the same NMOS chip as the computer, letting the chip interface directly with analog transducers.

Now three new high-performance single-chip microcomputers—the Intel® 8051, 8751, and 8031—extend the advantages of Integrated Electronics to whole new product areas. Thanks to Intel's new HMOS technology, the MCS-51™ family provides four times the program memory and twice the data memory as the 8048 on a single chip. New I/O and peripheral capabilities both increase the range of applicability and reduce total system cost. Depending on the use, processing throughput increases by two and one-half to ten times.

This Application Note is intended to introduce the reader to the MCS-51™ architecture and features. While it does not assume intimacy with the MCS-48™ product line on the part of the reader, he/she should be familiar with some microprocessor (preferably Intel's, of course) or have a background in computer programming and digital logic.

Family Overview

Pinout diagrams for the 8051, 8751, and 8031 are shown in Figure 1. The devices include the following features:

- Single-supply 5 volt operation using HMOS technology.
- 4096 bytes program memory on-chip (not on 8031).
- 128 bytes data memory on-chip.
- Four register banks.
- 128 User-defined software flags.
- 64 Kilobytes each program and external RAM addressability.
- One microsecond instruction cycle with 12 MHz crystal.
- 32 bidirectional I/O lines organized as four 8-bit ports (16 lines on 8031).
- Multiple mode, high-speed programmable Serial Port.
- Two multiple mode, 16-bit Timer/Counters.
- Two-level prioritized interrupt structure.
- Full depth stack for subroutine return linkage and data storage.
- Augmented MCS-48™ instruction set.
- Direct Byte and Bit addressability.
- Binary or Decimal arithmetic.
- Signed-overflow detection and parity computation.
- Hardware Multiple and Divide in 4 μsec.
- Integrated Boolean Processor for control applications.
- Upwardly compatible with existing 8048 software.

AFN-01502A

All three devices come in a standard 40-pin Dual In-Line Package, with the same pin-out, the same timing, and the same electrical characteristics. The primary difference between the three is the on-chip program memory—different types are offered to satisfy differing user requirements.

The 8751 provides 4K bytes of ultraviolet-Erasable, Programmable Read Only Memory (EPROM) for program development, prototyping, and limited production runs. (By convention, 1K means $2^{10} = 1024$. 1k—with a lower case "k"—equals $10^3 = 1000$.) This part may be individually programmed for a specific application using Intel's Universal PROM Programmer (UPP). If software bugs are detected or design specifications change the same part may be "erased" in a matter of minutes by exposure to ultraviolet light and reprogrammed with the modified code. This cycle may be repeated indefinitely during the design and development phase.

The final version of the software must be programmed into a large number of production parts. The 8051 has 4K bytes of ROM which are mask-programmed with the customer's order when the chip is built. This part is considerably less expensive, but cannot be erased or altered after fabrication.

The 8031 does not have any program memory on-chip, but may be used with up to 64K bytes of external standard or multiplexed ROMs, PROMs, or EPROMs. The 8031 fits well in applications requiring significantly larger or smaller amounts of memory than the 4K bytes provided by its two siblings.

(The 8051 and 8751 automatically access external program memory for all addresses greater than the 4096 bytes on-chip. The External Access input is an override for all internal program memory—the 8051 and 8751 will each emulate an 8031 when pin 31 is low.)

Throughout this Note, "8051" is used as a generic term. Unless specifically stated otherwise, the point applies equally to all three components. Table 1 summarizes the quantitative differences between the members of the MCS-48™ and MCS-51™ families.

The remainder of this Note discusses the various MCS-51™ features and how they can be used. Software and/or hardware application examples illustrate many of the concepts. Several isolated tasks (rather than one complete system design example) are presented in the hope that some of them will apply to the reader's experiences or needs.

A document this short cannot detail all of a computer system's capabilities. By no means will all the 8051 instructions be demonstrated; the intent is to stress new or unique MCS-51™ operations and instructions generally used in conjunction with each other. For additional hardware information refer to the Intel **MCS-51™ Family User's Manual,** publication number 121517. The assembly language and use of ASM51, the MCS-51™ assembler, are further described in the **MCS-51™ Macro Assembler User's Guide,** publication number 9800937.

The next section reviews some of the basic concepts of microcomputer design and use. Readers familiar with the 8048 may wish to skim through this section or skip directly to the next, "ARCHITECTURE AND ORGANIZATION."

Microcomputer Background Concepts

Most digital computers use the binary (base 2) number system internally. All variables, constants, alphanumeric characters, program statements, etc., are represented by groups of binary digits ("bits"), each of which has the value 0 or 1. Computers are classified by how many bits they can move or process at a time.

The MCS-51™ microcomputers contain an eight-bit central processing unit (CPU). Most operations process variables eight bits wide. All internal RAM and ROM, and virtually all other registers are also eight bits wide. An eight-bit ("byte") variable (shown in Figure 2) may assume one of $2^8 = 256$ distinct values, which usually represent integers between 0 and 255. Other types of numbers, instructions, and so forth are represented by one or more bytes using certain conventions.

For example, to represent positive and negative values, the most significant bit (D7) indicates the sign of the other seven bits—0 if positive, 1 if negative—allowing integer variables between -128 and +127. For integers with extremely large magnitudes, several bytes are manipulated together as "multiple precision" signed or unsigned integers—16, 24, or more bits wide.

Table 1. Features of Intel's Single-Chip Microcomputers

EPROM Program Memory	ROM Program Memory	External Program Memory	Program Memory (Int/Max)	Data Memory (Bytes)	Instr. Cycle Time	Input/ Output Pins	Interrupt Sources	Reg. Banks
—	8021	—	1K/1K	64	8.4 μSec	21	0	1
—	8022	—	2K/2K	64	8.4 μSec	28	2	1
8748	8048	8035	1K/4K	64	2.5 μSec	27	2	2
—	8049	8039	2K/4K	128	1.36 μSec	27	2	2
8751	8051	8031	4K/64K	128	1.0 μSec	32	5	4

AFN-01502A

The letters "MCS" have traditionally indicated a system or family of compatible Intel® microcomputer components, including CPUs, memories, clock generators, I/O expanders, and so forth. The numerical suffix indicates the microprocessor or microcomputer which serves as the cornerstone of the family. Microcomputers in the MCS-48™ family currently include the 8048-series (8035, 8048, & 8748), the 8049-series (8039 & 8049), and the 8021 and 8022; the family also includes the 8243, an I/O expander compatible with each of the microcomputers. Each computer's CPU is derived from the 8048, with essentially the same architecture, addressing modes, and instruction set, and a single assembler (ASM48) serves each.

The first members of the MCS-51™ family are the 8051, 8751, and 8031. The architecture of the 8051-series, while derived from the 8048, is not strictly compatible; there are more addressing modes, more instructions, larger address spaces, and a few other hardware differences. In this Application Note the letters "MCS-51" are used when referring to *architectural* features of the 8051-series—features which would be included on possible future microcomputers based on the 8051 CPU. Such products could have different amounts of memory (as in the 8048/8049) or different peripheral functions (as in the 8021 and 8022) while leaving the CPU and instruction set intact. ASM51 is the assembler used by all microcomputers in the 8051 family.

Two digit decimal numbers may be "packed" in an eight-bit value, using four bits for the binary code of each digit. This is called Binary-Coded Decimal (BCD) representation, and is often used internally in programs which interact heavily with human beings.

Alphanumeric characters (letters, numbers, punctuation marks, etc.) are often represented using the American Standard Code for Information Interchange (ASCII) convention. Each character is associated with a unique seven-bit binary number. Thus one byte may represent a single character, and a word or sequence of letters may be represented by a series (or "string") of bytes. Since the ASCII code only uses 128 characters, the most significant bit of the byte is not needed to distinguish between characters. Often D7 is set to 0 for all characters. In some coding schemes, D7 is used to indicate the "parity" of the other seven bits—set or cleared as necessary to ensure that the total number of "1" bits in the eight-bit code is even ("even parity") or odd ("odd parity"). The 8051 includes hardware to compute parity when it is needed.

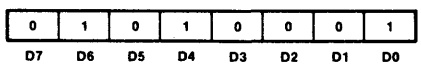


Figure 2. Representation of Bits Within an Eight-Bit "Byte" (Value shown = 01010001 Binary = 81 decimal).

A computer program consists of an ordered sequence of specific, simple steps to be executed by the CPU one-at-a-time. The method or sequence of steps used collectively to solve the user's application is called an "algorithm."

The program is stored inside the computer as a sequence of binary numbers, where each number corresponds to one of the basic operations ("opcodes") which the CPU is capable of executing. In the 8051, each program memory location is one byte. A complete instruction consists of a sequence of one or more bytes, where the first defines the operation to be executed and additional bytes (if needed) hold additional information, such as data values or variable addresses. No instruction is longer than three bytes.

The way in which binary opcodes and modifier bytes are assigned to the CPU's operations is called the computer's "machine language." Writing a program directly in machine language is time-consuming and tedious. Human beings think in words and concepts rather than encoded numbers, so each CPU operation and resource is given a name and standard abbreviation ("mnemonic"). Programs are more easily discussed using these standard mnemonics, or "assembly language," and may be typed into an Intel® Intellec® 800 or Series II® microcomputer development system in this form. The development system can mechanically translate the program from assembly language "source" form to machine language "object" code using a program called an "assembler." The MCS-51™ assembler is called ASM51.

There are several important differences between a computer's machine language and the assembly language used as a tool to represent it. The machine language or instruction set is the set of operations which the CPU can perform while a program is executing ("at run-time"), and is strictly determined by the microcomputer hardware design.

The assembly language is a standard (though more-or-less arbitrary) set of symbols including the instruction set mnemonics, but with additional features which further simplify the program design process. For example, ASM51 has controls for creating and formatting a program listing, and a number of directives for allocating variable storage and inserting arbitrary bytes of data into the object code for creating tables of constants.

AFN-01502A

In addition, ASM51 can perform sophisticated mathematical operations, computing addresses or evaluating arithmetic expressions to relieve the programmer from this drudgery. However, these calculations can only use information known at "assembly time."

For example, the 8051 performs arithmetic calculations at run-time, eight bits at a time. ASM51 can do similar operations 16 bits at a time. The 8051 can only do one simple step per instruction, while ASM51 can perform complex calculations in each line of source code. However, the operations performed by the assembler may only use parameter values fixed at assembly-time, not variables whose values are unknown until program execution begins.

For example, when the assembly language source line,

```
ADD     A,#(LOOP_COUNT + 1) * 3
```

is *assembled*, ASM51 will find the value of the previously-defined constant "LOOP_COUNT" in an internal symbol table, increment the value, multiply the sum by three, and (assuming it is between -256 and 255 inclusive) truncate the product to eight bits. When this instruction is *executed*, the 8051 ALU will just add that resulting constant to the accumulator.

Some similar differences exist to distinguish number system ("radix") specifications. The 8051 does all computations in binary (though there are provisions for then converting the result to decimal form). In the course of writing a program, though, it may be more convenient to specify constants using some other radix, such as base 10. On other occasions, it is desirable to specify the ASCII code for some character or string of characters without refering to tables. ASM51 allows several representations for constants, which are converted to binary as each instruction is assembled.

For example, binary numbers are represented in the assembly language by a series of ones and zeros (naturally), followed by the letter "B" (for Binary); octal numbers as a series of octal digits (0-7) followed by the letter "O" (for Octal) or "Q" (which doesn't stand for anything, but *looks* sort of like an "O" and is less likely to be confused with a zero).

Hexadecimal numbers are represented by a series of hexadecimal digits (0-9,A-F), followed by (you guessed it) the letter "H." A "hex" number must begin with a decimal digit; otherwise it would look like a user-defined symbol (to be discussed later). A "dummy" leading zero may be inserted before the first digit to meet this constraint. The character string "BACH" could be a legal label for a Baroque music synthesis routine; the string "0BACH" is the hexadecimal constant BAC_{16}. This is a case where adding 0 makes a big difference.

Decimal numbers are represented by a sequence of decimal digits, optionally followed by a "D." If a number has no suffix, it is assumed to be decimal—so it had better not contain any non-decimal digits. "0BAC" is not a legal representation for anything.

When an ASCII code is needed in a program, enclose the desired character between two apostrophes (as in '#') and the assembler will convert it to the appropriate code (in this case 23H). A string of characters between apostrophes is translated into a series of constants; 'BACH' becomes 42H, 41H, 43H, 48H.

These same conventions are used throughout the associated Intel documentation. Table 2 illustrates some of the different number formats.

2. ARCHITECTURE AND ORGANIZATION

Figure 3 blocks out the MCS-51™ internal organization. Each microcomputer combines a Central Processing Unit, two kinds of memory (data RAM plus program ROM or EPROM), Input/Output ports, and the mode,

Table 2. Notations Used to Represent Numbers

Bit Pattern	Binary	Octal	Hexa-Decimal	Decimal	Signed Decimal
0 0 0 0 0 0 0 0	0B	0Q	00H	0	0
0 0 0 0 0 0 0 1	1B	1Q	01H	1	+1
.....	..	...	...	..	....
0 0 0 0 0 1 1 1	111B	7Q	07H	7	+7
0 0 0 0 1 0 0 0	1000B	10Q	08H	8	+8
0 0 0 0 1 0 0 1	1001B	11Q	09H	9	+9
0 0 0 0 1 0 1 0	1010B	12Q	0AH	10	+10
.....	..	...	...	..	....
0 0 0 0 1 1 1 1	1111B	17Q	0FH	15	+15
0 0 0 1 0 0 0 0	10000B	20Q	10H	16	+16
.....	..	...	...	..	....
0 1 1 1 1 1 1 1	1111111B	177Q	7FH	127	+127
1 0 0 0 0 0 0 0	10000000B	200Q	80H	128	-128
1 0 0 0 0 0 0 1	10000001B	201Q	81H	129	-127
.....	.....	...	...	...	....
1 1 1 1 1 1 1 0	11111110B	376Q	0FEH	254	-2
1 1 1 1 1 1 1 1	11111111B	377Q	0FFH	255	-1

AFN-01502A

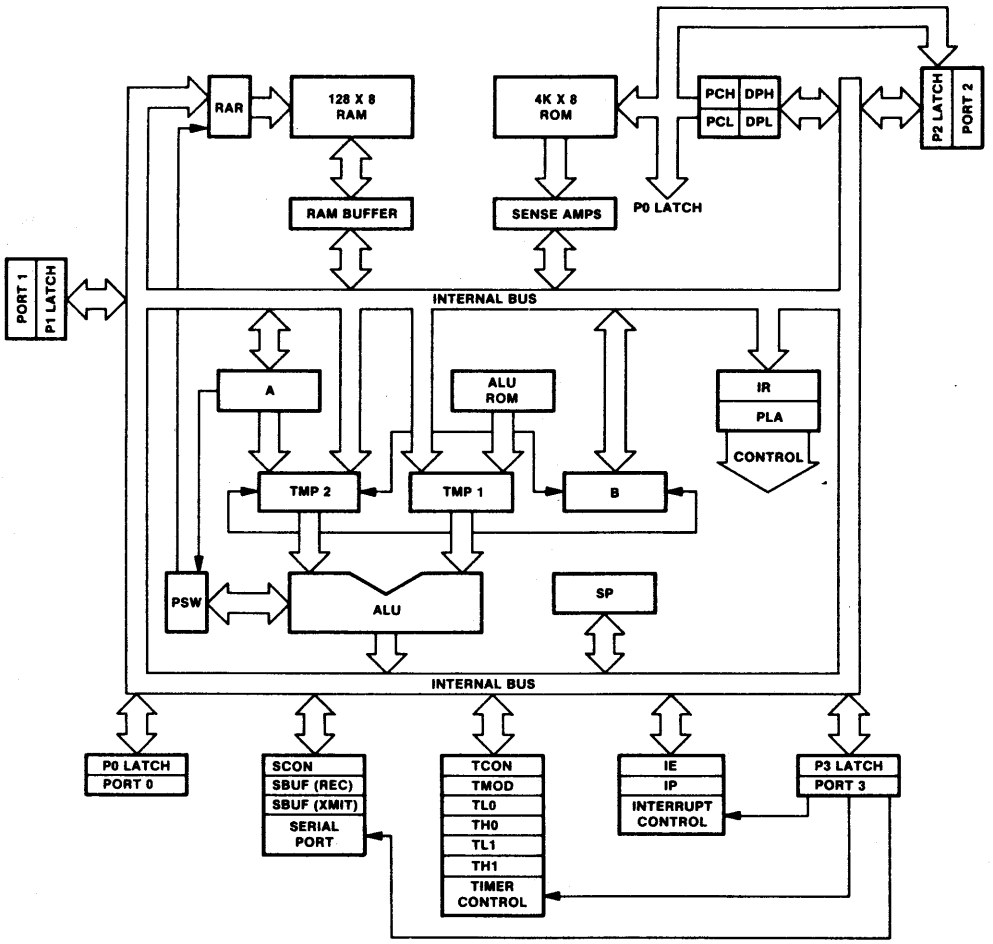


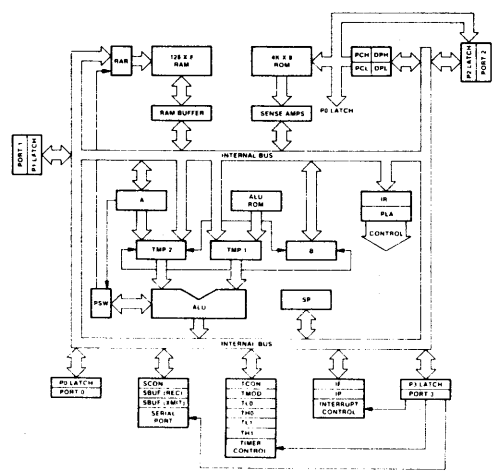
Figure 3. Block Diagram of 8051 Internal Structure

status, and data registers and random logic needed for a variety of peripheral functions. These elements communicate through an eight-bit data bus which runs throughout the chip, somewhat akin to indoor plumbing. This bus is buffered to the outside world through an I/O port when memory or I/O expansion is desired.

Let's summarize what each block does; later chapters dig into the CPU's instruction set and the peripheral registers in much greater detail.

Central Processing Unit

The CPU is the "brains" of the microcomputer, reading the user's program and executing the instructions stored therein. Its primary elements are an eight-bit Arithmetic/Logic Unit with associated registers A, B, PSW, and SP, and the sixteen-bit Program Counter and "Data Pointer" registers.



AFN-01502A

Arithmetic Logic Unit

The ALU can perform (as the name implies) arithmetic and logic functions on eight-bit variables. The former include basic addition, subtraction, multiplication, and division; the latter include the logical operations AND, OR, and Exclusive-OR, as well as rotate, clear, complement, and so forth. The ALU also makes conditional branching decisions, and provides data paths and temporary registers used for data transfers within the system. Other instructions are built up from these primitive functions: the addition capability can increment registers or automatically compute program destination addresses; subtraction is also used in decrementing or comparing the magnitude of two variables.

These primitive operations are automatically cascaded and combined with dedicated logic to build complex instructions such as incrementing a sixteen-bit register pair. To execute one form of the compare instruction, for example, the 8051 increments the program counter three times, reads three bytes of program memory, computes a register address with logical operations, reads internal data memory twice, makes an arithmetic comparison of two variables, computes a sixteen-bit destination address, and decides whether or not to make a branch—all in two microseconds!

An important and unique feature of the MCS-51 architecture is that the ALU can also manipulate one-bit as well as eight-bit data types. Individual bits may be set, cleared, or complemented, moved, tested, and used in logic computations. While support for a more primitive data type may initially seem a step backwards in an era of increasing word length, it makes the 8051 especially well suited for controller-type applications. Such algorithms *inherently* involve Boolean (true/false) input and output variables, which were heretofore difficult to implement with standard microprocessors. These features are collectively referred to as the MCS-51™ "Boolean Processor," and are described in the so-named chapter to come.

Thanks to this powerful ALU, the 8051 instruction set fares well at both real-time control and data intensive algorithms. A total of 51 separate operations move and manipulate three data types: Boolean (1-bit), byte (8-bit), and address (16-bit). All told, there are eleven addressing modes—seven for data, four for program sequence control (though only eight are used by more than just a few specialized instructions). Most operations allow several addressing modes, bringing the total number of instructions (operation/addressing mode combinations) to 111, encompassing 255 of the 256 possible eight-bit instruction opcodes.

Instruction Set Overview

Table 4 lists these 111 instructions classified into five groups:

- Arithmetic Operations
- Logical Operations for Byte Variables
- Data Transfer Instructions
- Boolean Variable Manipulation
- Program Branching and Machine Control

MCS-48™ programmers perusing Table 4 will notice the absence of special categories for Input/Output, Timer/Counter, or Control instructions. These functions are all still provided (and indeed many new functions are added), but as special cases of more generalized operations in other categories. To explicitly list all the useful instructions involving I/O and peripheral registers would require a table approximately four times as long.

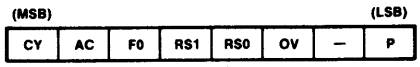
Observant readers will also notice that all of the 8048's page-oriented instructions (conditional jumps, JMPP, MOVP, MOVP3) have been replaced with corresponding but non-paged instructions. The 8051 instruction set is entirely *non*-page-oriented. The MCS-48™ "MOVP" instruction replacement and all conditional jump instructions operate relative to the program counter, with the actual jump address computed by the CPU during instruction execution. The "MOVP3" and "JMPP" replacements are now made relative to another sixteen-bit register, which allows the effective destination to be anywhere in the program memory space, regardless of where the instruction itself is located. There are even three-byte jump and call instructions allowing the destination to be *anywhere* in the 64K program address space.

The instruction set is designed to make programs efficient both in terms of code size and execution speed. No instruction requires more than three bytes of program memory, with the majority requiring only one or two bytes. Virtually all instructions execute in either one or two instruction cycles—one or two microseconds with a 12-MHz crystal—with the sole exceptions (multiply and divide) completing in four cycles.

Many instructions such as arithmetic and logical functions or program control, provide both a short and a long form for the same operation, allowing the programmer to optimize the code produced for a specific application. The 8051 usually fetches two instruction bytes per instruction cycle, so using a shorter form can lead to faster execution as well.

For example, any byte of RAM may be loaded with a constant with a three-byte, two-cycle instruction, but the commonly used "working registers" in RAM may be initialized in one cycle with a two-byte form. Any bit anywhere on the chip may be set, cleared, or complemented by a single three-byte logical instruction using two cycles. But critical control bits, I/O pins, and software flags may be controlled by two-byte, single cycle instructions. While three-byte jumps and calls can "go anywhere" in program memory, nearby sections of code may be reached by shorter relative or absolute versions.

AFN-01502A



Symbol	Position	Name and Significance
CY	PSW.7	Carry flag. Set/cleared by hardware or software during certain arithmetic and logical instructions.
AC	PSW.6	Auxiliary Carry flag. Set/cleared by hardware during addition or subtraction instructions to indicate carry or borrow out of bit 3.
F0	PSW.5	Flag 0 Set/cleared/tested by software as a user-defined status flag.
RS1	PSW.4	Register bank Select control bits 1 & 0. Set/cleared by software to determine
RS	PSW.3	working register bank (see Note).
OV	PSW.2	Overflow flag. Set/cleared by hardware during arithmetic instructions to indicate overflow conditions.
—	PSW.1	(reserved)
P	PSW.0	Parity flag. Set/cleared by hardware each instruction cycle to indicate an odd/even number of "one" bits in the accumulator, i.e., even parity.

Note— the contents of (RS1, RS0) enable the working register banks as follows:

(0,0)—Bank 0	(00H-07H)
(0,1)—Bank 1	(08H-0FH)
(1,0)—Bank 2	(10H-17H)
(1,1)—Bank 3	(18H-1FH)

Figure 4. PSW—Program Status Word Organization

A significant side benefit of an instruction set more powerful than those of previous single-chip microcomputers is that it is easier to generate applications-oriented software. Generalized addressing modes for byte and bit instructions reduce the number of source code lines written and debugged for a given application. This leads in turn to proportionately lower software costs, greater reliability, and faster design cycles.

Accumulator and PSW

The 8051, like its 8048 predecessor, is primarily an accumulator-based architecture: an eight-bit register called the accumulator ("A") holds a source operand and receives the result of the arithmetic instructions (addition, subtraction, multiplication, and division). The accumulator can be the source or destination for logical operations and a number of special data movement instructions, including table look-ups and external RAM expansion. Several functions apply exclusively to the accumulator: rotates, parity computation, testing for zero, and so on.

Many instructions implicitly or explicitly affect (or are affected by) several status flags, which are grouped together to form the Program Status Word shown in Figure 4.

(The period within entries under the Position column is called the "dot operator," and indicates a particular bit position within an eight-bit byte. "PSW.5" specifies bit 5 of the PSW. Both the documentation and ASM51 use this notation.)

The most "active" status bit is called the carry flag (abbreviated "C"). This bit makes possible multiple precision arithmetic operations including addition, subtraction, and rotates. The carry also serves as a "Boolean accumulator" for one-bit logical operations and bit manipulation instructions. The overflow flag (OV) detects when arithmetic overflow occurs on signed integer operands, making two's complement arithmetic possible. The parity flag (P) is updated after every instruction cycle with the even-parity of the accumulator contents.

The CPU does not control the two register-bank select bits, RS1 and RS0. Rather, they are manipulated by software to enable one of the four register banks. The usage of the PSW flags is demonstrated in the Instruction Set chapter of this Note.

Even though the architecture is accumulator-based, provisions have been made to bypass the accumulator in common instruction situations. Data may be moved from any location on-chip to any register, address, or indirect address (and vice versa), any register may be loaded with a constant, etc., all without affecting the accumulator. Logical operations may be performed against registers or variables to alter fields of bits—without using or affecting the accumulator. Variables may be incremented, decremented, or tested without using the accumulator. Flags and control bits may be manipulated and tested without affecting anything else.

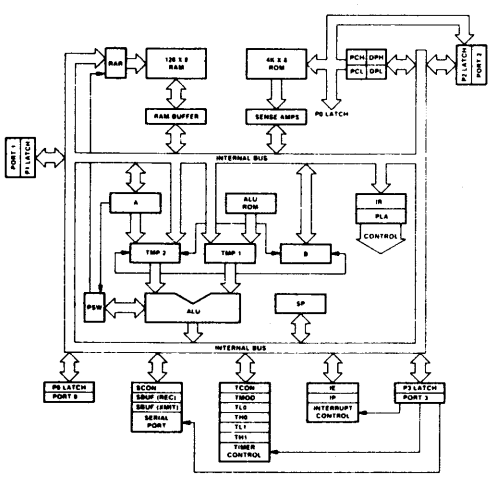
Other CPU Registers

A special eight-bit register ("B") serves in the execution of the multiply and divide instructions. This register is used in conjunction with the accumulator as the second input operand and to return eight-bits of the result.

The MCS-51 family processors include a hardware stack within internal RAM, useful for subroutine linkage,

passing parameters between routines, temporary variable storage, or saving status during interrupt service routines. The Stack Pointer (SP) is an eight-bit pointer register which indicates the address of the last byte pushed onto the stack. The stack pointer is automatically incremented or decremented on all push or pop instructions and all subroutine calls and returns. In theory, the stack in the 8051 may be up to a full 128 bytes deep. (In practice, even simple programs would use a handful of RAM locations for pointers, variables, and so forth—reducing the stack depth by that number.) The stack pointer defaults to 7 on reset, so that the stack will start growing up from location 8, just like in the 8048. By altering the pointer contents the stack may be relocated anywhere within internal RAM.

Finally, a 16-bit register called the data pointer (DPTR) serves as a base register in indirect jumps, table look-up instructions, and external data transfers. The high- and low-order halves of the data pointer may be manipulated as separate registers (DPH and DPL, respectively) or together using special instructions to load or increment all sixteen bits. Unlike the 8048, look-up tables can therefore start anywhere in program memory and be of arbitrary length.



Memory Spaces

Program memory is separate and distinct from data memory. Each memory type has a different addressing mechanism, different control signals, and a different function.

The program memory array (ROM or EPROM), like an elephant, is extremely large and never forgets information, even when power is removed. Program memory is used for information needed each time power is applied: initialization values, calibration constants, keyboard layout tables, etc., as well as the program itself. The program memory has a sixteen-bit address bus; its elements are addressed using the Program Counter or instructions which generate a sixteen-bit address.

To stretch our analogy just a bit, data memory is like a mouse: it is smaller and therefore quicker than program memory, and it goes into a random state when electrical power is applied. On-chip data RAM is used for variables which are determined or may change while the program is running.

A computer spends most of its time manipulating variables, not constants, and a relatively small number of variables at that. Since eight-bits is more than sufficient to uniquely address 128 RAM locations, the on-chip RAM address register is only one byte wide. In contrast to the program memory, data memory accesses need a single eight-bit value—a constant or another variable—to specify a unique location. Since this is the basic width of the ALU and the different memory types, those resources can be used by the addressing mechanisms, contributing greatly to the computer's operating efficiency.

The partitioning of program and data memory is extended to off-chip memory expansion. Each may be added independently, and each uses the same address and data busses, but with different control signals. External program memory is gated onto the external data bus by the $\overline{\text{PSEN}}$ (Program Store Enable) control output, pin 29. External data memory is read onto the bus by the $\overline{\text{RD}}$ output, pin 17, and written with data supplied from the microcomputer by the $\overline{\text{WR}}$ output, pin 16. (There is no control pin to write external program ROM, which is by definition Read Only.) While both types may be expanded to up to 64K bytes, the external data memory may optionally be expanded in 256 byte "pages" to preserve the use of P2 as an I/O port. This is useful with a relatively small expansion RAM (such as the Intel® 8155) or for addressing external peripherals.

Single-chip controller programs are finalized during the project design cycle, and are not modified after production. Intel's single-chip microcomputers are not "von Neumann" architectures common among main-frame and mini-computer systems: the MCS-51™ processor *data* memory—on-chip and external—may *not* be used for program code. Just as there is no write-control signal for program memory, there is no way for the CPU to execute instructions out of RAM. In return, this concession allows an architecture optimized for efficient controller applications: a large, fixed program located in ROM, a hundred or so variables in RAM, and different methods for efficiently addressing each.

(Von Neumann machines are helpful for software development and debug. An 8051 system could be modified to have a single off-chip memory space by gating together the two memory-read controls ($\overline{\text{PSEN}}$ and $\overline{\text{RD}}$) with a two-input AND gate (Figure 5). The CPU could then write data into the common memory array using $\overline{\text{WR}}$ and

AFN-01502A

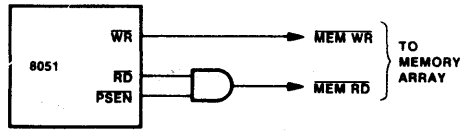


Figure 5. Combining External Program and Data Memory Arrays

external data transfer instructions, and read instructions or data with the AND gate output and data transfer or program memory look-up instructions.)

In addition to the memory arrays, there is (yet) another (albeit sparsely populated) physical address space. Connected to the internal data bus are a score of special-purpose eight-bit registers scattered throughout the chip. Some of these—B, SP, PSW, DPH, and DPL—have been discussed above. Others—I/O ports and peripheral function registers—will be introduced in the following sections. Collectively, these registers are designated as the "special-function register" address space. Even the accumulator is assigned a spot in the special-function register address space for additional flexibility and uniformity.

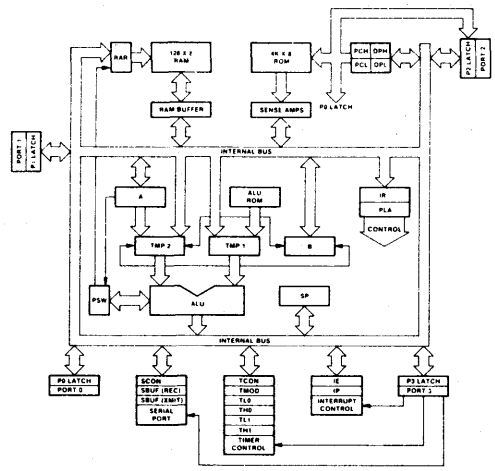
Thus, the MCS-51™ architecture supports several distinct "physical" address spaces, functionally separated at the hardware level by different addressing mechanisms, read and write control signals, or both:

- On-chip program memory;
- On-chip data memory;
- Off-chip program memory;
- Off-chip data memory;
- On-chip special-function registers.

What the *programmer sees,* though, are "logical" address spaces. For example, as far as the programmer is concerned, there is only one type of program memory, 64K bytes in length. The fact that it is formed by combining on- and off-chip arrays (split 4K/60K on the 8051 and 8751) is "invisible" to the programmer; the CPU automatically fetches each byte from the appropriate array, based on its address.

(Presumably, future microcomputers based on the MCS-51™ architecture may have a different physical split, with more or less of the 64K total implemented on-chip. Using the MCS-48™ family as a precedent, the 8048's 4K potential program address space was split 1K/3K between on- and off-chip arrays; the 8049's was split 2K/2K.)

Why go into such tedious details about address spaces? The logical addressing modes are described in the Instruction Set chapter in terms of physical address spaces. Understanding their differences now will pay off in understanding and using the chips later.



Input/Output Ports

The MCS-51™ I/O port structure is extremely versatile. The 8051 and 8751 each have 32 I/O pins configured as four eight-bit parallel ports (P0, P1, P2, and P3). Each pin will input or output data (or both) under software control, and each may be referenced by a wide repertoire of byte and bit operations.

In various operating or expansion modes, some of these I/O pins are also used for special input or output functions. Instructions which access external memory use Port 0 as a multiplexed address/data bus: at the beginning of an external memory cycle eight bits of the address are output on P0; later data is transferred on the same eight pins. External data transfer instructions which supply a sixteen-bit address, and any instruction accessing external program memory, output the high-order eight bits on P2 during the access cycle. (The 8031 *always* uses the pins of P0 and P2 for external addressing, but P1 and P3 are available for standard I/O.)

The eight pins of Port 3 (P3) each have a special function. Two external interrupts, two counter inputs, two serial data lines, and two timing control strobes use pins of P3 as described in Figure 6. Port 3 pins corresponding to functions not used are available for conventional I/O.

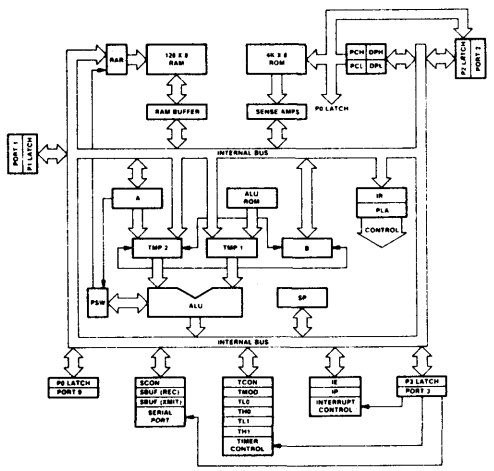
Even within a single port, I/O functions may be combined in many ways: input and output may be performed using different pins at the same time, or the same pins at different times; in parallel in some cases, and in serial in others; as test pins, or (in the case of Port 3) as additional special functions.

AFN-01502A

(MSB)							(LSB)
RD	WR	T1	T0	INT1	INT0	TXD	RXD

Symbol	Position	Name and Significance
RD	P3.7	Read data control output. Active low pulse generated by hardware when external data memory is read.
WR	P3.6	Write data control output. Active low pulse generated by hardware when external data memory is written.
T1	P3.5	Timer/counter 1 external input or test pin.
T0	P3.4	Timer/counter 0 external input or test pin.
INT1	P3.3	Interrupt 1 input pin. Low-level or falling-edge triggered.
INT0	P3.2	Interrupt 0 input pin. Low-level or falling-edge triggered.
TXD	P3.1	Transmit Data pin for serial port in UART mode. Clock output in shift register mode.
RXD	P3.0	Receive Data pin for serial port in UART mode. Data I/O pin in shift register mode.

Figure 6. P3—Alternate Special Functions of Port 3



Special Peripheral Functions

There are a few special needs common among control-oriented computer systems:

- keeping track of elapsed real-time;
- maintaining a count of signal transitions;
- measuring the precise width of input pulses;
- communicating with other systems or people;
- closely monitoring asynchronous external events.

Until now, microprocessor systems needed peripheral chips such as timer/counters, USARTs, or interrupt controllers to meet these needs. The 8051 integrates all of these capabilities on-chip!

Timer/Counters

There are two sixteen-bit multiple-mode Timer/Counters on the 8051, each consisting of a "High" byte (corresponding to the 8048 "T" register) and a low byte (similar to the 8048 prescaler, with the additional flexibility of being software-accessible). These registers are called, naturally enough, TH0, TL0, TH1, and TL1. Each pair may be independently software programmed to any of a dozen modes with a mode register designated TMOD (Figure 7), and controlled with register TCON (Figure 8).

The timer modes can be used to measure time intervals, determine pulse widths, or initiate events, with one-microsecond resolution, up to a maximum interval of 65,536 instruction cycles (over 65 milliseconds). Longer delays may easily be accumulated through software. Configured as a counter, the same hardware will accumulate external events at frequencies from D.C. to 500 KHz, with up to sixteen bits of precision.

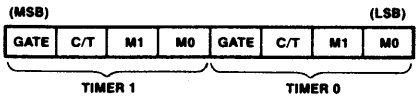
Serial Port Interface

Each microcomputer contains a high-speed, full-duplex, serial port which is software programmable to function in four basic modes: shift-register I/O expander, 8-bit UART, 9-bit UART, or interprocessor communications link. The UART modes will interface with standard I/O devices (e.g. CRTs, teletypewriters, or modems) at data rates from 122 baud to 31 kilobaud. Replacing the standard 12 MHz crystal with a 10.7 MHz crystal allows 110 baud. Even or odd parity (if desired) can be included with simple bit-handling software routines. Inter-processor communications in distributed systems takes place at 187 kilobaud with hardware for automatic address/data message recognition. Simple TTL or CMOS shift registers provide low-cost I/O expansion at a super-fast 1 Megabaud. The serial port operating modes are controlled by the contents of register SCON (Figure 9).

Interrupt Capability and Control

(Interrupt capability is generally considered a CPU function. It is being introduced here since, from an applications point of view, interrupts relate more closely to peripheral and system interfacing.)

AFN-01502A



GATE	Gating control. When set, Timer/counter "x" is enabled only while "INTx" pin is high and "TRx" control bit is set. When cleared, timer/counter is enabled whenever "TRx" control bit is set.
C/T	Timer or Counter Selector. Cleared for Timer operation (input from internal system clock). Set for Counter operation (input from "Tx" input pin).

M1	M0	Operating Mode
0	0	MCS-48 Timer. "TLx" serves as five-bit prescaler.
0	1	16-bit timer/counter. "THx" and "TLx" are cascaded; there is no prescaler.
1	0	8-bit auto-reload timer/counter. "THx" holds a value which is to be reloaded into "TLx" each time it overflows.
1	1	(Timer 0) TL0 is an eight-bit timer/counter controlled by the standard Timer 0 control bits. TH0 is an eight-bit timer only controlled by Timer 1 control bits.
1	1	(Timer 1) Timer/counter 1 stopped.

Figure 7. TMOD—Timer/Counter Mode Register

Symbol	Position	Name and Significance
TF1	TCON.7	Timer 1 overflow Flag. Set by hardware on timer/counter overflow. Cleared when interrupt processed.
TR1	TCON.6	Timer 1 Run control bit. Set/cleared by software to turn timer/counter on/off.
TF0	TCON.5	Timer 0 overflow Flag. Set by hardware on timer/counter overflow. Cleared when interrupt processed.
TR0	TCON.4	Timer 0 Run control bit. Set/cleared by software to turn timer/counter on/off.
IE1	TCON.3	Interrupt 1 Edge flag. Set by hardware when external interrupt edge detected. Cleared when interrupt processed.
IT1	TCON.2	Interrupt 1 Type control bit. Set/cleared by software to specify falling edge/low level triggered external interrupts.
IE0	TCON.1	Interrupt 0 Edge flag. Set by hardware when external interrupt edge detected. Cleared when interrupt processed.
IT0	TCON.0	Interrupt 0 Type control bit. Set/cleared by software to specify falling edge/low level triggered external interrupts.

Figure 8. TCON—Timer/Counter Control/Status Register

AFN-01502A

(MSB)							(LSB)
SM0	SM1	SM2	REN	TB8	RB8	TI	RI

Symbol	Position	Name and Significance
SM0	SCON.7	Serial port Mode control bit 0. Set/cleared by software (see note).
SM1	SCON.6	Serial port Mode control bit 1. Set/cleared by software (see note).
SM2	SCON.5	Serial port Mode control bit 2. Set by software to disable reception of frames for which bit 8 is zero.
REN	SCON.4	Receiver Enable control bit. Set/cleared by software to enable/disable serial data reception.
TB8	SCON.3	Transmit Bit 8. Set/cleared by hardware to determine state of ninth data bit transmitted in 9-bit UART mode.
RB8	SCON.2	Receive Bit 8. Set/cleared by hardware to indicate state of ninth data bit received.
TI	SCON.1	Transmit Interrupt flag. Set by hardware when byte transmitted. Cleared by software after servicing.
RI	SCON.0	Received Interrupt flag. Set by hardware when byte received. Cleared by software after servicing.

Note— the state of (SM0,SM1) selects:
(0,0)—Shift register I/O expansion.
(0,1)—8 bit UART, variable data rate.
(1,0)—9 bit UART, fixed data rate.
(1,1)—9 bit UART, variable data rate.

Figure 9. SCON—Serial Port Control/Status Register

These peripheral functions allow special hardware to monitor real-time signal interfacing without bothering the CPU. For example, imagine serial data is arriving from one CRT while being transmitted to another, and one timer/counter is tallying high-speed input transitions while the other measures input pulse widths. During all of this the CPU is thinking about something else.

But how does the CPU know when a reception, transmission, count, or pulse is finished? The 8051 programmer can choose from three approaches.

TCON and SCON contain status bits set by the hardware when a timer overflows or a serial port operation is completed. The first technique reads the control register into the accumulator, tests the appropriate bit, and does a conditional branch based on the result. This "polling" scheme (typically a three-instruction sequence though additional instructions to save and restore the accumulator may sometimes be needed) will surely be familiar to programmers used to multi-chip microcomputer systems and peripheral controller chips. This process is rather cumbersome, especially when monitoring multiple peripherals.

As a second approach, the 8051 can perform a conditional branch based on the state of any control or status bit or input pin in a single instruction; a four instruction sequence could poll the four simultaneous happenings mentioned above in just eight microseconds.

Unfortunately, the CPU must still drop what it's doing to test these bits. A manager cannot do his own work well if he is continuously monitoring his subordinates; they should interrupt him (or her) only when they need attention or guidance. So it is with machines: ideally, the CPU would not have to worry about the peripherals until they require servicing. At that time, it would postpone the background task long enough to handle the appropriate device, then return to the point where it left off.

This is the basis of the third and generally optimal solution, hardware interrupts. The 8051 has five interrupt sources: one from the serial port when a transmission or reception is complete, two from the timers when overflows occur, and two from input pins INT0 and INT1. Each source may be independently enabled or disabled to allow polling on some sources or at some times, and each may be classified as high or low priority. A high priority source can interrupt a low priority service routine; the manager's boss can interrupt conferences with subordinates. These options are selected by the interrupt enable and priority control registers, IE and IP (Figures 10 and 11).

Each source has a particular program memory address associated with it (Table 3), starting at 0003H (as in the 8048) and continuing at eight-byte intervals. When an event enabled for interrupts occurs the CPU automatically executes an internal subroutine call to the corresponding address. A user subroutine starting at this location (or jumped to from this location) then performs the instructions to service that particular source. After completing the interrupt service routine, execution returns to the background program.

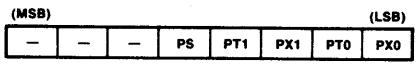
Table 3. 8051 Interrupt Sources and Service Vectors

Interrupt Source	Service Routine Starting Address
(Reset)	0000H
External 0	0003H
Timer/Counter 0	000BH
External 1	0013H
Timer/Counter 1	001BH
Serial Port	0023H

AFN-01502A

Symbol	Position	Name and Significance
EA	IE.7	Enable All control bit. Cleared by software to disable all interrupts, independent of the state of IE.4-IE.0.
—	IE.6	(reserved)
—	IE.5	(reserved)
ES	IE.4	Enable Serial port control bit. Set/cleared by software to enable/ disable interrupts from TI or RI flags.
ET1	IE.3	Enable Timer 1 control bit. Set/cleared by software to enable/disable interrupts from timer/counter 1.
EX1	IE.2	Enable External interrupt 1 control bit. Set/cleared by software to enable/ disable interrupts from INT1.
ET0	IE.1	Enable Timer 0 control bit. Set/cleared by software to enable/disable interrupts from timer/counter 0
EX0	IE.0	Enable External interrupt 0 control bit. Set/cleared by software to enable/ disable interrupts from INT0.

Figure 10. IE—Interrupt Enable Register



Symbol	Position	Name and Significance
—	IP.7	(reserved)
—	IP.6	(reserved)
—	IP.5	(reserved)
PS	IP.4	Serial port Priority control bit. Set/cleared by software to specify high/low priority interrupts for Serial port.
PT1	IP.3	Timer 1 Priority control bit. Set/cleared by software to specify high/low priority interrupts for timer/counter 1.
PX1	IP.2	External interrupt 1 Priority control bit. Set/cleared by software to specify high/low priority interrupts for INT1.
PT0	IP.1	Timer 0 Priority control bit. Set/cleared by software to specify high/low priority interrupts for timer/counter 0.
PX0	IP.0	External interrupt 0 Priority control bit. Set/cleared by software to specify high/low priority interrupts for INT0.

Figure 11. IP—Interrupt Priority Control Register

Table 4. MCS-51™ Instruction Set Description

ARITHMETIC OPERATIONS

Mnemonic		Description	Byte	Cyc
ADD	A,Rn	Add register to Accumulator	1	1
ADD	A,direct	Add direct byte to Accumulator	2	1
ADD	A,@Ri	Add indirect RAM to Accumulator	1	1
ADD	A,#data	Add immediate data to Accumulator	2	1
ADDC	A,Rn	Add register to Accumulator with Carry	1	1
ADDC	A,direct	Add direct byte to A with Carry flag	2	1
ADDC	A,@Ri	Add indirect RAM to A with Carry flag	1	1
ADDC	A,#data	Add immediate data to A with Carry flag	2	1
SUBB	A,Rn	Subtract register from A with Borrow	1	1
SUBB	A,direct	Subtract direct byte from A with Borrow	2	1
SUBB	A,@Ri	Subtract indirect RAM from A w/ Borrow	1	1
SUBB	A,#data	Subtract immed. data from A w/ Borrow	2	1
INC	A	Increment Accumulator	1	1
INC	Rn	Increment register	1	1
INC	direct	Increment direct byte	2	1
INC	@Ri	Increment indirect RAM	1	1
DEC	A	Decrement Accumulator	1	1
DEC	Rn	Decrement register	1	1
DEC	direct	Decrement direct byte	2	1
DEC	@Ri	Decrement indirect RAM	1	1
INC	DPTR	Increment Data Pointer	1	2
MUL	AB	Multiply A & B	1	4
DIV	AB	Divide A by B	1	4
DA	A	Decimal Adjust Accumulator	1	1

LOGICAL OPERATIONS

Mnemonic		Destination	Byte	Cyc
ANL	A,Rn	AND register to Accumulator	1	1
ANL	A,direct	AND direct byte to Accumulator	2	1
ANL	A,@Ri	AND indirect RAM to Accumulator	1	1
ANL	A,#data	AND immediate data to Accumulator	2	1
ANL	direct,A	AND Accumulator to direct byte	2	1
ANL	direct,#data	AND immediate data to direct byte	3	2
ORL	A,Rn	OR register to Accumulator	1	1
ORL	A,direct	OR direct byte to Accumulator	2	1
ORL	A,@Ri	OR indirect RAM to Accumulator	1	1
ORL	A,#data	OR immediate data to Accumulator	2	1
ORL	direct,A	OR Accumulator to direct byte	2	1
ORL	direct,#data	OR immediate data to direct byte	3	2
XRL	A,Rn	Exclusive-OR register to Accumulator	1	1
XRL	A,direct	Exclusive-OR direct byte to Accumulator	2	1
XRL	A,@Ri	Exclusive-OR indirect RAM to A	1	1
XRL	A,#data	Exclusive-OR immediate data to A	2	1
XRL	direct,A	Exclusive-OR Accumulator to direct byte	2	1
XRL	direct,#data	Exclusive-OR immediate data to direct	3	2
CLR	A	Clear Accumulator	1	1
CPL	A	Complement Accumulator	1	1
RL	A	Rotate Accumulator Left	1	1
RLC	A	Rotate A Left through the Carry flag	1	1
RR	A	Rotate Accumulator Right	1	1
RRC	A	Rotate A Right through Carry flag	1	1
SWAP	A	Swap nibbles within the Accumulator	1	1

DATA TRANSFER

Mnemonic		Description	Byte	Cyc
MOV	A,Rn	Move register to Accumulator	1	1
MOV	A,direct	Move direct byte to Accumulator	2	1
MOV	A,@Ri	Move indirect RAM to Accumulator	1	1
MOV	A,#data	Move immediate data to Accumulator	2	1
MOV	Rn,A	Move Accumulator to register	1	1
MOV	Rn,direct	Move direct byte to register	2	2
MOV	Rn,#data	Move immediate data to register	2	1
MOV	direct,A	Move Accumulator to direct byte	2	1
MOV	direct,Rn	Move register to direct byte	2	2
MOV	direct,direct	Move direct byte to direct	3	2
MOV	direct,@Ri	Move indirect RAM to direct byte	2	2
MOV	direct,#data	Move immediate data to direct byte	3	2
MOV	@Ri,A	Move Accumulator to indirect RAM	1	1
MOV	@Ri,direct	Move direct byte to indirect RAM	2	2
MOV	@Ri,#data	Move immediate data to indirect RAM	2	1
MOV	DPTR,#data16	Load Data Pointer with a 16-bit constant	3	2

DATA TRANSFER (cont.)

Mnemonic		Description	Byte	Cyc
MOVC	A,@A+DPTR	Move Code byte relative to DPTR to A	1	2
MOVC	A,@A+PC	Move Code byte relative to PC to A	1	2
MOVX	A,@Ri	Move External RAM (8-bit addr) to A	1	2
MOVX	A,@DPTR	Move External RAM (16-bit addr) to A	1	2
MOVX	@Ri,A	Move A to External RAM (8-bit addr)	1	2
MOVX	@DPTR,A	Move A to External RAM (16-bit addr)	1	2
PUSH	direct	Push direct byte onto stack	2	2
POP	direct	Pop direct byte from stack	2	2
XCH	A,Rn	Exchange register with Accumulator	1	1
XCH	A,direct	Exchange direct byte with Accumulator	2	1
XCH	A,@Ri	Exchange indirect RAM with A	1	1
XCHD	A,@Ri	Exchange low-order Digit ind. RAM w/A	1	1

BOOLEAN VARIABLE MANIPULATION

Mnemonic		Description	Byte	Cyc
CLR	C	Clear Carry flag	1	1
CLR	bit	Clear direct bit	2	1
SETB	C	Set Carry flag	1	1
SETB	bit	Set direct Bit	2	1
CPL	C	Complement Carry flag	1	1
CPL	bit	Complement direct bit	2	1
ANL	C,bit	AND direct bit to Carry flag	2	2
ANL	C,/bit	AND complement of direct bit to Carry	2	2
ORL	C,bit	OR direct bit to Carry flag	2	2
ORL	C,/bit	OR complement of direct bit to Carry	2	2
MOV	C,bit	Move direct bit to Carry flag	2	1
MOV	bit,C	Move Carry flag to direct bit	2	2

PROGRAM AND MACHINE CONTROL

Mnemonic		Description	Byte	Cyc
ACALL	addr11	Absolute Subroutine Call	2	2
LCALL	addr16	Long Subroutine Call	3	2
RET		Return from subroutine	1	2
RETI		Return from interrupt	1	2
AJMP	addr11	Absolute Jump	2	2
LJMP	addr16	Long Jump	3	2
SJMP	rel	Short Jump (relative addr)	2	2
JMP	@A+DPTR	Jump indirect relative to the DPTR	1	2
JZ	rel	Jump if Accumulator is Zero	2	2
JNZ	rel	Jump if Accumulator is Not Zero	2	2
JC	rel	Jump if Carry flag is set	2	2
JNC	rel	Jump if No Carry flag	2	2
JB	bit,rel	Jump if direct Bit set	3	2
JNB	bit,rel	Jump if direct Bit Not set	3	2
JBC	bit,rel	Jump if direct Bit is set & Clear bit	3	2
CJNE	A,direct,rel	Compare direct to A & Jump if Not Equal	3	2
CJNE	A,#data,rel	Comp. immed. to A & Jump if Not Equal	3	2
CJNE	Rn,#data,rel	Comp. immed. to reg. & Jump if Not Equal	3	2
CJNE	@Ri,#data,rel	Comp. immed. to ind. & Jump if Not Equal	3	2
DJNZ	Rn,rel	Decrement register & Jump if Not Zero	2	2
DJNZ	direct,rel	Decrement direct & Jump if Not Zero	3	2
NOP		No operation	1	1

Notes on data addressing modes:

Rn —Working register R0-R7
direct —128 internal RAM locations, any I/O port, control or status register
@Ri —Indirect internal RAM location addressed by register R0 or R1
#data —8-bit constant included in instruction
#data16 —16-bit constant included as bytes 2 & 3 of instruction
bit —128 software flags, any I/O pin, control or status bit

Notes on program addressing modes:

addr16 —Destination address for LCALL & LJMP may be anywhere within the 64-Kilobyte program memory address space.
addr11 —Destination address for ACALL & AJMP will be within the same 2-Kilobyte page of program memory as the first byte of the following instruction.
rel —SJMP and all conditional jumps include an 8-bit offset byte. Range is +127 -128 bytes relative to first byte of the following instruction.

3. INSTRUCTION SET AND ADDRESSING MODES

The 8051 instruction set is extremely regular, in the sense that most instructions can operate with variables from several different physical or logical address spaces. Before getting deeply enmeshed in the instruction set proper, it is important to understand the details of the most common data addressing modes. Whereas Table 4 summarizes the instructions set broken down by functional group, this chapter starts with the addressing mode classes and builds to include the related instructions.

Data Addressing Modes

MCS-51 assembly language instructions consist of an operation mnemonic and zero to three operands separated by commas. In two operand instructions the destination is specified first, then the source. Many byte-wide data

operations (such as ADD or MOV) inherently use the accumulator as a source operand and/or to receive the result. For the sake of clarity the letter "A" is specified in the source or destination field in all such instructions. For example, the instruction,

ADD A,<source>

will add the variable<source>to the accumulator, leaving the sum in the accumulator.

The operand designated "<source>" above may use any of four common logical addressing modes:

- Register—one of the working registers in the currently enabled bank.
- Direct—an internal RAM location, I/O port, or special-function register.
- Register-indirect—an internal RAM location, pointed to by a working register.
- Immediate data—an eight-bit constant incorporated into the instruction.

The first three modes provide access to the internal RAM and Hardware Register address spaces, and may therefore be used as source or destination operands; the last mode accesses program memory and may be a source operand only.

(It is hard to show a "typical application" of any instruction without involving instructions not yet described. The following descriptions use only the self-explanatory ADD and MOV instructions to demonstrate how the four addressing modes are specified and used. Subsequent examples will become increasingly complex.)

Register Addressing

The 8051 programmer has access to eight "working registers," numbered R0-R7. The least-significant three-bits of the instruction opcode indicate one register within this logical address space. Thus, a function code and operand address can be combined to form a short (one byte) instruction (Figure 12.a).

The 8051 assembly language indicates register addressing with the symbol Rn (where n is from 0 to 7) or with a symbolic name previously defined as a register by the EQUate or SET directives. (For more information on assembler directives see the Macro Assembler Reference Manual.)

Example 1—Adding Two Registers Together

```
;REGADR ADD CONTENTS OF REGISTER 1
;       TO CONTENTS OF REGISTER 0
;
REGADR: MOV     A,R0
        ADD     A,R1
        MOV     R0,A
```

There are four such banks of working registers, only one of which is active at a time. Physically, they occupy the first 32 bytes of on-chip data RAM (addresses 0-1FH). PSW bits 4 and 3 determine which bank is active. A hardware reset enables register bank 0; to select a different bank the programmer modifies PSW bits 4 and 3 accordingly.

Example 2—Selecting Alternate Memory Banks

```
        MOV     PSW.#00010000B  SELECT BANK 2
```

Register addressing in the 8051 is the same as in the 8048 family, with two enhancements: there are four banks rather than one or two, and 16 instructions (rather than 12) can access them.

Direct Byte Addressing

Direct addressing can access any on-chip variable or hardware register. An additional byte appended to the opcode specifies the location to be used (Figure 12.b).

Depending on the highest order bit of the direct address byte, one of two physical memory spaces is selected. When the direct address is between 0 and 127 (00H-7FH) one of the 128 low-order on-chip RAM locations is used. (Future microcomputers based on the MCS-51™ architecture may incorporate more than 128 bytes of on-chip RAM. Even if this is the case, only the low-order 128 bytes will be directly addressable. The remainder would be accessed indirectly or via the stack pointer.)

Example 3—Adding RAM Location Contents

```
;DIRADR ADD CONTENTS OF RAM LOCATION 41H
;       TO CONTENTS OF RAM LOCATION 40H
;
DIRADR: MOV     A,40H
        ADD     A,41H
        MOV     40H,A
```

All I/O ports and special function, control, or status registers are assigned addresses between 128 and 255 (80H-0FFH). When the direct address byte is between these limits the corresponding hardware register is accessed. For example, Ports 0 and 1 are assigned direct addresses 80H and 90H, respectively. A complete list is presented in Table 5. Don't waste your time trying to memorize the addresses in Table 5. Since programs using absolute addresses for function registers would be difficult to write or understand, ASM51 allows and understands the abbreviations listed instead.

Example 4—Adding Input Port Data to Output Port Data

```
;PRTADR ADD DATA INPUT ON PORT 1
;       TO DATA PREVIOUSLY OUTPUT
;       ON PORT 0
;
PRTADR: MOV     A,P0
        ADD     A,P1
        MOV     P0,A
```

Direct addressing allows all special-function registers in the 8051 to be read, written, or used as instruction operands. In general, this is the *only* method used for accessing I/O ports and special-function registers. If direct addressing is used with special-function register addresses other than those listed, the result of the instruction is undefined.

AFN-01502A

The 8048 does not have or need any generalized direct addressing mode, since there are only five special registers (BUS, P1, P2, PSW, & T) rather than twenty. Instead, 16 special 8048 opcodes control output bits or read or write each register to the accumulator. These functions are all subsumed by four of the 27 direct addressing instructions of the 8051.

Table 5. 8051 Hardware Register Direct Addresses

Register	Address	Function
P0	80H*	Port 0
SP	81H	Stack Pointer
DPL	82H	Data Pointer (Low)
DPH	83H	Data Pointer (High)
TCON	88H*	Timer register
TMOD	89H	Timer Mode register
TL0	8AH	Timer 0 Low byte
TL1	8BH	Timer 1 Low byte
TH0	8CH	Timer 0 High byte
TH1	8DH	Timer 1 High byte
P1	90H*	Port 1
SCON	98H*	Serial Port Control register
SBUF	99H	Serial Port data Buffer
P2	0A0H*	Port 2
IE	0A8H*	Interrupt Enable register
P3	0B0H*	Port 3
IP	0B8H*	Interrupt Priority register
PSW	0D0H*	Program Status Word
ACC	0E0H*	Accumulator (direct address)
B	0F0H*	B register

*= bit addressable register.

Register-Indirect Addressing

How can you handle variables whose locations in RAM are determined, computed, or modified while the program is running? This situation arises when manipulating sequential memory locations, indexed entries within tables in RAM, and multiple precision or string operations. Register or Direct addressing cannot be used, since their operand addresses are fixed at assembly time.

The 8051 solution is "register-indirect RAM addressing." R0 and R1 of each register bank may operate as index or pointer registers, their contents indicating an address into RAM. The internal RAM location so addressed is the actual operand used. The least significant bit of the instruction opcode determines which register is used as the "pointer" (Figure 12.c).

In the 8051 assembly language, register-indirect addressing is represented by a commercial "at" sign ("@") preceding R0, R1, or a symbol defined by the user to be equal to R0 or R1.

Example 5—Indirect Addressing

```
; INDADR ADD CONTENTS OF MEMORY LOCATION
;       ADDRESSED BY REGISTER 1
;       TO CONTENTS OF RAM LOCATION
;       ADDRESSED BY REGISTER 0
;
INDADR  MOV     A,@R0
        ADD     A,@R1
        MOV     @R0,A
```

Indirect addressing on the 8051 is the same as in the 8048 family, except that all eight bits of the pointer register contents are significant; if the contents point to a non-existent memory location (i.e., an address greater than 7FH on the 8051) the result of the instruction is undefined. (Future microcomputers based on the MCS-51™ architecture could implement additional memory in the on-chip RAM logical address space at locations above 7FH.) The 8051 uses register-indirect addressing for five new instructions plus the 13 on the 8048.

Immediate Addressing

When a source operand is a constant rather than a variable (i.e.—the instruction uses a value known at assembly time), then the constant can be incorporated into the instruction. An additional instruction byte specifies the value used (Figure 12.d).

The value used is fixed at the time of ROM manufacture or EPROM programming and may not be altered during program execution. In the assembly language immediate operands are preceded by a number sign ("#"). The operand may be either a numeric string, a symbolic variable, or an arithmetic expression using constants.

Example 6—Adding Constants Using Immediate Addressing

```
; IMMADR ADD THE CONSTANT 12 (DECIMAL)
;       TO THE CONSTANT 34 (DECIMAL)
;       LEAVE SUM IN ACCUMULATOR
;
IMMADR  MOV     A,#12
        ADD     A,#34
```

The preceding example was included for consistency; it has little practical value. Instead, ASM51 could compute the sum of two constants at assembly time.

Example 7—Adding Constants Using ASM51 Capabilities

```
; ASMSUM LOAD ACC WITH THE SUM OF
;       THE CONSTANT 12 (DECIMAL) AND
;       THE CONSTANT 34 (DECIMAL)
;
ASMSUM  MOV     A,#(12+34)
```

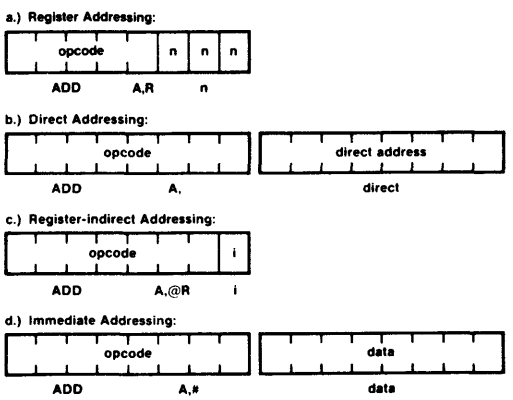


Figure 12. Data Addressing Machine Code Formats

AFN-01502A

Addressing Mode Combinations

The above examples all demonstrated the use of the four data-addressing modes in two-operand instructions (MOV, ADD) which use the accumulator as one operand. The operations ADDC, SUBB, ANL, ORL, and XRL (all to be discussed later) could be substituted for ADD in each example. The first three modes may be also be used for the XCH operation or, in combination with the Immediate Addressing mode (and an additional byte), loaded with a constant. The one-operand instructions INC and DEC, DJNZ, and CJNE may all operate on the accumulator, or may specify the Register, Direct, and Register-indirect addressing modes. Exception: as in the 8048, DJNZ cannot use the accumulator or indirect addressing. (The PUSH and POP operations cannot inherently address the accumulator as a special register either. However, all three can *directly* address the accumulator as one of the twenty special-function registers by putting the symbol "ACC" in the operand field.)

Advantages of Symbolic Addressing

Like most assembly or higher-level programming languages, ASM51 allows instructions or variables to be given appropriate, user-defined symbolic names. This is done for instruction lines by putting a label followed by a colon (":") before the instruction proper, as in the above examples. Such symbols must start with an alphabetic character (remember what distinguished BACH from 0BACH?), and may include any combination of letters, numbers, question marks ("?") and underscores ("_"). For very long names only the first 31 characters are relevant.

Assembly language programs may intermix upper- and lower-case letters arbitrarily, but ASM51 converts both to upper-case. For example, ASM51 will internally process an "I" for an "i" and, of course, "A_TOOTH" for "a_tooth."

The underscore character makes symbols easier to read and can eliminate potential ambiguity (as in the label for a subroutine to switch two entires on a stack, "S_EXCHANGE"). The underscore is significant, and would distinguish between otherwise-identical character strings.

ASM51 allows *all* variables (registers, ports, internal or external RAM addresses, constants, etc.) to be assigned labels according to these rules with the EQUate or SET directives.

Example 8—Symbolic Addressing of Variables Defined as RAM Locations

```
VAR_0   SET     20H
VAR_1   SET     21H
;
;SYMB_1 ADD CONTENTS OF VAR_1
;       TO CONTENTS OF VAR_0
;
SYMB_1: MOV     A,VAR_0
        ADD     A,VAR_1
        MOV     VAR_0,A
```

Notice from Table 4 that the MCS-51™ instruction set has relatively few instruction mnemonics (abbreviations) for the programmer to memorize. Different data types or addressing modes are determined by the operands specified, rather than variations on the mnemonic. For example, the mnemonic "MOV" is used by 18 different instructions to operate on three data types (bit, byte, and address). The fifteen versions which move byte variables between the logical address spaces are diagrammed in Figure 13. Each arrow shows the direction of transfer from source to destination.

Notice also that for most instructions allowing register addressing there is a corresponding direct addressing instruction and vice versa. This lets the programmer begin writing 8051 programs as if (s)he has access to 128 different registers. When the program has evolved to the point where the programmer has a fairly accurate idea how often each variable is used, he/she may allocate the working registers in each bank to the most "popular" variables. (The assembly cross-reference option will show exactly how often and where each symbol is referenced.) If symbolic addressing is used in writing the source program only the lines containing the symbol definition will need to be changed; the assembler will produce the appropriate instructions even though the rest of the program is left untouched. Editing only the first two lines of Example 8 will shrink the six-byte code segment produced in half.

How are instruction sets "counted"? There is no standard practice; different people assessing the same CPU using different conventions may arrive at different totals.

Each operation is then broken down according to the different addressing modes (or combinations of addressing modes) it can accommodate. The "CLR" mnemonic is used by two instructions with respect to bit variables ("CLR C" and "CLR bit") and once ("CLR A") with regards to bytes. This expansion yields the 111 separate instructions of Table 4.

The method used for the MCS-51® instruction set first breaks it down into "operations": a basic function applied to a single data type. For example, the four versions of the ADD instruction are grouped to form one operation — addition of eight-bit variables. The six forms of the ANL instruction for *byte* variables make up a different operation; the two forms of ANL which operate on *bits* are considered still another. The MOV mnemonic is used by three different operation classes, depending on whether bit, byte, or 16-bit values are affected. Using this terminology the 8051 can perform 51 different operations.

AFN-01502A

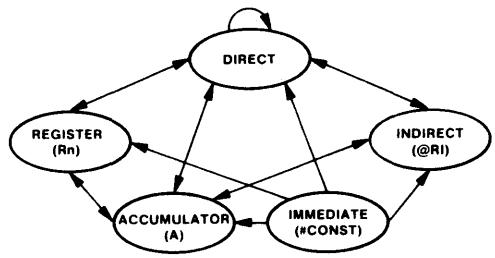


Figure 13. Road map for moving data bytes

Example 9—Redeclaring Example 8 Symbols as Registers

```
VAR_0   SET     R0
VAR_1   SET     R1
;
;SYMB_2 ADD CONTENTS OF VAR_1
;       TO CONTENTS OF VAR_0
;
SYMB_2: MOV     A, VAR_0
        ADD     A, VAR_1
        MOV     VAR_0, A
```

Arithmetic Instruction Usage — ADD, ADDC, SUBB and DA

The ADD instruction adds a byte variable with the accumulator, leaving the result in the accumulator. The carry flag is set if there is an overflow from bit 7 and cleared otherwise. The AC flag is set to the carry-out from bit 3 for use by the DA instruction described later. ADDC adds the previous contents of the carry flag with the two byte variables, but otherwise is the same as ADD.

The SUBB (subtract with borrow) instruction subtracts the byte variable indicated and the contents of the carry flag together from the accumulator, and puts the result back in the accumulator. The carry flag serves as a "Borrow Required" flag during subtraction operations; when a greater value is subtracted from a lesser value (as in subtracting 5 from 1) requiring a borrow into the highest order bit, the carry flag is set; otherwise it is cleared.

When performing signed binary arithmetic, certain combinations of input variables can produce results which seem to violate the Laws of Mathematics. For example, adding 7FH (127) to itself produces a sum of 0FEH, which is the two's complement representation of -2 (refer back to Table 2)! In "normal" arithmetic, two positive values can't have a negative sum. Similarly, it is normally impossible to subtract a positive value from a negative value and leave a positive result — but in two's complement there are instances where this too may happen. Fundamentally, such anomolies occur when the magnitude of the resulting value is too great to "fit" into the seven bits allowed for it; there is no one-byte two's complement representation for 254, the true sum of 127 and 127.

The MCS-51™ processors detect whether these situations occur and indicate such errors with the OV flag. (OV may be tested with the conditional jump instructions JB and JNB, described under the Boolean Processor chapter.)

At a hardware level, OV is set if there is a carry out of bit 6 but not out of bit 7, or a carry out of bit 7 but not out of bit 6. When adding signed integers this indicates a negative number produced as the sum of two positive operands, or a positive sum from two negative operands; on SUBB this indicates a negative result after subtracting a negative number from a positive number, or a positive result when a positive number is subtracted from a negative number.

The ADDC and SUBB instructions incorporate the previous state of the carry (borrow) flag to allow multiple precision calculations by repeating the operation with successively higher-order operand bytes. In either case, the carry must be cleared before the first iteration.

If the input data for a multiple precision operation is an unsigned string of integers, upon completion the carry flag will be set if an overflow (for ADDC) or underflow (for SUBB) occurs. With two's complement signed data (i.e., if the most significant bit of the original input data indicates the sign of the string), the overflow flag will be set if overflow or underflow occurred.

Example 10—String Subtraction with Signed Overflow Detection

```
;SUBSTR SUBTRACT STRING INDICATED BY R1
;       FROM STRING INDICATED BY R0 TO
;       PRECISION INDICATED BY R2.
;       CHECK FOR SIGNED UNDERFLOW WHEN DONE.
;
SUBSTR: CLR     C               ;BORROW= 0.
SUBS1:  MOV     A, @R0
        SUBB    A, @R1          ;SUBTRACT NEXT PLACE
        MOV     @R0, A
        INC     R0              ;BUMP POINTERS
        INC     R1
        DJNZ    R2, SUBS1               ;LOOP AS NEEDED
;               WHEN DONE, TEST IF OVERFLOW OCCURED
;               ON LAST ITERATION OF LOOP.
        JNB     OV, OV_OK
;       ....    ......          (OVERFLOW RECOVERY ROUTINE)
OV_OK:  RET                     ;RETURN
```

Decimal addition is possible by using the DA instruction in conjunction with ADD and/or ADDC. The eight-bit binary value in the accumulator resulting from an earlier addition of two variables (each a packed BCD digit-pair) is adjusted to form two BCD digits of four bits each. If the contents of accumulator bits 3-0 are greater than nine (xxxx1010-xxxx1111), or if the AC flag had been set, six is added to the accumulator producing the proper BCD digit in the low-order nibble. (This addition might itself set — but would not clear — the carry flag.) If the carry flag is set, or if the four high-order bits now exceed nine (1010xxxx-1111xxxx), these bits are incremented by six. The carry flag is left set if originally set or if either addition of six produces a carry out of the highest-order bit, indicating the sum of the original two BCD variables is greater than or equal to decimal 100.

AFN-01502A

Example 11—Two Byte Decimal Add with Registers and Constants

```
;BCDADD ADD THE CONSTANT 1,234 (DECIMAL) TO THE
;       CONTENTS OF REGISTER PAIR <R3><R2>
;       (ALREADY A 4 BCD-DIGIT VARIABLE)
;
BCDADD: MOV     A,R2
        ADD     A,#34H
        DA      A
        MOV     R2,A
        MOV     A,R3
        ADDC    A,#12H
        DA      A
        MOV     R3,A
        RET
```

Multiplication and Division

The instruction "MUL AB" multiplies the unsigned eight-bit integer values held in the accumulator and B-registers. The low-order byte of the sixteen-bit product is left in the accumulator, the higher-order byte in B. If the high-order eight-bits of the product are all zero the overflow flag is cleared; otherwise it is set. The programmer can poll OV to determine when the B register is non-zero and must be processed.

"DIV AB" divides the unsigned eight-bit integer in the accumulator by the unsigned eight-bit integer in the B-register. The integer part of the quotient is returned in the accumulator; the remainder in the B-register. If the B-register originally contained 00H then the overflow flag will be set to indicate a division error, and the values returned will be undefined. Otherwise OV is cleared.

The divide instruction is also useful for purposes such as radix conversion or separating bit fields of the accumulator. A short subroutine can convert an eight-bit unsigned binary integer in the accumulator (between 0 & 255) to a three-digit (two byte) BCD representation. The hundred's digit is returned in one register (HUND) and the ten's and one's digits returned as packed BCD in another (TENONE).

Example 12—Use of DIV Instruction for Radix Conversion

```
;BINBCD CONVERT 8-BIT BINARY VARIABLE IN ACC
;       TO 3-DIGIT PACKED BCD FORMAT
;       HUNDREDS' PLACE LEFT IN VARIABLE 'HUND',
;       TENS' AND ONES' PLACES IN 'TENONE'
;
HUND    EQU     21H
TENONE  EQU     22H
;
BINBCD: MOV     B,#100  ;DIVIDE BY 100 TO
        DIV     AB      ;DETERMINE NUMBER OF HUNDREDS
        MOV     HUND,A
        MOV     A,#10   ;DIVIDE REMAINDER BY 10 TO
        XCH     A,B     ;DETERMINE # OF TENS LEFT
        DIV     AB      ;TENS DIGIT IN ACC, REMAINDER IS ONES
                        ;DIGIT
        SWAP    A
        ADD     A,B     ;PACK BCD DIGITS IN ACC
        MOV     TENONE,A
        RET
```

The divide instruction can also separate eight bits of data in the accumulator into sub-fields. For example, packed BCD data may be separated into two nibbles by dividing the data by 16, leaving the high-nibble in the accumulator and the low-order nibble (remainder) in B. The two digits may then be operated on individually or in conjunction with each other. This example receives two packed BCD digits in the accumulator and returns the product of the two individual digits in packed BCD format in the accumulator.

Example 13—Implementing a BCD Multiply Using MPY and DIV

```
;MULBCD UNPACK TWO BCD DIGITS RECEIVED IN ACC,
;       FIND THEIR PRODUCT, AND RETURN PRODUCT
;       IN PACKED BCD FORMAT IN ACC.
;
MULBCD: MOV     B,#10H  ;DIVIDE INPUT BY 16
        DIV     AB      ;A & B HOLD SEPARATED DIGITS
                        ;(EACH RIGHT JUSTIFIED IN REGISTER).
        MUL     AB      ;A HOLDS PRODUCT IN BINARY FORMAT (0 -
                        ;99(DECIMAL) = 0 - 63H)
        MOV     B,#10   ;DIVIDE PRODUCT BY 10.
        DIV     AB      ;A HOLDS # OF TENS, B HOLDS REMAINDER
        SWAP    A
        ORL     A,B     ;PACK DIGITS
        RET
```

Logical Byte Operations — ANL, ORL, XRL

The instructions ANL, ORL, and XRL perform the logical functions AND, OR, and/or Exclusive-OR on the two byte variables indicated, leaving the results in the first. No flags are affected. (A word to the wise — do not vocalize the first two mnemonics in mixed company.)

These operations may use all the same addressing modes as the arithmetics (ADD, etc.) but unlike the arithmetics, they are not restricted to operating on the accumulator. Directly addressed bytes may be used as the destination with either the accumulator or a constant as the source. These instructions are useful for clearing (ANL), setting (ORL), or complementing (XRL) one or more bits in a RAM, output ports, or control registers. The pattern of bits to be affected is indicated by a suitable mask byte. Use immediate addressing when the pattern to be affected is known at assembly time (Figure 14); use the accumulator versions when the pattern is computed at run-time.

I/O ports are often used for parallel data in formats other than simple eight-bit bytes. For example, the low-order five bits of port 1 may output an alphabetic character code (hopefully) without disturbing bits 7-5. This can be a simple two-step process. First, clear the low-order five pins with an ANL instruction; then set those pins corresponding to ones in the accumulator. (This example assumes the three high-order bits of the accumulator are originally zero.)

Example 14—Reconfiguring Port Size with Logical Byte Instructions

```
OUT_PX: ANL   P1,#11100000B ;CLEAR BITS P1.4 - P1.0
        ORL   P1,A          ;SET P1 PINS CORRESONDING TO SET ACC
                            ; BITS.
        RET
```

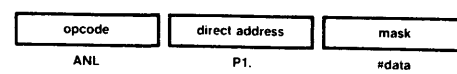


Figure 14. Instruction Pattern for Logical Operation Special Addressing Modes

AFN-01502A

In this example, low-order bits remaining high may "glitch" low for one machine cycle. If this is undesirable, use a slightly different approach. First, set all pins corresponding to accumulator one bits, then clear the pins corresponding to zeroes in low-order accumulator bits. Not all bits will change from original to final state at the same instant, but no bit makes an intermediate transition.

Example 15 — Reconfiguring I/O Port Size without Glitching

```
ALT_PX  ORL     P1,A
        ORL     A,#11100000B
        ANL     P1,A
        RET
```

Program Control — Jumps, Calls, Returns

Whereas the 8048 only has a single form of the simple jump instruction, the 8051 has three. Each causes the program to unconditionally jump to some other address. They differ in how the machine code represents the destination address.

LJMP (Long Jump) encodes a sixteen-bit address in the second and third instruction bytes (Figure 15.a); the destination may be anywhere in the 64 Kilobyte program memory address space.

The two-byte AJMP (Absolute Jump) instruction encodes its destination using the same format as the 8048: address bits 10 through 8 form a three bit field in the opcode and address bits 7 through 0 form the second byte (Figure 15.b). Address bits 15-12 are unchanged from the (incremented) contents of the P.C., so AJMP can only be used when the destination is known to be within the same 2K memory block. (Otherwise ASM51 will point out the error.)

A different two-byte jump instruction is legal with any nearby destination, regardless of memory block boundaries or "pages." SJMP (Short Jump) encodes the destination with a program counter-relative address in the second byte (Figure 15.c). The CPU calculates the destination at run-time by adding the signed eight-bit displacement value to the incremented P.C. Negative offset values will cause jumps up to 128 bytes backwards; positive values up to 127 bytes forwards. (SJMP with 00H in the machine code offset byte will proceed with the following instruction).

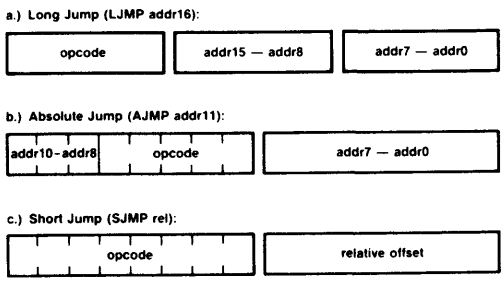


Figure 15. Jump Instruction Machine Code Formats

In keeping with the 8051 assembly language goal of minimizing the number of instruction mnemonics, there is a "generic" form of the three jump instructions. ASM51 recognizes the mnemonic JMP as a "pseudo-instruction," translating it into the machine instructions LJMP, AJMP, or SJMP, depending on the destination address.

Like SJMP, all conditional jump instructions use relative addressing. JZ (Jump if Zero) and JNZ (Jump if Not Zero) monitor the state of the accumulator as implied by their names, while JC (Jump on Carry) and JNC (Jump on No Carry) test whether or not the carry flag is set. All four are two-byte instructions, with the same format as Figure 15.c. JB (Jump on Bit), JNB (Jump on No Bit) and JBC (Jump on Bit then Clear Bit) can test any status bit or input pin with a three byte instruction; the second byte specifies which bit to test and the third gives the relative offset value.

There are two subroutine-call instructions, LCALL (Long Call) and ACALL (Absolute Call). Each increments the P.C. to the first byte of the following instruction, then pushes it onto the stack (low byte first). Saving both bytes increments the stack pointer by two. The subroutine's starting address is encoded in the same ways as LJMP and AJMP. The generic form of the call operation is the mnemonic CALL, which ASM51 will translate into LCALL or ACALL as appropriate.

The return instruction RET pops the high- and low-order bytes of the program counter successively from the stack, decrementing the stack pointer by two. Program execution continues at the address previously pushed: the first byte of the instruction immediately following the call.

When an interrupt request is recognized by the 8051 hardware, two things happen. Program control is automatically "vectored" to one of the interrupt service routine starting addresses by, in effect, forcing the CPU to process an LCALL instead of the next instruction. This automatically stores the return address on the stack. (Unlike the 8048, no status information is automatically saved.)

Secondly, the interrupt logic is disabled from accepting any other interrupts from the same or lower priority. After completing the interrupt service routine, executing an RETI (Return from Interrupt) instruction will return execution to the point where the background program was interrupted — just like RET — while restoring the interrupt logic to its previous state.

AFN-01502A

Operate-and-branch instructions — CJNE, DJNZ

Two groups of instructions combine a byte operation with a conditional jump based on the results.

CJNE (Compare and Jump if Not Equal) compares two byte operands and executes a jump if they disagree. The carry flag is set following the rules for subtraction: if the unsigned integer value of the first operand is less than that of the second it is set; otherwise, it is cleared. However, neither operand is modified.

The CJNE instruction provides, in effect, a one-instruction "case" statement. This instruction may be executed repeatedly, comparing the code variable to a list of "special case" value: the code segment following the instruction (up to the destination label) will be executed only if the operands match. Comparing the accumulator or a register to a series of constants is a convenient way to check for special handling or error conditions; if none of the cases match the program will continue with "normal" processing.

A typical example might be a word processing device which receives ASCII characters through the serial port and drives a thermal hard-copy printer. A standard routine translates "printing" characters to bit patterns, but control characters (<DEL>, <CR>, <LF>, <BEL>, <ESC> or <SP>) must invoke corresponding special routines. Any other character with an ASCII code less than 20H should be translated into the <NUL> value, 00H, and processed with the printing characters.

Example 16 — Case Statements Using CJNE

```
CHAR    EQU     R7      ;CHARACTER CODE VARIABLE
;
INTERP: CJNE    CHAR,#7FH,INTP_1
;               .....   (SPECIAL ROUTINE FOR RUBOUT CODE)
        RET
INTP_1: CJNE    CHAR,#07H,INTP_2
;               .....   (SPECIAL ROUTINE FOR BELL CODE)
        RET
INTP_2: CJNE    CHAR,#0AH,INTP_3
;               .....   (SPECIAL ROUTINE FOR LFEED CODE)
        RET
INTP_3: CJNE    CHAR,#0DH,INTP_4
;               .....   (SPECIAL ROUTINE FOR RETURN CODE)
        RET
INTP_4: CJNE    CHAR,#1BH,INTP_5
;               .....   (SPECIAL ROUTINE FOR ESCAPE CODE)
        RET
INTP_5: CJNE    CHAR,#20H,INTP_6
;               .....   (SPECIAL ROUTINE FOR SPACE CODE)
        RET
INTP_6: JC      PRINTC  ;JUMP IF CODE > 20H
        MOV     CHAR,#0 ;REPLACE CONTROL CHARACTERS WITH
                        ; NULL CODE
PRINTC:                 ;PROCESS STANDARD PRINTING
;               .....   ; CHARACTER
        RET
```

DJNZ (Decrement and Jump if Not Zero) decrements the register or direct address indicated and jumps if the result is not zero, without affecting any flags. This provides a simple means for executing a program loop a given number of times, or for adding a moderate time delay (from 2 to 512 machine cycles) with a single instruction. For example, a 99-usec. software delay loop can be added to code forcing an I/O pin low with only two instructions.

Example 17 — Inserting a Software Delay with DJNZ

```
        CLR     WR
        MOV     R2,#49
        DJNZ    R2,$
        SETB    WR
```

The dollar sign in this example is a special character meaning "the address of this instruction." It is useful in eliminating instruction labels on the same or adjacent source lines. CJNE and DJNZ (like all conditional jumps) use program-counter relative addressing for the destination address.

Stack Operations — PUSH, POP

The PUSH instruction increments the stack pointer by one, then transfers the contents of the single byte variable indicated (direct addressing only) into the internal RAM location addressed by the stack pointer. Conversely, POP copies the contents of the internal RAM location addressed by the stack pointer to the byte variable indicated, then decrements the stack pointer by one.

(Stack Addressing follows the same rules, and addresses the same locations as Register-indirect. Future microcomputers based on the MCS-51™ CPU could have up to 256 bytes of RAM for the stack.)

Interrupt service routines must not change any variable or hardware registers modified by the main program, or else the program may not resume correctly. (Such a change might look like a spontaneous random error.) Resources used or altered by the service routine (Accumulator, PSW, etc.) must be saved and restored to their previous value before returning from the service routine. PUSH and POP provide an efficient and convenient way to save register states on the stack.

Example 18 — Use of the Stack for Status Saving on Interrupts

```
LOC_TMP EQU     $       ;REMEMBER LOCATION COUNTER
;
        ORG     0003H   ;STARTING ADDRESS FOR INTERRUPT ROUTINE
        LJMP    SERVER  ;JUMP TO ACTUAL SERVICE ROUTINE LOCATED
                        ; ELSEWHERE
;
        ORG     LOC_TMP ;RESTORE LOCATION COUNTER
SERVER: PUSH    PSW
        PUSH    ACC     ;SAVE ACCUMULATOR (NOTE DIRECT ADDRESSING
                        ; NOTATION)
        PUSH    B       ;SAVE B REGISTER
        PUSH    DPL     ;SAVE DATA POINTER
        PUSH    DPH     ;
        MOV     PSW,#00001000B  ;SELECT REGISTER BANK 1
;               .....
;               .....
        POP     DPH     ;RESTORE REGISTERS IN REVERSE ORDER
        POP     DPL
        POP     B
        POP     ACC
        POP     PSW     ;RESTORE PSW AND RE-SELECT ORIGINAL
                        ; REGISTER BANK
        RETI            ;RETURN TO MAIN PROGRAM AND RESTORE
                        ; INTERRUPT LOGIC
```

If the SP register held 1FH when the interrupt was detected, then while the service routine was in progress the stack would hold the registers shown in Figure 16; SP would contain 26H.

The example shows the most general situation; if the service routine doesn't alter the B-register and data pointer, for example, the instructions saving and restoring those registers would not be necessary.

The stack may also pass parameters to and from subroutines. The subroutine can indirectly address the parameters derived from the contents of the stack pointer.

AFN-01502A

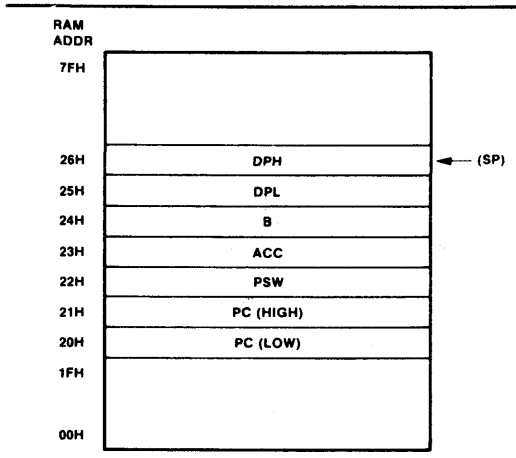


Figure 16. Stack contents during interrupt

One advantage here is simplicity. Variables need not be allocated for specific parameters, a potentially large number of parameters may be passed, and different calling programs may use different techniques for determining or handling the variables.

For example, the following subroutine reads out a parameter stored on the stack by the calling program, uses the low order bits to access a local look-up table holding bit patterns for driving the coils of a four phase stepper motor, and stores the appropriate bit pattern back in the same position on the stack before returning. The accumulator contents are left unchanged.

Example 19—Passing Variable Parameters to Subroutines Using the Stack

```
NXTPOS: MOV     R0, SP
        DEC     R0          ;ACCESS LOCATION PARAMETER PUSHED INTO
        DEC     R0
        XCH     A, @R0      ;READ INPUT PARAMETER AND SAVE
                            ; ACCUMULATOR
        ANL     A, #03H     ;MASK ALL BUT LOW-ORDER TWO BITS
        ADD     A, #2       ;ALLOW FOR OFFSET FROM MOVC TO TABLE
        MOVC    A, @A+PC    ;READ LOOK-UP TABLE ENTRY
        XCH     A, @R0      ;PASS BACK TRANSLATED VALUE AND RESTORE
                            ; ACC
        RET                 ;RETURN TO BACKGROUND PROGRAM
STPTBL: DB      01101111B       ;POSITION 0
        DB      01011111B       ;POSITION 1
        DB      10011111B       ;POSITION 2
        DB      10101111B       ;POSITION 3
```

The background program may reach this subroutine with several different calling sequences, all of which PUSH a value before calling the routine and POP the result after. A motor on Port 1 may be initialized by placing the desired position (zero) on the stack before calling the subroutine and outputing the results directly to a port afterwards.

Example 20—Sending and Receiving Data Parameters Via the Stack

```
        CLR     A
        PUSH    ACC
        CALL    NXTPOS
        POP     P1
```

If the position of the motor is determined by the contents of variable POSM1 (a byte in internal RAM) and the position of a second motor on Port 2 is determined by the data input to the low-order nibble of Port 2, a six-instruction sequence could update them both.

Example 21—Loading and Unloading Stack Direct from I/O Ports

```
POSM1   EQU     51

        PUSH    POSM1
        CALL    NXTPOS
        POP     P1
        PUSH    P2
        CALL    NXTPOS
        POP     P2
```

Data Pointer and Table Look-up instructions — MOV, INC, MOVC, JMP

The data pointer can be loaded with a 16-bit value using the instruction MOV DPTR, #data16. The data used is stored in the second and third instruction bytes, high-order byte first. The data pointer is incremented by INC DPTR. A 16-bit increment is performed; an overflow from the low byte will carry into the high-order byte. Neither instruction affects any flags.

The MOVC (Move Constant) instructions (MOVC A,@A+DPTR and MOVC A,@A+PC) read into the accumulator bytes of data from the program memory logical address space. Both use a form of indexed addressing: the former adds the unsigned eight-bit accumulator contents with the sixteen-bit data pointer register, and uses the resulting sum as the address from which the byte is fetched. A sixteen-bit addition is performed; a carry-out from the low-order eight bits may propagate through higher-order bits, but the contents of the DPTR are not altered. The latter form uses the incremented program counter as the "base" value instead of the DPTR (figure 17). Again, neither version affects the flags.

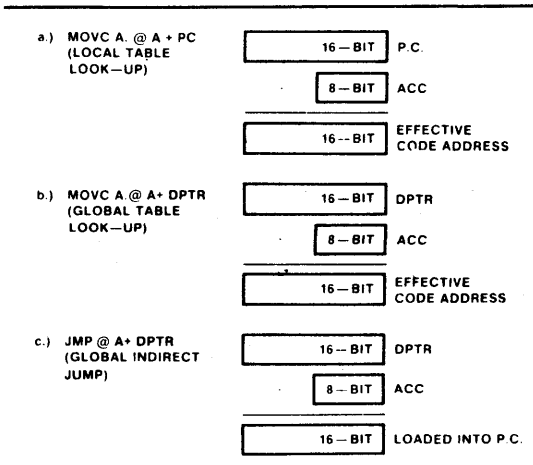


Figure 17. Operation of MOVC instructions

AFN-01502A

Each can be part of a three step sequence to access look-up tables in ROM. To use the DPTR-relative version, load the Data Pointer with the starting address of a look-up table; load the accumulator with (or compute) the index of the entry desired; and execute MOVC A,@A+DPTR. Unlike the similar MOVP3 instructions in the 8048, the table may be located anywhere in program memory. The data pointer may be loaded with a constant for short tables. Or to allow more complicated data structures, or tables with more than 256 entries, the values for DPH and DPL may be computed or modified with the standard arithmetic instruction set.

The PC-relative version has the advantage of not affecting the data pointer. Again, a look-up sequence takes three steps: load the accumulator with the index; compensate for the offset from the look-up instruction to the start of the table by adding the number of bytes separating them to the accumulator; then execute the MOVC A,@A+PC instruction.

Let's look at a non-trivial situation where this instruction would be used. Some applications store large multi-dimensional look-up tables of dot matrix patterns, non-linear calibration parameters, and so on in a linear (one-dimensional) vector in program memory. To retrieve data from the tables, variables representing matrix indices must be converted to the desired entry's memory address. For a matrix of dimensions (MDIMEN x NDIMEN) starting at address BASE and respective indices INDEXI and INDEXJ, the address of element (INDEXI, INDEXJ) is determined by the formula,

Entry Address = BASE + (NDIMEN x INDEXI) + INDEXJ

The code shown below can access any array with less than 255 entries (i.e., an 11x21 array with 231 elements). The table entries are defined using the Data Byte ("DB") directive, and will be contained in the assembly object code as part of the accessing subroutine itself.

Example 22 — Use of MPY and Data Pointer Instructions to Access Entries from a Multi-dimensional Look-Up Table in ROM

```
;MATRX1 LOAD CONSTANT READ FROM TWO DIMENSIONAL LOOK-UP
;       TABLE IN PROGRAM MEMORY INTO ACCUMULATOR
;       USING LOCAL TABLE LOOK-UP INSTRUCTION, 'MOVC    A,@A+PC
;       THE TOTAL NUMBER OF TABLE ENTRIES IS ASSUMED TO
;       BE SMALL, I.E. LESS THAN ABOUT 250 ENTRIES )
;       TABLE USED IN THIS EXAMPLE IS ( 11 X 21 )
;       DESIRED ENTRY ADDRESS IS GIVEN BY THE FORMULA,
;       [ (BASE ADDRESS) + (21 X INDEXI) + (INDEXJ) ]
;
INDEXI  EQU     R6      ;FIRST COORDINATE OF ENTRY (0-10)
INDEXJ  EQU     23H     ;SECOND COORDINATE OF ENTRY (0-20)
;
MATRX1: MOV     A,INDEXI
        MOV     B,#21
        MUL     AB
        ADD     A,INDEXJ
;       ALLOW FOR INSTRUCTION BYTE BETWEEN "MOVC" AND
;       ENTRY (0,0)
        INC     A
        MOVC    A,@A+PC
        RET
BASE1:  DB      1       ;(entry 0,0)
        DB      2       ;(entry 0,1)
;
        DB      21      ;(entry 0,20)
        DB      22      ;(entry 1,0)
;
        DB      42      ;(entry 1,20)
;
;
        DB      231     ;(entry 10,20)
```

There are several different means for branching to sections of code determined or selected at run time. (The single destination addresses incorporated into conditional and unconditional jumps are, of course, determined at assembly time). Each has advantages for different applications.

The most common is an N-way conditional jump based on some variable, with all of the potential destinations known at assembly time. One of a number of small routines is selected according to the value of an index variable determined while the program is running. The most efficient way to solve this problem is with the MOVC and an indirect jump instruction, using a short table of one byte offset values in ROM to indicate the relative starting addresses of the several routines.

JMP @A+DPTR is an instruction which performs an indirect jump to an address determined during program execution. The instruction adds the eight-bit unsigned accumulator contents with the contents of the sixteen-bit data pointer, just like MOVC A,@A+DPTR. The resulting sum is loaded into the program counter and is used as the address for subsequent instruction fetches. Again, a sixteen-bit addition is performed; a carry out from the low-order eight bits may propagate through the higher-order bits. In this case, neither the accumulator contents nor the data pointer is altered.

The example subroutine below reads a byte of RAM into the accumulator from one of four alternate address spaces, as selected by the contents of the variable MEMSEL. The address of the byte to be read is determined by the contents of R0 (and optionally R1). It might find use in a printing terminal application, where four different model printers all use the same ROM code but use different types and sizes of buffer memory for different speeds and options.

Example 23 — N-Way Branch and Computed Jump Instructions via JMP @ ADPTR

```
MEMSEL  EQU     R3
;
JUMP_4: MOV     A,MEMSEL
        MOV     DPTR,#JMPTBL
        MOVC    A,@A+DPTR
        JMP     @A+DPTR
JMPTBL: DB      MEMSP0-JMPTBL
        DB      MEMSP1-JMPTBL
        DB      MEMSP2-JMPTBL
        DB      MEMSP3-JMPTBL
MEMSP0: MOV     A,@R0   ;READ FROM INTERNAL RAM
        RET
MEMSP1: MOVX    A,@R0   ;READ FROM 256 BYTES OF EXTERNAL RAM
        RET
MEMSP2: MOV     DPL,R0
        MOV DPH,R1
        MOVX    A,@DPTR ;READ FROM 64K BYTES OF EXTERNAL RAM
        RET
MEMSP3: MOV     A,R1
        ANL     A,#07H
        ANL     P1,#11111000B
        ORL     P1,A
        MOVX    A,@R0   ;READ FROM 4K BYTES OF EXTERNAL RAM
        RET
```

Note that this approach is suitable whenever the size of jump table plus the length of the alternate routines is less than 256 bytes. The jump table and routines may be located anywhere in program memory, independent of 256-byte program memory pages.

AFN-01502A

For applications where up to 128 destinations must be selected, all of which reside in the same 2K page of program memory which may be reached by the two-byte absolute jump instructions, the following technique may be used. In the above mentioned printing terminal example, this sequence could "parse" 128 different codes for ASCII characters arriving via the 8051 serial port.

Example 24—N-Way Branch with 128 Optional Destinations

```
OPTION  EQU     R3
;
;
JMP128  MOV     A,OPTION
        RL      A               ;MULTIPLY BY 2 FOR 2 BYTE JUMP TABLE
        MOV     DPTR,#INSTBL    ;FIRST ENTRY IN JUMP TABLE
        JMP     @A+DPTR         ;JUMP INTO JUMP TABLE
;
INSTBL  AJMP    PROC00  ;128 CONSECUTIVE
        AJMP    PROC01  ;AJMP INSTRUCTIONS
        AJMP    PROC02
;
;
        AJMP    PROC7E
        AJMP    PROC7F
```

The destinations in the jump table (PROC00-PROC7F) are not all necessarily unique routines. A large number of special control codes could each be processed with their own unique routine, with the remaining printing characters all causing a branch to a common routine for entering the character into the output queue.

In those rare situations where even 128 options are insufficient, or where the destination routines may cross a 2K page boundary, the above approach may be modified slightly as shown below.

Example 25—256-Way Branch Using Address Look-Up Tables

```
RTEMP   EQU     R7
;
JMP256  MOV     DPTR,#ADRTBL    ;FIRST ENTRY IN TABLE OF ADDRESSES
        MOV     A,OPTION
        CLR     C
        RLC     A               ;MULTIPLY BY 2 FOR 2 BYTE JUMP TABLE
        JNC     LOW128
        INC     DPH
LOW128  MOV     RTEMP,A         ;SAVE ACC FOR HIGH BYTE READ
        MOVC    A,@A+DPTR       ;READ LOW BYTE FROM JUMP TABLE
        XCH     A,RTEMP
        INC     A
        MOVC    A,@A+DPTR       ;GET LOW-ORDER BYTE FROM TABLE
        PUSH    ACC
        MOV     A,RTEMP
        MOVC    A,@A+DPTR       ;GET HIGH-ORDER BYTE FROM TABLE
        PUSH    ACC
;       THE TWO ACC PUSHES HAVE PRODUCED
;       A "RETURN ADDRESS" ON THE STACK WHICH CORRESPONDS
;       TO THE DESIRED STARTING ADDRESS.
;       IT MAY BE REACHED BY POPPING THE STACK
;       INTO THE PC
        RET
;
ADRTBL  DW      PROC00  ;UP TO 256 CONSECUTIVE DATA
        DW      PROC01  ;WORDS INDICATING STARTING ADDRESSES
;
        DW      PROCFF
;
;
;       DUMMY CODE ADDRESS DEFINITIONS NEEDED BY ABOVE
;       TWO EXAMPLES:
;
PROC00  NOP
PROC01  NOP
PROC02  NOP
PROC7E  NOP
PROC7F  NOP
PROCFF  NOP
```

4. BOOLEAN PROCESSING INSTRUCTIONS

The commonly accepted terms for tasks at either end of the computational vs. control application spectrum are, respectively, "number-crunching" and "bit-banging". Prior to the introduction of the MCS-51™ family, nice number-crunchers made bad bit-bangers and vice versa. The 8051 is the industry's first single-chip microcomputer designed to crunch **and** bang. (In some circles, the latter technique is also referred to as "bit-twiddling". Either is correct.)

Direct Bit Addressing

A number of instructions operate on Boolean (one-bit) variables, using a direct bit addressing mode comparable to direct byte addressing. An additional byte appended to the opcode specifies the Boolean variable, I/O pin, or control bit used. The state of any of these bits may be tested for "true" or "false" with the conditional branch instructions JB (Jump on Bit) and JNB (Jump on Not Bit). The JBC (Jump on Bit and Clear) instruction combines a test-for-true with an unconditional clear.

As in direct byte addressing, bit 7 of the address byte switches between two physical address spaces. Values between 0 and 127 (00H-7FH) define bits in internal RAM locations 20H to 2FH (Figure 18a); address bytes between 128 and 255 (80H-0FFH) define bits in the 2 x "special-function" register address space (Figure 18b). If no 2 x "special-function" register corresponds to the direct bit address used the result of the instruction is undefined.

Bits so addressed have many wondrous properties. They may be set, cleared, or complemented with the two byte instructions SETB, CLR, or CPL. Bits may be moved to and from the carry flag with MOV. The logical ANL and ORL functions may be performed between the carry and either the addressed bit or its complement.

Bit Manipulation Instructions — MOV

The "MOV" mnemonic can be used to load an addressable bit into the carry flag ("MOV C, bit") or to copy the state of the carry to such a bit ("MOV bit, C"). These instructions are often used for implementing serial I/O algorithms via software or to adapt the standard I/O port structure.

It is sometimes desirable to "re-arrange" the order of I/O pins because of considerations in laying out printed circuit boards. When interfacing the 8051 to an immediately adjacent device with "weighted" input pins, such as keyboard column decoder, the corresponding pins are likely to be not aligned (Figure 19).

There is a trade-off in "scrambling" the interconnections with either interwoven circuit board traces or through software. This is extremely cumbersome (if not impossible) to do with byte-oriented computer architectures. The 8051's unique set of Boolean instructions makes it simple to move individual bits between arbitrary locations.

AFN-01502A

a.) RAM Bit Addresses.

RAM BYTE	(MSB)							(LSB)
7FH								
2FH	7F	7E	7D	7C	7B	7A	79	78
2EH	77	76	75	74	73	72	71	70
2DH	6F	6E	6D	6C	6B	6A	69	68
2CH	67	66	65	64	63	62	61	60
2BH	5F	5E	5D	5C	5B	5A	59	58
2AH	57	56	55	54	53	52	51	50
29H	4F	4E	4D	4C	4B	4A	49	48
28H	47	46	45	44	43	42	41	40
27H	3F	3E	3D	3C	3B	3A	39	38
26H	37	36	35	34	33	32	31	30
25H	2F	2E	2D	2C	2B	2A	29	28
24H	27	26	25	24	23	22	21	20
23H	1F	1E	1D	1C	1B	1A	19	18
22H	17	16	15	14	13	12	11	10
21H	0F	0E	0D	0C	0B	0A	09	08
20H	07	06	05	04	03	02	01	00
1FH 18H	Bank 3							
17H 10H	Bank 2							
0FH 08H	Bank 1							
07H 00H	Bank 0							

b.) Hardware Register Bit Addresses.

Direct Byte Address	(MSB) Bit Addresses (LSB)								Hardware Register Symbol
0FFH									
0F0H	F7	F6	F5	F4	F3	F2	F1	F0	B
0E0H	E7	E6	E5	E4	E3	E2	E1	E0	ACC
0D0H	D7	D6	D5	D4	D3	D2	D1	D0	PSW
0B8H	–	–	–	BC	BB	BA	B9	B8	IP
0B0H	B7	B6	B5	B4	B3	B2	B1	B0	P3
0A8H	AF	–	–	AC	AB	AA	A9	A8	IE
0A0H	A7	A6	A5	A4	A3	A2	A1	A0	P2
98H	9F	9E	9D	9C	9B	9A	99	98	SCON
90H	97	96	95	94	93	92	91	90	P1
88H	8F	8E	8D	8C	8B	8A	89	88	TCON
80H	87	86	85	84	83	82	81	80	P0

Figure 18. Bit Address Maps

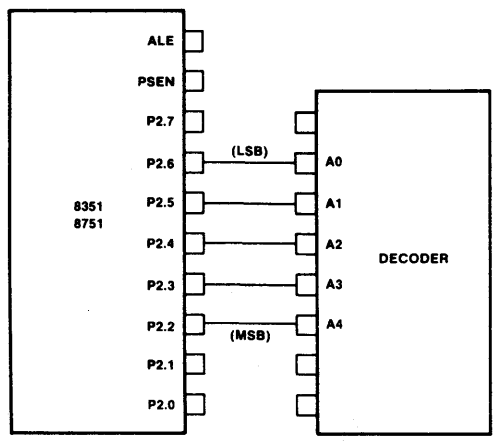


Figure 19. "Mismatch" Between I/O port and Decoder

Example 26 — Re-ordering I/O Port Configuration

```
OUT_P2: RRC     A       ;MOVE ORIGINAL ACC.0 INTO CY
        MOV     P2.6,C  ;STORE CARRY TO PIN P26
        RRC     A       ;MOVE ORIGINAL ACC.1 INTO CY
        MOV     P2.5,C  ;STORE CARRY TO PIN P25
        RRC     A       ;MOVE ORIGINAL ACC.2 INTO CY
        MOV     P2.4,C  ;STORE CARRY TO PIN P24
        RRC     A       ;MOVE ORIGINAL ACC.3 INTO CY
        MOV     P2.3,C  ;STORE CARRY TO PIN P23
        RRC     A       ;MOVE ORIGINAL ACC.4 INTO CY
        MOV     P2.2,C  ;STORE CARRY TO PIN P22
        RET
```

Solving Combinatorial Logic Equations — ANL, ORL

Virtually all hardware designers are familiar with the problem of solving complex functions using combinatorial logic. The technologies involved may vary greatly, from multiple contact relay logic, vacuum tubes, TTL, or CMOS to more esoteric approaches like fluidics, but in each case the goal is the same: a Boolean (true/false) function is computed on a number of Boolean variables.

AFN-01502A

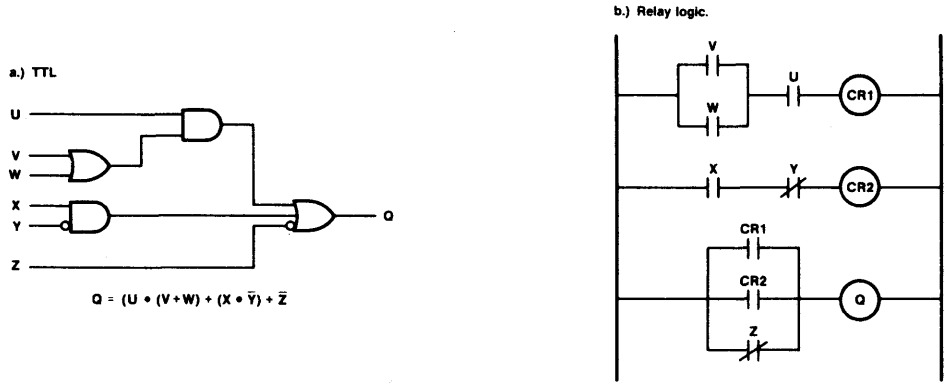


Figure 20. Implementations of Boolean functions

Figure 20 shows the logic diagram for an arbitrary function of six variables named U through Z using standard logic and relay logic symbols. Each is a solution of the equation,

$$Q = (U \cdot (V + W)) + (X \cdot \overline{Y}) + \overline{Z}$$

(While this equation could be reduced using Karnaugh Maps or algebraic techniques, that is not the purpose of this example. Even a minor change to the function equation would require re-reducing from scratch.)

Most digital computers can solve equations of this type with standard word-wide logical instructions and conditional jumps. Still, such software solutions seem somewhat sloppy because of the many paths through the program the computation can take.

Assume U and V are input pins being read by different input ports, W and X are status bits for two peripheral controllers (read as I/O ports), and Y and Z are software flags set or cleared earlier in the program. The end result must be written to an output pin on some third port.

For the sake of comparison we will implement this function with software drawn from three proper subsets of the MCS-51™ instruction set. The first two implementations follow the flow chart shown in Figure 21. Program flow would embark on a route down a test-and-branch tree and leaves either the "True" or "Not True" exit ASAP. These exits then write the output port with the data previously written to the same port with the result bit respectively one or zero.

In the first case, we assume there are no instructions for addressing individual bits other than special flags like the carry. This is typical of many older microprocessors and mainframe computers designed for number-crunching. MCS-51™ mnemonics are used here, though for most other machines the issue would be even further clouded by their use of operation-specific mnemonics like INPUT, OUTPUT, LOAD, STORE, etc., instead of the universal MOV.

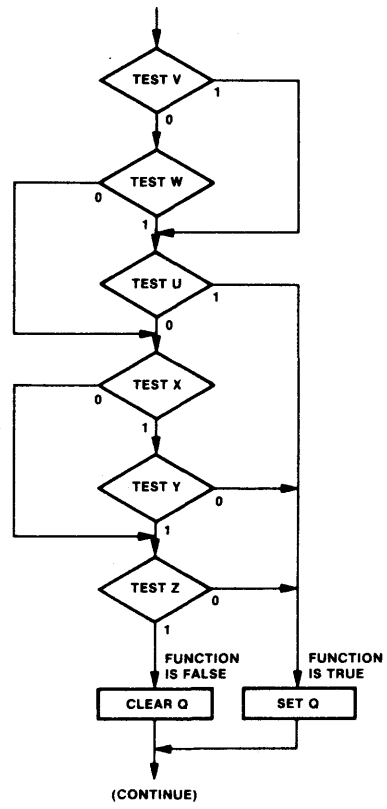


Figure 21. Flow chart for tree-branching logic implementation

AFN-01502A

Example 27—Software Solution to Logic Function of Figure 20, Using only Byte-Wide Logical Instructions

```
;BFUNC1 SOLVE A RANDOM LOGIC FUNCTION OF 6
;       VARIABLES BY LOADING AND MASKING THE APPROPRIATE
;       BITS IN THE ACCUMULATOR, THEN EXECUTING CONDITIONAL
;       JUMPS BASED ON ZERO CONDITION.
;       (APPROACH USED BY BYTE-ORIENTED ARCHITECTURES.)
;       BYTE AND MASK VALUES CORRESPOND TO RESPECTIVE
;       BYTE ADDRESS AND BIT POSITION.
;
OUTBUF  EQU     22H     ;OUTPUT PIN STATE MAP
;
TESTV:  MOV     A,P2
        ANL     A,#00000100B
        JNZ     TESTU
        MOV     A,TCON
        ANL     A,#00100000B
        JZ      TESTX
TESTU:  MOV     A,P1
        ANL     A,#00000010B
        JNZ     SETQ
TESTX:  MOV     A,TCON
        ANL     A,#00001000B
        JZ      TESTZ
        MOV     A,20H
        ANL     A,#00000001B
        JZ      SETQ
TESTZ:  MOV     A,21H
        ANL     A,#00000010B
        JZ      SETQ
CLRQ:   MOV     A,OUTBUF
        ANL     A,#11110111B
        JMP     OUTQ
SETQ:   MOV     A,OUTBUF
        ORL     A,#00001000B
OUTQ:   MOV     OUTBUF,A
        MOV     P3,A
```

Cumbersome, to say the least, and error prone. It would be hard to prove the above example worked in all cases without an exhaustive test.

Each move/mask/conditional jump instruction sequence may be replaced by a single bit-test instruction thanks to direct bit addressing. But the algorithm would be equally convoluted.

Example 28—Software Solution to Logic Function of Figure 20, Using only Bit-Test Instructions

```
;BFUNC2 SOLVE A RANDOM LOGIC FUNCTION OF 6
;       VARIABLES BY DIRECTLY POLLING EACH BIT.
;       (APPROACH USING MCS-51 UNIQUE BIT-TEST
;       INSTRUCTION CAPABILITY.)
;       SYMBOLS USED IN LOGIC DIAGRAM ASSIGNED TO
;       CORRESPONDING 8051 BIT ADDRESSES.
;
U       BIT     P1.1
V       BIT     P2.2
W       BIT     TF0
X       BIT     IE1
Y       BIT     20H.0
Z       BIT     21H.1
Q       BIT     P3.3
;
TEST_V: JB      V,TEST_U
        JNB     W,TEST_X
TEST_U: JB      U,SET_Q
TEST_X: JNB     X,TEST_Z
        JNB     Y,SET_Q
TEST_Z: JNB     Z,SET_Q
CLR_Q:  CLR     Q
        JMP     NXTTST
SET_Q:  SETB    Q
NXTTST:                 ;(CONTINUATION OF PROGRAM)
```

A more elegant and efficient 8051 implementation uses the Boolean ANL and ORL functions to generate the output function using straight-line code. These instructions perform the corresponding logical operations between the carry flag ("Boolean Accumulator") and the addressed bit, leaving the result in the carry. Alternate forms of each instruction (specified in the assembly language by placing a slash before the bit name) use the complement of the bit's state as the input operand.

These instructions may be "strung together" to simulate a multiple input logic gate. When finished, the carry flag contains the result, which may be moved directly to the destination or output pin. No flow chart is needed — it is simple to code directly from the logic diagrams in Figure 20.

Example 29—Software Solution to Logic Function of Figure 20, Using the MCS-51 (TM) Unique Logical Instructions on Boolean Variables

```
;BFUNC3 SOLVE A RANDOM LOGIC FUNCTION OF 6
;       VARIABLES USING STRAIGHT-LINE LOGICAL INSTRUCTIONS
;       ON MCS-51 BOOLEAN VARIABLES.
;
        MOV     C,V
        ORL     C,W     ;OUTPUT OF OR GATE
        ANL     C,U     ;OUTPUT OF TOP AND GATE
        MOV     F0,C    ;SAVE INTERMEDIATE STATE
        MOV     C,X
        ANL     C,/Y    ;OUTPUT OF BOTTOM AND GATE
        ORL     C,F0    ;INCLUDE VALUE SAVED ABOVE
        ORL     C,/Z    ;INCLUDE LAST INPUT VARIABLE
        MOV     Q,C     ;OUTPUT COMPUTED RESULT
```

Simplicity itself. Fast, flexible, reliable, easy to design, and easy to debug.

The Boolean features are useful and unique enough to warrant a complete Application Note of their own. Additional uses and ideas are presented in Application Note AP-70, **Using the Intel® MCS-51® Boolean Processing Capabilities,** publication number 121519.

5. ON-CHIP PERIPHERAL FUNCTION OPERATION AND INTERFACING

I/O Ports

The I/O port versatility results from the "quasi-bidirectional" output structure depicted in Figure 22. (This is effectively the structure of ports 1, 2, and 3 for normal I/O operations. On port 0 resistor R2 is disabled except during multiplexed bus operations, providing

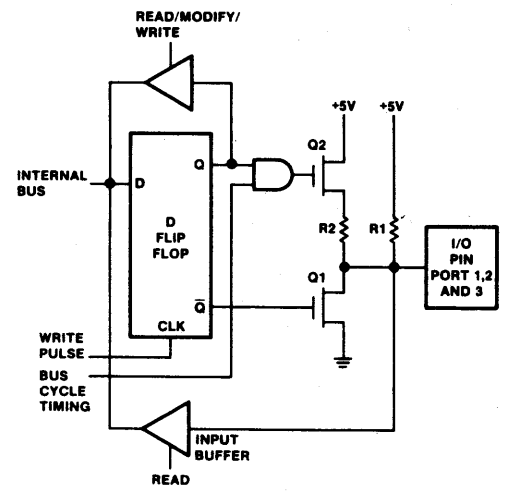


Figure 22. Pseudo-bidirectional I/O port circuitry

AFN-01502A

essentially open-collector outputs. For full electrical characteristics see the User's Manual.)

An output latch bit associated with each pin is updated by direct addressing instructions when that port is the destination. The latch state is buffered to the outside world by R1 and Q1, which may drive a standard TTL input. (In TTL terms, Q1 and R1 resemble an open-collector output with a pull-up resistor to Vcc.)

R2 and Q2 represent an "active pull-up" device enabled momentarily when a 0 previously output changes to a 1. This "jerks" the output pin to a 1 level more quickly than the passive pull-up, improving rise-time significantly if the pin is driving a capacitive load. Note that the active pull-up is **only** activated on 0-to-1 transitions at the output latch (unlike the 8048, in which Q2 is activated whenever a 1 is written out).

Operations using an input port or pin as the source operand use the logic level of the pin itself, rather than the output latch contents. This level is affected by both the microcomputer itself and whatever device the pin is connected to externally. The value read is essentially the "OR-tied" function of Q1 and the external device. If the external device is high-impedence, such as a logic gate input or a three state output in the third state, then reading a pin will reflect the logic level previously output. To use a pin for input, the corresponding output latch must be set. The external device may then drive the pin with either a high or low logic signal. Thus the same port may be used as both input and output by writing ones to all pins used as inputs on output operations, and ignoring all pins used as output on an input operation.

In one operand instructions (INC, DEC, DJNZ and the Boolean CPL) the output latch rather than the input pin level is used as the source data. Similarly, two operand instructions using the port as both one source and the destination (ANL, ORL, XRL) use the output latches. This ensures that latch bits corresponding to pins used as inputs will not be cleared in the process of executing these instructions.

The Boolean operation JBC tests the output latch bit, rather than the input pin, in deciding whether or not to jump. Like the byte-wise logical operations, Boolean operations which modify individual pins of a port leave the other bits of the output latch unchanged.

A good example of how these modes may play together may be taken from the host-processor interface expected by an 8243 I/O expander. Even though the 8051 does not include 8048-type instructions for interfacing with an 8243, the parts can be interconnected (Figure 23) and the protocol may be emulated with simple software.

Example 30 — Mixing Parallel Output, Input, and Control Strobes on Port 2

```
; IN8243 INPUT DATA FROM AN 8243 I/O EXPANDER
;       CONNECTED TO P23-P20
;       P25 & P24 MIMIC CS/ & PROG
;       P27-P26 USED AS INPUTS
;       PORT TO BE READ IN ACC

IN8243  ORL     A,#11010000B
        MOV     P2,A        ;OUTPUT INSTRUCTION CODE
        CLR     P2.4        ;FALLING EDGE OF PROG
        ORL     P2,#00001111B   ;SET FOR INPUT
        MOV     A,P2        ;READ INPUT DATA
        SETB    P2.4        ;RETURN PROG HIGH
        SETB    P2.5        ;DE-SELECT CHIP
```

Serial Port and Timer applications

Configuring the 8051's Serial Port for a given data rate and protocol requires essentially three short sections of software. On power-up or hardware reset the serial port and timer control words must be initialized to the appropriate values. Additional software is also needed in the transmit routine to load the serial port data register and in the receive routine to unload the data as it arrives.

This is best illustrated through an arbitrary example. Assume the 8051 will communicate with a CRT operating at 2400 baud (bits per second). Each character is transmitted as seven data bits, odd parity, and one stop bit. This results in a character rate of 2400/10=240 characters per second.

For the sake of clarity, the transmit and receive subroutines are driven by simple-minded software status polling code rather than interrupts. (It might help to refer back to Figures 7-9 showing the control word formats.) The serial port must be initialized to 8-bit UART mode (M0, M1=01), enabled to receive all messages (M2=0, REN=1). The flag indicating that the transmit register is free for more data will be artificially set in order to let the output software know the output register is available. This can all be set up with one instruction.

Example 31 — Serial Port Mode and Control Bits

```
;SPINIT INITIALIZE SERIAL PORT
;       FOR 8-BIT UART MODE
;       & SET TRANSMIT READY FLAG.
;
SPINIT: MOV     SCON,#01010010B
```

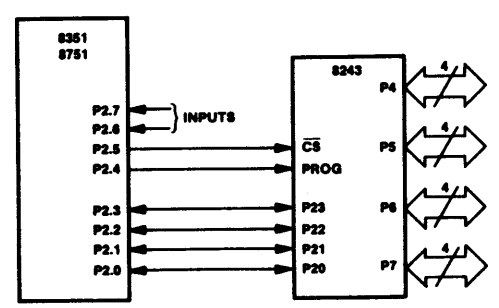


Figure 23. Connecting an 8051 with an 8243 I/O Expander

AFN-01502A

Timer 1 will be used in auto-reload mode as a data rate generator. To achieve a data rate of 2400 baud, the timer must divide the 1 MHz internal clock by 32 x (desired data rate):

$$\frac{1 \times 10^6}{(32)\ (2400)}$$

which equals 13.02 rounded down to 13 instruction cycles. The timer must reload the value -13, or 0F3H. (ASM51 will accept both the signed decimal or hexadecimal representations.)

Example 32—Initializing Timer Mode and Control Bits

```
;T1INIT  INITIALIZE TIMER 1 FOR
;        AUTO-RELOAD AT 32*2400 HZ
;        (TO USED AS GATED 16-BIT COUNTER )
;
T1INIT:  MOV     TCON,#11010010B
         MOV     TH1,#-13
         SETB    TR1
```

A simple subroutine to transmit the character passed to it in the accumulator must first compute the parity bit, insert it into the data byte, wait until the transmitter is available, output the character, and return. This is nearly as easy said as done.

Example 33—Code for UART Output, Adding Parity, Transmitter Loading

```
;SP_OUT  ADD ODD PARITY TO ACC AND
;        TRANSMIT WHEN SERIAL PORT READY.
;
SP_OUT:  MOV     C,P
         CPL     C
         MOV     ACC.7,C
         JNB     TI,$
         CLR     TI
         MOV     SBUF,A
         RET
```

A simple minded routine to wait until a character is received, set the carry flag if there is an odd-parity error, and return the masked seven-bit code in the accumulator is equally short.

Example 34—Code for UART Reception and Parity Verification

```
;SP_IN   INPUT NEXT CHARACTER FROM SERIAL PORT
;        SET CARRY IFF ODD-PARITY ERROR.
;
SP_IN:   JNB     RI,$
         CLR     RI
         MOV     A,SBUF
         MOV     C,P
         CPL     C
         ANL     A,#7FH
         RET
```

6. SUMMARY

This Application Note has described the architecture, instruction set, and on-chip peripheral features of the first three members of the MCS-51™ microcomputer family. The examples used throughout were admittedly (and necessarily) very simple. Additional examples and techniques may be found in the MCS-51™ User's Manual and other application notes written for the MCS-48™ and MCS-51™ families.

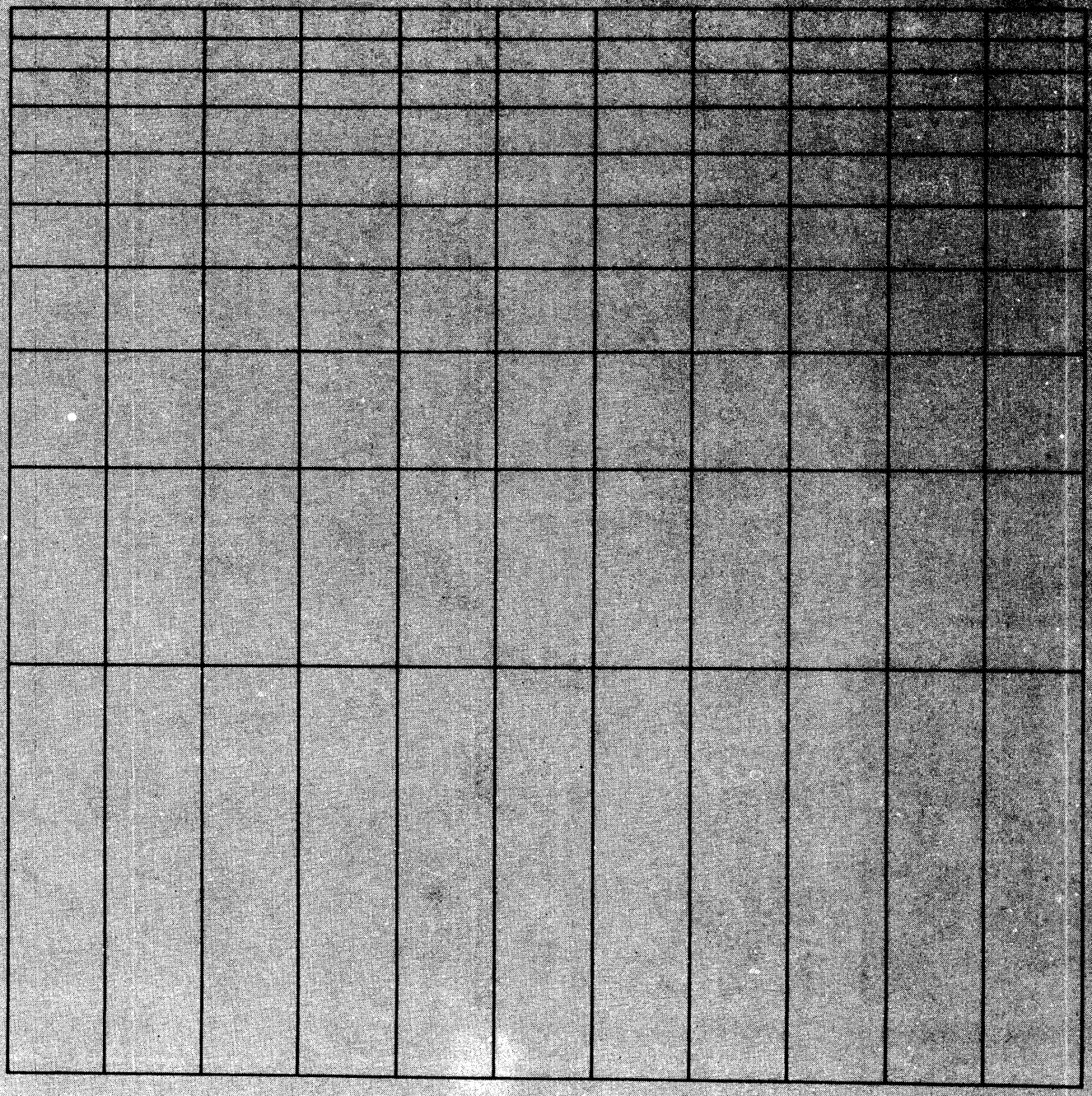
Since its introduction in 1977, the MCS-48™ family has become the industry standard single-chip microcomputer. The MCS-51™ architecture expands the addressing capabilities and instruction set of its predecessor while ensuring flexibility for the future, and maintaining basic software compatability with the past.

Designers already familiar with the 8048 or 8049 will be able to take with them the education and experience gained from past designs as ever-increasing system performance demands force them to move on to state-of-the-art products. Newcomers will find the power and regularity of the 8051 instruction set an advantage in streamlining both the learning and design processes.

Microcomputer system designers will appreciate the 8051 as basically a single-chip solution to many problems which previously required board-level computers. Designers of real-time control systems will find the high execution speed, on-chip peripherals, and interrupt capabilities vital in meeting the timing constraints of products previously requiring discrete logic designs. And designers of industrial controllers will be able to convert ladder diagrams directly from tested-and-true TTL or relay-logic designs to microcomputer software, thanks to the unique Boolean processing capabilities.

It has not been the intent of this note to gloss over the difficulty of designing microcomputer-based systems. To be sure, the hardware and software design aspects of any new computer system are nontrivial tasks. However, the system speed and level of integration of the MCS-51™ microcomputers, the power and flexibility of the instruction set, and the sophisticated assembler and other support products combine to give both the hardware and software designer as much of a head start on the problem as possible.

Appendix C
Using the Intel MCS-51 Boolean Processing Capabilities



1. INTRODUCTION

The Intel microcontroller family now has three new members -- the Intel® 8031, 8051, and 8751 single-chip microcomputers. These devices, shown in Figure 1, will allow whole new classes of products to benefit from recent advances in Integrated Electronics. Thanks to Intel's new HMOS® technology, they provide larger program and data memory spaces, more flexible I/O and peripheral capabilities, greater speed, and lower system cost than any previous-generation single-chip microcomputer.

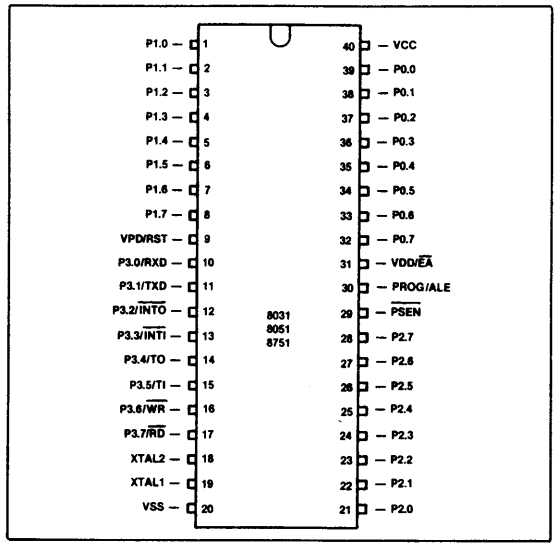


Figure 1. 8051 Family Pinout Diagram.

Table 1 summarizes the quantitative differences between the members of the MCS-48™ and 8051 families. The 8751 contains 4K bytes of EPROM program memory fabricated on-chip, while the 8051 replaces the EPROM with 4K bytes of lower-cost mask-programmed ROM. The 8031 has no program memory on-chip; instead, it accesses up to 64K bytes of program memory from external memory. Otherwise, the three new family members are identical. Throughout this Note, the term "8051" will represent all members of the 8051 Family, unless specifically stated otherwise.

The CPU in each microcomputer is one of the industry's fastest and most efficient for numerical calculations on byte operands. But controllers often deal with bits, not bytes: in the real world, switch contacts can only be open or closed, indicators should be either lit or dark, motors are either turned on or off, and so forth. For such control situations the most significant aspect of the MCS-51™ architecture is its complete hardware support for one-bit, or *Boolean* variables (named in honor of Mathematician George Boole) as a separate data type.

The 8051 incorporates a number of special features which support the direct manipulation and testing of individual bits and allow the use of single-bit variables in performing logical operations. Taken together, these features are referred to as the MCS-51™ *Boolean Processor*. While the bit-processing capabilities alone would be adequate to solve many control applications, their true power comes when they are used in conjunction with the microcomputer's byte-processing and numerical capabilities.

Many concepts embodied by the Boolean Processor will certainly be new even to experienced microcomputer system designers. The purpose of this Application Note is to explain these concepts and show how they are used. It is assumed the reader has read Application Note AP-69, **An Introduction to the Intel® MCS-51™ Single-Chip Microcomputer Family,** publication number 121518, or has been exposed to Intel's single-chip microcomputer product lines.

For detailed information on these parts refer to the **Intel MCS-51™ Family User's Manual**, publication number 121517. The instruction set, assembly language, and use of the 8051 assembler (ASM51) are further described in the **MCS-51™ Macro Assembler User's Guide**, publication number 9800937.

2. BOOLEAN PROCESSOR OPERATION

The Boolean Processing capabilities of the 8051 are based on concepts which have been around for some time. Digital computer systems of widely varying designs all have four functional elements in common (Figure 2):

Table 1. Features of Intel's Single-chip Microcomputers.

EPROM Program Memory	ROM Program Memory	External Program Memory	Program Memory (Int/Max)	Data Memory (Bytes)	Instr. Cycle Time	Input/ Output Pins	Interrupt Sources	Reg. Banks
—	8021	—	1K/1K	64	10 μSec	21	0	1
—	8022	—	2K/2K	64	10 μSec	28	2	1
8748	8048	8035	1K/4K	64	2.5 μSec	27	2	2
—	8049	8039	2K/4K	128	1.36μSec	27	2	2
8751	8051	8031	4K/64K	128	1.0 μSec	32	5	4

01489A

- a central processor (CPU) with the control, timing, and logic circuits needed to execute stored instructions;
- a memory to store the sequence of instructions making up a program or algorithm;
- data memory to store variables used by the program; and
- some means of communicating with the outside world.

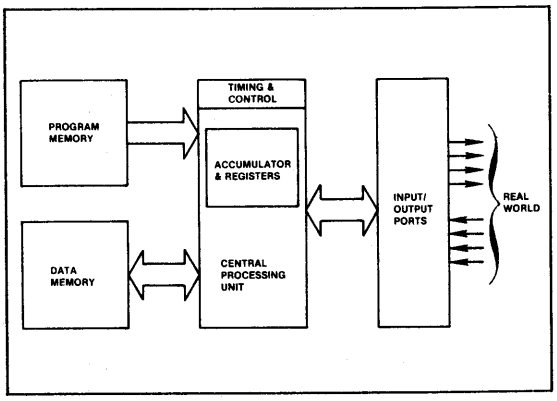


Figure 2. Block Diagram for Abstract Digital Computer.

The CPU usually includes one or more accumulators or special registers for computing or storing values during program execution. The instruction set of such a processor generally includes, at a minimum, operation classes to perform arithmetic or logical functions on program variables, move variables from one place to another, cause program execution to jump or conditionally branch based on register or variable states, and instructions to call and return from subroutines. The program and data memory functions sometimes share a single memory space, but this is not always the case. When the address spaces are separated, program and data memory need not even have the same basic word width.

A digital computer's flexibility comes in part from combining simple fast operations to produce more complex (albeit slower) ones, which in turn link together eventually solving the problem at hand. A four-bit CPU executing multiple precision subroutines can, for example, perform 64-bit addition and subtraction. The subroutines could in turn be building blocks for floating-point multiplication and division routines. Eventually, the four-bit CPU can simulate a far more complex "virtual" machine.

In fact, *any* digital computer with the above four functional elements can (given time) complete *any* algorithm (though the proverbial room full of chimpanzees at word processors might first re-create Shakespeare's classics and this Application Note)! This fact offers little consolation to product designers who want programs to run as quickly as possible. By definition, a real-time control algorithm *must* proceed quickly enough to meet the preordained speed constraints of other equipment.

One of the factors determining how long it will take a microcomputer to complete a given chore is the number of instructions it must execute. What makes a given computer architecture particularly well-or poorly-suited for a class of problems is how well its instruction set matches the tasks to be performed. The better the "primative" operations correspond to the steps taken by the control algorithm, the lower the number of instructions needed, and the quicker the program will run. All else being equal, a CPU supporting 64-bit arithmetic directly could clearly perform floating-point math faster than a machine bogged-down by multiple-precision subroutines. In the same way, direct support for bit manipulation naturally leads to more efficient programs handling the binary input and output conditions inherent in digital control problems.

Processing Elements

The introduction stated that the 8051's bit-handling capabilities alone would be sufficient to solve some control applications. Let's see how the four basic elements of a digital computer - a CPU with associated registers, program memory, addressable data RAM, and I/O capability - relate to Boolean variables.

CPU. The 8051 CPU incorporates special logic devoted to executing several bit-wide operations. All told, there are 17 such instructions, all listed in Table 2. Not shown are 94 other (mostly byte-oriented) 8051 instructions.

Program Memory. Bit-processing instructions are fetched from the same program memory as other arithmetic and logical operations. In addition to the instructions of Table 2, several sophisticated program control features like multiple addressing modes, subroutine nesting, and a two-level interrupt structure are useful in structuring Boolean Processor-based programs.

Boolean instructions are one, two, or three bytes long, depending on what function they perform. Those involving only the carry flag have either a single-byte opcode or an opcode followed by a conditional-branch destination byte (Figure 3.a). The more general instructions add a "direct address" byte after the opcode to specify the bit affected, yielding two or three byte encodings (Figure 3.b). Though this format allows potentially 256 directly addressable bit locations, not all of them are implemented in the 8051 family.

01489A

Table 2. MCS-51™ Boolean Processing Instruction Subset.

Mnemonic		Description	Byte	Cyc
SETB	C	Set Carry flag	1	1
SETB	bit	Set direct Bit	2	1
CLR	C	Clear Carry flag	1	1
CLR	bit	Clear direct bit	2	1
CPL	C	Complement Carry flag	1	1
CPL	bit	Complement direct bit	2	1
MOV	C,bit	Move direct bit to Carry flag	2	1
MOV	bit,C	Move Carry flag to direct bit	2	2
ANL	C,bit	AND direct bit to Carry flag	2	2
ANL	C,/bit	AND complement of direct bit to Carry flag	2	2
ORL	C,bit	OR direct bit to Carry flag	2	2
ORL	C,/bit	OR complement of direct bit to Carry flag	2	2
JC	rel	Jump if Carry is flag is set	2	2
JNC	rel	Jump if No Carry flag	2	2
JB	bit,rel	Jump if direct Bit set	3	2
JNB	bit,rel	Jump if direct Bit Not set	3	2
JBC	bit,rel	Jump if direct Bit is set & Clear bit	3	2

Address mode abbreviations:

C — Carry flag.

bit — 128 software flags, any I/O pin, control or status bit

rel — All conditional jumps include an 8-bit offset byte. Range is +127/-128 bytes relative to first byte of the following instruction.

Data Memory. The instructions in Figure 3.b can operate directly upon 144 general purpose bits forming the Boolean processor "RAM." These bits can be used as sofware flags or to store program variables. Two operand instructions use the CPU's carry flag ("C") as a special one-bit register; in a sense, the carry is a "Boolean accumulator" for logical operations and data transfers.

Input/Output. All 32 I/O pins can be addressed as individual inputs, outputs, or both, in any combination. Any pin can be a control strobe output, status (Test) input, or serial I/O link implemented via software. An additional 33 individually addressable bits reconfigure, control, and monitor the status of the CPU and all on-chip peripheral functions (timer/counters, serial port modes, interrupt logic, and so forth).

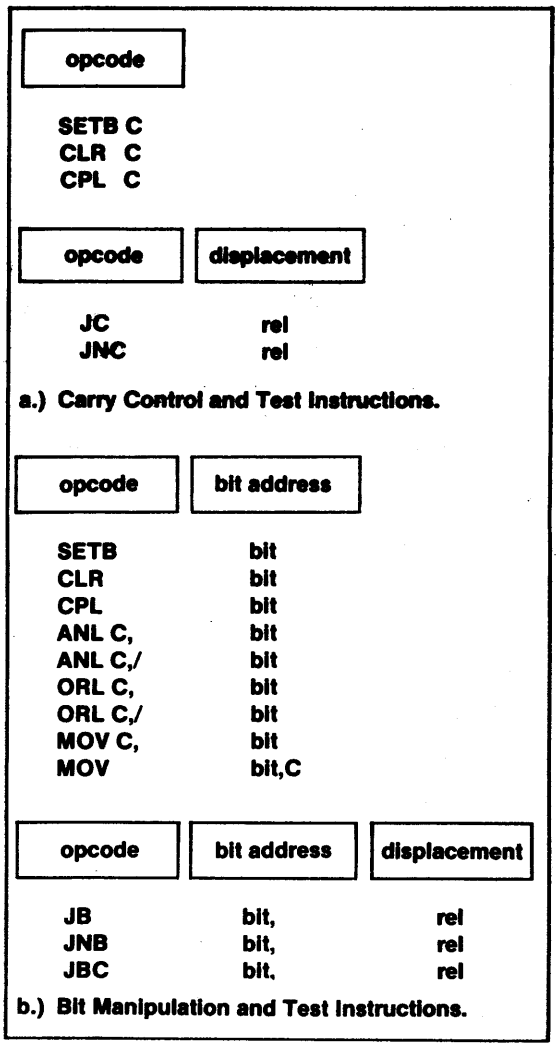


Figure 3. Bit Addressing Instruction Formats.

Direct Bit Addressing

The most significant bit of the direct address byte selects one of two groups of bits. Values between 0 and 127 (00H and 7FH) define bits in a block of 32 bytes of on-chip RAM, between RAM addresses 20H and 2FH (Figure 4.a). They are numbered consecutively from the lowest-order byte's lowest-order bit through the highest-order byte's highest-order bit.

Bit addresses between 128 and 255(80H and 0FFH) correspond to bits in a number of special registers, mostly used for I/O or peripheral control. These positions are numbered with a different scheme than RAM: the five high-order address bits match those of the register's own address, while the three low-order bits identify the bit position within that register (Figure 4.b).

01469A

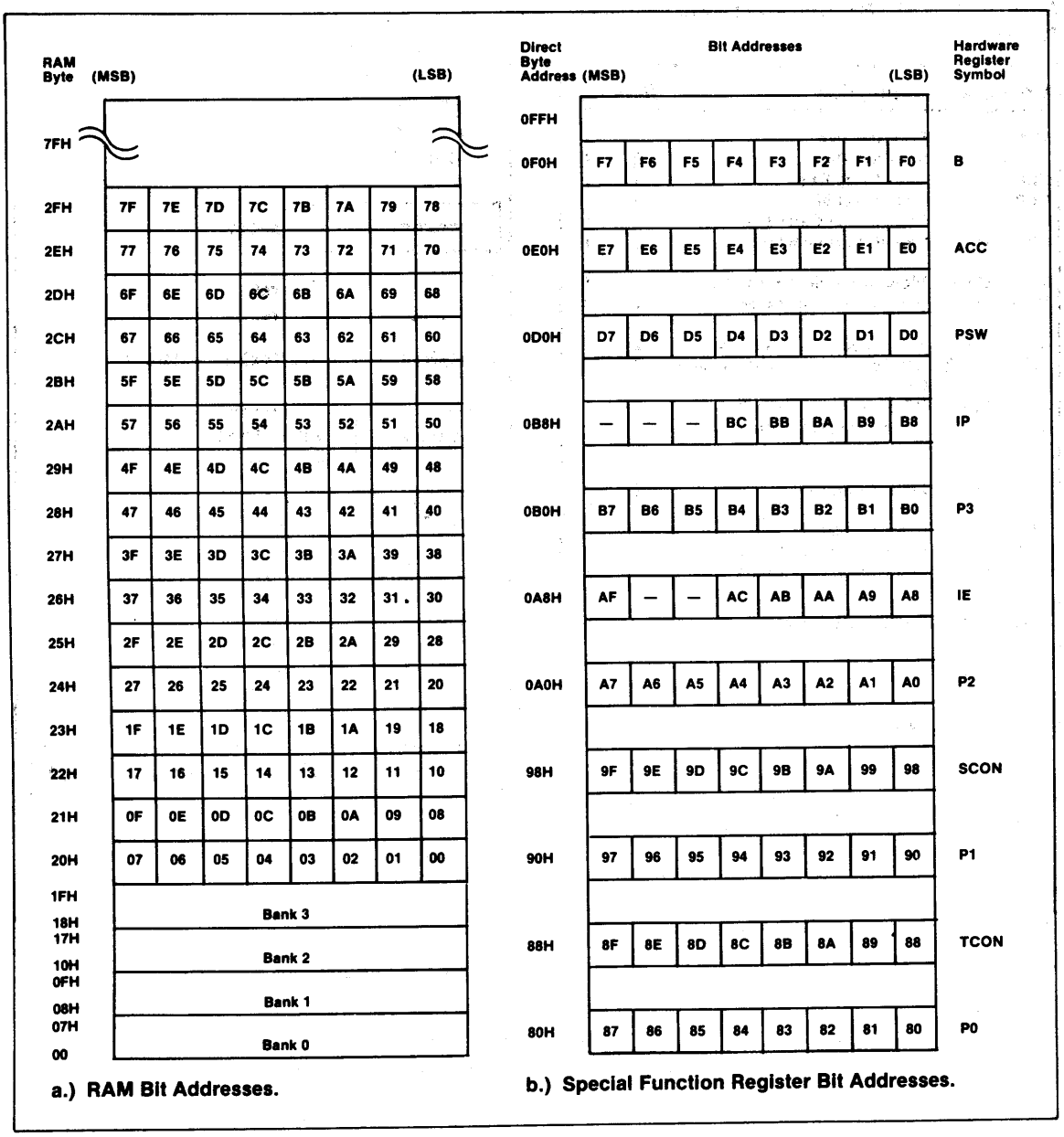


Figure 4. Bit Address Maps.

Notice the column labeled "Symbol" in Figure 5. Bits with special meanings in the PSW and other registers have corresponding symbolic names. General-purpose (as opposed to carry-specific) instructions may access the carry like any other bit by using the mnemonic CY in place of C. P0, P1, P2, and P3 are the 8051's four I/O ports; secondary functions assigned to each of the eight pins of P3 are shown in Figure 6.

Figure 7 shows the last four bit addressable registers. TCON (Timer Control) and SCON (Serial port Control) control and monitor the corresponding peripherals, while IE (Interrupt Enable) and IP (Interrupt Priority) enable and prioritize the five hardware interrupt sources. Like the reserved hardware register addresses, the five bits not implemented in IE and IP should not be accessed; they can *not* be used as software flags.

01489A

(MSB) (LSB)

CY	AC	F0	RS1	RS0	OV	—	P

Symbol	Position	Name and significance
CY	PSW.7	Carry flag. Set/cleared by hardware or software during certain arithmetic and logical instructions.
AC	PSW.6	Auxiliary Carry flag. Set/cleared by hardware during addition or subtraction instructions to indicate carry or borrow out of bit 3.
F0	PSW.5	Flag 0. Set/cleared/tested by software as a user-defined status flag.
RS1	PSW.4	Register bank Select control bits 1 & 0. Set/cleared by software to determine working register bank (see Note).
RS0	PSW.3	
OV	PSW.2	Overflow flag. Set/cleared by hardware during arithmetic instructions to indicate overflow conditions.
—	PSW.1	(reserved)
P	PSW.0	Parity flag. Set/cleared by hardware each instruction cycle to indicate an odd/-even number of "one" bits in the accumulator, i.e., even parity.

Note - the contents of (RS1, RS0) enable the working register banks as follows:

(0,0) - Bank 0	(00H-07H)
(0,1) - Bank 1	(08H-0FH)
(1,0) - Bank 2	(10H-17H)
(1,1) - Bank 3	(18H-1FH)

Figure 5. PSW - Program Status Word organization.

(MSB) (LSB)

RD	WR	T1	T0	INT1	INT0	TXD	RXD

Symbol	Position	Name and significance
RD	P3.7	Read data control output. Active low pulse generated by hardware when external data memory is read.
WR	P3.6	Write data control output. Active low pulse generated by hardware when external data memory is written.
T1	P3.5	Timer/counter 1 external input or test pin.
T0	P3.4	Timer/counter 0 external input or test pin.
INT1	P3.3	Interrupt 1 input pin. Low-level or falling-edge triggered.
INT0	P3.2	Interrupt 0 input pin. Low-level or falling-edge triggered.
TXD	P3.1	Transmit Data pin for serial port in UART mode. Clock output in shift register mode.
RXD	P3.0	Receive Data pin for serial port in UART mode. Data I/O pin in shift register mode.

Figure 6. P3 - Alternate I/O Functions of Port 3.

01489A

(MSB) (LSB)

TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

Symbol Position Name and significance

TF1 TCON.7 Timer 1 overflow Flag. Set by hardware on timer/counter overflow. Cleared when interrupt processed.

TR1 TCON.6 Timer 1 Run control bit. Set/cleared by software to turn timer/counter on/off.

TF0 TCON.5 Timer 0 overflow Flag. Set by hardware on timer/counter overflow. Cleared when interrupt processed.

TR0 TCON.4 Timer 0 Run control bit. Set/cleared by software to turn timer/counter on/off.

IE1 TCON.3 Interrupt 1 Edge flag. Set by hardware when external interrupt edge detected. Cleared when interrupt processed.

IT1 TCON.2 Interrupt 1 Type control bit. Set/cleared by software to specify falling edge/low level triggered external interrupts.

IE0 TCON.1 Interrupt 0 Edge flag. Set by hardware when external interrupt edge detected. Cleared when interrupt processed.

IT0 TCON.0 Interrupt 0 Type control bit. Set/cleared by softrware to specify falling edge/low level triggered external interrupts.

a.) TCON - Timer/Counter Control/status register.

(MSB) (LSB)

SM0	SM1	SM2	REN	TB8	RB8	TI	RI

Symbol Position Name and significance

SM0 SCON.7 Serial port Mode control bit 0. Set/cleared by software (see note).

SM1 SCON.6 Serial port Mode control bit 1. Set/cleared by software (see note).

SM2 SCON.5 Serial port Mode control bit 2. Set by software to disable reception of frames for which bit 8 is zero.

REN SCON.4 Receiver Enable control bit. Set/cleared by software to enable/disable serial data reception.

TB8 SCON.3 Transmit Bit 8. Set/cleared by hardware to determine state of ninth data bit transmitted in 9-bit UART mode.

RB8 SCON.2 Receive Bit 8. Set/cleared by hardware to indicate state of ninth data bit received.

TI SCON.1 Transmit Interrupt flag. Set by hardware when byte transmitted. Cleared by software after servicing.

RI SCON.0 Receive Interrupt flag. Set by hardware when byte received. Cleared by software after servicing.

Note - the state of (SM0,SM1) selects:
(0,0) - Shift register I/O expansion.
(0,1) - 8 bit UART, variable data rate.
(1,0) - 9 bit UART, fixed data rate.
(1,1) - 9 bit UART, variable data rate.

b.) SCON - Serial Port Control/status register.

Figure 7. Peripheral Configuration Registers.

01489A

(MSB) (LSB)

EA	—	—	ES	ET1	EX1	ET1	EX0

Symbol	Position	Name and significance
EA	IE.7	Enable All control bit. Cleared by software to disable all interrupts, independent of the state of IE.4 - IE.0.
—	IE.6	(reserved)
—	IE.5	
ES	IE.4	Enable Serial port control bit. Set/cleared by software to enable/ disable interrupts from TI or RI flags.
ET1	IE.3	Enable Timer 1 control bit. Set/cleared by software to enable/ disable interrupts from timer/counter 1.
EX1	IE.2	Enable External interrupt 1 control bit. Set/cleared by software to enable/disable interrupts from INT1.
ET0	IE.1	Enable Timer 0 control bit. Set/cleared by software to enable/ disable interrupts from timer/counter 0.
EX0	IE.0	Enable External interrupt 0 control bit. Set/cleared by software to enable/disable interrupts from INT0.

c.) IE - Interrupt Enable Register.

(MSB) (LSB)

—	—	—	PS	PT1	PX1	PT0	PX0

Symbol	Position	Name and significance
—	IP.7	(reserved)
—	IP.6	(reserved)
—	IP.5	(reserved)
PS	IP.4	Serial port Priority control bit. Set/cleared by software to specify high/low priority interrupts for Serial port.
PT1	IP.3	Timer 1 Priority control bit. Set/cleared by software to specify high/low priority interrupts for timer/counter 1.
PX1	IP.2	External interrupt 1 Priority control bit. Set/cleared by software to specify high/low priority interrupts for INT1.
PT0	IP.1	Timer 0 Priority control bit. Set/cleared by software to specify high/low priority interrupts for timer/counter 0.
PX0	IP.0	External interrupt 0 Priority control bit. Set/cleared by software to specify high/low priority interrupts for INT0.

d.) IP - Interrupt Priority Control Register.

Figure 7. (continued)

Addressable Register Set. There are 20 special function registers in the 8051, but the advantages of bit addressing only relate to the 11 described below. Five potentially bit-addressable register addresses (0C0H, 0C8H, 0D8H, 0E8H, & 0F8H) are being reserved for possible future expansion in microcomputers based on the MCS-51™ architecture. Reading or writing non-existent registers in the 8051 series is pointless, and may cause unpredictable results. Byte-wide logical operations can be used to manipulate bits in all *non*-bit addressable registers and RAM.

The accumulator and B registers (A and B) are normally involved in byte-wide arithmetic, but their individual bits can also be used as 16 general software flags. Added with the 128 flags in RAM, this gives 144 general purpose variables for bit-intensive programs. The program status word (PSW) in Figure 5 is a collection of flags and machine status bits including the carry flag itself. Byte operations acting on the PSW can therefore affect the carry.

01489A

Instruction Set

Having looked at the bit variables available to the Boolean Processor, we will now look at the four classes of instructions that manipulate these bits. It may be helpful to refer back to Table 2 while reading this section.

State Control. Addressable bits or flags may be set, cleared, or logically complemented in one instruction cycle with the two-byte instructions SETB, CLR, and CPL. (The "B" affixed to SETB distinguishes it from the assembler "SET" directive used for symbol definition.) SETB and CLR are analogous to loading a bit with a constant: 1 or 0. Single byte versions perform the same three operations on the carry.

The MCS-51™ assembly language specifies a bit address in any of three ways:

- by a number or expression corresponding to the direct bit address (0-255);
- by the name or address of the register containing the bit, the *dot operator* symbol (a period: "."), and the bit's position in the register (7-0);
- in the case of control and status registers, by the predefined assembler symbols listed in the first columns of Figures 5-7.

Bits may also be given user-defined names with the assembler "BIT" directive and any of the above techniques. For example, bit 5 of the PSW may be cleared by any of the four instructions,

```
USR_FLG BIT   PSW.5     ; User Symbol Definition
;         ...   .....
          CLR   0D5H    ; Absolute Addressing
          CLR   PSW.5   ; Use of Dot Operator
          CLR   F0      ; Pre-Defined Assembler
                        ;              Symbol
          CLR   USR_FLG ; User-Defined Symbol
```

Data Transfers. The two-byte MOV instructions can transport any addressable bit to the carry in one cycle, or copy the carry to the bit in two cycles. A bit can be moved between two arbitrary locations via the carry by combining the two instructions. (If necessary, push and pop the PSW to preserve the previous contents of the carry.) These instructions can replace the multi-instruction sequence of Figure 8, a program structure appearing in controller applications whenever flags or outputs are conditionally switched on or off.

Logical Operations. Four instructions perform the logical-AND and logical-OR operations between the carry and another bit, and leave the results in the carry. The instruction mnemonics are ANL and ORL; the absence or presence of a

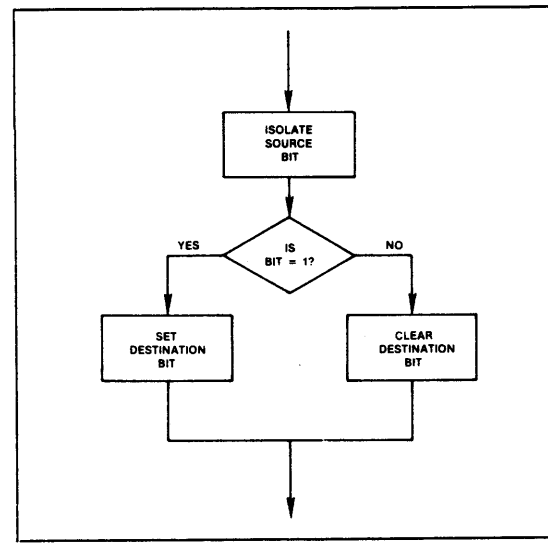


Figure 8. Bit Transfer Instruction Operation.

slash mark ("/") before the source operand indicates whether to use the positive-logic value or the logical complement of the addressed bit. (The source operand itself is never affected.)

Bit-test Instructions. The conditional jump instructions "JC rel" (Jump on Carry) and "JNC rel" (Jump on Not Carry) test the state of the carry flag, branching if it is a one or zero, respectively. (The letters "rel" denote relative code addressing.) The three-byte instructions "JB bit, rel" and "JNB bit,rel" (Jump on Bit and Jump on Not Bit) test the state of *any* addressable bit in a similar manner. A fifth instruction combines the Jump on Bit and Clear operations. "JBC bit,rel" conditionally branches to the indicated address, then clears the bit in the same two cycle instruction. This operation is the same as the MCS-48™ "JTF" instructions.

All 8051 conditional jump instructions use program counter-relative addressing, and all execute in two cycles. The last instruction byte encodes a signed displacement ranging from -128 to +127. During execution, the CPU adds this value to the incremented program counter to produce the jump destination. Put another way, a conditional jump to the immediately following instruction would encode 00H in the offset byte.

A section of program or subroutine written using only relative jumps to nearby addresses will have the same machine code independent of the code's location. An assembled routine may be repositioned anywhere in memory, even crossing memory page boundaries, without having to modify the program or recompute destination addresses. To facilitate this flexibility, there is an unconditional "Short Jump" (SJMP) which uses relative addressing as well. Since a pro-

01489A

grammer would have quite a chore trying to compute relative offset values from one instruction to another, ASM51 automatically computes the displacement needed given only the destination address or label. An error message will alert the programmer if the destination is "out of range."

(The so-called "Bit Test" instructions implemented on many other microprocessors simply perform the logical-AND operation between a byte variable and a constant mask, and set or clear a zero flag depending on the result. This is essentially equivalent to the 8051 "MOV C,bit" instruction. A second instruction is then needed to conditionally branch based on the state of the zero flag. This does *not* constitute abstract bit-addressing in the MCS-51™ sense. A flag exists only as a field within a register; to reference a bit the programmer must know and specify both the encompassing register and the bit's position therein. This constraint severely limits the flexibility of symbolic bit addressing and reduces the machine's code-efficiency and speed.)

Interaction with Other Instructions. The carry flag is also affected by the instructions listed in Table 3. It can be rotated through the accumulator, and altered as a side effect of arithmetic instructions. Refer to the User's Manual for details on how these instructions operate.

Simple Instruction Combinations

By combining general purpose bit operations with certain addressable bits, one can "custom build" several hundred useful instructions. All eight bits of the PSW can be tested directly with conditional jump instructions to monitor (among other things) parity and overflow status. Programmers can take advantage of 128 software flags to keep track of operating modes, resource usage, and so forth.

The Boolean instructions are also the most efficient way to control or reconfigure peripheral and I/O registers. All 32 I/O lines become "test pins," for example, tested by conditional jump instructions. Any output pin can be toggled (complemented) in a single instruction cycle. Setting or clearing the Timer Run flags (TR0 and TR1) turn the timer/counters on or off; polling the same flags elsewhere lets the program determine if a timer is running. The respective overflow flags (TF0 and TF1) can be tested to determine when the desired period or count has elapsed, then cleared in preparation for the next repetition. (For the record, these bits are all part of the TCON register, Figure 7.a. Thanks to symbolic bit addressing, the programmer only needs to remember the mnemonic associated with each function. In other words, don't bother memorizing control word layouts.)

In the MCS-48® family, instructions corresponding to some of the above functions require specific opcodes. Ten different opcodes serve to clear/complement the software flags F0 and F1, enable/disable each interrupt, and start/stop the timer. In the 8051 instruction set, just three opcodes (SETB,

Table 3. Other Instructions Affecting the Carry Flag.

Mnemonic	Description	Byte	Cyc
ADD A,Rn	Add register to Accumulator	1	1
ADD A,direct	Add direct byte to Accumulator	2	1
ADD A,@Ri	Add indirect RAM to Accumulator	1	1
ADD A,#data	Add immediate data to Accumulator	2	1
ADDC A,Rn	Add register to Accumulator with Carry flag	1	1
ADDC A, direct	Add direct byte to Accumulator with Carry flag	2	1
ADDC A,@Ri	Add indirect RAM to Accumulator with Carry flag	1	1
ADDC A,#data	Add immediate data to Acc with Carry flag	2	1
SUBB A,Rn	Subtract register from Accumulator with borrow	1	1
SUBB A,direct	Subtract direct byte from Acc with borrow	2	1
SUBB A,@Ri	Subtract indirect RAM from Acc with borrow	1	1
SUBB A,#data	Subtract immediate data from Acc with borrow	2	1
MUL AB	Multiply A & B	1	4
DIV AB	Divide A by B	1	4
DA A	Decimal Adjust Accumulator	1	1
RLC A	Rotate Accumulator Left through the Carry flag	1	1
RRC A	Rotate Accumulator Right through Carry flag	1	1
CJNE A,direct,rel	Compare direct byte to Acc & Jump if Not Equal	3	2
CJNE A,#data,rel	Compare immediate to Acc & Jump if Not Equal	3	2
CJNE Rn,#data,rel	Compare immed to register & Jump if Not Equal	3	2
CJNE @Ri,#data,rel	Compare immed to indirect & Jump if Not Equal	3	2

CLR, CPL) with a direct bit address appended perform the same functions. Two test instructions (JB and JNB) can be combined with bit addresses to test the software flags, the 8048 I/O pins T0, T1, and INT, and the eight accumulator bits, replacing 15 more different instructions.

Table 4.a shows how 8051 programs implement software flag and machine control functions associated with special

using awkward sequences of other basic operations. As mentioned earlier, any CPU can solve any problem given enough time.

Quantitatively, the differences between a solution allowed by the 8051 and those required by previous architectures are numerous. What the 8051 Family buys you is a faster, cleaner, lower-cost solution to microcontroller applications.

The opcode space freed by condensing many specific 8048

Table 4.a. Contrasting 8048 and 8051 Bit Control and Testing Instructions.

8048 Instruction	Bytes	Cycles	uSec	8x51 Instruction		Bytes	Cycles & uSec
Flag Control							
CLR C	1	1	2.5	CLR	C	1	1
CPL F0	1	1	2.5	CPL	F0	2	1
Flag Testing							
JNC offset	2	2	5.0	JNC	rel	2	2
JF0 offset	2	2	5.0	JB	F0,rel	3	2
JB7 offset	2	2	5.0	JB	ACC.7,rel	3	2
Peripheral Polling							
JT0 offset	2	2	5.0	JB	T0,rel	3	2
JNI offset	2	2	5.0	JNB	INT0,rel	3	2
JTF offset	2	2	5.0	JBC	TF0,rel	3	2
Machine and Peripheral Control							
STRT T	1	1	2.5	SETB	TR0	2	1
EN I	1	1	2.5	SETB	EX0	2	1
DIS TCNTI	1	1	2.5	CLR	ET0	2	1

Table 4.b. Replacing 8048 Instruction sequences with single 8x51 Instructions.

8048 Instructions	Bytes	Cycles	uSec	8051 Instructions	Bytes	Cycles & uSec
Flag Control Set carry: CLR C CPL C	= 2	2	5.0	SETB C	1	1
Set Software Flag: CLR F0 CPL F0	= 2	2	5.0	SETB F0	2	1

opcodes in the 8048. In every case the MCS-51™ solution requires the same number of machine cycles, and executes 2.5 times faster.

3. BOOLEAN PROCESSOR APPLICATIONS

So what? Then what does all this buy you?

Qualitatively, nothing. All the same capabilities *could* be (and often have been) implemented on other machines

instructions into a few general operations has been used to add new functionality to the MCS-51™ architecture - both for byte and bit operations. 144 software flags replace the 8048's two. These flags (and the carry) may be directly set, not just cleared and complemented, and all can be tested for either state, not just one. Operating mode bits previously inaccessible may be read, tested, or saved. Situations where the 8051 instruction set provides new capabilities are contrasted with 8048 instruction sequences in Table 4.b. Here the 8051 speed advantage ranges from 5x to 15x!

01489A

Table 4b. (Continued)

8048 Instructions		Bytes	Cycles	uSec	8x51 Instructions	Bytes	Cycles & uSec
Turn Off Output Pin: ANL P1,#0FBH	=	2	2	5.0	CLR P1.2	2	1
Complement Output Pin: IN A,P1 XRL A,#04H OUTL P1,A	=	4	6	15.0	CPL P1.2	2	1

Clear Flag in RAM:
MOV R0,#FLGADR
MOV A,@R0
ANL A,#FLGMASK
MOV @R0,A
	=	6	6	15.0	CLR USER_FLG	2	1

Flag Testing Jump if Software Flag is 0: JF0 $+4 JMP offset	=	4	4	10.0	JNB F0,rel	3	2
Jump if Accumulator bit is 0: CPL A JB7 offset CPL A	=	4	4	10.0	JNB ACC.7,rel	3	2
Peripheral Polling Test if Input Pin is Grounded: IN A,P1 CPL A JB3 offset	=	4	5	12.5	JNB P1.3,rel	3	2
Test if Interrupt Pin is High: JNI $+4 JMP offset	=	4	4	10.0	JB INT0,rel	3	2

Combining Boolean and byte-wide instructions can produce great synergy. An MCS-51™ based application will prove to be:

- simpler to write since the architecture correlates more closely with the problems being solved;
- easier to debug because more individual instructions have no unexpected or undesirable side-effects;
- more byte efficient due to direct bit addressing and program counter relative branching;
- faster running because fewer bytes of instruction need to be fetched and fewer conditional jumps are processed;
- lower cost because of the high level of system-intergration within one component.

These rather unabashed claims of excellence shall not go unsubstantiated. The rest of this chapter examines less trivial tasks simplified by the Boolean processor. The first three compare the 8051 with other microprocessors; the last two go into 8051-based system designs in much greater depth.

Design Example #1 - Bit Permutation

First off, we'll use the bit-transfer instructions to permute a lengthy pattern of bits.

A steadily increasing number of data communication products use encoding methods to protect the security of sensitive information. By law, interstate financial transactions involving the Federal banking system must be transmitted using the Federal Information Processing *Data Encryption Standard* (DES).

Basically, the DES combines eight bytes of "plaintext" data (in binary, ASCII, or any other format) with a 56-bit "key", producing a 64-bit encrypted value for transmission. At the receiving end the same algorithm is applied to the incoming data using the same key, reproducing the original eight byte message. The algorithm used for these permutations is fixed; different user-defined keys ensure data privacy.

01489A

It is not the purpose of this note to describe the DES in any detail. Suffice it to say that encryption/decryption is a long, iterative process consisting of rotations, exclusive -OR operations, function table look-ups, and an extensive (and quite bizarre) sequence of bit permutation, packing, and unpacking steps. (For further details refer to the June 21, 1979 issue of **Electronics** magazine.) The bit manipulation steps are included, it is rumored, to impede a general purpose digital supercomputer trying to "break" the code. Any algorithm implementing the DES with previous generation microprocessors would spend virtually all of its time diddling bits.

The bit manipulation performed is typified by the Key Schedule Calculation represented in Figure 9. This step is repeated 16 times for each key used in the course of a transmission. In essence, a seven-byte, 56-bit "Shifted Key Buffer" is transformed into an eight-byte, "Permutation Buffer" without altering the shifted Key. The arrows in Figure 9 indicate a few of the translation steps. Only six bits of each byte of the Permutation Buffer are used; the two high-order bits of each byte are cleared. This means only 48 of the 56 Shifted Key Buffer bits are used in any one iteration.

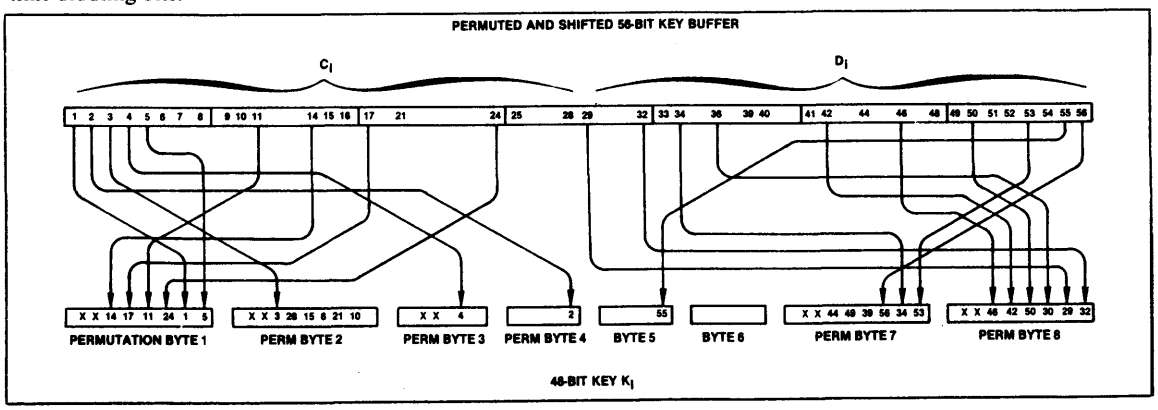


Figure 9. DES Key Schedule Transformation.

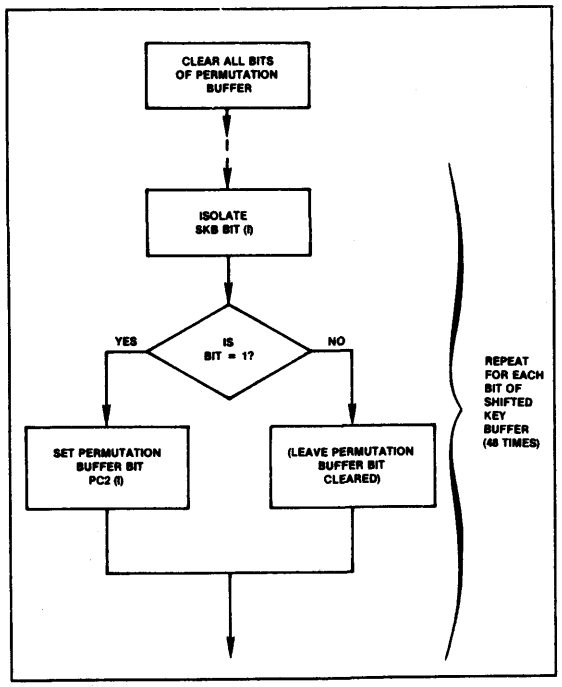


Figure 10.a. Flowchart for Key permutation attempted with a byte processor.

Different microprocessor architectures would best implement this type of permutation in different ways. Most approaches would share the steps of Figure 10.a:

- Initialize the Permutation Buffer to default state (ones or zeroes);
- Isolate the state of a bit of a byte from the Key Buffer. Depending on the CPU, this might be accomplished by rotating a word of the Key Buffer through a carry flag or testing a bit in memory or an accumulator against a mask byte;
- Perform a conditional jump based on the carry or zero flag if the Permutation Buffer default state is correct;
- Otherwise reverse the corresponding bit in the permutation buffer with logical operations and mask bytes.

Each step above may require several instructions. The last three steps must be repeated for all 48 bits. Most microprocessors would spend 300 to 3,000 microseconds on each of the 16 iterations.

Notice, though, that this flow chart looks a lot like Figure 8. The Boolean Processor can permute bits by simply moving

01489A

them from the source to the carry to the destination—a total of two instructions taking four bytes and three microseconds per bit. Assume the Shifted Key Buffer and Permutation Buffer both reside in bit-addressable RAM, with the bits of the former assigned symbolic names SKB_1, SKB_2, . . . SKB_56, and that the bytes of the latter are named PB_1, . . . PB_8. Then working from Figure 9, the software for the permutation algorithm would be that of Example 1.a. The total routine length would be 192 bytes, requiring 144 microseconds.

The algorithm of Figure 10.b is just slightly more efficient in this time-critical application and illustrates the synergy of an integrated byte and bit processor. The bits needed for each byte of the Permutation Buffer are assimilated by loading each bit into the carry (1 usec.) and shifting it into the accumulator (1 usec.). Each byte is stored in RAM when completed. Forty-eight bits thus need a total of 112 instructions, some of which are listed in Example 1.b.

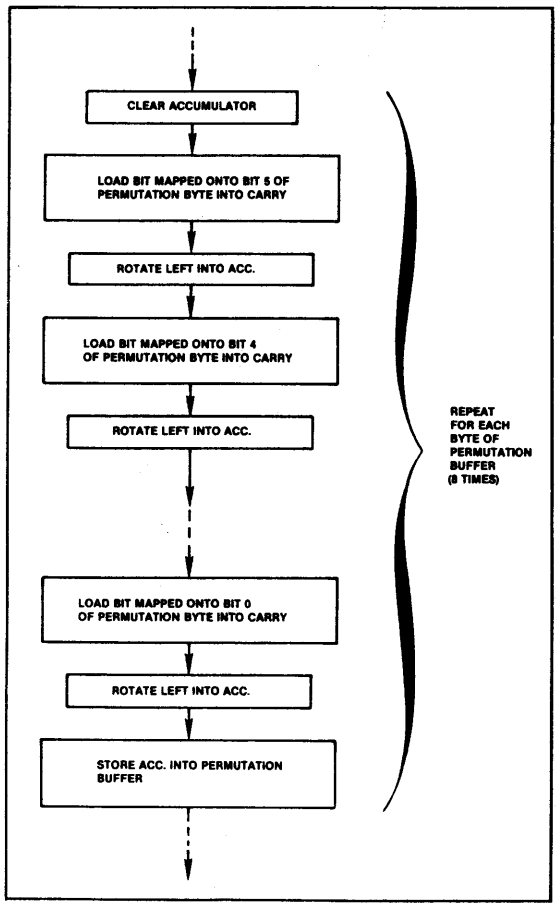


Figure 10.b. DES Key Permutation with Boolean Processor.

Worst-case execution time would be 112 microseconds, since each instruction takes a single cycle. Routine length would also decrease, to 168 bytes. (Actually, in the context of the complete encryption algorithm, each permuted byte would be processed as soon as it is assimilated—saving memory and cutting execution time by another 8 usec.)

Example 1. DES Key Permutation Software.

a.) "Brute Force" technique.

```
        MOV     C,SKB_1
        MOV     PB_1.1,C
        MOV     C,SKB_2
        MOV     PB_4.0,C
        MOV     C,SKB_3
        MOV     PB_2.5,C
        MOV     C,SKB_4
        MOV     PB_1.0,C
;       ...     .....
;       ...     .....
        MOV     C,SKB_55
        MOV     PB_5.0,C
        MOV     C,SKB_56
        MOV     PB_7.2,C
```

b.) Using Accumulator to Collect Bits.

```
        CLR     A
        MOV     C,SKB_14
        RLC     A
        MOV     C,SKB_17
        RLC     A
        MOV     C,SKB_11
        RLC     A
        MOV     C,SKB_24
        RLC     A
        MOV     C,SKB_1
        RLC     A
        MOV     C,SKB_5
        RLC     A
        MOV     PB_1,A
;       ...     .....
;       ...     .....
        MOV     C,SKB_29
        RLC     A
        MOV     C,SKB_32
        RLC     A
        MOV     PB_8,A
```

To date, most banking terminals and other systems using the DES have needed special boards or peripheral controller chips just for the encryption/decryption process, and still more hardware to form a serial bit stream for transmission (Figure 11.a). An 8051 solution could pack most of the entire system onto the one chip (Figure 11.b). The whole DES algorithm would require less than one-

01489A

fourth of the on-chip program memory, with the remaining bytes free for operating the banking terminal (or whatever) itself.

Moreover, since transmission and reception of data is performed through the on-board UART, the unencrypted data (plaintext) never even exists outside the microcomputer! Naturally, this would afford a high degree of security from data interception.

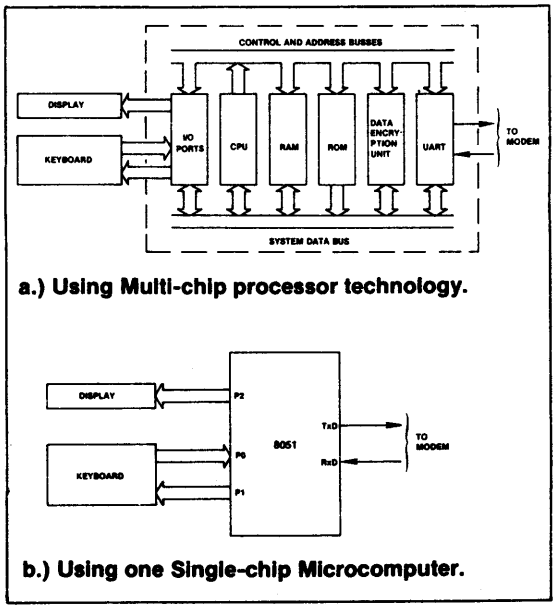


Figure 11. Secure Banking Terminal Block Diagram.

Design Example #2 - Software Serial I/O

An exercise often imposed on beginning microcomputer students is to write a program simulating a UART. (See, for example, Application Notes AP24, AP29, and AP49.) Though doing this with the 8051 Family may appear to be a moot point (given that the hardware for a full UART is on-chip), it is still instructive to see how it would be done, and maintains a product line tradition.

As it turns out, the 8051 microcomputers can receive or transmit serial data via software very efficiently using the Boolean instruction set. Since any I/O pin may be a serial input or output, several serial links could be maintained at once.

Figures 12.a and 12.b show algorithms for receiving or transmitting a byte of data. (Another section of program would invoke this algorithm eight times, synchronizing it with a start bit, clock signal, software delay, or timer interrupt.) Data is received by testing an input pin, setting the carry to the same state, shifting the carry into a data buffer, and saving the partial frame in internal RAM. Data is transmitted by shifting an output buffer through the carry, and generating each bit on an output pin.

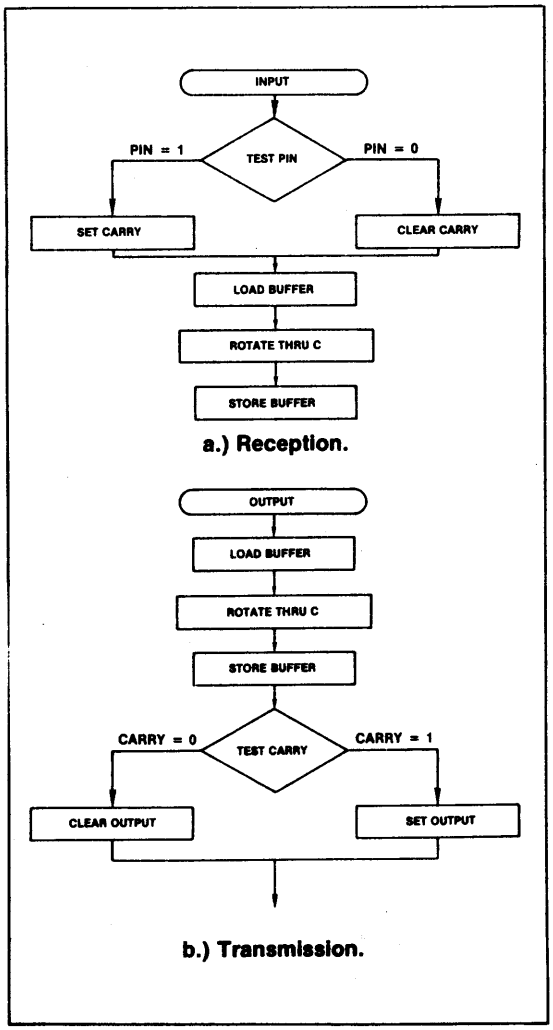


Figure 12. Serial I/O Algorithms.

A side-by-side comparison of the software for this common "bit-banging" application with three different microprocessor architectures is shown in Table 5.a and 5.b. The 8051 solution is more efficient than the others on every count!

01489A

Table 5. Serial I/O Programs for Various Microprocessors.

a.) Input Routine.

8085		8048		8051	
IN	SERPORT			MOV	C,SERPIN
ANI	MASK	CLR	C		
JZ	LO	JNT0	LO		
CMC		CPL	C		
LO: LXI	HL,SERBUF	MOV	R0,#SERBUF		
MOV	A,M	MOV	A,@R0	MOV	A,SERBUF
RR		RRC	A	RRC	A
MOV	M,A	MOV	@R0,A	MOV	SERBUF,A
RESULTS:					
8 INSTRUCTIONS		7 INSTRUCTIONS		4 INSTRUCTIONS	
14 BYTES		9 BYTES		7 BYTES	
56 STATES		9 CYCLES		4 CYCLES	
19 uSEC.		22.5 uSEC.		4 uSEC.	

b.) Output Routine.

8085		8048		8051	
LXI	HL,SERBUF	MOV	R0,#SERBUF		
MOV	A,M	MOV	A,@R0	MOV	A,SERBUF
RR		RRC	A	RRC	A
MOV	M,A	MOV	@R0,A	MOV	SERBUF,A
IN	SERPORT				
JC	HI	JC	HI		
LO: ANI	NOT MASK	ANL	SERPRT,#NOT MASK	MOV	SERPIN,C
JMP	CNT	JMP	CNT		
HI: ORI	MASK	HI: ORL	SERPRT,#MASK		
CNT: OUT	SERPORT	CNT:			
RESULTS:					
10 INSTRUCTIONS		8 INSTRUCTIONS		4 INSTRUCTIONS	
20 BYTES		13 BYTES		7 BYTES	
72 STATES		11 CYCLES		5 CYCLES	
24 uSEC.		27.5 uSEC.		5 uSEC.	

Design Example #3 - Combinatorial Logic Equations

Next we'll look at some simple uses for bit-test instructions and logical operations. (This example is also presented in Application Note AP-69.)

Virtually all hardware designers have solved complex functions using combinatorial logic. While the hardware involved may vary from relay logic, vacuum tubes, or TTL or to more esoteric technologies like fluidics, in each case the goal is the same: to solve a problem represented by a logical function of several Boolean variables.

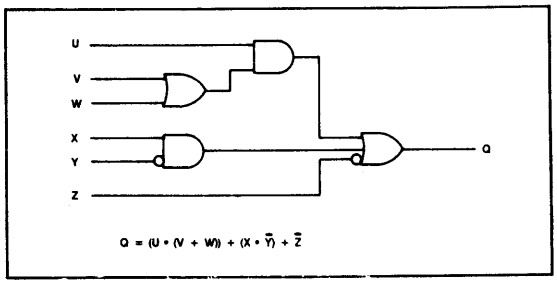
Figure 13 shows TTL and relay logic diagrams for a function of the six variables U through Z. Each is a solution of the equation,

$$Q = (U \cdot (V + W)) + (X \cdot \overline{Y}) + \overline{Z} \quad .$$

Equations of this sort might be reduced using Karnaugh Maps or algebraic techniques, but that is not the purpose of this example. As the logic complexity increases, so does the difficulty of the reduction process. Even a minor change to the function equations as the design evolves would require tedious re-reduction from scratch.

01489A

Figure 13. Hardware Implementations of Boolean functions.



a.) Using TTL:

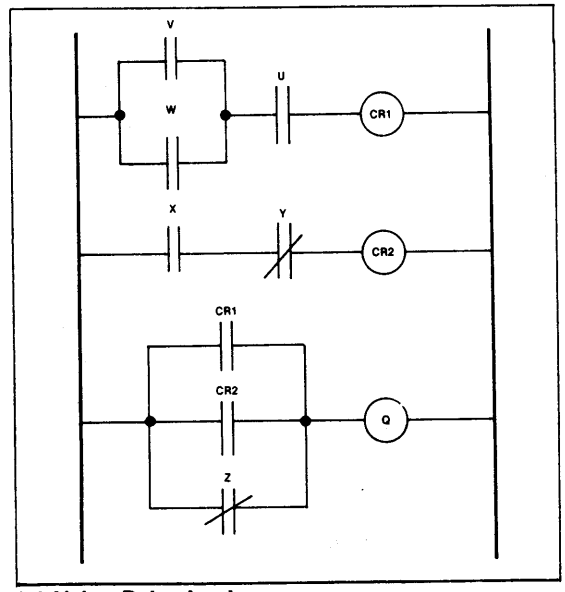
For the sake of comparison we will implement this function three ways, restricting the software to three proper subsets of the MCS-51™ instruction set. We will also assume that U and V are input pins from different input ports, W and X are status bits for two peripheral controllers, and Y and Z are software flags set up earlier in the program. The end result must be written to an output pin on some third port. The first two implementations follow the flow-chart shown in Figure 14. Program flow would embark on a route down a test-and-branch tree and leaves either the "True" or "Not True" exit ASAP — as soon as the proper result has been determined. These exits then rewrite the output port with the result bit respectively one or zero.

Other digital computers must solve equations of this type with standard word-wide logical instructions and conditional jumps. So for the first implementation, we won't use any generalized bit-addressing instructions. As we shall soon see, being constrained to such an instruction subset produces somewhat sloppy software solutions. MCS-51™ mnemonics are used in Example 2.a; other machines might further cloud the situation by requiring operation-specific mnemonics like INPUT, OUTPUT, LOAD, STORE, etc., instead of the MOV mnemonic used for all variable transfers in the 8051 instruction set.

The code which results is cumbersome and error prone. It would be difficult to prove whether the software worked for all input combinations in programs of this sort. Furthermore, execution time will vary widely with input data.

Thanks to the direct bit-test operations, a single instruction can replace each move / mask / conditional jump sequence in Example 2.a, but the algorithm would be equally convoluted (see Example 2.B). To lessen the confusion "a bit" each input variable is assigned a symbolic name.

A more elegant and efficient implementation (Example 2.c) strings together the Boolean ANL and ORL functions to generate the output function with straight-line code.



b.) Using Relay Logic:

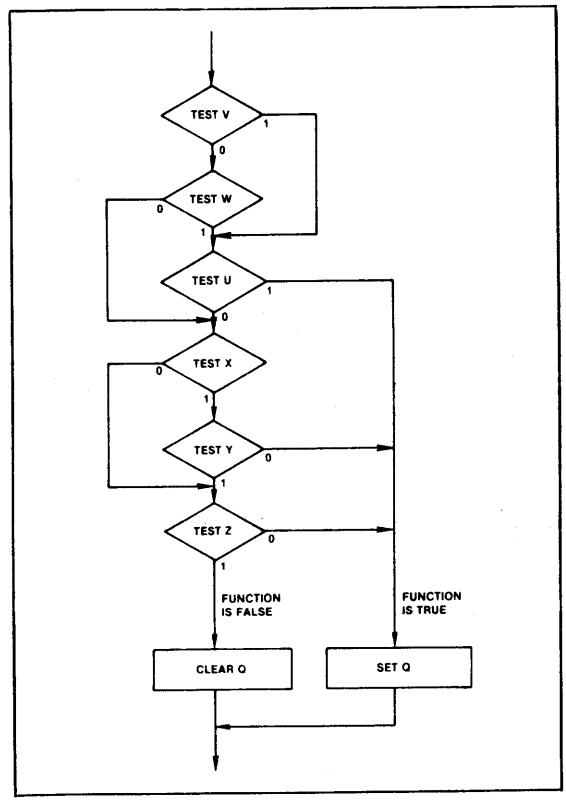


Figure 14. Flow chart for tree-branching algorithm.

01489A

When finished, the carry flag contains the result, which is simply copied out to the destination pin. No flow chart is needed—code can be written directly from the logic diagrams in Figure 14. The result is simplicity itself: fast, flexible, reliable, easy to design, and easy to debug.

An 8051 program can simulate an N-input AND or OR gate with at most N+1 lines of source program—one for each input and one line to store the results. To simulate NAND and NOR gates, complement the carry after computing the function. When some inputs to the gate have "inversion bubbles," perform the ANL or ORL operation on inverted operands. When the first input is inverted, either load the operand into the carry and then complement it, or use DeMorgan's Theorem to convert the gate to a different form.

Example 2. Software Solutions to Logic Function of Figure 13.

a.) Using only byte-wide logical instructions.

```
;BFUNC1 SOLVE RANDOM LOGIC FUNCTION
;       OF 6 VARIABLES BY LOADING AND
;       MASKING THE APPROPRIATE BITS
;       IN THE ACCUMULATOR, THEN
;       EXECUTING CONDITIONAL JUMPS
;       BASED ON ZERO CONDITION.
;       (APPROACH USED BY BYTE-
;       ORIENTED ARCHITECTURES.)
;       BYTE AND MASK VALUES
;       CORRESPOND TO RESPECTIVE BYTE
;       ADDRESS AND BIT POSITIONS.
;
OUTBUF  DATA 22H  ;OUTPUT PIN STATE MAP
;
TESTV:  MOV  A,P2
        ANL  A,#00000100B
        JNZ  TESTU
        MOV  A,TCON
        ANL  A,#00100000B
        JZ   TESTX
TESTU:  MOV  A,P1
        ANL  A,#00000010B
        JNZ  SETQ
TESTX:  MOV  A,TCON
        ANL  A,#00001000B
        JZ   TESTZ
        MOV  A,20H
        ANL  A,#00000001B
        JZ   SETQ
TESTZ:  MOV  A,21H
        ANL  A,#00000010B
        JZ   SETQ
CLRQ:   MOV  A,OUTBUF
        ANL  A,#11110111B
        JMP  OUTQ
SETQ:   MOV  A,OUTBUF
        ORL  A,#00001000B
OUTQ:   MOV  OUTBUF,A
        MOV  P3,A
```

b.) Using only bit-test instructions.

```
;BFUNC2 SOLVE A RANDOM LOGIC FUNCTION
;       OF 6 VARIABLES BY DIRECTLY
;       POLLING EACH BIT.
;       (APPROACH USING MCS-51 UNIQUE
;       BIT-TEST INSTRUCTION CAPABILITY.)
;       SYMBOLS USED IN LOGIC DIAGRAM
;       ASSIGNED TO CORRESPONDING 8x51
;       BIT ADDRESSES.
;
U       BIT  P1.1
V       BIT  P2.2
W       BIT  TF0
X       BIT  IE1
Y       BIT  20H.0
Z       BIT  21H.1
Q       BIT  P3.3
;       ...  ....
TEST_V: JB   V,TEST_U
        JNB  W,TEST_X
TEST_U: JB   U,SET_Q
TEST_X: JNB  X,TEST_Z
        JNB  Y,SET_Q
TEST_Z: JNB  Z,SET_Q
CLR_Q:  CLR  Q
        JMP  NXTTST
SET_Q:  SETB Q
NXTTST:              ;(CONTINUATION OF
                     ;PROGRAM)
```

c.) Using logical operations on Boolean variables.

```
;FUNC3  SOLVE A RANDOM LOGIC FUNCTION
;       OF 6 VARIABLES USING
;       STRAIGHT-LINE LOGICAL
;       INSTRUCTIONS ON MCS-51 BOOLEAN
;       VARIABLES.
;
MOV  C,V
ORL  C,W   ;OUTPUT OF OR GATE
ANL  C,U   ;OUPUT OF TOP AND GATE
MOV  F0,C  ;SAVE INTERMEDIATE STATE
MOV  C,X
ANL  C,/Y  ;OUTPUT OF BOTTOM AND GATE
ORL  C,F0  ;INCLUDE VALUE SAVED ABOVE
ORL  C,/Z  ;INCLUDE LAST INPUT VARIABLE
MOV  Q,C   ;OUTPUT COMPUTED RESULT
```

01489A

An upper-limit can be placed on the complexity of software to simulate a large number of gates by summing the total number of inputs and outputs. The *actual* total should be somewhat shorter, since calculations can be "chained," as shown above. The output of one gate is often the first input to another, bypassing the intermediate variable to eliminate two lines of source.

Design Example #4 - Automotive Dashboard Functions

Now let's apply these techniques to designing the software for a complete controller system. This application is patterned after a familiar real-world application which isn't nearly as trivial as it might first appear: automobile turn signals.

Imagine the three position turn lever on the steering column as a single-pole, triple-throw toggle switch. In its central position all contacts are open. In the up or down positions contacts close causing corresponding lights in the rear of the car to blink. So far very simple.

Two more turn signals blink in the front of the car, and two others in the dashboard. All six bulbs flash when an emergency switch is closed. A thermo-mechanical relay (accessible under the dashboard in case it wears out) causes the blinking.

Applying the brake pedal turns the tail light filaments on constantly... unless a turn is in progress, in which case the blinking tail light is not affected. (Of course, the front turn signals and dashboard indicators are not affected by the brake pedal.) Table 6 summarizes these operating modes.

But we're not done yet. Each of the exterior turn signal (but not the dashboard) bulbs has a second, somewhat dimmer filament for the parking lights. Figure 15 shows TTL circuitry which could control all six bulbs. The signals labeled "High Freq." and "Low Freq." represent two square-wave inputs. Basically, when one of the turn switches is closed or the emergency switch is activated the low frequency signal (about 1 Hz) is gated through to the appropriate dashboard indicator(s) and turn signal(s). The rear signals are also activated when the brake pedal is depressed provided a turn is not being made in the same direction. When the parking light switch is closed the higher frequency oscillator is gated to each front and rear turn signal, sustaining a low-intensity background level. (This is to eliminate the need for additional parking light filaments.)

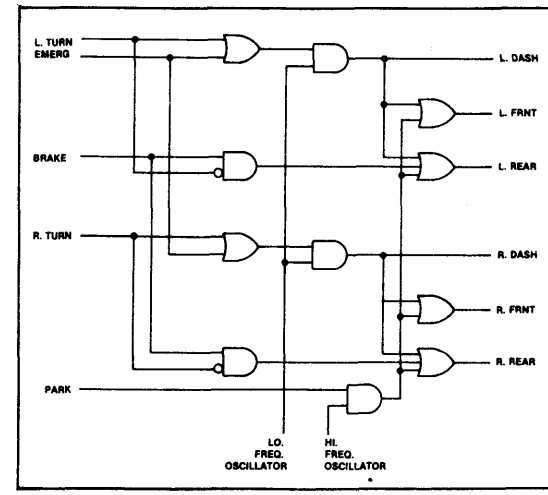


Figure 15. TTL logic implementation of automotive turn signals.

Table 6. Truth table for turn-signal operation.

INPUT SIGNALS				OUTPUT SIGNALS			
BRAKE SWITCH	EMERG. SWITCH	LEFT TURN SWITCH	RIGHT TURN SWITCH	LEFT FRONT & DASH	RIGHT FRONT & DASH	LEFT REAR	RIGHT REAR
0	0	0	0	OFF	OFF	OFF	OFF
0	0	0	1	OFF	BLINK	OFF	BLINK
0	0	1	0	BLINK	OFF	BLINK	OFF
0	1	0	0	BLINK	BLINK	BLINK	BLINK
0	1	0	1	BLINK	BLINK	BLINK	BLINK
0	1	1	0	BLINK	BLINK	BLINK	BLINK
1	0	0	0	OFF	OFF	ON	ON
1	0	0	1	OFF	BLINK	ON	BLINK
1	0	1	0	BLINK	OFF	BLINK	ON
1	1	0	0	BLINK	BLINK	ON	ON
1	1	0	1	BLINK	BLINK	ON	BLINK
1	1	1	0	BLINK	BLINK	BLINK	ON

01489A

In most cars, the switching logic to generate these functions requires a number of multiple-throw contacts. As many as 18 conductors thread the steering column of some automobiles solely for turn-signal and emergency blinker functions. (The author discovered this recently to his astonishment and dismay when replacing the whole assembly because of one burned contact.)

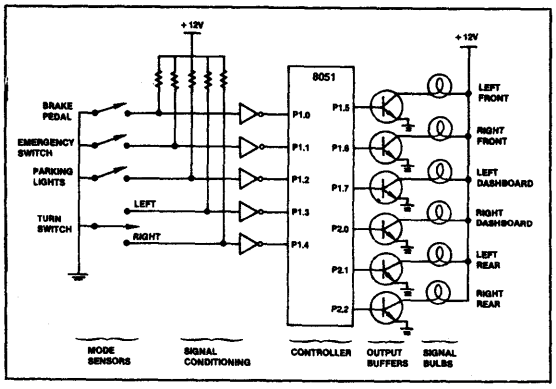
A multiple-conductor wiring harness runs to each corner of the car, behind the dash, up the steering column, and down to the blinker relay below. Connectors at each termination for each filament lead to extra cost and labor during construction, lower reliability and safety, and more costly repairs. And considering the system's present complexity, increasing its reliability or detecting failures would be quite difficult.

There are two reasons for going into such painful detail describing this example. First, to show that the messiest part of many system designs is determining what the controller should do. Writing the software to solve these functions will be comparatively easy. Secondly, to show the many potential failure points in the system. Later we'll see how the peripheral functions and intelligence built into a microcomputer (with a little creativity) can greatly reduce external interconnections and mechanical part count.

The Single-chip Solution

The circuit shown in Figure 16 indicates five input pins to the five input variables—left-turn select, right-turn select, brake pedal down, emergency switch on, and parking lights on. Six output pins turn on the front, rear, and dashboard indicators for each side. The microcomputer implements all logical functions through software, which periodically updates the output signals as time elapses and input conditions change.

Figure 16. Microcomputer Turn-signal Connections.



Design Example #3 demonstrated that symbolic addressing with user-defined bit names makes code and documentation easier to write and maintain. Accordingly, we'll assign these I/O pins names for use throughout the program. (The format of this example will differ somewhat from the others. Segments of the overall program will be presented in sequence as each is described.)

```
;
;          INPUT PIN DECLARATIONS:
;    (ALL INPUTS ARE POSITIVE-TRUE LOGIC)
;
BRAKE   BIT P1.0   ; BRAKE PEDAL DEPRESSED
EMERG   BIT P1.1   ; EMERGENCY BLINKER
                     ACTIVATED
PARK    BIT P1.2   ; PARKING LIGHTS ON
L_TURN  BIT P1.3   ; TURN LEVER DOWN
R_TURN  BIT P1.4   ; TURN LEVER UP
;
;          OUTPUT PIN DECLARATIONS:
;
L_FRNT  BIT P1.5   ; FRONT LEFT-TURN
                     INDICATOR
R_FRNT  BIT P1.6   ; FRONT RIGHT-TURN
                     INDICATOR
L_DASH  BIT P1.7   ; DASHBOARD LEFT-TURN
                     INDICATOR
R_DASH  BIT P2.0   ; DASHBOARD RIGHT-TURN
                     INDICATOR
L_REAR  BIT P2.1   ; REAR LEFT-TURN
                     INDICATOR
R_REAR  BIT P2.2   ; REAR RIGHT-TURN
                     INDICATOR
;
```

Another key advantage of symbolic addressing will appear further on in the design cycle. The locations of cable connectors, signal conditioning circuitry, voltage regulators, heat sinks, and the like all affect P.C. board layout. It's quite likely that the somewhat arbitrary pin assignment defined early in the software design cycle will prove to be less than optimum; rearranging the I/O pin assignment could well allow a more compact module, or eliminate costly jumpers on a single-sided board. (These considerations apply especially to automotive and other cost-sensitive applications needing single-chip controllers.) Since other architectures mask bytes or use "clever" algorithms to isolate bits by rotating them into the carry, re-routing an input signal (from bit 1 of port 1, for example, to bit 4 of port 3) could require extensive modifications throughout the software.

The Boolean Processor's direct bit addressing makes such changes absolutely trivial. The number of the port containing the pin is irrelevent, and masks and complex program structures are not needed. Only the initial Boolean varia-

```
;                ...          .....
; INTERRUPT RATE SUBDIVIDER
SUB_DIV          DATA    20H
; HIGH-FREQUENCY OSCILLATOR BIT
HI_FREQ          BIT     SUB_DIV.0
; LOW-FREQUENCY OSCILLATOR BIT
LO_FREQ          BIT     SUB_DIV.7
;                ...
                 ORG     0000H
JMP              INIT
;                ...          .....
                 ORG     100H
; PUT TIMER 0 IN MODE 1
INIT:            MOV     TMOD,#00000001B
; INITIALIZE TIMER REGISTERS
                 MOV     TL0,#0
                 MOV     TH0,#-16
; SUBDIVIDE INTERRUPT RATE BY 244
                 MOV     SUB_DIV,#244
; ENABLE TIMER INTERRUPTS
                 SETB    ET0
; GLOBALLY ENABLE ALL INTERRUPTS
                 SETB    EA
; START TIMER
                 SETB    TR0
;
; (CONTINUE WITH BACKGROUND PROGRAM)
;

; PUT TIMER 0 IN MODE 1
; INITIALIZE TIMER REGISTERS

; SUBDIVIDE INTERRUPT RATE BY 244
; ENABLE TIMER INTERRUPTS
; GLOBALLY ENABLE ALL INTERRUPTS
; START TIMER
```

ble declarations need to be changed; ASM51 automatically adjusts all addresses and symbolic references to the reassigned variables. The user is assured that no additional debugging or software verification will be required.

Timer 0 (one of the two on-chip timer/counters) replaces the thermo-mechanical blinker relay in the dashboard controller. During system initialization it is configured as a timer in mode 1 by setting the least significant bit of the timer mode register (TMOD). In this configuration the low-order byte (TL0) is incremented every machine cycle, overflowing and incrementing the high-order byte (TH0) every 256 μSec. Timer interrupt 0 is enabled so that a hardware interrupt will occur each time TH0 overflows. (For details of the numerous timer operating modes see the MCS-51™ User's Manual.)

An eight-bit variable in the bit-addressable RAM array will be needed to further subdivide the interrupts via software. The lowest-order bit of this counter toggles very fast to modulate the parking lights; bit 7 will be "tuned" to approximately 1 Hz for the turn- and emergency-indicator blinking rate.

Loading TH0 with -16 will cause an interrupt after 4.096 msec. The interrupt service routine reloads the high-order byte of timer 0 for the next interval, saves the CPU registers likely to be affected on the stack, and then decrements SUB_DIV. Loading SUB_DIV. with 244 initially and each time it decrements to zero will produce a 0.999 second period for the highest-order bit.

```
ORG   000BH     ; TIMER 0 SERVICE VECTOR
MOV   TH0,#-16
PUSH  PSW
PUSH  ACC
PUSH  B
DJNZ  SUB_DIV,T0SERV
MOV   SUB_DIV,#244
```

The code to sample inputs, perform calculations, and update outputs—the real "meat" of the signal controller algorithm—may be performed either as part of the interrupt service routine or as part of a background program loop. The only concern is that it must be executed at least several dozen times per second to prevent parking light flickering. We will assume the former case, and insert the code into the timer 0 service routine.

First, notice from the logic diagram (Figure 15) that the subterm (PARK · H_FREQ), asserted when the parking lights are to be on dimly, figures into four of the six output functions. Accordingly, we will first compute that term and save it in a temporary location named "DIM". The PSW contains two general purpose flags: F0, which corresponds to the 8048 flag of the same name, and PSW.1. Since The PSW has been saved and will be restored to its previous state after servicing the interrupt, we can use either bit for temporary storage.

```
DIM   BIT       PSW.1  ; DECLARE TEMP.
                         STORAGE FLAG
...   .....
MOV   C,PARK             ; GATE PARKING
                           LIGHT SWITCH
ANL   HI_FREQ            ; WITH HIGH
                           FREQUENCY
                           SIGNAL
MOV   DIM,C              ; AND SAVE IN
                           TEMP. VARIABLE.
```

This simple three-line section of code illustrates a remarkable point. The software indicates in very abstract terms exactly what function is being performed, independent of

01489A-22

the hardware configuration. The fact that these three bits include an input pin, a bit within a program variable, and a software flag in the PSW is totally invisible to the programmer.

Now generate and output the dashboard left turn signal.

```
;
   MOV C,L_TURN        ; SET CARRY IF
                         TURN
   ORL C,EMERG         ; OR EMERGENCY
                         SELECTED.
   ANL C,LO_FREQ       ; GATE IN 1 HZ
                         SIGNAL
   MOV L_DASH,C        ; AND OUTPUT TO
                         DASHBOARD.
```

To generate the left front turn signal we only need to add the parking light function in F0. But notice that the function in the carry will also be needed for the rear signal. We can save effort later by saving its current state in F0.

```
;
   MOV F0,C            ; SAVE FUNCTION
                         SO FAR.
   ORL C,DIM           ; ADD IN PARKING
                         LIGHT FUNCTION
   MOV L_FRNT,C        ; AND OUTPUT TO
                         TURN SIGNAL.
```

Finally, the rear left turn signal should also be on when the brake pedal is depressed, provided a left turn is not in progress.

```
   MOV C,BRAKE         ; GATE BRAKE
                         PEDAL SWITCH
   ANL C,/L_TURN       ; WITH TURN
                         LEVER.
   ORL C,F0            ; INCLUDE TEMP.
                         VARIABLE FROM
                         DASH
   ORL C,DIM           ; AND PARKING
                         LIGHT FUNCTION
   MOV L_REAR,C        ; AND OUTPUT TO
                         TURN SIGNAL.
```

Now we have to go through a similar sequence for the right-hand equivalents to all the left-turn lights. This also gives us a chance to see how the code segments above look when combined.

```
   MOV C,R_TURN        ; SET CARRY IF
                         TURN
   ORL C,EMERG         ; OR EMERGENCY
                         SELECTED.
   ANL C,LO_FREQ       ; IF SO, GATE IN 1
                         HZ SIGNAL
   MOV R_DASH,C        ; AND OUTPUT TO
                         DASHBOARD.
   MOV F0,C            ; SAVE FUNCTION
                         SO FAR.
   ORL C,DIM           ; ADD IN PARKING
                         LIGHT FUNCTION
   MOV R_FRNT,C        ; AND OUTPUT TO
                         TURN SIGNAL.
   MOV C,BRAKE         ; GATE BRAKE
                         PEDAL SWITCH
   ANL C,/R_TURN       ; WITH TURN
                         LEVER.
   ORL C,F0            ; INCLUDE TEMP.
                         VARIABLE FROM
                         DASH
   ORL C,DIM           ; AND PARKING
                         LIGHT FUNCTION
   MOV R_REAR,C        ; AND OUTPUT TO
                         TURN SIGNAL.
```

(The perceptive reader may notice that simply rearranging the steps could eliminate one instruction from each sequence.)

Now that all six bulbs are in the proper states, we can return from the interrupt routine, and the program is finished. This code essentially needs to reverse the status saving steps at the beginning of the interrupt.

```
   POP B               ; RESTORE CPU
                         REGISTERS.
   POP ACC
   POP PSW
   RETI
```

Program Refinements. The luminescence of an incandescent light bulb filament is generally non-linear; the 50% duty cycle of HI_FREQ may not produce the desired intensity. If the application requires, duty cycles of 25%, 75%, etc. are easily achieved by ANDing and ORing in additional low-order bits of SUB_DIV. For example, 30 Hz signals of seven different duty cycles could be produced by considering bits 2—0 as shown in Table 7. The only software change required would be to the code which sets-up variable DIM:

```
   MOV C,SUB_DIV.1     ; START WITH 50
                         PERCENT
   ANL C,SUB_DIV.0     ; MASK DOWN TO 25
                         PERCENT
   ORL C,SUB_DIV.2     ; AND BUILD BACK TO
                         62 PERCENT
   MOV DIM,C           ; DUTY CYCLE FOR
                         PARKING LIGHTS.
```

01489A

Table 7. Non-trivial Duty Cycles.

SUB_DIV BITS								DUTY CYCLES						
7	6	5	4	3	2	1	0	12.5%	25.0%	37.5%	50.0%	62.5%	75.0%	87.5%
X	X	X	X	X	0	0	0	OFF	OFF	OFF	OFF	OFF	OFF	OFF
X	X	X	X	X	0	0	1	OFF	OFF	OFF	OFF	OFF	OFF	ON
X	X	X	X	X	0	1	0	OFF	OFF	OFF	OFF	OFF	ON	ON
X	X	X	X	X	0	1	1	OFF	OFF	OFF	OFF	ON	ON	ON
X	X	X	X	X	1	0	0	OFF	OFF	OFF	ON	ON	ON	ON
X	X	X	X	X	1	0	1	OFF	OFF	ON	ON	ON	ON	ON
X	X	X	X	X	1	1	0	OFF	ON	ON	ON	ON	ON	ON
X	X	X	X	X	1	1	1	ON	ON	ON	ON	ON	ON	ON

Interconnections increase cost and decrease reliability. The simple buffered pin-per-function circuit in Figure 16 is insufficient when many outputs require higher-than-TTL drive levels. A lower-cost solution uses the 8051 serial port in the shift-register mode to augment I/O. In mode 0, writing a byte to the serial port data buffer (SBUF) causes the data to be output sequentially through the "RXD" pin while a burst of eight clock pulses is generated on the "TXD" pin. A shift register connected to these pins (Figure 17) will load the data byte as it is shifted out. A number of special peripheral driver circuits combining shift-register inputs with high drive level outputs have been introduced recently.

Cascading multiple shift registers end-to-end will expand the number of outputs even further. The data rate in the I/O expansion mode is one megabaud, or 8 usec. per byte. This is the mode which the serial port defaults to following a reset, so no initialization is required.

The software for this technique uses the B register as a "map" corresponding to the different output functions. The program manipulates these bits instead of the output pins. After all functions have been calculated the B register is shifted by the serial port to the shift-register/driver. (While some outputs may glitch as data is shifted through them, at 1 Megabaud most people wouldn't notice. Some shift registers provide an "enable" bit to hold the output states while new data is being shifted in.)

This is where the earlier decision to address bits symbolically throughout the program is going to pay off. This major I/O restructuring is nearly as simple to implement as rearranging the input pins. Again, only the bit declarations need to be changed.

```
L_FRNT  BIT  B.0  ; FRONT LEFT-TURN
                   INDICATOR
R_FRNT  BIT  B.1  ; FRONT RIGHT-TURN
                   INDICATOR
L_DASH  BIT  B.2  ; DASHBOARD LEFT-TURN
                   INDICATOR
R_DASH  BIT  B.3  ; DASHBOARD RIGHT-TURN
                   INDICATOR
```

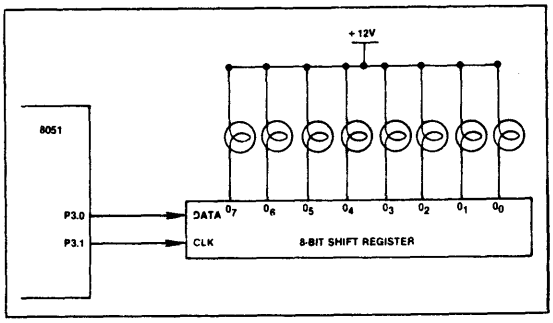


Figure 17. Output expansion using serial port.

```
L_REAR  BIT  B.4  ; REAR LEFT-TURN
                   ; INDICATOR
R_REAR  BIT  B.5  ; REAR RIGHT-TURN
                   ; INDICATOR
```

The original program to compute the functions need not change. After computing the output variables, the control map is transmitted to the buffered shift register through the serial port:

```
MOV  SBUF,B  ;LOAD BUFFER AND TRANSMIT
```

The Boolean Processor solution holds a number of advantages over older methods. Fewer switches are required. Each is simpler, requiring fewer poles and lower current contacts. The flasher relay is eliminated entirely. Only six filaments are driven, rather than 10. The wiring harness is therefore simpler and less expensive—one conductor for each of the six lamps and each of the five sensor switches. The fewer conductors use far fewer connectors. The whole system is more reliable.

And since the system is much simpler it would be feasible to implement redundancy and or fault detection on the four main turn indicators. Each could **still** be a standard double filament bulb, but with the filaments driven in parallel to tolerate single-element failures.

Even with redundancy, the lights will eventually fail. To handle this inescapable fact current or voltage sensing

01489A

circuits on each main drive wire can verify that each bulb and its high-current driver is functioning properly. Figure 18 shows one such circuit.

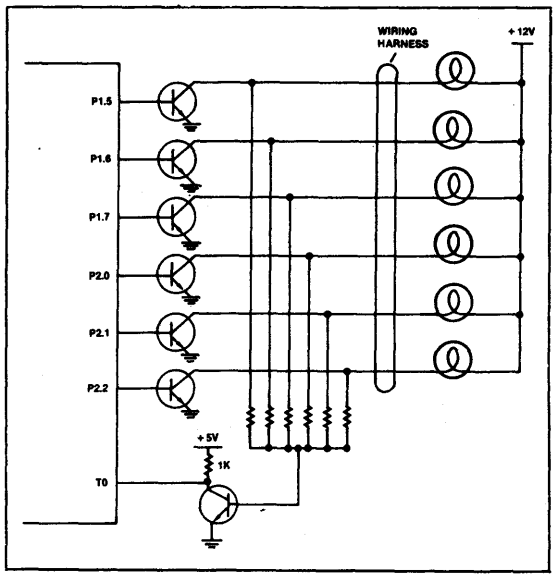


Figure 18.

Assume all of the lights are turned on except one; i.e., all but one of the collectors are grounded. For the bulb which is turned off, if there is continuity from +12 V through the bulb base and filament, the control wire, all connectors, and the P.C. board traces, and if the transistor is indeed not shorted to ground, then the collector will be pulled to +12 V. This turns on the base of Q8 through the corresponding resistor, and grounds the input pin, verifying that the bulb circuit is operational. The continuity of each circuit can be checked by software in this way.

Now turn *all* the bulbs on, grounding all the collectors. Q7 should be turned off, and the Test pin should be high. However, a control wire shorted to +12 V or an open-circuited drive transistor would leave one of the collectors at the higher voltage even now. This too would turn on Q7, indicating a different type of failure. Software could perform these checks once per second by executing the routine every time the software counter SUB_DIV is reloaded by the interrupt routine.

```
DJNZ SUB_DIV,T0SERV
MOV  SUB_DIV,#244       ; RELOAD COUNTER
ORL  P1,#11100000B      ; SET CONTROL
                          OUTPUTS HIGH
ORL  P2,#00000111B
CLR  L_FRNT             ; FLOAT DRIVE
                          COLLECTOR
JB   T0,FAULT           ; T0 SHOULD BE
                          PULLED LOW
SETB L_FRNT             ; PULL COLLECTOR
                          BACK DOWN
CLR  L_DASH
JB   T0,FAULT
SETB L_DASH
CLR  L_REAR
JB   T0,FAULT
SETB L_REAR
CLR  R_FRNT
JB   T0,FAULT
SETB R_FRNT
CLR  R_DASH
JB   T0,FAULT
SETB R_DASH
CLR  R_REAR
JB   T0,FAULT
SETB R_REAR
;
; WITH ALL COLLECTORS GROUNDED, T0
  SHOULD BE HIGH
; IF SO, CONTINUE WITH INTERRUPT ROUTINE.
  JB   T0,T0SERV
FAULT:                  ; ELECTRICAL FAILURE
                        ;PROCESSING ROUTINE
                        ; (LEFT TO READER'S
                        ; IMAGINATION)
T0SERV:                 ; CONTINUE WITH
                        ;INTERRUPT PROCESSING
;
;
```

The complete assembled program listing is printed in Appendix A. The resulting code consists of 67 program statements, not counting declarations and comments, which assemble into 150 bytes of object code. Each pass through the service routine requires (coincidently) 67 usec, plus 32 usec once per second for the electrical test. If executed every 4 msec as suggested this software would typically reduce the throughput of the background program by less than 2%.

Once a microcomputer has been designed into a system, new features suddenly become virtually free. Software could make the emergency blinkers flash alternately or at a rate faster than the turn signals. Turn signals could override the emergency blinkers. Adding more bulbs would allow multiple tail light sequencing and syncopation — true flash factor, so to speak.

Design Example #5 - Complex Control Functions

Finally, we'll mix byte and bit operations to extend the use of 8051 into extremely complex applications.

Programmers can arbitrarily assign I/O pins to input and output functions only if the total does not exceed 32, which is insufficient for applications with a very large number of input variables. One way to expand the number of inputs is with a technique similar to multiplexed-keyboard scanning.

01489A

Figure 19 shows a block diagram for a moderately complex programmable industrial controller with the following characteristics:

- 64 input variable sensors;
- 12 output signals;
- Combinational and sequential logic computations;
- Remote operation with communications to a host processor via a high-speed full-duplex serial link;
- Two prioritized external interrupts;
- Internal real-time and time-of-day clocks.

While many microprocessors could be programmed to provide these capabilities with assorted peripheral support chips, an 8051 microcomputer needs **no** other integrated circuits!

The 64 input sensors are logically arranged as an 8x8 matrix. The pins of Port 1 sequentially enable each column of the sensor matrix; as each is enabled Port 0 reads in the state of each sensor in that column. An eight-byte block in bit-addressable RAM remembers the data as it is read in so that after each complete scan cycle there is an internal map of the current state of all sensors. Logic functions can then directly address the elements of the bit map.

The computer's serial port is configured as a nine-bit UART, transferring data at 17,000 bytes-per-second. The ninth bit may distinguish between address and data bytes.

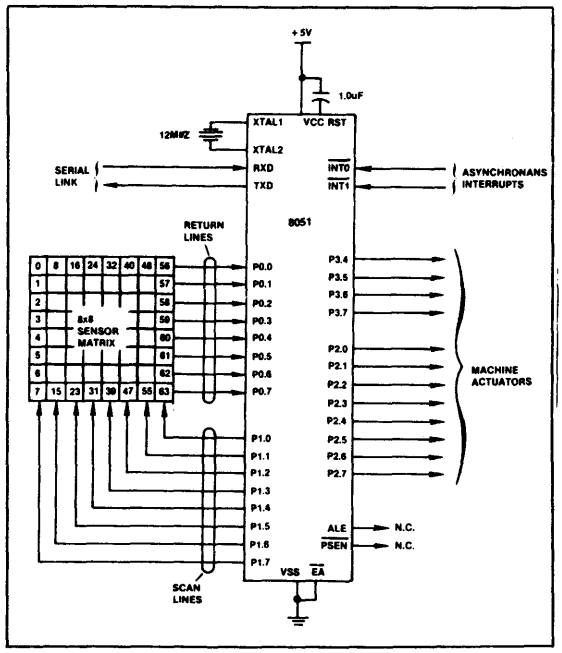


Figure 19. Block diagram of 64-input machine controller.

The 8051 serial port can be configured to detect bytes with the address bit set, automatically ignoring all others. Pins INT0 and INT1 are interrupts configured respectively as high-priority, falling-edge triggered and low-priority, low-level triggered. The remaining 12 I/O pins output TTL-level control signals to 12 actuators.

There are several ways to implement the sensor matrix circuitry, all logically similar. Figure 20.a shows one possibility. Each of the 64 sensors consists of a pair of simple switch contacts in series with a diode to permit multiple contact closures throughout the matrix.

The scan lines from Port 1 provide eight un-encoded active-high scan signals for enabling columns of the matrix. The return lines on rows where a contact is closed are pulled high and read as logic ones. Open return lines are pulled to ground by one of the 40 kohm resistors and are read as zeroes. (The resistor values must be chosen to ensure all return lines are pulled above the 2.0 V logic threshold, even in the worst-case, where all contacts in an enabled column are closed.) Since P0 is provided open-collector outputs and high-impedance MOS inputs its input loading may be considered negligible.

The circuits in Figures 20.b—20.d are variations on this theme. When input signals must be electrically isolated from the computer circuitry as in noisy industrial environments, phototransistors can replace the switch diode pairs **and** provide optical isolation as in Figure 20.b. Additional opto-isolators could also be used on the control output and special signal lines.

The other circuits assume that input signals are already at TTL levels. Figure 20.c uses octal three-state buffers enabled by active-low scan signals to gate eight signals onto Port 0. Port 0 is available for memory expansion or peripheral chip interfacing between sensor matrix scans. Eight-to-one multiplexers in Figure 20.d select one of eight inputs for each return line as determined by encoded address bits output on three pins of Port 1. (Five more output pins are thus freed for more control functions.) Each output can drive at least one standard TTL or up to 10 low-power TTL loads without additional buffering.

Going back to the original matrix circuit, Figure 21 shows the method used to scan the sensor matrix. Two complete bit maps are maintained in the bit-addressable region of the RAM: one for the current state and one for the previous state read for each sensor. If the need arises, the program could then sense input transitions and/or debounce contact closures by comparing each bit with its earlier value.

01489A

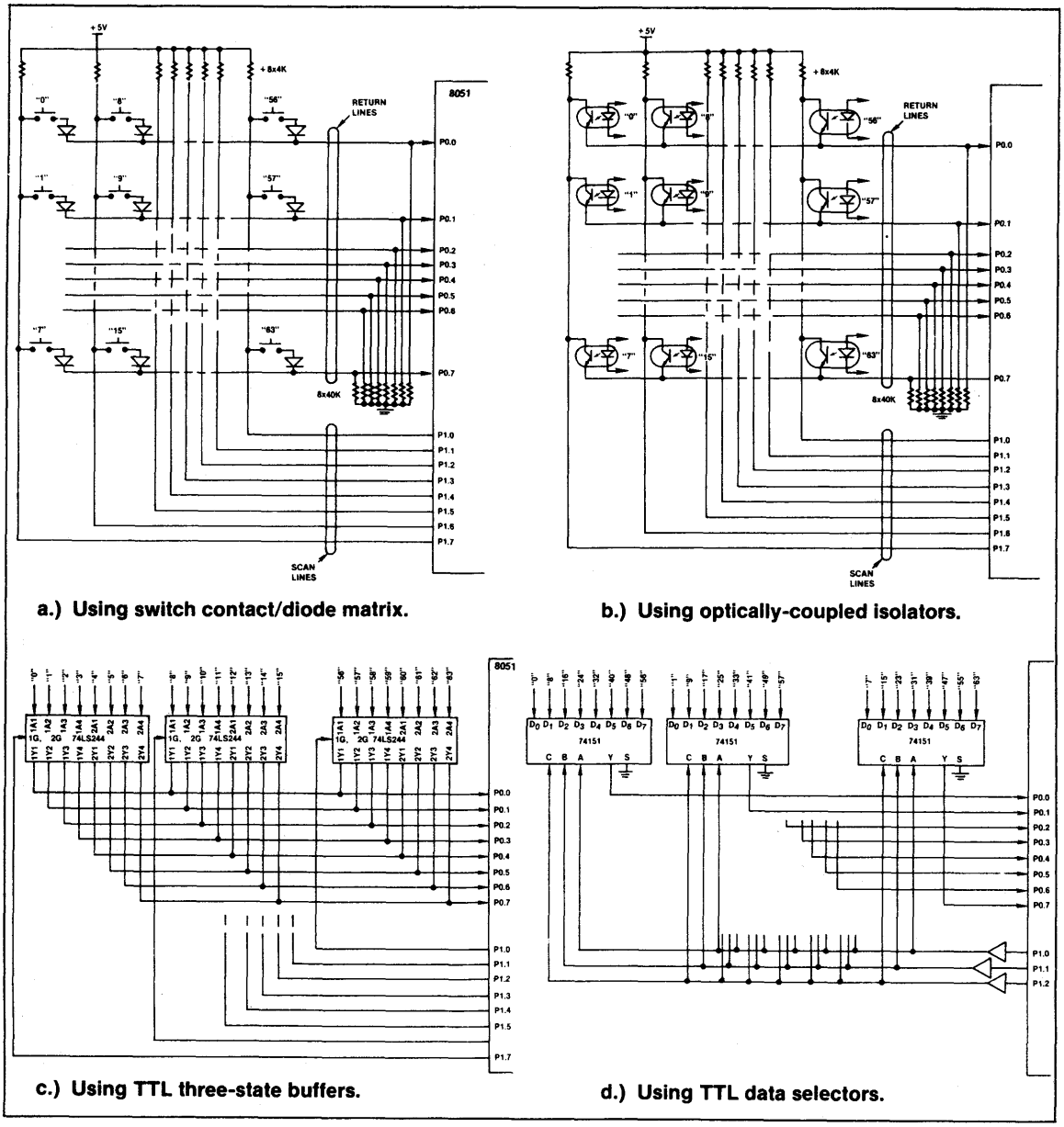


Figure 20. Sensor Matrix Implementation Methods.

The code in Example 3 implements the scanning algorithm for the circuits in Figure 20.a. Each column is enabled by setting a single bit in a field of zeroes. The bit maps are positive logic; ones represent contacts that are closed or isolators turned on.

Example 3.

```
INPUT_SCAN:     ; SUBROUTINE TO READ
                  CURRENT STATE
                ; OF 64 SENSORS AND
                  SAVE IN RAM 20H-27H.
    MOV R0,#20H     ; INITIALIZE
                    ; POINTERS
    MOV R1,#28H     ; FOR BIT MAP
                    ; BASES.
```

01489A

```
        MOV  A,#80H       ; SET FIRST BIT IN
                            ACC.
SCAN:   MOV  P1,A         ; OUTPUT TO SCAN
                            LINES.
        RR   A            ; SHIFT TO ENABLE
                            NEXT COLUMN
                            NEXT.
        MOV  R2,A         ; REMEMBER CUR-
                            RENT SCAN
                            POSITION.
        MOV  A,P0         ; READ RETURN
                            LINES.
        XCH  A,@R0        ; SWITCH WITH
                            PREVIOUS MAP
                            BITS.
        MOV  @R1,A        ; SAVE PREVIOUS
                            STATE AS WELL.
        INC  R0           ; BUMP POINTERS.
        INC  R1
        MOV  A,R2         ; RELOAD SCAN LINE
                            MASK
        JNB  ACC.7,SCAN   ; LOOP UNTIL ALL
                            EIGHT COLUMNS
                            READ.
        RET
```

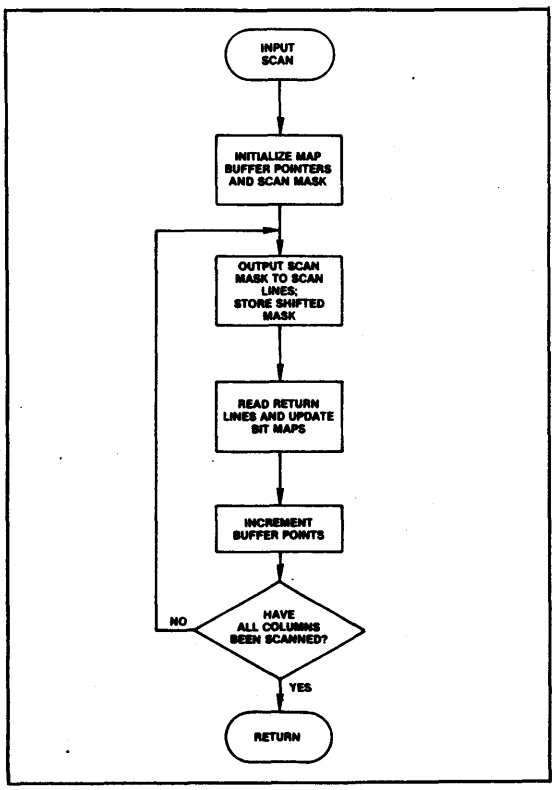


Figure 21. Flowchart for reading in sensor matrix.

What happens after the sensors have been scanned depends on the individual application. Rather than inventing some artificial design problem, software corresponding to commonplace logic elements will be discussed.

Combinatorial Output Variables. An output variable which is a simple (or not so simple) combinational function of several input variables is computed in the spirit of Design Example 3. All 64 inputs are represented in the bit maps; in fact, the sensor numbers in Figure 20 correspond to the absolute bit addresses in RAM! The code in Example 4 activates an actuator connected to P2.2 when sensors 12, 23, and 34 are closed and sensors 45 and 56 are open.

Example 4.

Simple Combinatorial Output Variables.

```
; SET P2.2 = (12) (23) (34) (/45) (/56)
        MOV  C,12
        ANL  C,23
        ANL  C,34
        ANL  C,/45
        ANL  C,/56
        MOV  P2.2,C
```

Intermediate Variables. The examination of a typical relay-logic ladder diagram will show that many of the rungs control *not* outputs but rather relays whose contacts figure into the computation of other functions. In effect, these relays indicate the state of intermediate variables of a computation.

The MCS-51™ solution can use any directly addressable bit for the storage of such intermediate variables. Even when all 128 bits of the RAM array are dedicated (to input bit maps in this example), the accumulator, PSW, and B register provide 18 additional flags for intermediate variables.

For example, suppose switches 0 through 3 control a safety interlock system. Closing any of them should deactivate certain outputs. Figure 22 is a ladder diagram for this situation. The interlock function could be recomputed for every output affected, or it may be computed once and saved (as implied by the diagram). As the program proceeds this bit can qualify each output.

Example 5. Incorporating Override signal into actuator outputs.

```
        CALL INPUT_SCAN
        MOV  C,0
        ORL  C,1
        ORL  C,2
        ORL  C,3
        MOV  F0,C
;       ....   .....
```

01489A-26

```
;       COMPUTE FUNCTION 0
;
        ANL   C,/F0
        MOV   P1.0,C
;       ....  .....
;       COMPUTE FUNCTION 1
;
        ANL   C,/F0
        MOV   P1.1,C
;       ....  .....
;       COMPUTE FUNCTION 2
;
        ANL   C,/F0
        MOV   P1.2,C
;       ....  .....
```

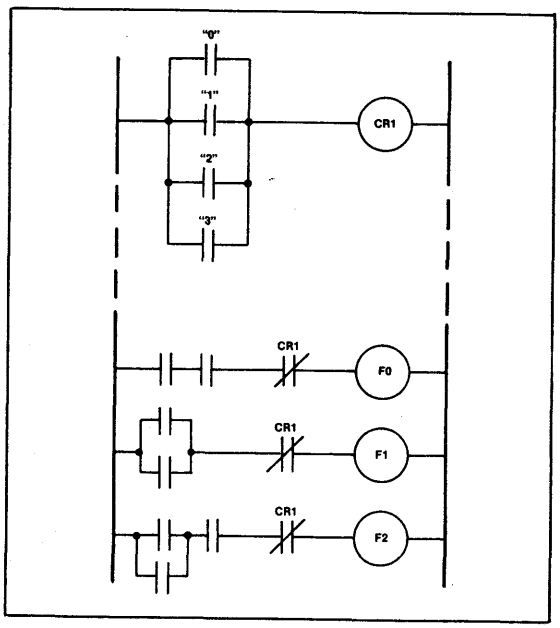


Figure 22. Ladder diagram for output override circuitry.

Latching Relays. A latching relay can be forced into either the ON or OFF state by two corresponding input signals, where it will remain until forced onto the opposite state—analogous to a TTL Set/Reset flip-flop. The relay is used as an intermediate variable for other calculations. In the previous example, the emergency condition could be remembered and remain active until an "emergency cleared" button is pressed.

Any flag or addressable bit may represent a latching relay with a few lines of code (see Example 6).

Example 6. Simulating a latching relay.

```
;L_SET   SET FLAG 0 IF C=1
L_SET:    ORL   C,F0
          MOV   F0,C
;

;L_RSET RESET FLAG 0 IF C=1
L_RSET:  CPS   C
          ANL   C,F0
          MOV   F0,C
;
```

Time Delay Relays. A time delay relay does not respond to an input signal until it has been present (or absent) for some predefined time. For example, a ballast or load resistor may be switched in series with a D.C. motor when it is first turned on, and shunted from the circuit after one second. This sort of time delay may be simulated by an interrupt routine driven by one of the two 8051 timer/counters. The procedure followed by the routine depends heavily on the details of the exact function needed; time-outs or time delays with resettable or non-resettable inputs are possible. If the interrupt routine is executed every 10 milliseconds the code in Example 7 will clear an intermediate variable set by the background program after it has been active for two seconds.

Example 7. Code to clear USRFLG after a fixed time delay.

```
          JNB   USR_FLG,NXTTST
          DJNZ DLAY_COUNT,NXTTST
          CLR   USR_FLG
          MOV   DLAY_COUNT,#200
NXTTST: ;..   .....
```

Serial Interface to Remote Processor. When it detects emergency conditions represented by certain input combinations (such as the earlier Emergency Override), the controller could shut down the machine immediately and/or alert the host processor via the serial port. Code bytes indicating the nature of the problem could be transmitted to a central computer. In fact, at 17,000 bytes-per-second, the entire contents of both bit maps could be sent to the host processor for further analysis in less than a millisecond! If the host decides that conditions warrant, it could alert other remote processors in the system that a problem exists and specify which shut-down sequence each should initiate. For more information on using the serial port, consult the MCS-51™ User's Manual.

Response Timing.
One difference between relay and programmed industrial controllers (when each is considered as a "black box") is their respective reaction times to input changes. As reflected by a ladder diagram, relay systems contain a

large number of "rungs" operating in parallel. A change in input conditions will begin propagating through the system immediately, possibly affecting the output state within milliseconds.

Software, on the other hand, operates sequentially. A change in input states will not be detected until the next time an input scan is performed, and will not affect the outputs until that section of the program is reached. For that reason the raw speed of computing the logical functions is of extreme importance.

Here the Boolean processor pays off. *Every instruction mentioned in this Note* completes in one or two microseconds—the *minimum* instruction execution time for many other microcontrollers! A ladder diagram containing a hundred rungs, with an average of four contacts per rung can be replaced by approximately five hundred lines of software. A complete pass through the entire matrix scanning routine and all computations would require about a millisecond; less than the time it takes for most relays to change state.

A programmed controller which simulates each Boolean function with a subroutine would be less efficient by at least an order of magnitude. Extra software is needed for the simulation routines, and each step takes longer to execute for three reasons: several byte-wide logical instructions are executed per user program step (rather than one Boolean operation); most of those instructions take longer to execute with microprocessors performing multiple off-chip accesses; and calling and returning from the various subroutines requires overhead for stack operations.

In fact, the speed of the Boolean Processor solution is likely to be much faster than the system requires. The CPU might use the time left over to compute feedback parameters, collect and analyze execution statistics, perform system diagnostics, and so forth.

Additional functions and uses.

With the building-block basics mentioned above many more operations may be synthesized by short instruction sequences.

Exclusive-OR. There are no common mechanical devices or relays analogous to the Exclusive-OR operation, so this instruction was omitted from the Boolean Processor. However, the Exclusive-OR or Exclusive-NOR operation may be performed in two instructions by conditionally complementing the carry or a Boolean variable based on the state of any other testable bit.

```
; EXCLUSIVE-OR FUNCTION IMPOSED ON CARRY
; USING F0 IS INPUT VARIABLE.
XOR_F0: JNB   F0,XORCNT  ; ("JB" FOR X-NOR)
        CPL   C
XORCNT:...    .....
```

XCH. The contents of the carry and some other bit may be exchanged (switched) by using the accumulator as temporary storage. Bits can be moved into and out of the accumulator simultaneously using the Rotate-through-carry instructions, though this would alter the accumulator data.

```
; EXCHANGE CARRY WITH USRFLG
XCHBIT: RLC   A
        MOV   C,USR_FLG
        RRC   A
        MOV   USR_FLG,C
        RLC   A
```

Extended Bit Addressing. The 8051 can directly address 144 general-purpose bits for all instructions in Figure 3.b. Similar operations may be extended to any bit anywhere on the chip with some loss of efficiency.

The logical operations AND, OR, and Exclusive-OR are performed on byte variables using six different addressing modes, one of which lets the source be an immediate mask, and the destination any directly addressable byte. Any bit may thus be set, cleared, or complemented with a three-byte, two-cycle instruction if the mask has all bits but one set or cleared.

Byte variables, registers, and indirectly addressed RAM may be moved to a bit addressable register (usually the accumulator) in one instruction. Once transferred, the bits may be tested with a conditional jump, allowing any bit to be polled in 3 microseconds—still much faster than most architectures—or used for logical calculations. (This technique can also simulate additional bit addressing modes with byte operations.)

Parity of bytes or bits. The parity of the current accumulator contents is always available in the PSW, from whence it may be moved to the carry and further processed. Error-correcting Hamming codes and similar applications require computing parity on groups of isolated bits. This can be done by conditionally complementing the carry flag based on those bits or by gathering the bits into the accumulator (as shown in the DES example) and then testing the parallel parity flag.

Multiple byte shift and CRC codes.

Though the 8051 serial port can accommodate eight- or nine-bit data transmissions, some protocols involve much

01489A-30

longer bit streams. The algorithms presented in Design Example 2 can be extended quite readily to 16 or more bits by using multi-byte input and output buffers.

Many mass data storage peripherals and serial communications protocols include Cyclic Redundancy (CRC) codes to verify data integrity. The function is generally computed serially by hardware using shift registers and Exclusive-OR gates, but it can be done with software. As each bit is received into the carry, appropriate bits in the multi-byte data buffer are conditionally complemented based on the incoming data bit. When finished, the CRC register contents may be checked for zero by ORing the two bytes in the accumulator.

4. SUMMARY

A truly unique facet of the Intel MCS-51™ microcomputer family design is the collection of features optimized for the one-bit operations so often desired in real-world, real-time control applications. Included are 17 special instructions, a Boolean accumulator, implicit and direct addressing modes, program and mass data storage, and many I/O options. These are the world's first single-chip microcomputers able to efficiently manipulate, operate on, and transfer either bytes or individual bits as data.

This Application Note has detailed the information needed by a microcomputer system designer to make full use of these capabilities. Five design examples were used to contrast the solutions allowed by the 8051 and those required by previous architectures. Depending on the individual application, the 8051 solution will be easier to design, more reliable to implement, debug, and verify, use less program memory, and run up to an order of magnitude faster than the same function implemented on previous digital computer architectures.

Combining byte- and bit-handling capabilities in a single microcomputer has a strong synergistic effect: the power of the result exceeds the power of byte- and bit-processors laboring individually. Virtually all user applications will benefit in some ways from this duality. Data intensive applications will use bit addressing for test pin monitoring or program control flags; control applications will use byte manipulation for parallel I/O expansion or arithmetic calculations.

It is hoped that these design examples give the reader an appreciation of these unique features and suggest ways to exploit them in his or her own application.

01489A

```
ISIS-II MCS-51 MACRO ASSEMBLER V1.0
OBJECT MODULE PLACED IN :F0:AP70.HEX
ASSEMBLER INVOKED BY:  :f1:asm51 ap70.src date(328)
LOC  OBJ             LINE     SOURCE

                        1     $XREF TITLE(AP-70 APPENDIX)
                        2     ;*******************************************************
                        3     ;
                        4     ;       THE FOLLOWING PROGRAM USES THE BOOLEAN INSTRUCTION SET
                        5     ;       OF THE INTEL 8051 MICROCOMPUTER TO PERFORM A NUMBER OF
                        6     ;       AUTOMOTIVE DASHBOARD CONTROL FUNCTIONS RELATING TO
                        7     ;       TURN SIGNAL CONTROL, EMERGENCY BLINKERS, BRAKE LIGHT
                        8     ;       CONTROL, AND PARKING LIGHT OPERATION.
                        9     ;       THE ALGORITHMS AND HARDWARE ARE DESCRIBED IN DESIGN
                       10     ;       EXAMPLE #4 OF INTEL APPLICATION NOTE AP-70,
                       11     ;               "USING THE INTEL MCS-51(TM)
                       12     ;              BOOLEAN PROCESSING CAPABILITIES"
                       13     ;
                       14     ;*******************************************************
                       15     ;
                       16     ;       INPUT PIN DECLARATIONS:
                       17     ;        (ALL INPUTS ARE POSITIVE-TRUE LOGIC.
                       18     ;       INPUTS ARE HIGH WHEN RESPECTIVE SWITCH CONTACT IS CLOSED.)
                       19     ;
0090                   20     BRAKE   BIT     P1.0    ; BRAKE PEDAL DEPRESSED
0091                   21     EMERG   BIT     P1.1    ; EMERGENCY BLINKER ACTIVATED
0092                   22     PARK    BIT     P1.2    ; PARKING LIGHTS ON
0093                   23     L_TURN  BIT     P1.3    ; TURN LEVER DOWN
0094                   24     R_TURN  BIT     P1.4    ; TURN LEVER UP
                       25     ;
                       26     ;       OUTPUT PIN DECLARATIONS:
                       27     ;       (ALL OUTPUTS ARE POSITIVE TRUE LOGIC.
                       28     ;       BULB IS TURNED ON WHEN OUTPUT PIN IS HIGH.)
                       29     ;
0095                   30     L_FRNT  BIT     P1.5    ; FRONT LEFT-TURN INDICATOR
0096                   31     R_FRNT  BIT     P1.6    ; FRONT RIGHT-TURN INDICATOR
0097                   32     L_DASH  BIT     P1.7    ; DASHBOARD LEFT-TURN INDICATOR
00A0                   33     R_DASH  BIT     P2.0    ; DASHBOARD RIGHT-TURN INDICATOR
00A1                   34     L_REAR  BIT     P2.1    ; REAR LEFT-TURN INDICATOR
00A2                   35     R_REAR  BIT     P2.2    ; REAR RIGHT-TURN INDICATOR
                       36     ;
00A3                   37     S_FAIL  BIT     P2.3    ; ELECTRICAL SYSTEM FAULT INDICATOR
                       38     ;
                       39     ;       INTERNAL VARIABLE DEFINITIONS:
                       40     ;
0020                   41     SUB_DIV DATA    20H             ; INTERRUPT RATE SUBDIVIDER
0000                   42     HI_FREQ BIT     SUB_DIV.0       ; HIGH-FREQUENCY OSCILLATOR BIT
0007                   43     LO_FREQ BIT     SUB_DIV.7       ; LOW-FREQUENCY OSCILLATOR BIT
                       44     ;       ...     .....
00D1                   45     DIM     BIT     PSW.1           ; PARKING LIGHTS ON FLAG
                       46     ;
                       47     ;=======================================================
                       48 +1  $EJECT
```

```
LOC  OBJ             LINE     SOURCE

                       49             ORG     0000H           ; RESET VECTOR
0000 020040            50             LJMP    INIT
                       51     ;
000B                   52             ORG     000BH           ; TIMER 0 SERVICE VECTOR
000B 758CF0            53             MOV     TH0,#-16        ; HIGH TIMER BYTE ADJUSTED TO CONTROL INT. RATE
000E C0D0              54             PUSH    PSW             ; EXECUTE CODE TO SAVE ANY REGISTERS USED BELOW
0010 0154              55             AJMP    UPDATE          ;     (CONTINUE WITH REST OF ROUTINE)
                       56     ;
0040                   57             ORG     0040H
0040 758A00            58     INIT:   MOV     TL0,#0          ; ZERO LOADED INTO LOW-ORDER BYTE AND
0043 758CF0            59             MOV     TH0,#-16        ; -16 IN HIGH-ORDER BYTE GIVES 4 MSEC PERIOD
0046 758961            60             MOV     TMOD,#01100001B ; 8-BIT AUTO RELOAD COUNTER MODE FOR TIMER 1,
                       61                                     ;  16-BIT TIMER MODE FOR TIMER 0 SELECTED
0049 7520F4            62             MOV     SUB_DIV,#244    ; SUBDIVIDE INTERRUPT RATE BY 244 FOR 1 HZ
004C D2A9              63             SETB    ET0             ; USE TIMER 0 OVERFLOWS TO INTERRUPT PROGRAM
004E D2AF              64             SETB    EA              ; CONFIGURE IE TO GLOBALLY ENABLE INTERRUPTS
0050 D28C              65             SETB    TR0             ; KEEP INSTRUCTION CYCLE COUNT UNTIL OVERFLOW
0052 80FE              66             SJMP    $               ; START BACKGROUND PROGRAM EXECUTION
                       67     ;
                       68     ;
0054 D52038            69     UPDATE: DJNZ    SUB_DIV,TOSERV  ; EXECUTE SYSTEM TEST ONLY ONCE PER SECOND
0057 7520F4            70             MOV     SUB_DIV,#244    ; GET VALUE FOR NEXT ONE SECOND DELAY AND
                       71     ;                               ; GO THROUGH ELECTRICAL SYSTEM TEST CODE:
005A 4390E0            72             ORL     P1,#11100000B   ; SET CONTROL OUTPUTS HIGH
005D 43A007            73             ORL     P2,#00000111B
0060 C295              74             CLR     L_FRNT          ; FLOAT DRIVE COLLECTOR
0062 20B428            75             JB      T0,FAULT        ; T0 SHOULD BE PULLED LOW
0065 D295              76             SETB    L_FRNT          ; PULL COLLECTOR BACK DOWN
0067 C297              77             CLR     L_DASH          ; REPEAT SEQUENCE FOR L_DASH,
0069 20B421            78             JB      T0,FAULT
006C D297              79             SETB    L_DASH
006E C2A1              80             CLR     L_REAR          ;               L_REAR,
0070 20B41A            81             JB      T0,FAULT
0073 D2A1              82             SETB    L_REAR
0075 C296              83             CLR     R_FRNT          ;               R_FRNT,
0077 20B413            84             JB      T0,FAULT
007A D296              85             SETB    R_FRNT
007C C2A0              86             CLR     R_DASH          ;               R_DASH,
007E 20B40C            87             JB      T0,FAULT
0081 D2A0              88             SETB    R_DASH
0083 C2A2              89             CLR     R_REAR          ;               AND R_REAR.
0085 20B405            90             JB      T0,FAULT
0088 D2A2              91             SETB    R_REAR
                       92     ;
                       93     ;       WITH ALL COLLECTORS GROUNDED, T0 SHOULD BE HIGH
                       94     ;       IF SO, CONTINUE WITH INTERRUPT ROUTINE.
                       95     ;
008A 20B402            96             JB      T0,TOSERV
008D B2A3              97     FAULT:  CPL     S_FAIL          ; ELECTRICAL FAILURE PROCESSING ROUTINE
                       98                                     ; (TOGGLE INDICATOR ONCE PER SECOND)
                       99 +1  $EJECT
```

```
LOC  OBJ            LINE     SOURCE

                     100      ;       CONTINUE WITH INTERRUPT PROCESSING:
                     101      ;
                     102      ;   1)  COMPUTE LOW BULB INTENSITY WHEN PARKING LIGHTS ARE ON.
                     103      ;
008F A201            104      TOSERV: MOV     C,SUB_DIV.1         ; START WITH 50 PERCENT,
0091 8200            105              ANL     C,SUB_DIV.0         ; MASK DOWN TO 25 PERCENT,
0093 7202            106              ORL     C,SUB_DIV.2         ; BUILD BACK TO 62.5 PERCENT,
0095 8292            107              ANL     C,PARK              ; GATE WITH PARKING LIGHT SWITCH,
0097 92D1            108              MOV     DIM,C               ; AND SAVE IN TEMP. VARIABLE.
                     109      ;
                     110      ;   2)  COMPUTE AND OUTPUT LEFT-HAND DASHBOARD INDICATOR.
                     111      ;
0099 A293            112              MOV     C,L_TURN            ; SET CARRY IF TURN
009B 7291            113              ORL     C,EMERG             ; OR EMERGENCY SELECTED.
009D 8207            114              ANL     C,LO_FREQ           ; IF SO, GATE IN 1 HZ SIGNAL
009F 9297            115              MOV     L_DASH,C            ; AND OUTPUT TO DASHBOARD.
                     116      ;
                     117      ;   3)  COMPUTE AND OUTPUT LEFT-HAND FRONT TURN SIGNAL.
                     118      ;
00A1 92D5            119              MOV     F0,C                ; SAVE FUNCTION SO FAR.
00A3 72D1            120              ORL     C,DIM               ; ADD IN PARKING LIGHT FUNCTION
00A5 9295            121              MOV     L_FRNT,C            ; AND OUTPUT TO TURN SIGNAL.
                     122      ;
                     123      ;   4)  COMPUTE AND OUTPUT LEFT-HAND REAR TURN SIGNAL.
                     124      ;
00A7 A290            125              MOV     C,BRAKE             ; GATE BRAKE PEDAL SWITCH
00A9 B073            126              ANL     C,/L_TURN           ; WITH TURN LEVER.
00AB 72D5            127              ORL     C,F0                ; INCLUDE TEMP. VARIABLE FROM DASH
00AD 72D1            128              ORL     C,DIM               ; AND PARKING LIGHT FUNCTION
00AF 92A1            129              MOV     L_REAR,C            ; AND OUTPUT TO TURN SIGNAL.
                     130      ;
                     131      ;   5)  REPEAT ALL OF ABOVE FOR RIGHT-HAND COUNTERPARTS.
                     132      ;
00B1 A294            133              MOV     C,R_TURN            ; SET CARRY IF TURN
00B3 7291            134              ORL     C,EMERG             ; OR EMERGENCY SELECTED.
00B5 8207            135              ANL     C,LO_FREQ           ; IF SO, GATE IN 1 HZ SIGNAL
00B7 92A0            136              MOV     R_DASH,C            ; AND OUTPUT TO DASHBOARD.
00B9 92D5            137              MOV     F0,C                ; SAVE FUNCTION SO FAR.
00BB 72D1            138              ORL     C,DIM               ; ADD IN PARKING LIGHT FUNCTION
00BD 9296            139              MOV     R_FRNT,C            ; AND OUTPUT TO TURN SIGNAL.
00BF A290            140              MOV     C,BRAKE             ; GATE BRAKE PEDAL SWITCH
00C1 B094            141              ANL     C,/R_TURN           ; WITH TURN LEVER.
00C3 72D5            142              ORL     C,F0                ; INCLUDE TEMP. VARIABLE FROM DASH
00C5 72D1            143              ORL     C,DIM               ; AND PARKING LIGHT FUNCTION
00C7 92A2            144              MOV     R_REAR,C            ; AND OUTPUT TO TURN SIGNAL.
                     145      ;
                     146      ;       RESTORE STATUS REGISTER AND RETURN.
                     147      ;
00C9 D0D0            148              POP     PSW                 ; RESTORE PSW
00CB 32              149              RETI                        ; AND RETURN FROM INTERRUPT ROUTINE
                     150      ;
                     151              END
```

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


NAME      TYPE     VALUE AND REFERENCES

BRAKE .   N BSEG   0090H  20# 125 140
DIM . .   N BSEG   00D1H  45# 108 120 128 138 143
EA. . .   N BSEG   00AFH  64
EMERG .   N BSEG   0091H  21# 113 134
ETO . .   N BSEG   00A9H  63
FO. . .   N BSEG   00D5H  119 127 137 142
FAULT .   L CSEG   008DH  75 78 81 84 87 90 97#
HI_FREQ   N BSEG   0000H  42#
INIT. .   L CSEG   0040H  50 58#
L_DASH.   N BSEG   0097H  32# 77 79 115
L_FRNT.   N BSEG   0095H  30# 74 76 121
L_REAR.   N BSEG   00A1H  34# 80 82 129
L_TURN.   N BSEG   0093H  23# 112 126
LO_FREQ   N BSEG   0007H  43# 114 135
P1. . .   N DSEG   0090H  20 21 22 23 24 30 31 32 72
P2. . .   N DSEG   00A0H  33 34 35 37 73
PARK. .   N BSEG   0092H  22# 107
PSW . .   N DSEG   00D0H  45 54 148
R_DASH.   N BSEG   00A0H  33# 86 88 136
R_FRNT.   N BSEG   0096H  31# 83 85 139
R_REAR.   N BSEG   00A2H  35# 89 91 144
R_TURN.   N BSEG   0094H  24# 133 141
S_FAIL.   N BSEG   00A3H  37# 97
SUB_DIV   N DSEG   0020H  41# 42 43 62 69 70 104 105 106
TO. . .   N BSEG   00B4H  75 78 81 84 87 90 96
TOSERV.   L CSEG   008FH  69 96 104#
THO . .   N DSEG   008CH  53 59
TLO . .   N DSEG   008AH  58
TMOD. .   N DSEG   0089H  60
TRO . .   N BSEG   008CH  65
UPDATE.   L CSEG   0054H  55 69#


ASSEMBLY COMPLETE, NO ERRORS FOUND
```

Introduction

A recent NASA invention enables considerable energy savings when using the common induction electric motor. It can be used with existing motors, since it requires no modification to the motor. Typical energy savings of 10 to 60% can be realized, depending on the amount of motor loading present. This invention is the direct result of an analysis, by NASA, of Solar Heating and Cooling Systems to reduce the power consumed by pump and fan motors used in these systems. It is applicable to both single phase and 3 phase motors.

Since the induction motor is widely used in many of the same systems that can take advantage of the 8022 microcontroller, this invention can provide a significant benefit to many commercial and consumer products. Examples include the following:

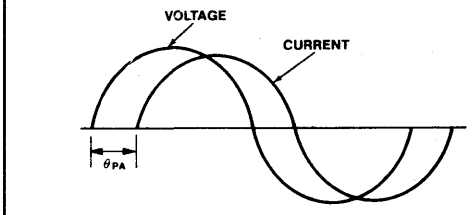
—space heating and cooling systems
—heat pumps
—solar collector controllers
—liquid or chemical process control
—large industrial motor control
—refrigeration units
—swimming pool controllers
—washing machines
—dishwashers.

This application note will explain how this invention can be implemented with an 8022 microcontroller with a minimum amount of external hardware. The note is organized into the following sections:

1. Power Factor Controller theory of operation.
2. Description of the 8022 microcontroller in this application.
3. Hardware description.
4. Software description and listing.
5. Conclusion.

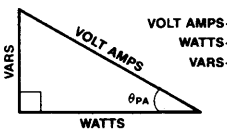
Theory of Operation

The concept of the Power Factor Controller (PFC), conceived and developed by NASA aerospace engineer Frank Nola,[1] is to reduce the voltage applied to the motor when it is partially loaded, resulting in significant energy savings. At full load, the induction motor must have a high flux density in order to perform adequately. Under this condition the motor is running at peak efficiency with a power factor of about 0.8. At less than full load, however, this high flux density is still supported by the high current set up in the field coils. The resulting power factor can drop to as low as 0.1 (FIG. 2). The losses associated with this high current, under the lightly loaded condition, is primarily in the form of heat. This is a loss which can be paid for twice if the motor is in a refrigerated or otherwise cooled system.



θPA is the phase angle between the voltage and current waveforms (the current will always lag the voltage in an induction motor).

Figure 1.



VOLT AMPS—APPARANT POWER—IEFF VEFF
WATTS— AVERAGE POWER —IEFF VEFF COS θPA
VARS— REACTIVE POWER —IEFF VEFF SIN θPA

The Power Factor is equal to the cosine of the phase angle. It represents the ratio of the total power required to be supplied by the power company to the power recorded by a watt meter.

$$\text{Power Factor} = \frac{\text{Average Power}}{\text{Apparent Power}} = \text{Cos}\,\theta_{PA}$$

The Power Factor ranges from 0 to 1. For a purely resistive load, the voltage and current are in phase, θ_{PA} is zero, and the P.F. is unity. the Apparent Power and Average Power are equal.

For a purely reactive load, that is, one containing no resistance, the phase difference between the voltage and current is either plus or minus 90° (purely inductive being plus 90°, purely capacitive being minus 90°), the P.F. is therefore zero, resulting in the Average Power being zero.

Between these two extreme cases, there are the general networks for which the P.F. can range from zero to unity. Therefore, for a constant magnitude of the current and voltage, the Average Power (Watts) can range from zero to a maximum value equal to the Apparent Power, depending on the phase angle.

Figure 2. Complex Power and the Power Triangle

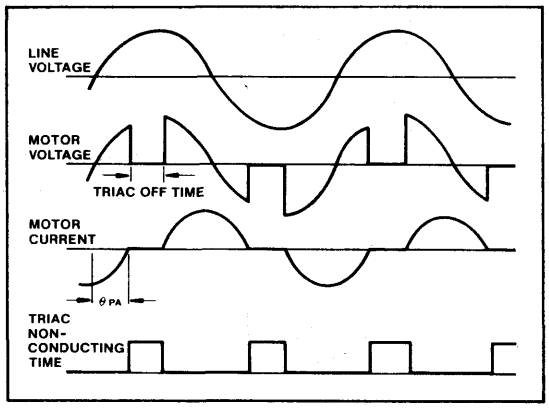


Figure 3.

[1] NASA Tech Brief MFS-23280, Power Factor Controller. Copies may be obtained by writing to: Director, Technology Utilization Office, Marshall Space Flight Center, AL, 35812. The patent for the PFC is owned by the U.S. Government. Licenses for commercial development are available at no charge. Contact: Patent Counsel, Marshall Space Flight Center, AL, 35812.

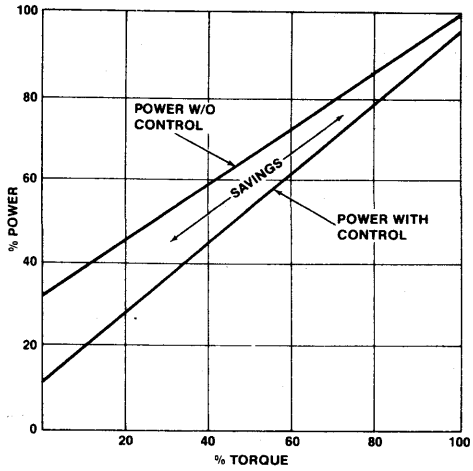
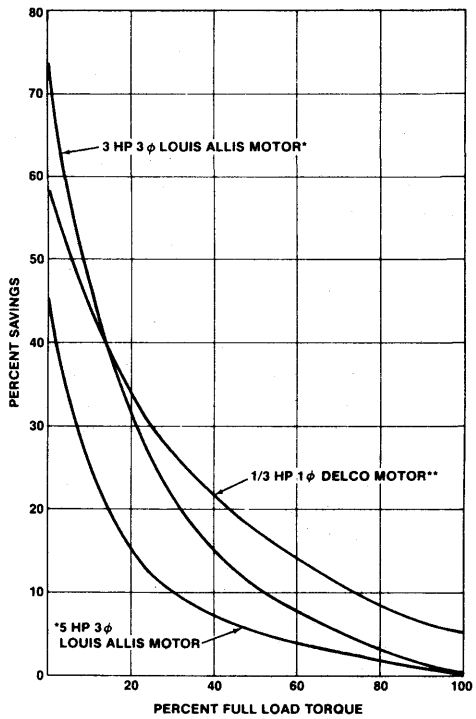


Figure 4. Typical Power Savings [2] for Single Phase Motor

In this partial load condition, the motor would produce the same torque at essentially the same speed even if much weaker magnetic forces were set up, by decreasing the current which produces these fields. The electric motor on its own, can't recognize this condition, however, and will continue to draw near the high current used under full load, even when operating under no load.

The principle of operation of the PFC is to measure the shift in phase angle between motor voltage and current



*DATA OBTAINED FROM AUBURN UNIVERSITY REPORT
**NASA/MARSHALL SPACE FLIGHT CENTER (MSFC) DATA

Figure 5. Power Factor Controller Percent Savings Vs. Torque for Various Motors[3]

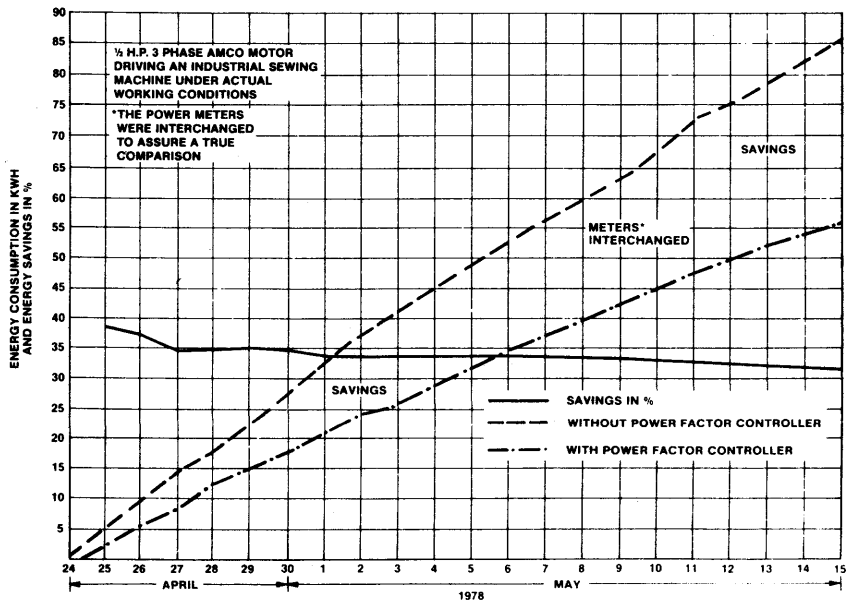


Figure 6. Energy Consumption as a Function of Time[4]

[2]NASA TECH BRIEF MFS-23280
[3]Ibid.
[4]Ibid.

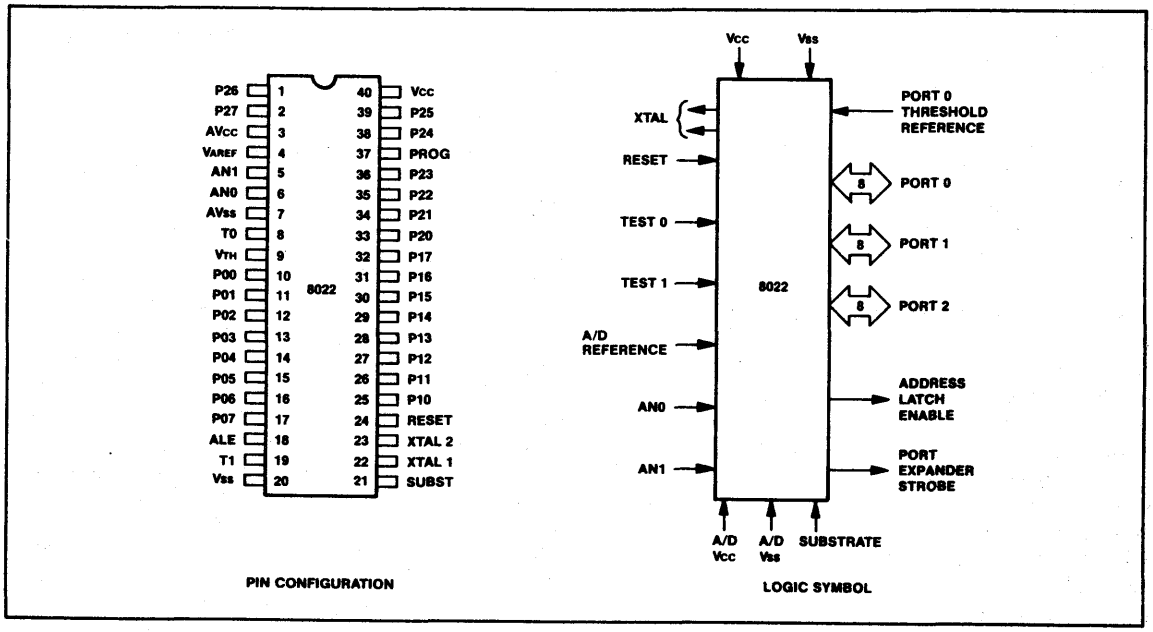


Figure 7.

as the load changes. As the monitored phase angle increases, a solid-state switch (triac) is used to reduce the voltage applied to the motor (FIG. 3). This increases motor slip and reduces the phase angle to a predetermined value. The feedback loop forces the motor to run at a constant, optimum, phase angle selected by the user, thereby minimizing wasted power. Since phase shift can be sampled as often as every power-line cycle, the 8022 can immediately respond to changes in the load and the motor will always run at near-peak efficiency, supplying only the amount of power required.

The PFC was originally tested at the Marshall Space Flight Center on 40 motors ranging from $^1/_{12}$ hp to 5 hp, both single phase and 3 phase.[5] Most motors exhibited a 40–60% savings at no load and 0–10% savings when fully loaded. The more expensively built 3-phase motors being the more efficient. When the device was used in 15 typical applications with single phase motors, including a drill press and vacuum pump, the typical savings were 30–40% when idling and 10% when loaded. The vacuum pump even ran about 25 degrees C cooler with the device. In a 500 hour test of two identical machines performing the same task, the piece of equipment with the PFC used 33% less energy (FIG. 6).

The device was also tested extensively by the Auburn University and many companies, all coming to the same conclusion. The Power Factor Controller is proven to offer significant savings to the user that, by the way, go beyond the actual direct power reduction mentioned. These multiple savings include the motor running cooler and quieter, extending the motor's life as well as saving in an air conditioned environment since there is less heat generated, and also by actually controlling the power factor to a desired value. This will benefit large users of motors with cyclic loads who are often charged by the Power Company for a poor power factor and are forced to correct this condition with large capacitor banks or a synchronous condenser.

[5]Ibid.

The 8022

The PFC developed by NASA consists of discrete electronic parts, designed to produce an analog controller. The 8022 microcontroller brings the benefit of a programmable computer to this application so that intelligent control of the total system can be effected. The 8022 is especially suited for an analog environment as it contains a 2-channel 8-bit A/D converter, zero-cross detection circuitry, and comparitor inputs with a controllable threshold. All of these capabilities are integrated into a single chip, along with 2048 bytes of program storage, 64 bytes of RAM, an 8-bit internal timer/event counter, external interrupt input, two high current drive outputs, and three 8-bit I/O ports. The result is a complete microcomputer system on a single chip. If the reader desires more information about the 8022, (s)he may refer to the MCS-48 User's Manual and application note AP-56, "Designing With Intel's 8022 Microcontroller," for a complete description of the 8022.

The 8022 is already being used to control many systems which utilize an induction motor. The extra code required for the PFC operation is only 154 bytes (less than 8% of the available program store). The main program would, of course, have to be modified to facilitate the PFC's function. This can easily be performed by placing the PFC operation in an interrupt routine. The total number of pins delegated to the PFC operation is 4: three I/O pins and the T1 zero-cross input. One of these I/O pins is dedicated to control Vth.

These three interface lines: the voltage-cross input, the current-cross input, and the triac gate control, are all that is required for the PFC application. The 8022 simply measures the amount of lag time between the voltage zero-crossing and the current zero-crossing. This measured time is then converted to a phase angle and subsequently compared to the desired phase angle. As the load changes and the measured phase angle shifts either greater than or less than the desired value, the 8022 will either lengthen or shorten the triac off time. The result is that the motor only gets the amount of current needed to drive the instantaneous load.

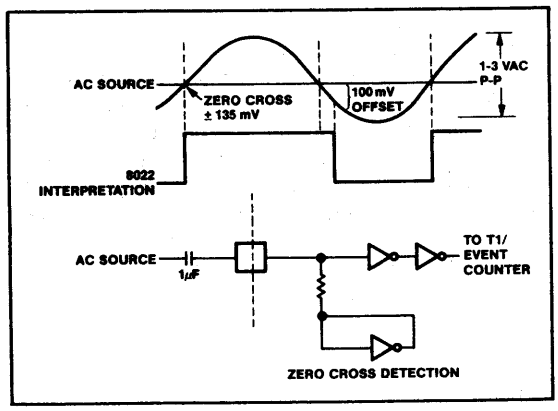


Figure 8.

Hardware Description

To perform the PFC function, the motor was placed in series with a triac which cuts out portions of the applied voltage, and a small series resistor which was used for measuring the current (FIG. 9). To isolate the 8022 from the AC line voltage, a small audio transformer was placed across the current sensing resistor, and an opto-isolator was used between the gate of the triac and the pin driving it. The voltage signal was taken from the secondary side of the power supply transformer.

Aside from the isolation, the only other interface hardware was some simple signal conditioning. Since the T1 zero-cross input requires a 1-3 VAC p-p signal, two diodes to DC ground clipped the AC signal nicely at 1.5 VAC p-p. A resistor for current limiting and a 1 microfarad capacitor to T1 complete the voltage zero-cross.

One of the port 0 comparator inputs was used for the current zero-cross detection. The 8022 had to be able to recognize when the current waveform was approaching zero from both the positive side and the negative side since it doesn't cross zero at any particular point, but rather approaches it from either side then remains zero until the triac fires again (FIG. 3). This was done by having two separate thresholds for the comparator. One for the negative slope crossing and one for the positive. To accomplish this, the current waveform was first boosted up with a DC level so it would be entirely positive. VTH was then biased to this DC level by two resistors. The threshold would be changed by about 0.1v either side of this DC level under software control by writing either a logic "1" or "0" to P17 when anticipating either a positive approach or a negative approach, respectively. This would place the 100k ohm resistor in parallel with either resistor thereby decreasing its value and subsequently raising or lowering the threshold.

The remaining hardware was included to aid in the development of this application note, but is not necessary for the system's operation. The remaining 6 pins on port zero were used to input the desired phase angle. The remaining 7 pins on port 1 were used to control the 7 segments of a display. The upper nibble of port 2 was used to control up to 10 digits of a display. The user would then select the desired phase angle at will, with the actual phase angle being written to the display.

With this pin assignment the remaining pins include the lower nibble of port 2 which can be used with the Intel 8243 I/O expander, adding four 4-bit I/O ports. The T0 pin can still be used as either a testable input, the interrupt request pin, or both. Finally, the two channels of the A/D converter are available, allowing the chip to interface directly with analog transducers. Aside from the Power Factor Controller and the direct user interface with the 8022 (via a keypad or thumbwheel switches and the display), there are still enough pins left over for almost any controller application.

Hardware not required for the PFC operation is shaded in the schematic.

Software Description

To simplify the software, the triac "offtime" is quantized in units of 0.27ms. This represents one time-unit of the timer (operating with a 3.58 MHz TV crystal). The "offtime" is therefore incremented or decremented by one of these units (or remained unchanged) when it is

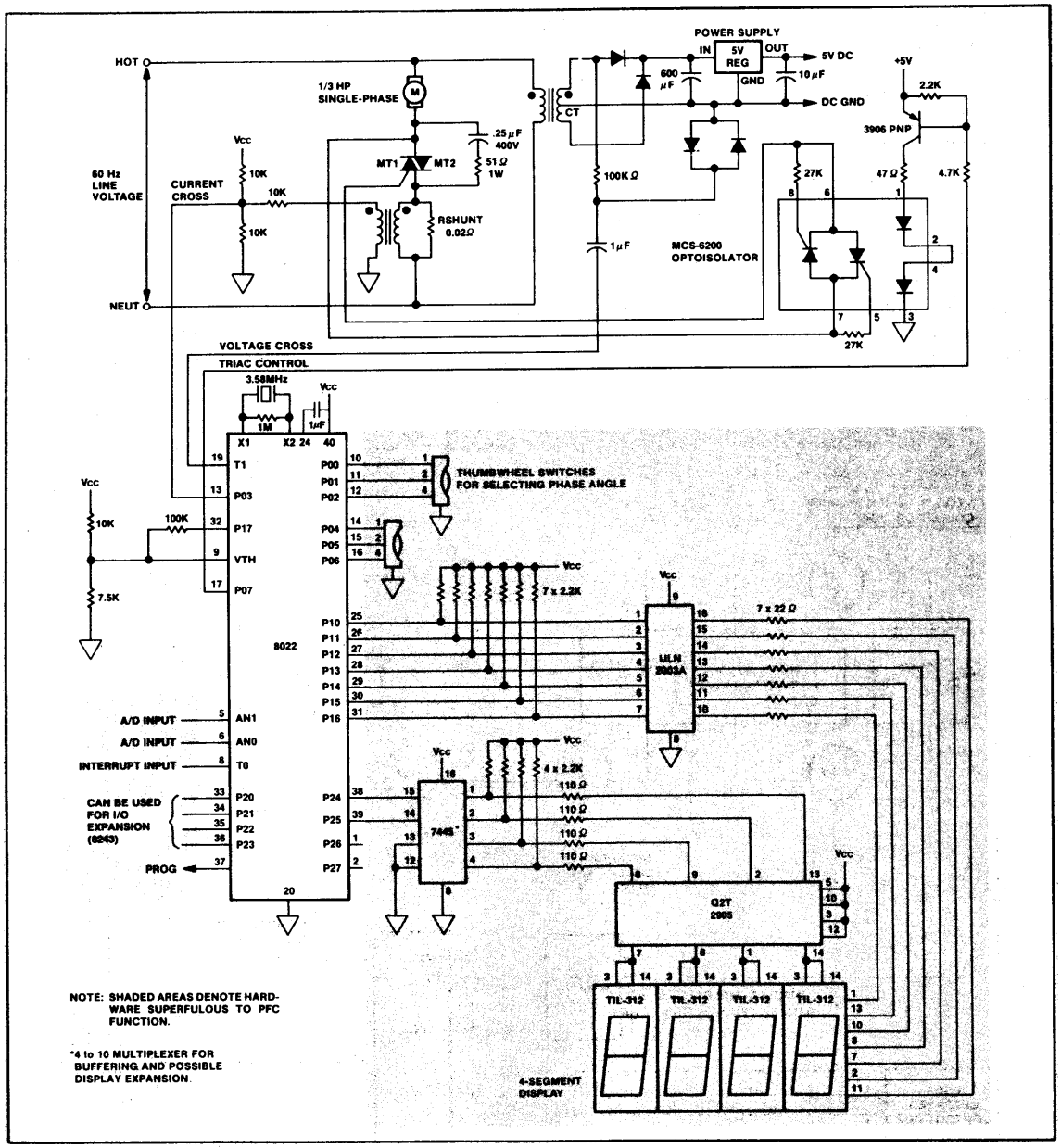


Figure 9.

reviewed for adjustment every cycle. This proves to be an adequate reaction to abrupt changes in the load. If however, the phase angle will only be sampled every 3 cycles, for example, on an interrupt, a simple routine has been included to change the "offtime" by more than one increment should the phase angle change more than 12 degrees between samplings. This is included to compensate for extremely abrupt load changes.

For simplicity, the phase angle is also measured in units of approximately 6 degrees, to coincide with the timer unit of 0.27ms. The lookup table consists of these quantized values. For a steady phase angle, the desired phase angle should be chosen to be one of these values. If any value between two of these are chosen, the phase angle oscillates between these two values and will never equal the chosen value, since this value is not in the lookup

table. This can be rectified by using a table lookup and interpolation instead of a simple table lookup (see AP-24; "Application Techniques for the MCS-48 Family"). There is no apparent benefit in doing this, however, as the value-searching over a 6 degree range doesn't seem to cause any problems.

The software also distinguishes between that necessary for the PFC only, and that used to include the user interface features. The BCD-to-binary routine was adapted from AP-49, "Serial I/O and Math Utilities for the 8049 Microcomputer," as was the binary-to-BCD routine. The total bytes of program storage consumed by the PFC only, is 154, and the total for this entire application is 328.

The software flowchart and complete listing follow.

Conclusion

The advantages of controlling a system with a programable microcontroller are evident. The added advantage of conserving power with the 8022 in the same application is realized in this application note. As the Power Factor Controller will be used in more and more applications as an energy-saving feature, the 8022 is ideally suited to implement it.

To illustrate this, the following application shows the 8022 controlling a Heat Pump / Solar Heating system.

Figure 10 illustrates the type of solar heating system that was the target of the NASA invention. The 8022 can control this system, and in addition, implement the PFC function on the motors driving the heatpump compressor and fan. The A/D can be used to monitor the temperatures in the system, and the various pumps and fans can be activated so as to maximize the heat extracted from the solar radiation. As an example of this, the flow rate in the solar-collector circuit can be adjusted by the 8022 so that heat can be collected even on cloudy days. (Lowering the flow rate lets the fluid remain in the collectors longer so that the heat radiation has a longer period in which to heat the fluid.) The 2K bytes of program storage available on the 8022 would make the addition of features such as a real-time clock for controlling the temperature to different levels at different times of the day, and dual-level thermostat settings, possible at little or no incremental product cost.

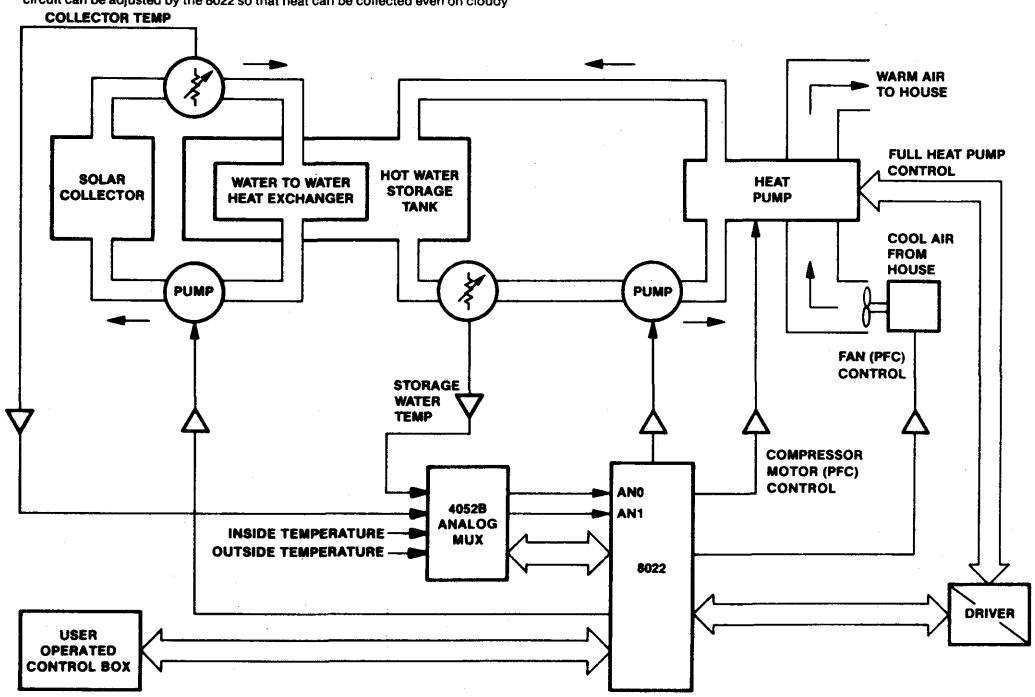
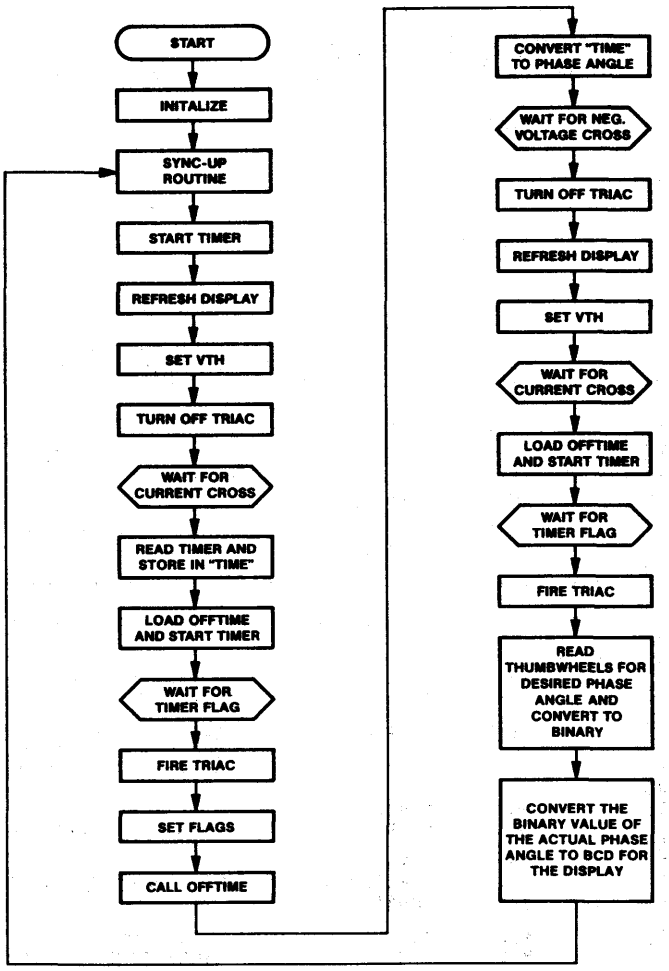


Figure 10.



Program Flow Chart

```
ASM48 AP PAGELENGTH(88)

ISIS-II MCS-48/UPI-41 MACRO ASSEMBLER, V2.0                PAGE    1

  LOC  OBJ          SEQ          SOURCE STATEMENT

                      1 $MOD22 SYMBOLS MACROFILE XREF
                      2
                      3 ;*****************************************************************
                      4 ;       DEFINITIONS OF MACROS
                      5 ;*****************************************************************
                      6
                      7 LDA       MACRO   PAR1
                      8           MOV     R0,#PAR1
                      9           MOV     A,@R0
                     10           ENDM
                     11
                     12 STA       MACRO   PAR1
                     13           MOV     R0,#PAR1
                     14           MOV     @R0,A
                     15           ENDM
                     16
0000 1498            17 INIT:     CALL    INITZ              ;INITIALIZE
                     18
                     19 ;*****************************************************************
                     20 ;           THE FOLLOWING IS A SYNC-UP ROUTINE TO ASSURE
                     21 ;            A START ON A RAISING VOLTAGE ZERO-CROSS
                     22 ;*****************************************************************
                     23
0002 5606            24 PWRON:    JT1     VOWAIT             ;JUMP ON T1=1
0004 0402            25           JMP     PWRON
                     26
0006 460A            27 VOWAIT:   JNT1    WAITST             ;JUMP ON T1=0
0008 0406            28           JMP     VOWAIT
                     29
000A 560E            30 WAITST:   JT1     START              ;JUMP ON T1=1
000C 040A            31           JMP     WAITST
                     32
                     33 ;*****************************************************************
                     34 ;         END OF SYNC-UP                ==START PROGRAM==
                     35 ;*****************************************************************
                     36
000E 27              37 START:    CLR     A
000F 62              38           MOV     T,A                ;CLEAR PRESCALER
0010 55              39           STRT    T                  ;START TIMER
                     40
0011 3400            41           CALL    REFRSH             ;REFRESH THE DISPLAY
                     42                                      ;VTH SET FOR UPCOMING CURRENT AUTO-
                     43                                      ;MATICALLY IN REFRSH ROUTINE (P17=0)
                     44
0013 23FF            45           MOV     A,#0FFH
0015 90              46           OUTL    P0,A               ;TURN OFF TRIAC
                     47
0016 08              48 IPWAIT:   IN      A,P0
0017 5308            49           ANL     A,#00001000B       ;WAIT FOR CURRENT CROSS
0019 C616            50           JZ      IPWAIT
                     51
001B 42              52           MOV     A,T                ;READ TIMER
001C AA              53           MOV     TIME,A             ;STORE TIMER VALUE
                     54
001D 14BB            55           CALL    TRIAC
                     56
                     57 $EJECT
```

NOTE: SHADED AREAS DENOTE SOFTWARE SUPERFULOUS TO PFC FUNCTION.

```
ISIS-II MCS-48/UPI-41 MACRO ASSEMBLER, V2.0                PAGE    2

 LOC  OBJ          SEQ          SOURCE STATEMENT

                    58
                    59 ;*************************************************************************
                    60 ;                          SET THE FLAGS
                    61 ;*************************************************************************
                    62
 001F 97            63           CLR     C                  ;CLEAR THE CARRY FLAG
                    64
 0020 FC            65           MOV     A,SETVAL
 0021 37            66           CPL     A
 0022 17            67           INC     A                  ;2'S COMPLIMENT SETVAL
 0023 6D            68           ADD     A,PHSANG           ;PHSANG-SETVAL = DELTA
                    69
 0024 C64F          70           JZ      LOOK               ;IF PHSANG=SETVAL, DONT CHANGE ANYTHING
                    71
                    72           STA     DELTA
 0026 B826          73+          MOV     R0,#DELTA
 0028 A0            74+          MOV     @R0,A
                    75
 0029 5380          76           ANL     A,#10000000B       ;TEST FOR NEG
 002B C638          77           JZ      OTCALC             ;IF POS DO NOT SET FLAG
 002D 1E            78           INC     PHLTSV             ;SET PHSANG L.T. SETVAL FLAG
                    79
                    80           LDA     DELTA
 002E B826          81+          MOV     R0,#DELTA
 0030 F0            82+          MOV     A,@R0
 0031 030A          83           ADD     A,#10D             ;10-DELTA
 0033 5380          84           ANL     A,#10000000B
 0035 C638          85           JZ      OTCALC
 0037 1F            86           INC     DLGT10             ;SET DELTA G.T. 10 FLAG IF NEG
                    87
                    88 ;*************************************************************************
                    89 ;                  CHANGE OFFTIME PER FLAGS NEWLY SET
                    90 ;*************************************************************************
                    91
 0038 FE            92 OTCALC:   MOV     A,PHLTSV           ;DECREMENT OR INCREMENT?
 0039 C649          93           JZ      DECREM             ;IF PHLTSV IS CLEAR THEN DECREMENT
                    94
 003B FB            95 INCREM:   MOV     A,OFFTIM           ;IF OFFTIME=255 OR 254 DON'T INCREMENT
 003C 37            96           CPL     A
 003D C64C          97           JZ      CLRFLG
 003F 07            98           DEC     A
 0040 C64C          99           JZ      CLRFLG
                   100
 0042 1B           101           INC     OFFTIM             ;INCREMENT
                   102
 0043 FF           103           MOV     A,DLGT10           ;SHOULD IT INCREMENT TWICE?
 0044 C64C         104           JZ      CLRFLG
 0046 1B           105           INC     OFFTIM
 0047 044C         106           JMP     CLRFLG
                   107
 0049 FB           108 DECREM:   MOV     A,OFFTIM
 004A 07           109           DEC     A
 004B AB           110           MOV     OFFTIM,A           ;DECREMENT
                   111
 004C 27           112 CLRFLG:   CLR     A
 004D AE           113           MOV     PHLTSV,A
 004E AF           114           MOV     DLGT10,A           ;CLEAR FLAGS
                   115
                   116 $EJECT
```

```
ISIS-II MCS-48/UPI-41 MACRO ASSEMBLER, V2.0                PAGE    3

  LOC  OBJ          SEQ          SOURCE STATEMENT

                    117
                    118 ;*****************************************************************
                    119 ;                LOOKUP VALUE FOR "PHSANG" FROM "TIME"
                    120 ;*****************************************************************
                    121
  004F 23C9         122 LOOK:    MOV      A,#(LOW TAB)     ;BOTTOM OF TABLE
  0051 6A           123          ADD      A,TIME
  0052 A3           124          MOVP     A,@A             ;GET PHSANG VALUE
  0053 AD           125          MOV      PHSANG,A
                    126
                    127 ;*****************************************************************
                    128 ;                WAIT FOR NEGATIVE GOING VOLTAGE CROSS
                    129 ;*****************************************************************
                    130
  0054 4658         131 HOLD:    JNT1     CONTON
  0056 0454         132          JMP      HOLD
                    133
  0058 23FF         134 CONTON:  MOV      A,#0FFH          ;TURN OFF TRIAC
  005A 90           135          OUTL     P0,A
                    136
  005B 3400         137          CALL     REFRSH           ;REFRESH THE DISPLAY
                    138          LDA      SEGDAT
  005D B824         139+         MOV      R0,#SEGDAT
  005F F0           140+         MOV      A,@R0
  0060 4380         141          ORL      A,#10000000B     ;SET VTH FOR DOWN GOING CURRENT-
  0062 39           142          OUTL     P1,A             ;BY SETTING P17
                    143
  0063 08           144 INWAIT:  IN       A,P0             ;WAIT FOR CURRENT CROSS
  0064 5308         145          ANL      A,#00001000B
  0066 9663         146          JNZ      INWAIT
                    147
  0068 14BB         148          CALL     TRIAC
                    149
                    150 ;*****************************************************************
                    151 ;                   READ THUMBWHEELS
                    152 ;                AND CONVERT TO BINARY
                    153 ;*****************************************************************
                    154
  006A 08           155          IN       A,P0
  006B 5377         156          ANL      A,#01110111B
                    157          STA      BINTR            ;STORE ACC IN BINTR
  006D B820         158+         MOV      R0,#BINTR
  006F A0           159+         MOV      @R0,A
                    160
  0070 14A7         161          CALL     SELRB1
                    162
                    163          LDA      BINTR
  0072 B820         164+         MOV      R0,#BINTR
  0074 F0           165+         MOV      A,@R0
                    166
  0075 344A         167          CALL     CONBIN           ;CONVERT THUMBWHEELS TO BINARY
                    168
                    169          STA      BINTR            ;STORE ACC IN BINTR
  0077 B820         170+         MOV      R0,#BINTR
  0079 A0           171+         MOV      @R0,A
                    172
  007A 14B1         173          CALL     SELRB0
                    174
                    175          LDA      BINTR            ;MOV BINTR TO ACC
  007C B820         176+         MOV      R0,#BINTR
  007E F0           177+         MOV      A,@R0
  007F AC           178          MOV      SETVAL,A
                    179
                    180 $EJECT
```

```
ISIS-II MCS-48/UPI-41 MACRO ASSEMBLER, V2.0                PAGE    4

 LOC  OBJ          SEQ          SOURCE STATEMENT

                    181
                    182 ;*****************************************************************
                    183 ;       CONVERT BINARY VALUE OF THE PHASE ANGLE (PHSANG)
                    184 ;                          TO BCD (ANGLE)
                    185 ;*****************************************************************
                    186
 0080 FD            187           MOV     A,PHSANG
                    188           STA     BCDTR           ;STORE PHASE ANGLE
 0081 B821          189+          MOV     R0,#BCDTR
 0083 A0            190+          MOV     @R0,A
                    191
 0084 14A7          192           CALL    SELRB1
                    193
                    194           LDA     BCDTR
 0086 B821          195+          MOV     R0,#BCDTR
 0088 F0            196+          MOV     A,@R0
                    197
 0089 3437          198           CALL    CNBCD
                    199
                    200           STA     BCDTR
 008B B821          201+          MOV     R0,#BCDTR
 008D A0            202+          MOV     @R0,A
                    203
 008E 14B1          204           CALL    SELRB0
                    205
                    206           LDA     BCDTR
 0090 B821          207+          MOV     R0,#BCDTR
 0092 F0            208+          MOV     A,@R0
                    209           STA     ANGLE
 0093 B823          210+          MOV     R0,#ANGLE
 0095 A0            211+          MOV     @R0,A
                    212
                    213
 0096 040A          214           JMP     WAITST          ;GO BACK TO BEGINNING AND REPEAT
                    215
                    216 ;*****************************************************************
                    217 ;*****************************************************************
                    218 ;*****************************************************************
                    219 ;   ------------------- END OF MAIN PROGRAM --------------------
                    220 ;*****************************************************************
                    221 ;*****************************************************************
                    222 ;*****************************************************************
                    223
                    224 $EJECT
```

```
ISIS-II MCS-48/UPI-41 MACRO ASSEMBLER, V2.0                PAGE    5

  LOC  OBJ          SEQ          SOURCE STATEMENT

                    225
                    226 INITZ:
                    227
                    228 ;*****************************************************************
                    229 ;                    ALLOCATE    RAM
                    230 ;*****************************************************************
                    231
  0020              232 BINTR    EQU     20H
  0021              233 BCDTR    EQU     21H
  0022              234 PASTOR   EQU     22H
  0023              235 ANGLE    EQU     23H
  0024              236 SEGDAT   EQU     24H
  0025              237 DISFLG   EQU     25H
  0026              238 DELTA    EQU     26H
                    239
  0002              240 TIME     EQU     R2
  0003              241 OFFTIM   EQU     R3
  0004              242 SETVAL   EQU     R4
  0005              243 PHSANG   EQU     R5
  0006              244 PHLTSV   EQU     R6
  0007              245 DLGT10   EQU     R7
                    246
                    247 ;*****************************************************
                    248 ;       FOR USE IN CONBIN AND CONBCD
                    249 ;       ***********************
                    250 ;
  0002              251 RA       EQU     R2
  0003              252 CONBY    EQU     R3
  0004              253 TCNT     EQU     R4
  0001              254 DIGPN    EQU     1
                    255
                    256 ;*****************************************************
                    257
  0098 BC29         258          MOV     R4,#29H            ;SETVAL=41
  009A BD29         259          MOV     R5,#29H            ;PHSANG=41
  009C BBFF         260          MOV     R3,#0FFH           ;SET OFFTIM TO 255
                    261
  009E 237F         262          MOV     A,#7FH             ;TURN ON TRIAC
  00A0 90           263          OUTL    P0,A
                    264
  00A1 97           265          CLR     C
  00A2 27           266          CLR     A
                    267          STA     DISFLG             ;CLEAR DISPLAY FLAG
  00A3 B825         268+         MOV     R0,#DISFLG
  00A5 A0           269+         MOV     @R0,A
                    270
  00A6 83           271          RET
                    272
                    273 $EJECT
```

```
ISIS-II MCS-48/UPI-41 MACRO ASSEMBLER, V2.0                PAGE    6

  LOC  OBJ          SEQ          SOURCE STATEMENT

                     274
                     275 ;*****************************************************************
                     276 ;         THESE NEXT TWO SUBROUTINES ACT LIKE A BANK SELECT
                     277 ;              EXCEPT THAT R0 AND R1 GET WIPED OUT
                     278 ;*****************************************************************
                     279
                     280
                     281
                     282
                     283
                     284
                     285 SELRB1:
  00A7 B918          286             MOV     R1,#18H
  00A9 B807          287             MOV     R0,#07H
  00AB F0            288 BAK1:       MOV     A,@R0
  00AC A1            289             MOV     @R1,A
  00AD 19            290             INC     R1
  00AE E8AB          291             DJNZ    R0,BAK1
  00B0 83            292             RET
                     293
                     294 ;*****************************************************************
                     295
                     296
                     297 SELRB0:
  00B1 B918          298             MOV     R1,#18H
  00B3 B807          299             MOV     R0,#07H
  00B5 F1            300 BAK2:       MOV     A,@R1
  00B6 A0            301             MOV     @R0,A
  00B7 19            302             INC     R1
  00B8 E8B5          303             DJNZ    R0,BAK2
  00BA 83            304             RET
                     305
                     306 $EJECT


ISIS-II MCS-48/UPI-41 MACRO ASSEMBLER, V2.0                PAGE    7

  LOC  OBJ          SEQ          SOURCE STATEMENT

                     307
                     308 ;*****************************************************************
                     309 ;                  THIS ROUTINE FIRES THE TRIAC AFTER A
                     310 ;                          PREDETERMINED OFFTIME
                     311 ;*****************************************************************
                     312
                     313
                     314
                     315
                     316 TRIAC:
                     317
  00BB 65            318             STOP    TCNT            ;STOP TIMER
  00BC 16BE          319             JTF     $+2             ;CLEAR TIMER FLAG
                     320
  00BE FB            321             MOV     A,OFFTIM        ;SET PRESCALER TO OFFTIME
  00BF 62            322             MOV     T,A
                     323
  00C0 55            324             STRT    T               ;START TIMER
                     325
  00C1 16C5          326 CNTDWN:     JTF     FIRE            ;WAIT FOR TIMER FLAG
  00C3 04C1          327             JMP     CNTDWN
                     328
  00C5 237F          329 FIRE:       MOV     A,#7FH          ;FIRE
  00C7 90            330             OUTL    P0,A            ;TRIAC
                     331
  00C8 83            332             RET
                     333
  00C9               334 TAB         EQU     $
                     335
                     336 $EJECT
```

```
ISIS-II MCS-48/UPI-41 MACRO ASSEMBLER, V2.0                PAGE    8

 LOC  OBJ          SEQ          SOURCE STATEMENT

                    337
                    338 ;*************************************************************
                    339 ;*********      THIS ROUTINE REFRESHES THE DISPLAY     *******
                    340 ;*************************************************************
                    341
 0100               342          ORG     100H
                    343 REFRSH:
 0100 23F0          344          MOV     A,#0F0H
 0102 3A            345          OUTL    P2,A            ;TURN OFF DIGITS
                    346
                    347          LDA     DISFLG
 0103 B825          348+         MOV     R0,#DISFLG
 0105 F0            349+         MOV     A,@R0
 0106 C60E          350          JZ      NOSWAP          ;CHECK DISPLAY FLAG
                    351          LDA     ANGLE
 0108 B823          352+         MOV     R0,#ANGLE
 010A F0            353+         MOV     A,@R0
 010B 47            354          SWAP    A
 010C 2411          355          JMP     CONT2
                    356
                    357 NOSWAP: LDA      ANGLE
 010E B823          358+         MOV     R0,#ANGLE
 0110 F0            359+         MOV     A,@R0
 0111 530F          360 CONT2:   ANL     A,#0FH          ;MASK UPPER NIBBLE
 0113 032D          361          ADD     A,#(LOW TABL)
 0115 A3            362          MOVP    A,@A            ;GET CHAR CODE
                    363
                    364          STA     SEGDAT
 0116 B824          365+         MOV     R0,#SEGDAT
 0118 A0            366+         MOV     @R0,A
 0119 39            367          OUTL    P1,A            ;TURN ON THE SEGMENTS
                    368
                    369          LDA     DISFLG
 011A B825          370+         MOV     R0,#DISFLG
 011C F0            371+         MOV     A,@R0
 011D C623          372          JZ      DIG1
                    373
 011F 2310          374 DIG2:    MOV     A,#10H
 0121 2424          375          JMP     CONT3
                    376
 0123 27            377 DIG1:    CLR     A
                    378
 0124 3A            379 CONT3:   OUTL    P2,A            ;TURN ON THE DIGIT
                    380
                    381          LDA     DISFLG
 0125 B825          382+         MOV     R0,#DISFLG
 0127 F0            383+         MOV     A,@R0
 0128 37            384          CPL     A
                    385          STA     DISFLG          ;CHANGE DISPLAY FLAG
 0129 B825          386+         MOV     R0,#DISFLG
 012B A0            387+         MOV     @R0,A
                    388
 012C 83            389          RET
                    390
                    391 TABL:
 012D 3F            392          DB      00111111B       ; 0
 012E 06            393          DB      00000110B       ; 1
 012F 5B            394          DB      01011011B       ; 2
 0130 4F            395          DB      01001111B       ; 3
 0131 66            396          DB      01100110B       ; 4
 0132 6D            397          DB      01101101B       ; 5
 0133 7D            398          DB      01111101B       ; 6
 0134 07            399          DB      00000111B       ; 7
 0135 7F            400          DB      01111111B       ; 8
 0136 67            401          DB      01100111B       ; 9
                    402
 0137               403 HERE1    EQU     $
                    404
                    405 $EJECT
```

```
ISIS-II MCS-48/UPI-41 MACRO ASSEMBLER, V2.0                PAGE    9

  LOC  OBJ          SEQ          SOURCE STATEMENT

                    406
                    407 ;*****************************************************************************
                    408 ;
                    409 ;                       LOOKUP  TABLE  FOR THE PHASE ANGLE
                    410 ;
                    411 ;*****************************************************************************
                    412
  00C9              413           ORG     TAB
                    414
  00C9 00           415           DB      00H
  00CA 06           416           DB      06H
  00CB 0C           417           DB      0CH
  00CC 11           418           DB      11H
  00CD 17           419           DB      17H
  00CE 1D           420           DB      1DH
  00CF 23           421           DB      23H
  00D0 29           422           DB      29H
  00D1 2E           423           DB      2EH
  00D2 34           424           DB      34H
  00D3 3A           425           DB      3AH
  00D4 40           426           DB      40H
  00D5 46           427           DB      46H
  00D6 4B           428           DB      4BH
  00D7 51           429           DB      51H
  00D8 57           430           DB      57H
  00D9 5D           431           DB      5DH
  00DA 63           432           DB      63H
  00DB 68           433           DB      68H
  00DC 6E           434           DB      6EH
  00DD 74           435           DB      74H
  00DE 7A           436           DB      7AH
  00DF 80           437           DB      80H
  00E0 85           438           DB      85H
  00E1 8B           439           DB      8BH
  00E2 91           440           DB      91H
  00E3 97           441           DB      97H
  00E4 97           442           DB      97H
  00E5 97           443           DB      97H
  00E6 97           444           DB      97H
                    445
                    446 $EJECT
```

```
ISIS-II MCS-48/UPI-41 MACRO ASSEMBLER, V2.0                PAGE   10

 LOC  OBJ          SEQ          SOURCE STATEMENT

                    447
 0137               448           ORG     HERE1
                    449
                    450 $       INCLUDE(CBNBCD)
                =   451 ;*****************************************************************
                =   452 ;*
                =   453 ;*                  CONBCD
                =   454 ;*
                =   455 ;*****************************************************************
                =   456 ;*
                =   457 ;*                  THIS UTILITY CONVERTS AN 8 BIT BINARY VALUE TO BCD
                =   458 ;*                  AT ENTRY:
                =   459 ;*                           A = 8 BIT BINARY VALUE
                =   460 ;*
                =   461 ;*                  AT EXIT:
                =   462 ;*                           A = BCD VALUE
                =   463 ;*****************************************************************
                =   464
 0005           =   465 TEMP1   SET     R5
 0006           =   466 TEMP2   SET     R6
                =   467 ;
                =   468 ;! CONVERT TO BCD
                =   469 CBNBCD:
                =   470 ;! BCDACC:=0
 0137 BE00      =   471         MOV     TEMP2,#00
                =   472 ;! COUNT:=8
 0139 BB08      =   473         MOV     COUNT,#08
                =   474 ;! REPEAT
                =   475 NEXTCOR:
                =   476 ;2    BIN:=BIN*2
 013B 97        =   477         CLR     C
 013C F7        =   478         RLC     A
                =   479 ;2    BCD:=BCD*2+CARRY
 013D AD        =   480         MOV     TEMP1,A
 013E FE        =   481 BCDOOC: MOV     A,TEMP2
 013F 7E        =   482         ADDC    A,TEMP2
 0140 57        =   483         DA      A
 0141 AE        =   484         MOV     TEMP2,A
 0142 FD        =   485         MOV     A,TEMP1
                =   486 ;2    IF CARRY FROM BCDACC GOTO ERROR EXIT
 0143 F649      =   487         JC      BCDCOB
                =   488 ;2    COUNT:=COUNT-1
                =   489 ;! UNTIL COUNT=0
 0145 EB3B      =   490         DJNZ    COUNT,BCDCOR
 0147 97        =   491         CLR     C       ; CLEAR CARRY TO INDICATE NORMAL TERMINATION
 0148 FE        =   492         MOV     A,TEMP2 ; PUT BCD VALUE IN ACCUMULATOR FOR RETURN
                =   493 ;! END CONVERT TO BCD
 0149 83        =   494 BCDCOB: RET
                =   495 $EJECT
```

```
ISIS-II MCS-48/UPI-41 MACRO ASSEMBLER, V2.0          PAGE   11

 LOC  OBJ          SEQ          SOURCE STATEMENT

                    496 $        INCLUDE(CONB2)
                  = 497 ;*********************************************************************
                  = 498 ;*
                  = 499 ;*       CONBIN
                  = 500 ;*
                  = 501 ;*=======================================================================
                  = 502 ;*
                  = 503 ;*       THIS UTILITY CONVERTS A 2 DIGIT BCD VALUE TO BINARY
                  = 504 ;*       AT ENTRY:
                  = 505 ;*         A = BCD VALUE
                  = 506 ;*
                  = 507 ;*       AT EXIT:
                  = 508 ;*         A = EIGHT BITS OF THE BINARY RESULT
                  = 509 ;*
                  = 510 ;*********************************************************************
                  = 511 ;
                  = 512 ;
                  = 513 ;
0005              = 514 TEMP1    SET      R5
0006              = 515 TEMP2    SET      R6
0007              = 516 TEMP3    SET      R7
                  = 517 ;
                  = 518 ;1 CONVERT_TO_BINARY
                  = 519 CONBIN:
                  = 520 ;1 COUNT:=DIGITPAIR
014A BB01         = 521          MOV      COUNT,#DIGPR
                  = 522 ;  REPEAT
                  = 523 CONBLP:
                  = 524 ;     BIN:=BIN*10
014C AF           = 525          MOV      TEMP3,A
014D 47           = 526          SWAP     A
014E 530F         = 527          ANL      A,#0FH
0150 345A         = 528          CALL     CONB10
                  = 529 ;     BIN:=BIN+MEM(R0)[7-4]
0152 AE           = 530          MOV      TEMP2,A
0153 FF           = 531          MOV      A,TEMP3
0154 530F         = 532          ANL      A,#0FH
0156 6E           = 533          ADD      A,TEMP2
                  = 534 ;      COUNT:=COUNT-1
                  = 535 ;  UNTIL COUNT=0
0157 EB4C         = 536          DJNZ     COUNT,CONBLP
                  = 537 ;  END CONVERT_TO_BINARY
0159 83           = 538 CONBER: RET
                  = 539 ;
                  = 540 $EJECT
```

```
ISIS-II MCS-48/UPI-41 MACRO ASSEMBLER, V2.0                PAGE   12

  LOC  OBJ          SEQ          SOURCE STATEMENT

                  = 541 ;
                  = 542 ;       UTILITY TO MULTIPLY BIN BY 10
                  = 543 ;       CARRY WILL BE SET IF OVERFLOW OCCURS
                  = 544 ;
  015A AD         = 545 CONB10: MOV     TEMP1,A ; SAVE A
                  = 546
  015B 97         = 547         CLR     C
  015C F7         = 548         RLC     A       ; BIN:=BIN*2
                  = 549
  015D F7         = 550         RLC     A       ; BIN:=BIN*4
                  = 551
  015E 6D         = 552         ADD     A,TEMP1 ; BIN:=BIN*5
                  = 553
  015F F7         = 554         RLC     A       ; BIN:=BIN*10
                  = 555
  0160 83         = 556         RET
                  = 557
                  = 558 ;
                  = 559 $EJECT
```

```
ISIS-II MCS-48/UPI-41 MACRO ASSEMBLER, V2.0                PAGE   13

  LOC  OBJ          SEQ          SOURCE STATEMENT

                    560          END


USER SYMBOLS
ANGLE  0023    BAK1   00AB    BAK2   00B5    BCDCOB 013B    BCDCOD 0149
BCDOC  013E    BCDTR  0021    BINTR  0020    CLRFLG 004C    CNBCD  0137
CNTDWN 00C1    CONB10 015A    CONBER 0159    CONBIN 014A    CONBLP 014C
CONT2  0111    CONT3  0124    CONTON 0058    COUNT  0003    DECREM 0049
DELTA  0026    DIG1   0123    DIG2   011F    DIGPR  0001    DISFLG 0025
DLGT10 0007    FIRE   00C5    HERE1  0137    HOLD   0054    ICNT   0004
INCREM 003B    INIT   0000    INITZ  0098    INWAIT 0063    IPWAIT 0016
LDA    0000    LOOK   004F    NOSWAP 010E    OFFTIM 0003    OTCALC 0038
PASTOR 0022    PHLTSV 0006    PHSANG 0005    PWRON  0002    REFRSH 0100
SEGDAT 0024    SELRB0 00B1    SELRB1 00A7    SETVAL 0004    STA    0001
START  000E    TAB    00C9    TABL   012D    TEMP1  0005    TEMP2  0006
TEMP3  0007    TIME   0002    TRIAC  00BB    VOWAIT 0006    WAITST 000A
XA     0002

ASSEMBLY COMPLETE,  NO ERRORS
```

```
ISIS-II ASSEMBLER SYMBOL CROSS REFERENCE, V2.0                    PAGE  1

ANGLE   210    235#   352    358
BAK1    288#   291
BAK2    300#   303
BCDCOB  475#   490
BCDCOD  487    494#
BCDOC   481#
BCDTR   189    195    201    207    233#
BINTR   158    164    170    176    232#
CLRFLG   97     99    104    106    112#
CNBCD   198    469#
CNTDWN  326#   327
CONB10  528    545#
CONBER  538#
CONBIN  167    519#
CONBLP  523#   536
CONT2   355    360#
CONT3   375    379#
CONTON  131    134#
COUNT   252#   473    490    521    536
DECREM   93    108#
DELTA    73     81    238#
DIG1    372    377#
DIG2    374#
DIGPR   254#   521
DISFLG  237#   268    348    370    382    386
DLGT10   86    103    114    245#
FIRE    326    329#
HERE1   403#   448
HOLD    131#   132
ICNT    253#
INCREM   95#
INIT     17#
INITZ    17    226#
INWAIT  144#   146
IPWAIT   48#    50
LDA       7#    80    138    163    175    194    206    347    351    357    369    381
LOOK     70    122#
NOSWAP  350    357#
OFFTIM   95    101    105    108    110    241#   321
OTCALC   77     85     92#
PASTOR  234#
PHLTSV   78     92    113    244#
PHSANG   68    125    187    243#
PWRON    24#    25
REFRSH   41    137    343#
SEGDAT  139    236#   365
SELRB0  173    204    297#
SELRB1  161    192    285#
SETVAL   65    178    242#
STA      12#    72    157    169    188    200    209    267    364    385
START    30     37#
TAB     122    334#   413
TABL    361    391#
TEMP1   465#   480    485    514#   545    552
TEMP2   466#   471    481    482    484    492    515#   530    533
TEMP3   516#   525    531
TIME     53    123    240#
TRIAC    55    148    316#
VOWAIT   24     27#    28
WAITST   27     30#    31    214
XA      251#

CROSS REFERENCE COMPLETE
```

INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, CA 95051 (408) 987-8080

Printed in USA/TP 44/0480/30K SC HG

16 BIT MICROPROCESSORS

- OVERVIEW
- 8088/8086
- 8087
- IAPX 286

The Software Crisis

The growth of the electronics industry in the past decade has been phenomenal. The 1980's promise to be equally rewarding, as VLSI makes possible new and more complex applications for microelectronics.

The ability to build increasingly complex devices has been the driving force behind the success of the industry. This growth in complexity will continue in the 1980's. Our task will be to exploit this driving force in a way that will help you profit from this increasing complexity.

The challenge of the 1970's was to reduce the cost of the electronic functions needed to store and process data. Our approach was to integrate more and more *microcomputer hardware* on a silicon chip at continually decreasing prices.

The challenge of the 1980's is different. Now we must reduce the cost of electronic solutions; that is, reduce the cost you incur in using our devices to build products. Solving this problem will require a shift from the component integration that was prevalent in the 1970's to concentration on *system level* integration in the 1980's.

The reason for this shift is a direct result of the success of the 1970's. The declining price of LSI devices made it possible to put microelectronics into everything from toys to automobiles to appliances. At the same time, the capability of the microcomputer was also growing. Used first for simple logic replacement, the microcomputer became powerful enough for dedicated computer applications and then moved on to become a building block for user-programmable computer systems.

We can now talk about putting the power of a mainframe CPU on a single chip. This buys you nothing as a customer, however, unless you can use that power. Hardware is computing *potential;* it must be harnessed and driven by software to be useful.

Calculating the requirements for computer programmers for all the new microcomputer products, we come up with a need for about *one million* programmers by 1990! When we look at the fact that U.S. electronic engineering schools produced only about 17,000 graduates in 1979, the challenge looks even greater. It is clear that our success is leading to what we call the "software crisis".

There is an answer. In the 1980's, we will replicate our past success by integrating *standardized software.* That will help you develop and maintain microcomputer applications less expensively.

The present level of integration is characterized by the integrated central processor such as our 8086: a CPU on a chip. We have developed and are introducing additional iAPX 86,88 support processors, such as a math coprocessor, the 8087, and an input/ output processor, the 8089. Either of the devices, in conjunction with an 8086 or 8088 CPU, can drive an extremely powerful math or I/O oriented system.

In the future, we will extend the process by integrating operating system and software functions into the processing hardware. Engineering and programming costs will be reduced substantially, and you will have an upward compatible interface for future products. Our goal in the 1980's is to deliver the optimum level of system integration for the complexity level of each of your applications. What this will give you is a *minimum total cost* solution for your application.

This point is best illustrated in terms of dollars. In the early 1970's, producing a simple system using components cost a few hundred thousand dollars. Now, the typical cost of system implementation nears a million dollars. The complex system of the 80's will cost perhaps five million dollars to implement, assuming the programmers can be found to do the job! By providing the optimum level of system integration, as described, your costs are reduced dramatically (see Figure 1).

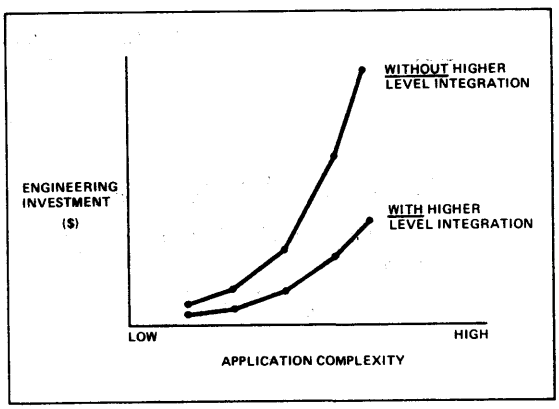


Figure 1. Typical Application Total Engineering Investment

This approach was the rationale for the design of the evolving Microsystem 80 series. You will see that much of our investment in the future of the iAPX 86 and iAPX 88 product lines is aimed at integrating software into the hardware to offer you the most cost effective solution for your application needs in the 1980's.

The iAPX 86,88 provides the foundation for meeting our objectives for the 1980's: to offer you the world's highest performance VLSI systems, and to provide you the total solution at the lowest possible cost.

Evolving Application and Software Complexity

Today's complex microsystem applications range from dedicated 8-bit control processors to multi-tasking 8-bit, 16-bit, and 32-bit systems. Intel plans to provide a product line that will address the full range of these microsystems.

As the complexity of your product line increases — for example, from a small single user data processor to a large multi-user data processor, or from an intelligent terminal to a shared

logic word processor — the level of software complexity increases *more* than proportionally. Application software is rapidly headed in the direction of the full blown "mainframe-style" multiprocessing environment. This application trend is the basis for the "software crisis".

The iAPX 86 and iAPX 88 address this growing software complexity head on. They provide an architecture *designed* for high level languages. High level languages such as PASCAL and FORTRAN are much more cost effective than assembly language for large software applications.

High level languages offer a major productivity improvement in application development. They use simple, algorithmic statements, independent of processor architecture to define the application. This reduces errors and produces programs that are far easier to understand, debug, and maintain than if done in assembly language. HLL programming increases programmer productivity, benefiting you with lower costs and earlier completion. You realize returns on your investment quicker and reduce the total investment necessary.

Table 1 shows a case study comparing costs of assembly language coding and HLL programming. In this study we have used the generally accepted program development rate of about 10 lines of debugged code per day — *regardless of programming language.*

This example shows that pure assembly language coding is no longer a cost effective alternative in large applications. The price of a line of code, whether assembly or HLL, is estimated to be $50-60. HLL programming can produce over four times the output of assembly coding per dollar. The cost implications of the wrong choice in a large, complex application can be enormous.

Table 1. HLL vs. Assembly Language Cost Analysis

1975

1000 Lines of Assembly Code ($20/line)

At 10 Lines Debugged Code per Man Day, and $40, 000 per Man Year = ½ Man Year

Or **≃ $20, 000**

1 Person, 6 Months

1980

13, 000 Lines of Assembly Code ($50.00/line) or 3, 000 Lines of HLL Code ($50.00/line)

At 10 Lines Debugged Code per Day and $125, 000 per Man Year =

Assembly Language	High Level Language
6.5 Man Years	1.5 Man Years
≃ $800, 000	**≃ $180, 000**
3 People, 2+ Years	2 People, 9 Months

Microsystem 80
Introduction

1

iAPX 86 AND iAPX 88 ARCHITECTURE — THE FOUNDATION FOR THE FUTURE

Overview

iAPX 86,88 is an evolving family of microprocessors and peripherals. The family partitions processing functions among general data processors (8086 and 8088), specialized coprocessors like the 8087 numeric data processor, and I/O channel processors (the 8089).

Four key architectural concepts shaped the data processor designs. All four reflect the family's role as vehicles for modular, high level language programming (in addition to assembly language programming). The four architectural concepts are memory segmentation, the operand addressing structure, the operation register set, and the instruction encoding scheme. They are distinct departures from the minicomputer architectural styles of the 1960's and 1970's.

These earlier architectures (minicomputers) were designed for assembly language programming which emphasizes register based data and linear programs. Over the last decade, large software development projects shifted their programming to high level languages which employ modular programming and memory based data. The iAPX 86,88 memory segmentation scheme is intended for modular programs. It supports the static and dynamic memory requirements of program modules, as well as their communication needs. The iAPX 86,88 registers are designed for fast high level language execution. The scheme employs specialized registers and implicit register usage. You will derive significant performance and memory utilization improvements directly from these architectural features.

The four concepts are discussed in the following sections. They are:

- Memory segmentation for modular programming, evolution to memory management and protection
- Addressing structure for high level programming languages
- Operation register set for computation
- Instruction set encoding for memory efficiency and execution speed

Memory Segmentation for Modular Programming

Large programs (10-100K bytes) are not generally written in assembly language. They are developed in individually compiled modules in high level languages. Modular program development techniques, program libraries, compatible linking, and project management tools are often requirements in such an environment. A complex application program might be composed of multiple processes, with each process constructed from multiple modules. Processes send messages to each other for communication, while modules generally share common data when needed. Ideally, these inter-module communication paths are well structured and disciplined.

The iAPX 86,88 segmentation scheme is optimized for the reference patterns of computer programs. Four segment registers are provided in a segment register file. Memory references are relative to automatically selected code segment (CS) and data segment (DS) registers. The module shares a stack segment (SS) with all other modules of the process (task). The module may share a global data segment with other modules in the process; for example, to send and receive messages between modules. This segment is accessed explicitly with the extra segment (ES) register.

This scheme is highly efficient because constant program references to code and data, as well as the stack, have *automatic* segment selection. This results in minimized instruction length. Only 16 bits are required to address anywhere in the full megabyte address range. Only infrequent inter-module communications require the extra prefix bits to explicitly override the automatic segment selection.

There are two other significant advantages to the segment register concept. First, it separates segment base addresses from offset addresses which are relative to the segment base. Only offset addresses are used within object modules. This supports position-independent, dynamically relocatable modules. You merely have to alter the CS and DS register contents to move a module, rather than relinking the whole task and reloading. This structure employs short addresses (16 rather than 20-bit) for efficient use of memory.

The second advantage of iAPX 86,88 segmentation is that it can be extended to include memory management and multi-level protection. The contents and width of segmentation registers are independent of the rest of the instruction set. The architecture can be made to address additional memory and provide access rights and limit checking. Using the mainframe concept of memory based segment tables, this structure can also support virtual memory. Further, since only four registers are active in the file at a time, these features can be accomplished on the CPU chip itself, avoiding the access delays of off-chip memory management.

In summary, memory segmentation has several ultimate benefits for the end user. It provides for simplified hardware and faster, modular software development, more easily maintainable code, and provides an orderly way for the architecture to grow.

Addressing Structure for High Level Programming Languages

The iAPX 86,88 architecture employs an operand addressing scheme complementing the memory segmentation scheme. There are four components in an address. They are the segment, base, index, and displacement. The segment component was just described. A base register is dedicated to both the data and stack segments. These base registers may

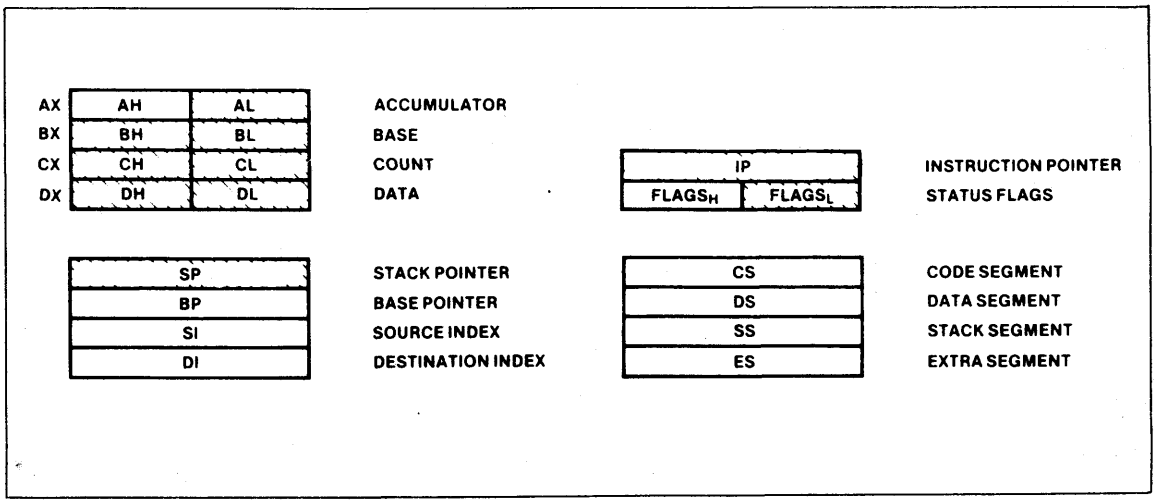


Figure 2. iAPX 86/10, 88/10 Register Model

also be used when accessing the extra (global) segment. They are used for holding the base address of a data structure.

Two index registers are provided for use with the base registers to dynamically select any element from a based data structure. Eight or sixteen-bit fixed displacements may be added to any of these address forms. The complete register file is shown in Figure 2 and the addressing structure is shown in Figure 3.

Referring to Figure 3, an iAPX 86,88 operand address contains up to four components: a segment (S), a base (B), an index (I), and a displacement (d). The segment component is automatically selected for the code, data, and stack segments. An explicit segment selection is required for data references in the extra segment. Any combination of the remaining three address components is permitted in virtually all memory reference instructions, with at least one always being present.

Block and string data are extensions to this scheme. They use different assumptions for source and destination segments, but the segments are still implicitly accessed. Immediate operands are also supported.

The iAPX 86,88 is a two operand machine (source and destination). It supports source/destination operand combinations of register/memory, memory/register, memory/memory (string operations only), immediate/register, and immediate/memory. The various address combinations of S, B, I, and d correspond to common data structures used in high level language programming. Such data structures can therefore be implemented easily in assembly language as well.

Figure 3 shows the correspondence between the most common iAPX 86,88 address modes and various data types in high level programming languages. The S component is

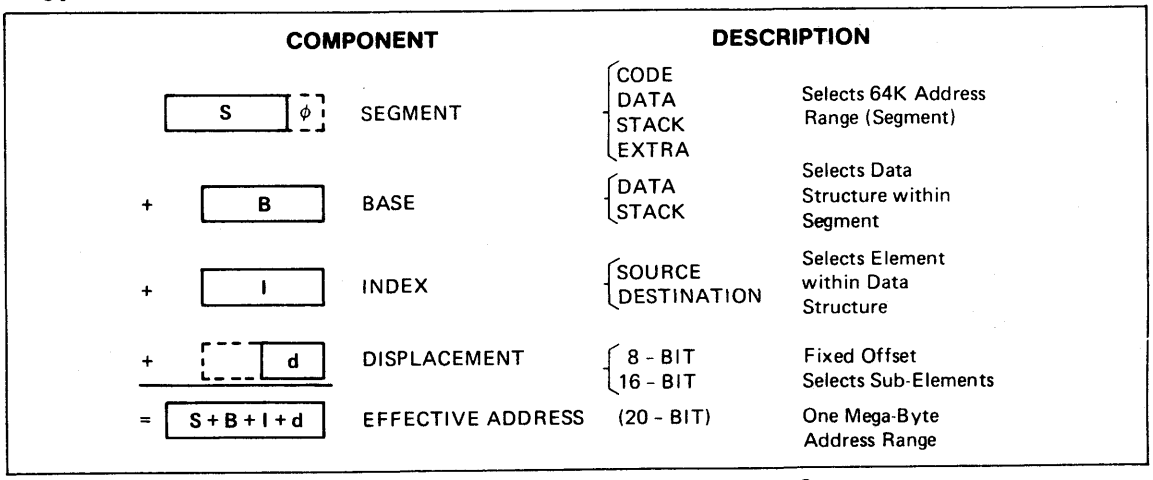


Figure 3. iAPX 86,88 Four Component Addressing Structure

implicit; the stack base (BP) assumes the stack segment; no B component, or use of the data base (BX), assumes the data segment. The less commonly used address modes are not shown.

The stack base (BP) is a concept borrowed from the family of P-machines "developed" as ideal PASCAL vehicles. P-machines term this register the "mark pointer". It always points to the base of the current local data area in the stack segment. This permits efficient local addressing in block-structured languages such as PASCAL and PL/M. In these languages, procedures are invoked by pushing their parameters on the stack, calling the procedure, and then allocating their local data area on the stack. The iAPX 86,88 return instruction then removes the parameters from the stack, as is done in the P-machines.

Operation Register Set for Computation

The Intel iAPX 86,88 line is truly a complete family of microprocessors. The iAPX 86/10 and iAPX 88/10 are the general data processor members of the family, while the 8089 is the I/O processor family member. In addition, the CPU itself has an interface for attaching coprocessors. Coprocessors provide specialized operation set extensions that benefit the application by performing special purpose logic to increase performance.

The iAPX 86/20 Numeric Data Processor is an example of this concept. Using an 8086 with an 8087 coprocessor (CPU extension) it provides a *one hundred-fold performance boost* over the iAPX 86/10 for a wide range of numeric operations. The full computational capability of the iAPX 86,88 family can therefore span a much broader range than is possible with a single microprocessor. This technique has been used successfully in the mainframe and minicomputer industries to provide instruction set options for scientific, commercial, text processing, or other special purpose applications.

An 8087 extends the iAPX 86 or iAPX 88 architecture to include additional data types, registers, and instructions. The 8086 or 8088, with an 8087 coprocessor, operates on 16, 32, and 64-bit integers, 32, 64 and 80-bit floating point numbers, and up to 18 digit packed BCD numbers. Data conversions and calculations are performed in the 8087 and are transparent to the programmer.

The iAPX 86/10 and iAPX 88/10 CPUs alone can perform arithmetic operations on signed and unsigned 8 and 16-bit binary integers as well as packed and unpacked decimal integers. The full complement of logical operations are provided as well. Interesting new features are the string operations. Six primitive string instructions (move, skip, search, compare, set, and translate) are standard. When combined with special control operators, complex string manipulations are possible with two or three *instructions*.

Instruction Set Encoding for Memory Efficiency and Execution Speed

The iAPX 86 uses a byte oriented instruction stream while operating with a 16-bit data bus. To accomplish this, the processor is subdivided into two independent parallel processors called the bus interface unit (BIU) and the execution unit (EU). The iAPX 88 employs an identical execution unit and is 100% code compatible with iAPX 86, yet it interfaces to an 8-bit wide data bus BIU. The bus interface unit is an independent processor that prefetches instructions. Instruction fetch time is therefore mostly overlapped with other iAPX 86,88 processor activity. The bus interface unit permits either instructions or data to be placed in memory without regard to word boundaries. (An array of five byte records in PASCAL can be referenced without requiring an additional byte of padding to word align the records.) Processor subdivision into the BIU and EU has the additional benefit of minimizing the effect of wait states and bus hold time on CPU efficiency.

Instruction set encoding is substantially improved when instructions are composed in byte multiples instead of words. Instructions in the iAPX 86,88 vary from one to six bytes in length (not counting optional prefix bytes). The average instruction is three bytes long. In a word aligned machine the same information would occupy four bytes. This and the features described above give the iAPX 86,88 roughly a 30% program space savings over other architectures.

PROCESSOR PARTITIONING

Beyond efficient support for high level languages, the iAPX 86 and iAPX 88 establish the foundation for the family to build on in the 1980's. The family uses increasing levels of integration to significantly reduce software, hardware, and development investment.

The iAPX 86/10 and iAPX 88/10 general purpose processors employ external module integration. Specialized system functions are distributed among optimized components and removed from the host processor. The CPU is freed to become the system manager and resource allocator rather than doing "all things for all programs". The family also includes the 8087 Numeric Data Processor and the 8089 I/O Channel Processor.

These processors are optimized to address the three main functions in a computer environment: data processing and control, arithmetic computation, and input/output. The 8087 and 8089 are described below.

The 8087 Numeric Processor Extension (NPX) adds over 50 numeric opcodes and eight 80-bit registers to the host processor to provide more extensive data and numeric processing capability. It performs floating point and trans-

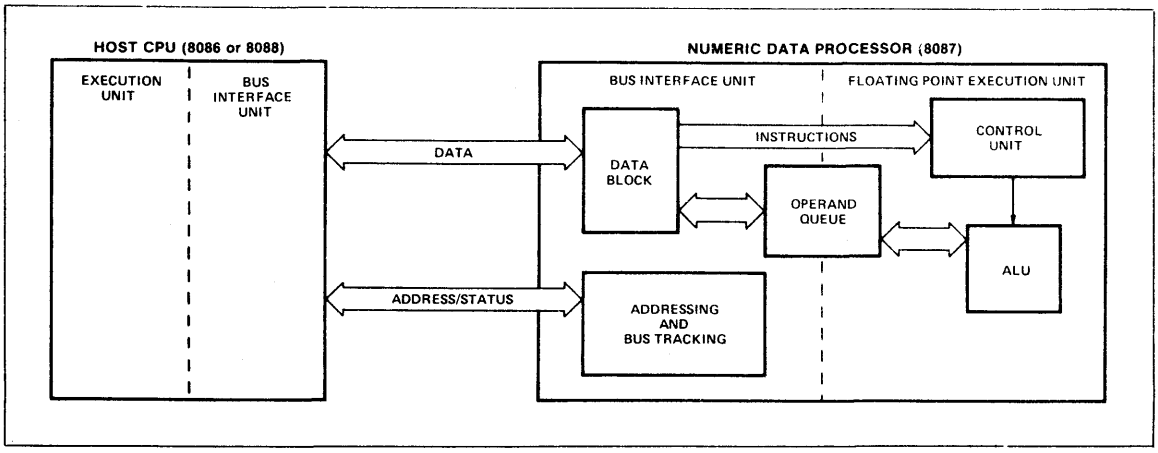


Figure 4. Numeric Data Processor Block Diagram

cendental (trigonometric) functions, processes decimal operands up to 18 digits without roundoff, and performs exact arithmetic on integers up to 64 bits long. Another feature of the NDP, with important benefits to you, is that it is compatible with the proposed IEEE floating point standards. It can be used in applications requiring high speed computation such as numerical analysis, accounting and financial applications, the sciences, and engineering. Throughput increases in such applications up to *100 times* current speeds are typical (See Figure 4.)

The 8089 Input/Output Processor (IOP) is an independent microprocessor that optimizes input/output operations. The objective of the IOP is to remove all I/O details from application software. It responds to CPU direction but executes its own instruction stream in parallel with other processors. I/O transfers of either 8 or 16-bit data can be done at rates up to 1.25 megabytes per second. The IOP therefore combines the attributes of both a CPU and a DMA controller to provide a powerful I/O subsystem. An important feature of the IOP is that it can be physically isolated from the application CPU. The advantage to you is that I/O subsystem changes or upgrades can be made without any impact to application software. (See Figure 5.)

Summarizing, there are several advantages to external module integration:

- System tasks may be allocated to special purpose processors designed for optimal task handling
- Simultaneous operation (parallel processing) provides highest system performance
- Isolated system functions minimize the effect of modifications, local failures, or errors on the rest of the system

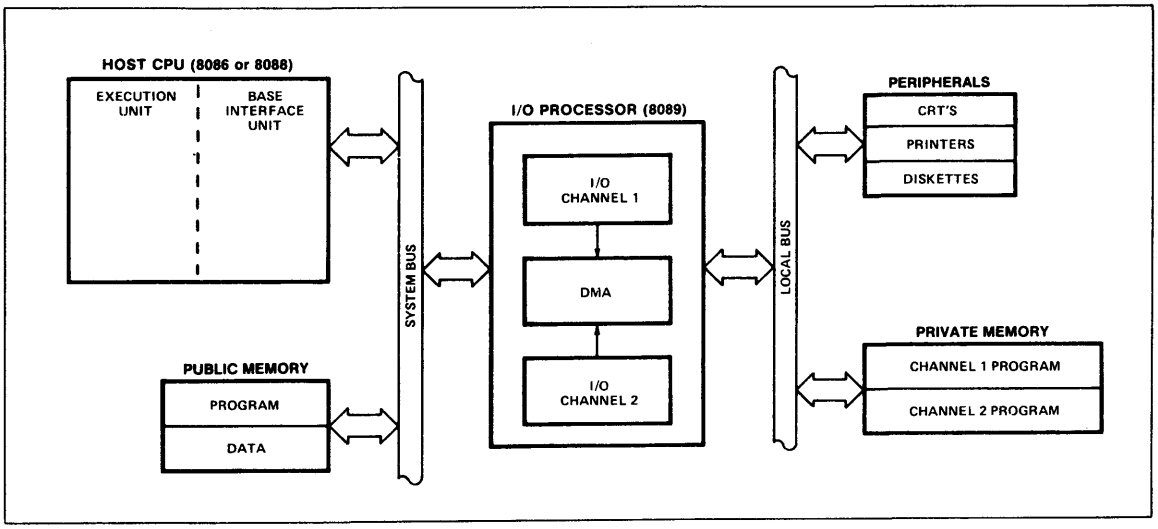


Figure 5. I/O Processor Block Diagram

- The iAPX 86,88 family of processors allows division of the application into small, manageable tasks for parallel development, while providing built-in hardware facilities for coordinating processor interaction. With the iAPX 86,88 approach you can implement high performance systems far more quickly and easily than would otherwise be possible.

DEVELOPMENT TOOLS

Development Systems

Development systems are a unique combination of hardware and software tools which increase your product development productivity. With Intel development products, you will shorten the development cycle and reduce your time to market.

Development systems from Intel provide an upgradable spectrum of tools ranging from stand alone development systems to future networks of specialized work stations. Intel eliminates your risk of development system obsolescence by guaranteeing product upgradability and compatibility. This guarantee protects your capital investment.

For small to medium size projects, the Intellec™ development system is available in many configurations at low cost. For small projects, these systems have nominal program memory with floppy disks as peripheral storage devices. Minimum configurations may be upgraded to provide increased performance, increased memory, and increased mass storage via hard disk. These more powerful configurations support medium sized projects.

The Intellec Series II/85 is a good example of such a system. It is a complete microcomputer development system integrated into one compact package. The Model 225 includes a CPU with 64K bytes of RAM, 4K bytes of ROM, a 2000 character CRT, detachable full ASCII keyboard, and a 250K byte floppy disk drive. The powerful ISIS-II Disk Operating System software allows you to efficiently develop and debug iAPX 86,88 programs. Optional storage peripherals provide over 2 million and 7.3 million bytes of storage on floppy and hard disk, respectively.

Distributed development configurations address the range of medium to large sized projects. These configurations connect multiple standalone development systems to more powerful support resources such as mainframes and their peripherals.

In addition to the Intellec® development system, Intel offers several products to help you debug and test your hardware and software. In-Circuit-Emulators, such as ICE-86™ and ICE-88™, are available to emulate your product environment. They increase development productivity substantially. Another software tool, RBF-89, helps you debug 8089 software under ICE control. With these tools, software development time can be reduced dramatically — lowering your total investment.

High Level Languages

Programming languages are the key to developing an application. Intel programming languages serve three purposes in your design. First, they are your primary design tool. Intel's breadth of languages and extended features give you the maximum ability to properly design and plan your program. Second, Intel languages are a communication vehicle between programmers during implementation and later during modification. Standard high level languages allow programmers to better communicate what the programs do. Third, Intel languages are designed in conjunction with Intel microsystems to provide the greatest code efficiency and execution speed. Intel languages speed implementation of your design and reduce maintenance costs.

MDS-311 is a set of software development tools for iAPX 86 and iAPX 88 applications. It is a complete set of software products that run on the Intellec Model-800 and Series-II development systems. The software tools provided include PL/M-86, high level programming language, and the ASM-86 assembler. Two utilities, LINK86 and LOC86, are supplied to link separately compiled or assembled program modules into executable tasks. The Library Manager, LIB86, lets you maintain a library of iAPX 86 or iAPX 88 object modules. These modules can then be linked in with new programs without being recompiled. This simplifies and speeds your development. Common code (e.g. a subroutine) only has to be developed and compiled once. Intel code converters, such as CONV86, are very useful tools for migrating 8080 or 8085, Z80, and 6809 assembly language programs to the iAPX 86 or iAPX 88. They convert assembly source code to ASM86 source code. This will help you make a rapid transition and cut redevelopment costs substantially.

Intel will provide a variety of languages for both systems and applications to facilitate development of your product. You can choose the language (or languages) which best suits your product needs and the expertise of your staff. ASM86, the assembly language, and PL/M-86, the systems oriented high level language, are both currently available. PASCAL, FORTRAN, and BASIC will be offered in the near future, and COBOL is planned after that.

Intel's languages also run on your final product. Your product's function is significantly increased when packaged with language translators. They allow your customers to tailor your products for their environment. Intel's languages will save implementation time and free resources to work on the value-added portion of your product.

SINGLE BOARD COMPUTERS ACCELERATE YOUR MICROSYSTEM SUCCESS

In addition to the increased integration of functions in VLSI components, there is a strong trend today to implement microsystem applications with single board compu-

ters. This allows the design engineer to:

- Easily configure reliable and cost-effective systems using iSBC and iSBX standard products.
- Overcome the shortage of qualified engineers and technicians.
- Get the end product to market quickly.
- Focus on the application.
- Offset the increasing cost of capital.

In addition, using iSBC single board computers and iSBX expansion products in your design reduces the number of risks that you must face in all phases of the product life cycle. The four major risk areas that Intel iSBC and iSBX products will help you overcome are as follows:

1. Limited Resources

Using a fully tested board computer, which incorporates the key elements of processor, memory and I/O, helps overcome today's critical shortage of engineers, programmers and technicans. Implementing iSBC boards and iSBX MULTIMODULES in your design reduces increasing capital costs in production, QC, and test. It is estimated that using iSBC boards can save up to $200,000 per board design.

2. Time to Market Dictates Success or Failure

With inflation running at its current rate, the amount of time it takes to get a product from an idea to the market becomes critical. A delay of a few months can collapse your return on investment.

Experience shows that the first company that gets its product to the marketplace usually dominates that market. You can get your product to the market months earlier using standard off-the-shelf iSBC, iSBX and Real-Time Executive (iRMX) Software modules. Intel's large board manufacturing and distribution capability enables you to respond to your market demand rapidly and in a cost-effective manner.

3. Solution Completeness and Project Credibility

Microprocessor based solutions for today's problems are commonplace and are expected to succeed. A broad spectrum of compatible system components in the iSBC, iSBX, and iRMX product line increase the probability of being right the first time. General purpose iSBC board solutions are easy to customize through the use of iSBX modules from Intel, or your own design.

4. Coping with the Technology/Complexity Avalanche

iSBC and iSBX products incorporate the latest in VLSI shortly after their initial introduction. With increasing system complexity Intel's design process and testing reduces the risk of "gremlin" bugs which multiply with complexity and evade diagnosis. Standards used throughout the product family such as the de facto industry standard MULTIBUS, EIA, IEEE etc. provide a smooth transition for your product to new and changing processor, memory and I/O technologies.

Intel's single board computer product family is continuing to reduce your risk and protect your investment in the future by expanding iSBC and iSBX products in three dimensions: processors, memory, and I/O.

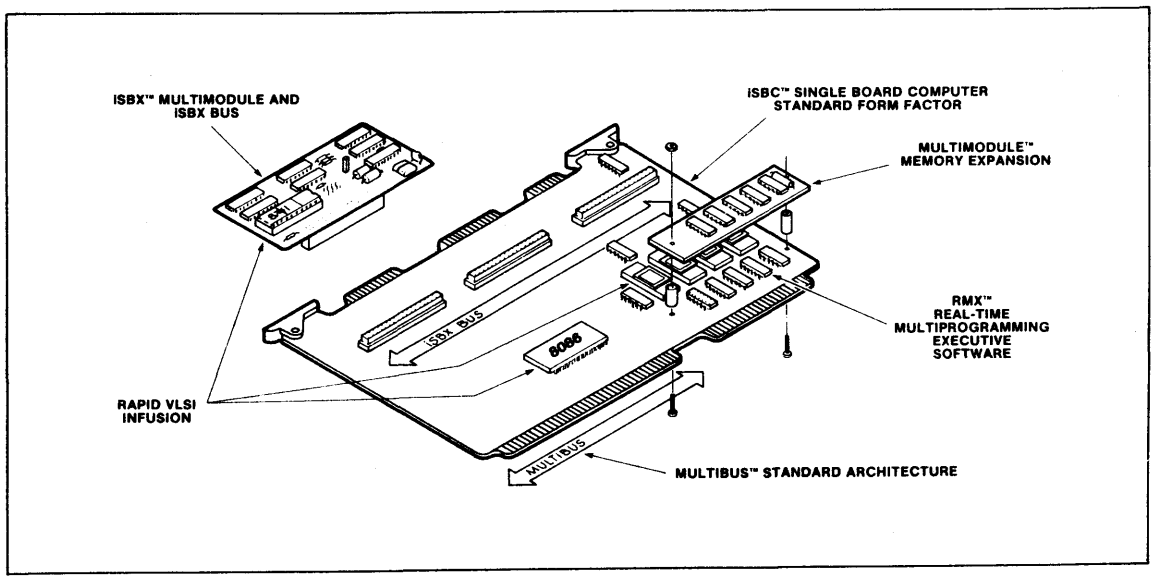


Figure 6. Single Board Computer (iSBC 86/12™)

SUMMARY

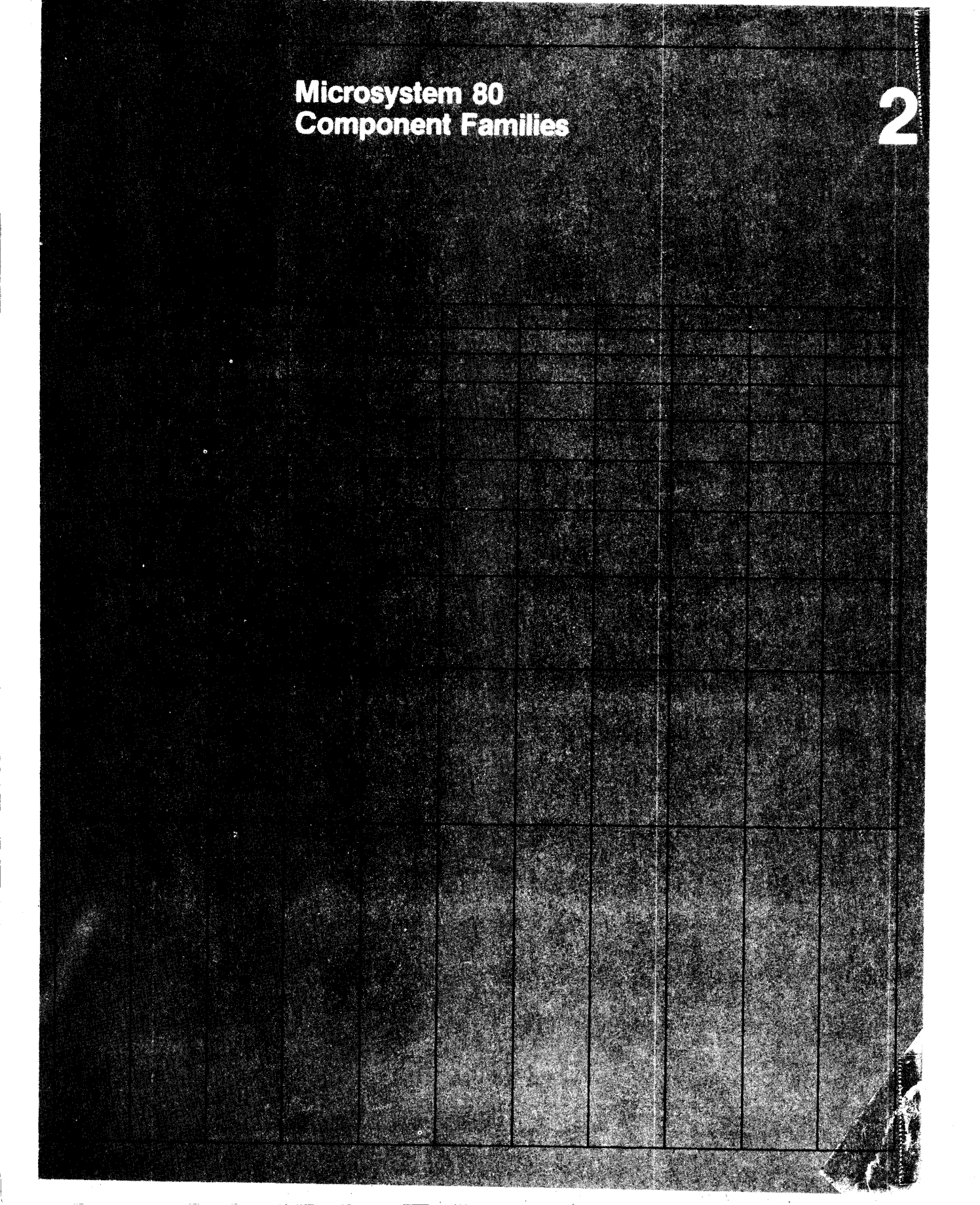
Intel's iAPX 86,88 multiple processor family is designed for modular programming in high level as well as assembly languages.

- Its memory segmentation scheme is optimized for the reference needs of computer programs, and is separate from the operand addressing structure.
- The structure for addressing operands within segments directly supports the various data types found in high level programming languages.
- The family provides an operation register set to support general computation requirements. It also provides for optimized operation register sets to do specialized data processing functions with its inherent multi- and coprocessor support.
- The family uses optimized instruction encoding for high performance and memory efficiency
- The family is well supported with development tools and single board computer products.

This architecture provides the foundation for solving the application needs in the 1980's. It makes a noted departure from architectures of the 1960's and 1970's — based on Intel's intent to minimize software and hardware product costs for you, the end user.

Microsystem 80 Component Families

2



iAPX 86, 88

- **Complete high performance microprocessor systems**
- **Multiple independent processing modules**
- **Flexible, intelligent I/O channel functions**
- **Multiple asynchronous bus structures supported**
- **MULTIBUS™ system compatible interface**
- **24 Operand addressing modes, 1 megabyte addressability**
- **Implements the proposed IEEE floating point standard**
- **14 general data processing registers plus an 8x80 bit numeric register stack**
- **Bit, byte, word, real, decimal and block operations**

The components in the iAPX 86, 88 product lines have been designed to operate together in diverse combinations within the systematic framework of the overall family architecture, as shown in Figure 1. In this way a single family of components can be used to solve a wide array of microcomputing problems. A component mix can be tailored to fit the performance needs of an application precisely, without having to pay for unneeded capabilities that may be bundled into more monolithic, CPU-centered architectures. Using the same family of components across multiple systems limits the learning curve problem and builds on past experience. Finally, the modular structure of the family architecture provides an orderly way for systems to grow and change.

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

1. System functions are distributed among specialized components.

2. Multiprocessing capabilities are inherent in the hardware.

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

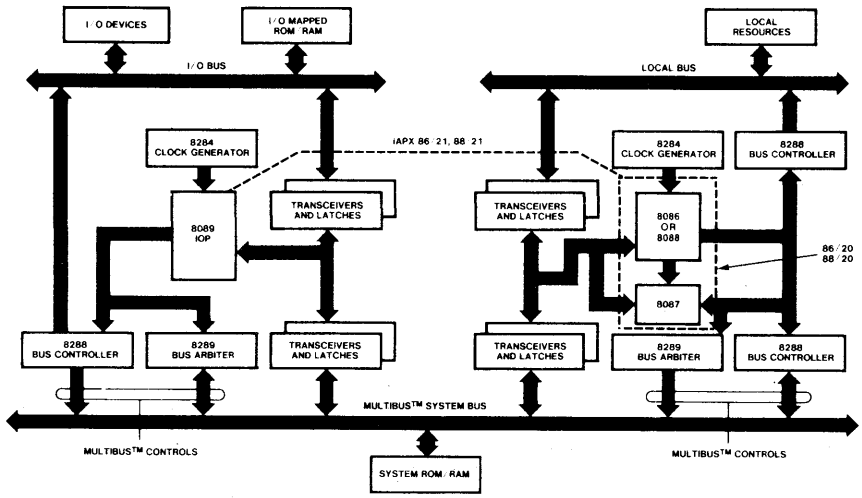


Figure 1. iAPX 86, 88 Multiprocessing System

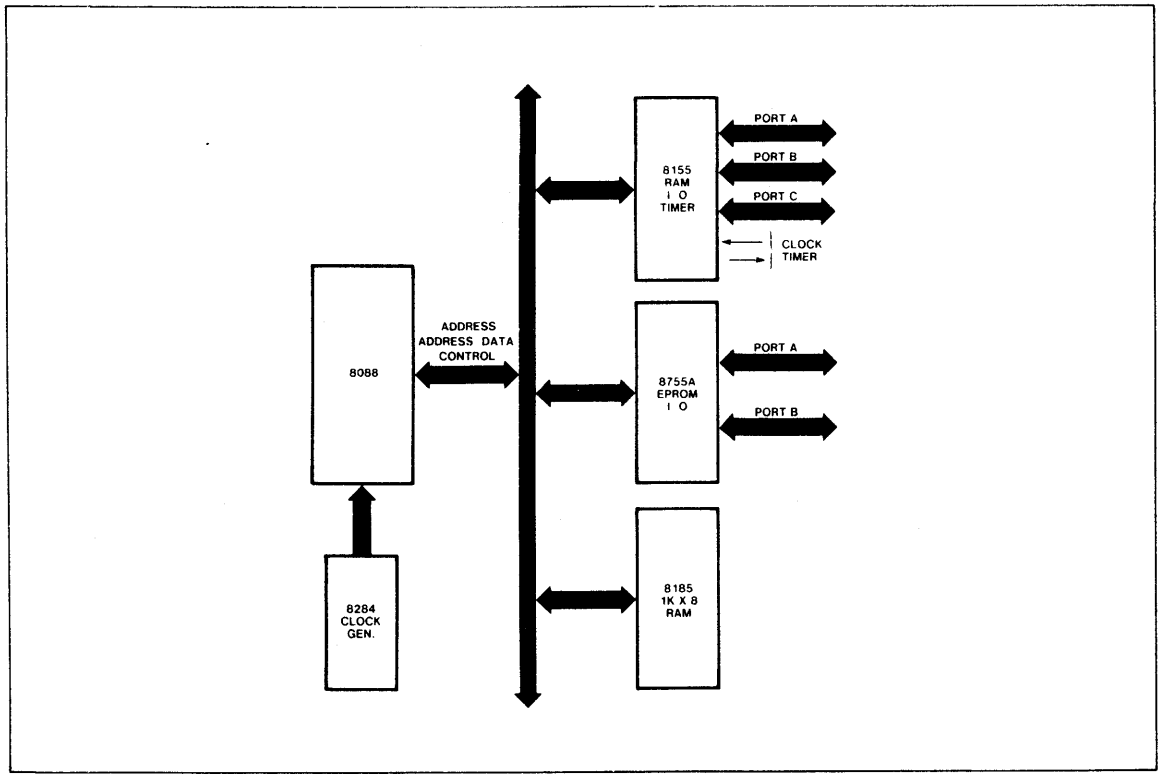


Figure 2. iAPX 88 Allows For A Highly Integrated, High Performance System Using 8085 Family of Components

MICROPROCESSORS

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

—5 MHz (200 ns cycle time); and 8 MHz (86/10) are available.

—System can be constructed for both 8 & 16 bit data paths

—Processors operate on 8-, 16-, 32-, 64-bit, character and string data types; internal data paths are at least 16-bits wide.

—Up to 1 megabyte of memory can be addressed, along with a separate 64K byte I/O.

—The address/data and status interfaces of the processors are compatible (the address and data buses are time multiplexed at the processor, allowing each to be compatible with a common set of bipolar bus support components.

MULTIPROCESSING

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

—System tasks may be allocated to special-purpose processors whose designs are optimized to perform specific (or classes of) tasks simply and efficiently;

—Very high levels of performance can be attained when processors can execute simultaneously (parallel/distributed processing);

—Reliability can be improved by isolating system functions so that a failure or error in one part of the system has a limited effect on the rest of the system;

—The natural partitioning of the system promotes parallel development of subsystems, breaks the application into smaller, more manageable tasks, and helps isolate the effects of system modifications.

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

The architecture supports two types of processors: independent processors and coprocessors. An independent

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

The iAPX 86, 88 product line architecture also supports a second type of processor, called a coprocessor. Coprocessor "hooks" have been designed into the iAPX 86/10 and iAPX 88/10 to allow this type of processor to be accommodated in the future. A coprocessor differs from an independent processor in that it obtains its instructions from the independent processor or host. The coprocessor monitors instructions fetched by the host and recognizes certain of these as its own and executes them. A coprocessor, in effect, extends the instruction set (and architecture) of its host processor.

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

For mutual exclusion, each processor has a LOCK (bus lock) signal which a program may activate to prevent other processors from obtaining a shared system bus. The IOP may lock the bus during a DMA transfer to ensure both that the transfer completes in the shortest possible time and that another processor does not access the target of the transfer (e.g., a buffer) while it is being updated. Each of the processors has an instruction that examines and updates a memory byte with the bus locked. This instruction can be used to implement a semaphore mechanism for controlling the access of multiple processors to shared resources. (A semaphore is a variable that indicates whether a resource, such as a buffer or a pointer, is "available" or "in use".

BUS ORGANIZATION

Figure 3 summarizes the iAPX 86, 88 bus structure. There are two different types of buses: system and local. Both buses may be shared by multiple processors, i.e., both are multimaster buses. Microprocessors are always connected to a local bus, and memory and I/O components usually reside on a system bus. The iAPX 86, 88 bus interface components link a local bus to a system bus.

Local Bus

The local bus is optimized for use by the iAPX 86, 88 microprocessors. Since standard memory and I/O components are not attached to the local bus, information can be multiplexed and encoded to make very efficient use of processor pins (certain MCS-85 peripheral components can be directly connected to the local bus). This allows several pins to be dedicated to coordinating the activity of multiple processors sharing the local bus. Multiple processors connected to the same local bus are said to be local to each other; processors on different local buses are said to be remote to each other, or configured remotely. Both independent processors and coprocessors may share a local bus; on-chip arbitration logic determines which processor drives the bus. Because the processors on the local bus share the same bus interface components, the local configuration of multiple processors provides a compact and inexpensive multiprocessing system.

System Bus

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

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

Processing Modules and Bus Topology

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

to all processing modules on the public bus. 8289 bus arbiters in each processing module control the access of the modules to the public bus and hence to the public memory and I/O.

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

iAPX 86/10, 88/10

- Direct memory addressing capability to 1 MByte
- 5 or 8 MHz clock rate
- 24 Operand addressing modes
- Bit, byte, word, and block operations
- Memory symmetric, segmented architecture for direct high level language support
- MULTIBUS™ system compatible interface
- Multiprocessor, coprocessor and I/O processor extensions supported

The iAPX 86/10, 88/10 are microprocessors whose resources are tailored for efficiency in high level language environments. Most high-level languages store variables in memory; the 86/10, 88/10 symmetrical instruction set

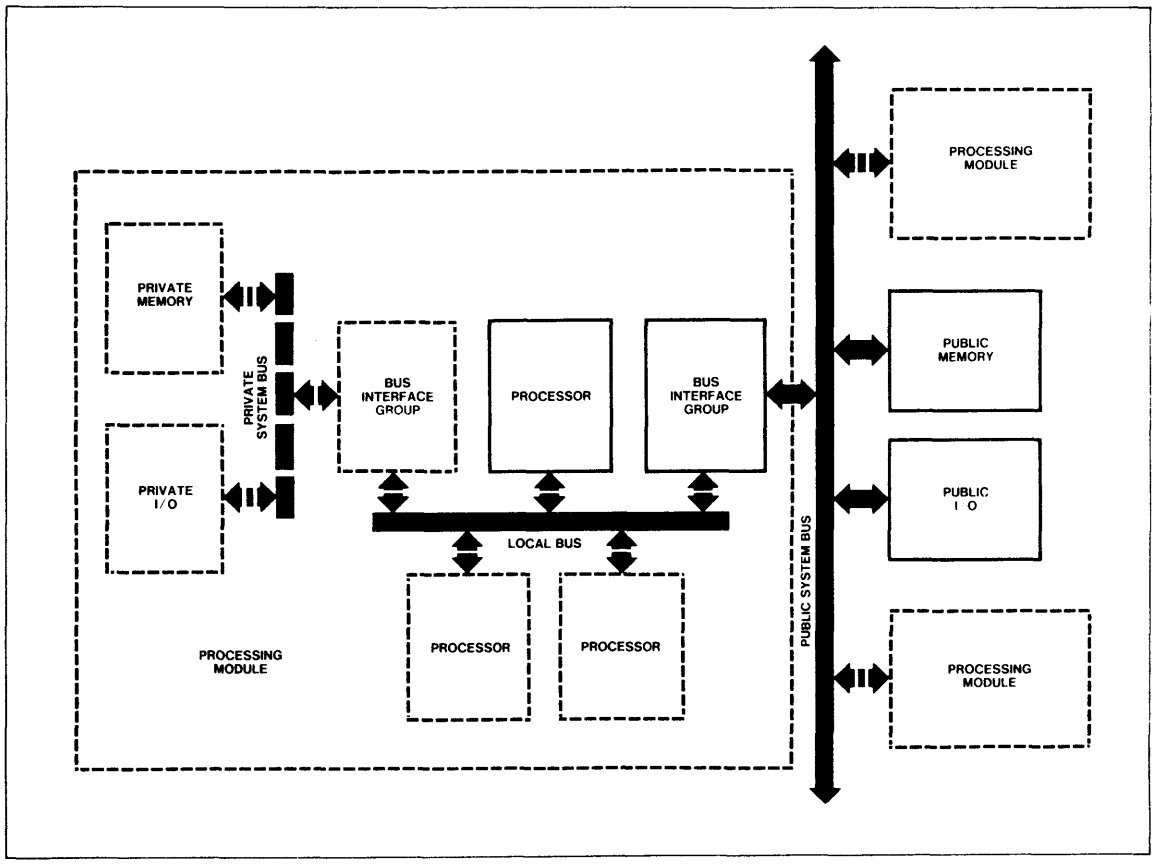


Figure 3. Generalized iAPX 86, 88 Bus Structure

supports direct operation on memory operands, including operands on the stack. The hardware addressing modes provide efficient, straightforward implementations of based variables, arrays, arrays of structures and other high-level language data constructs. A powerful set of memory-to-memory string operation is available for efficient character data manipulation. Of course, routines with critical performance requirements that cannot be met with high level languages may be written in ASM-86 (the 86/10, 88/10 assembly language) and linked with code.

The 86/10 and 88/10 support for high-level languages does not come at the expense of efficient coding. The 86/10, 88/10 instruction set provides byte granularity for all instructions and many optimized versions of more general instructions that are selected automatically by Intel's compiler's and assembler to produce the most compact code possible.

The base architecture of the CPU's themselves allows for extensibility. The CPU's small context makes it especially attractive for larger multiuser, multitasking systems. The 86/10, 88/10 segmented address space and explicit segment control makes it possible to extend the architecture easily to the memory protected environment required in these multiuser systems as well. Multilevel memory protection and a full virtual memory structure can be achieved by interpreting the explicit segment pointers of the 86/10, 88/10 implementation as logical sectors within a virtual address space described by memory resident tables that contain base, limit, and access rights information about each virtual segment. In this way application code written for the 86/10, 88/10 can be transferred directly to the protected environment.

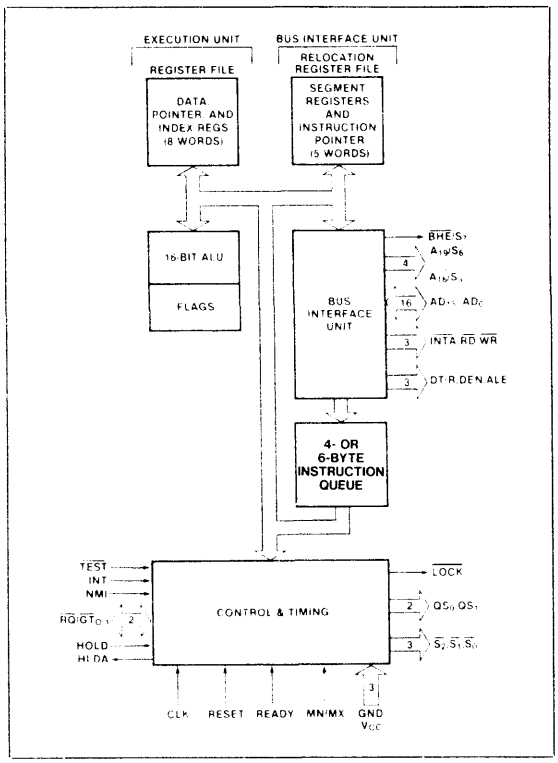


Figure 4. iAPX 86/10, 88/10 CPU Functional Block Diagram

The large application domain of the 86/10 and 88/10 is made possible primarily by the processors' dual operating modes (minimum and maximum mode) and built-in multiprocessing features. Several of the 40 CPU pins have dual functions that are selected by a strapping pin. Configured in minimum mode, these pins transfer control signals directly to memory and input/output devices. In maximum mode these same pins take on different functions that are helpful in medium to large systems, especially systems with multiple processors. The control functions assigned to these pins in minimum mode are assumed by a support chip, the 8288 bus controller.

The high performance of the 86/10 and 88/10 is realized by combining a 16-bit internal data path with a pipelined architecture that allows instructions to be prefetched during spare bus cycles. This makes maximum use of memory bandwidth at the same time actual memory speed (and cost) requirements are relaxed. Also contributing to performance is a compact instruction format that enables more instructions to be fetched in a given amount of time.

iAPX 86/20, 88/20 Numeric Data Processors

- Standard 86/10, 88/10 instruction set plus over 50 new numeric instructions
- Implements the proposed IEEE floating point standard
- High performance, up to 60,000 floating point instructions per second
- Automatic mixed mode arithmetic
- 14 general data processing registers plus an 8X80 bit numeric register stack
- 24 addressing modes available, one megabyte addressability
- Bit, byte, word, real, decimal and block operations
- MULTIBUS™ system compatible interface

The Intel 86/20 and 88/20 are high performance numeric data processors packaged in two 40-pin packages as shown in Figure 1. The 86/20 and 88/20 fully conform to the proposed IEEE floating point standard. The 86/20 and 88/20 provide 68 numeric instructions beyond those

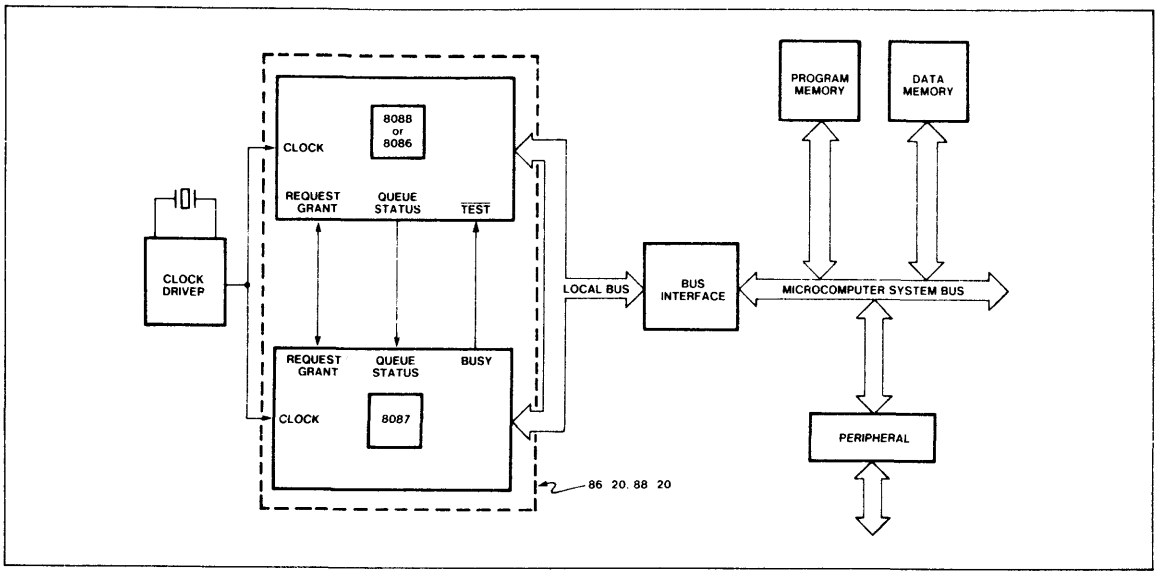


Figure 5. 86/20, 88/20

provided by the iAPX 86/10 and iAPX 88/10, providing a complete solution to high performance numeric and data processing.

PROCESSOR OVERVIEW

The numeric data processor performs arithmetic and logical operations on a variety of data types. The 86/20 and 88/20 have an extended register set over the 86/10 and 88/10 CPUs. Figure 6 presents this register view graphically.

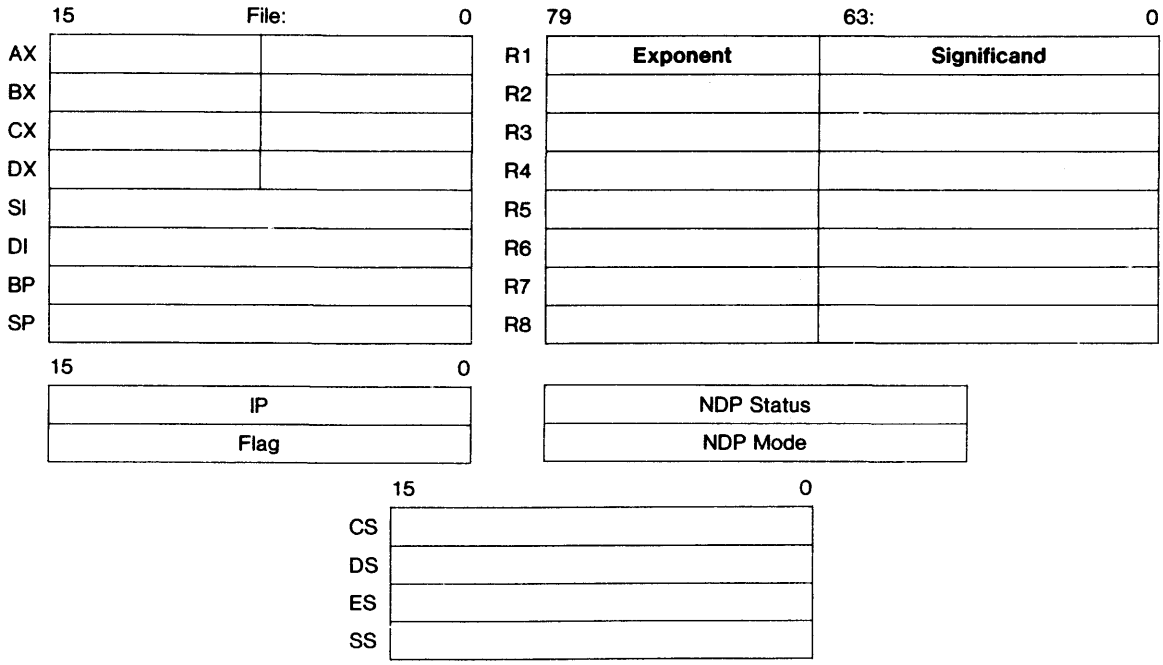


Figure 6. 86/20, 88/20 Architecture

The 86/20, 88/20 support 13 data types. These include six standard types and also seven numeric types, as shown in Figures 7A and 7B.

Format	Length
Byte	8 bits
Word	16 bits
Pointer	32 bits
ASCII	1 char/byte
Packed BCD	2 digits/byte
String	1-64K bytes

Figure 7A. Standard Data Types

DATA FORMATS	RANGE	PRECISION	MOST SIGNIFICANT BYTE
			7 07 07 07 07 07 07 07 07 07 0
Word integer	10^4	16 BITS	I_{15} I_0 TWO'S COMPLEMENT
Short integer	10^9	32 BITS	I_{31} I_0 TWO'S COMPLEMENT
Long integer	10^{18}	64 BITS	I_{63} I_0 TWO'S COMPLEMENT
Packed BCD	10^{18}	18 DIGITS	S — $D_{17}D_{16}$ D_1 D_0
Short real	$10^{\pm 38}$	24 BITS	S E_7 E_0 F_1 F_{23} F_0 IMPLICIT
Long real	$10^{\pm 308}$	53 BITS	S E_{10} E_0 F_1 F_{52} F_0 IMPLICIT
Temporary real	$10^{\pm 4932}$	64 BITS	S E_{14} E_0 F_0 F_{63}

INTEGER: I

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

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

BIAS=127 FOR SHORT REAL

1023 FOR LONG REAL

16383 FOR TEMP REAL

Figure 7B. Numeric Data Types

Internally, the NDP holds all numeric data in a temporary real format (80 bits). This format has extended range and precision. There are key contributors to the NDP's ability to consistently deliver stable, expected results. NDP load and store instructions convert operands represented in memory as 16, 32 or 64 bit integers, 32 or 64 bit floating point numbers (with the range of $+/- 10^{4932}$) or 18 digit packed BCD numbers into temporary real format and vice versa. The fact that these conversions are made, and that calculations are performed on converted numbers, is transparent to the programmer. Integer operands yield correct integer results, decimal operands yield correct decimal results. You control errors such as round off, underflow and overflow in intermediate calculations.

In addition to the standard 86/10, 88/10 instruction set for data manipulation and control, the NDP supports 68 numeric instructions for floating point, trignometric, logrithmic and exponential functions. Example times for several numerical operations are shown in Figure 8. Overall 86/20 system performance is typically 100–130K Whetstones/sec (depending on compiler efficiency), over 50 times the performance of an 86/10 class processor for this scientific benchmark.

APPLICATION AREAS

Viewed strictly from the standpoint of raw speed, the 86/20, 88/20 enable serious computation-intensive tasks to be performed by microprocessors for the first time. The 86/20 and 88/20 provide more than just numeric performance, however. By combining advances made by numerical analysts in the past several years, the NDP's provide a level of usability that surpasses existing minicomputer and mainframe arithmetic units.

Beyond traditional numerics support for scientific applications, the 86/20 and 88/20 have built-in facilities for "commercial" computing. They can process decimal numbers of up to 18 digits without roundoff errors, and perform exact arithmetic on integers as large as 64 bits, allowing many accounting and financial programs to utilize 86/20 or 88/20 performance.

Instruction	Approximate Execution Time (μs)	
	86/20 (5 MHz Clock)	86/10 Emulation of 86/20 (5 MHz Clock)
Add/Subtract Magnitude	14/18	1,600
Multiply (Single Precision)	18	1,600
Multiply (Double Precision)	27	2,100
Divide	39	3,200
Compare	10	1,300
Load (Single Precision)	9	1,700
Store (Single Precision)	17	1,200
Square Root	36	19,600
Tangent	110	13,000
Exponentiation	130	17,100

Figure 8. Execution Time for Selected 86/20 Actual and Emulated Numeric Instructions

PROGRAMMING INTERFACE

Numeric programs for NDP's can be written in ASM-86, the 86/20, 88/20 common assembly language. ASM-86 provides directives for defining all numeric data types and mnemonics for all instructions. All 86/10, 88/10 addressing modes are available in the 86/20, 88/20 and may be used to access memory-based numeric operands, enabling convenient processing of numeric arrays, structures, based variables, etc.

NDP routines may also be written in PL/M-86, Fortran-86, and Pascal-86, Intel's high-level languages for the 86/20 and 88/20. These languages provide the programmer with access to many numeric facilities while reducing the programmer's need to understand the architecture of the system.

I/O PROCESSOR

- Complete I/O capability for iAPX product lines
- Memory-based communication with CPU
- Dual channels with full independent register files
- Allows mixed interface of 8/16 bit peripherals to 8/16 bit processor buses
- Supports local or remote I/O processing with I/O oriented instructions set
- Flexible, intelligent channel functions including: translation, search, word assembly/disassembly and DMA from port to memory, port to port or memory to memory

The IOP performs the function of an intelligent I/O channel for the iAPX product line and with its processing power can remove I/O overhead from the iAPX 86, 88, 188, 186, 286, 432. It may operate completely in parallel with a host processor giving dramatically improved performance in I/O intensive applications. The IOP provides two I/O channels, each supporting a transfer rate up to 1.25 megabyte/sec at the standard clock frequency of 5 MHz. Memory based communication between the IOP and CPU enhances system flexibility and encourages software modularity, yielding more reliable, easier to develop systems. The iAPX architecture provides for multiple IOP's in a system to incrementally increase performance.

The IOP continues the trend of simplifying the CPU's "view" of I/O devices by off loading another level of control from the CPU (Figure 10). The CPU performs an I/O operation by depositing a message in memory that describes the function to be performed; the IOP reads the message, carries out the operation and notifies the CPU when it has finished. All I/O devices appear to the CPU as transmitting and receiving complete blocks of data; the IOP can make both byte- and word-level transfers invisible to the CPU. The IOP assumes all device controller overhead, performs both programmed and DMA transfers, and allows recovery from "soft" I/O errors without CPU intervention; all of these activities may be performed while the CPU is executing other tasks.

In the remote configuration, one or more IOPs share a common system bus with the CPU (Figure 11). Access to this bus is controlled by 8289 Bus Arbiters. The IOP's I/O bus, however, is physically separated from the CPU in the remote configuration. Two IOPs can share the local I/O bus. Any number of remote IOPs may be contained in a system, configured in remote clusters of one or two. The local I/O bus need not be the same physical width as the shared system bus, allowing an IOP, for example, to interface 8-bit peripherals to an 86/10, 88/10. In the remote configuration, the IOP can access local I/O devices and memory without using the shared system bus, thereby reducing system bus contention with the CPU.

Contention can further be reduced by locating the IOP's channel programs in the local I/O space. The IOP can then also fetch instructions without accessing the system bus. CPU/IOP communication blocks must be located in shared (global) system memory, however, so that both processors can access them. The remote configuration thus increases the degree to which an IOP and a CPU can operate in parallel and thereby increases a system's throughput potential.

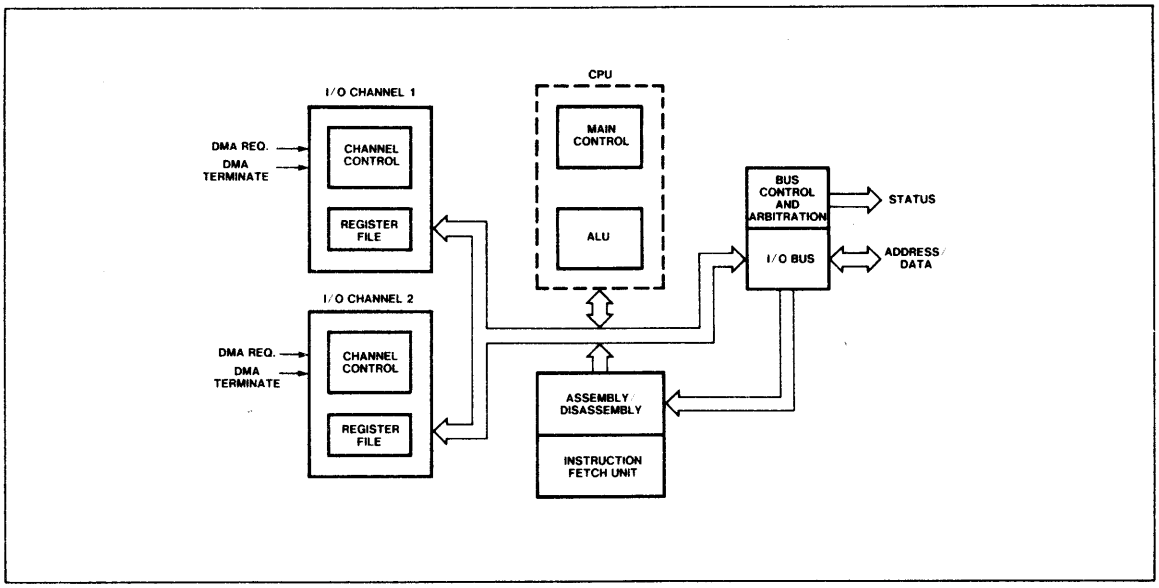


Figure 9. I/O Processor Block Diagram

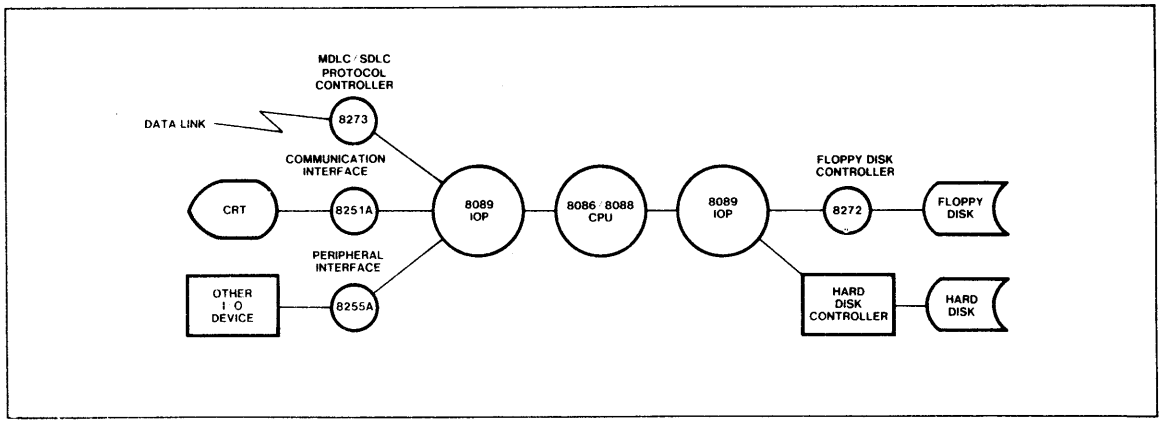


Figure 10. Typical I/O Control

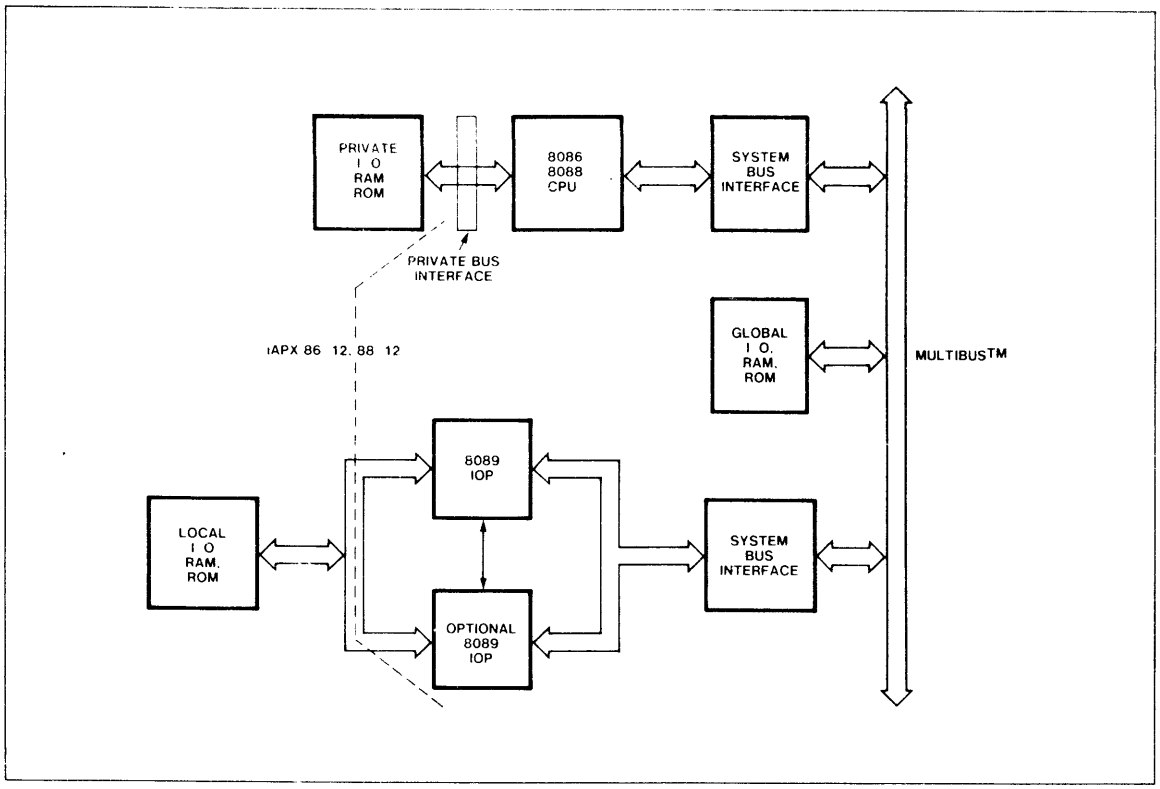


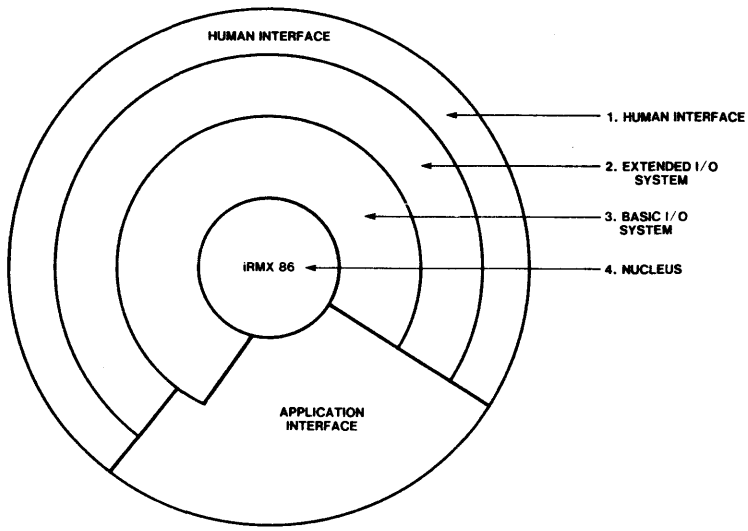
Figure 11. iAPX 86/12, 88/12 System

iRMX 86™
Operating System

- **Structured environment**
- **User extensible**
- **Powerful error handlers**
- **EPROM or RAM based**
- **User configurable**
- **Comprehensive I/O system**
- **Extensive debugging aids**
- **Full human interface**
- **Real-time priority oriented scheduling**

The iRMX 86 Real-Time Operating System is an easy-to-use, sophisticated, software system which operates on Intel iAPX 86 processors. The iRMX 86 Operating System extends the iAPX 86 architecture providing a structured, efficient environment for application programs. Services provided by the iRMX 86 Operating System include facilities for concurrent program execution, resource and information sharing, servicing asynchronous events, and interactive control over system resources and utilities. In addition, the iRMX 86 Operating System provides all major real-time facilities including priority-based system resource allocation; the means to concurrently monitor and control multiple external events; real-time clock control; and interrupt handling and task dispatching.

Because the services provided by the operating system are user selectable, application specific operating systems can be created. The iRMX 86 operating system thus eliminates the need for custom operating system design and hence reduces development time costs and risk. The iRMX 86 Operating System is a full featured operating system including a device independent input/output sub-system, a human interface sub-system with command language interpreter and ASCII console interface, a terminal handler, and an interactive debugger.



iRMX 86™

The iRMX 86 Operating System provides users of the Intel iAPX 86 simple, easy-to-use tools for creating a wide range of application systems. The most important features of the iRMX 86 Operating System are:

Structured Environment: The iRMX 86 Operating System provides a consistent structure from application to application thus allowing experience gained on one system to be easily transferred to others. Often entire programs may be used in multiple applications.

User Configurable: iRMX 86 Operating System based applications may be configured from a wide range of facilities, selecting only those which meet the specific requirements of the application system. The resultant system contains only the modules necessary for its use, allowing the iRMX 86 Operating System to be cost effectively applied in a wide range of application environments from performance oriented industrial applications, to security conscious data processing systems. The iRMX 86 Operating System is constructed in a thoroughly modular manner with the full range of facilities being offered in library modules, allowing easy selection of the exact features required. This eliminates overhead that might have been incurred for facilities not required.

User Extensible: The iRMX 86 Operating System provides a framework in which to extend the system and have these extensions look like facilities provided by Intel. These extensions include not only the ability to add custom system calls, but also the ability to add operating system data structures. Check-out of these extensions is easily accomplished by using the debugging sub-system because the extensions look like an integral part of the operating system. Because the operating system is designed for user extensibility, the extensions can be added in a symmetric manner resulting in a homogenous customized operating system.

EPROM or RAM Based: iRMX 86 can be EPROM-resident or loaded from a mass storage device into RAM depending upon application requirements. Being able to place all of the software in EPROM offers two benefits. First, if the application is to be used in a harsh environment mass storage devices cannot be used because of the danger of contamination. Second, if the application is small, the expense and overhead of mass storage devices need not be incurred.

Extensive Debugging Subsystem: The iRMX 86 Operating System provides interactive software debugging. The debugger permits both register and memory examination and modification. Task-oriented debugging is accomplished by symbolically setting breakpoints according to tasks rather than absolute memory locations. When a breakpoint has occurred, all operating system lists can be viewed so the system state can be determined.

NUCLEUS

The iRMX 86 nucleus provides a foundation upon which a variety of application systems can be built. A wide range of multi-programming, multi-tasking, and real-time oriented facilities are included. Extensive task to task communication and control permits easy task synchronization and data transfer thus allowing concurrent applications greater control over their execution environment.

Real-Time Priority Oriented Scheduler: The iRMX 86 Scheduler insures that the highest priority task ready to execute is given system control because the scheduler recognizes 255 software priority levels. Also, the system supports 64 hardware priority levels allowing the application system to be responsive to its external environment.

Error Handling Sub-system: The iRMX 86 Operating System provides extensive error handling and reporting mechanisms. Both excessive system loading and user operator errors can be reported causing system debug time to be shortened. The flexibility of the error handling subsystem allows errors to be serviced directly by the user task or sent to a specific error handler.

I/O SYSTEM

Comprehensive I/O System: The iRMX 86 Operating System offers a device independent I/O system which provides a standard interface for application programs to communicate with all I/O devices. A hierarchical directory structure provides quick, efficient file access. A standardized device driver interface allows users to easily create custom device drivers. A wide range of standard drivers are available; iSBC 204 Single Density Diskette Controller, iSBC 206 Hard Disk Controller, and terminal handler (via on-board 8251 USART device).

HUMAN INTERFACE

Human Interface Subsystem: The iRMX 86 Human Interface provides interactive control of system resources and utilities. iRMX 86 system utilities include file directories, copy files, rename files, etc. The Intel-supplied command line interpreter is table driven allowing easy modification by the user for application specific requirements. An application specific command language can be created. When using the command line interpreter this application language will be translated so that the appropriate user program will be invoked.

intel

The 8086 Family User's Manual

October 1979

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

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

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

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

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

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

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

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

Table of Contents

CHAPTER 1
INTRODUCTION

PAGE

Manual Organization 1-1
8086 Family Architecture 1-1
Functional Distribution 1-1
Microprocessors 1-2
Interrupt Controller 1-3
Bus Interface Components 1-3
Multiprocessing 1-3
Bus Organization 1-4
Local Bus 1-4
System Bus 1-5
Processing Modules 1-6
Bus Implementation Examples 1-6
Development Aids 1-12

CHAPTER 2
THE 8086 AND 8088 CENTRAL PROCESSING UNITS

Processor Overview 2-1
Processor Architecture 2-3
Execution Unit 2-5
Bus Interface Unit 2-5
General Registers 2-6
Segment Registers 2-7
Instruction Pointer 2-7
Flags 2-7
8080/8085 Register and Flag Correspondence ... 2-8
Mode Selection 2-8
Memory 2-8
Storage Organization 2-8
Segmentation 2-10
Physical Address Generation 2-11
Dynamically Relocatable Code 2-13
Stack Implementation 2-14
Dedicated and Reserved Memory Locations 2-14
8086/8088 Memory Access Differences 2-15
Input/Output 2-15
Input/Output Space 2-16
Restricted I/O Locations 2-16
8086/8088 Memory Access Differences 2-16
Memory-Mapped I/O 2-16
Direct Memory Access 2-17
8089 Input/Output Processor (IOP) 2-17
Multiprocessing Features 2-17
Bus Lock 2-17
WAIT and $\overline{\text{TEST}}$ 2-18
Escape 2-19
Request/Grant Lines 2-20
Multibus™ Architecture 2-21

PAGE

8289 Bus Arbiter 2-22
Processor Control and Monitoring 2-22
Interrupts 2-22
External Interrupts 2-22
Internal Interrupts 2-24
Interrupt Pointer Table 2-25
Interrupt Procedures 2-26
Single-Step (Trap) Interrupt 2-28
Breakpoint Interrupt 2-28
System Reset 2-29
Instruction Queue Status 2-29
Processor Halt 2-29
Status Lines 2-30
Instruction Set 2-30
Data Transfer Instructions 2-31
General Purpose Data Transfers 2-31
Address Object Transfers 2-32
Flag Transfers 2-32
Arithmetic Instructions 2-33
Arithmetic Data Formats 2-33
Arithmetic Instructions and Flags 2-34
Addition 2-35
Subtraction 2-36
Multiplication 2-36
Division 2-37
Bit Manipulation Instructions 2-38
Logical 2-38
Shifts 2-39
Rotates 2-39
String Instructions 2-40
Program Transfer Instructions 2-43
Unconditional Transfers 2-43
Conditional Transfers 2-45
Iteration Control 2-45
Interrupt Instructions 2-46
Processor Control Instructions 2-47
Flag Operations 2-47
External Synchronization 2-48
No Operation 2-48
Instruction Set Reference Information 2-48
Addressing Modes 2-68
Register and Immediate Operands 2-68
Memory Addressing Modes 2-68
The Effective Address 2-68
Direct Addressing 2-69
Register Indirect Addressing 2-69
Based Addressing 2-70
Indexed Addressing 2-70

PAGE

Based Indexed Addressing 2-71
String Addressing 2-72
I/O Port Addressing 2-72
Programming Facilities 2-72
Software Development Overview 2-73
PL/M-86 2-75
Statements and Comments 2-75
Data Definition 2-75
Assignment Statement 2-77
Program Flow Statements 2-79
Procedures 2-81
ASM-86 2-83
Statements 2-83
Constants 2-84
Defining Data 2-85
Records 2-85
Structures 2-87
Addressing Modes 2-87
Segment Control 2-88
Procedures 2-90
LINK-86 2-90
LOC-86 2-90
LIB-86 2-91
OH-86 2-91
CONV-86 2-92
Sample Programs 2-92
Programming Guidelines and Examples 2-96
Programming Guidelines 2-96
Segments and Segment Registers 2-96
Self-Modifying Code 2-96
Input/Output 2-97
Operating Systems 2-97
Interrupt Service Procedures 2-99
Stack-Based Parameters 2-100
Flag Images 2-100
Programming Examples 2-100
Procedures 2-100
Jumps and Calls 2-105
Records 2-110
Dynamic Code Relocation 2-113
Memory-Mapped I/O 2-115
Breakpoints 2-117
Interrupt Procedures 2-119
String Operations 2-125

CHAPTER 3
THE 8089 INPUT/OUTPUT PROCESSOR

Processor Overview 3-1
Evolution 3-1
Principles of Operation 3-2
CPU/IOP Communications 3-2
Channels 3-4
Channel Programs (Task Blocks) 3-4

PAGE

DMA Transfers 3-5
Bus Configurations 3-5
A Sample Transaction 3-10
Applications 3-12

Processor Architecture 3-13
Common Control Unit (CCU) 3-13
Arithmetic/Logic Unit (ALU) 3-13
Assembly/Disassembly Registers 3-14
Instruction Fetch Unit 3-14
Bus Interface Unit (BIU) 3-16
Channels 3-16
I/O Control 3-16
Registers 3-17
Program Status Word 3-18
Tag Bits 3-19
Concurrent Channel Operation 3-20
Memory 3-21
Storage Organization 3-22
Dedicated and Reserved Memory Locations 3-23
Dynamic Relocation 3-23
Memory Access 3-24
Input/Output 3-25
Programmed I/O 3-25
I/O Instructions 3-25
Device Addressing 3-26
I/O Bus Transfers 3-26
DMA Transfers 3-27
Preparing the Device Controller 3-27
Preparing the Channel 3-27
Beginning the Transfer 3-31
DMA Transfer Cycle 3-32
Following the Transfer 3-33
Multiprocessing Features 3-34
Bus Arbitration 3-34
Request/Grant Line 3-35
8289 Bus Arbiter 3-36
Bus Arbitration for IOP Configurations 3-36
Bus Load Limit 3-36
Bus Lock 3-37
Processor Control and Monitoring 3-37
Initialization 3-37
Channel Commands 3-40
DRQ (DMA Request) 3-43
EXT (External Terminate) 3-43
Interrupts 3-43
Status Lines 3-43
Instruction Set 3-44
Data Transfer Instructions 3-44
Arithmetic Instructions 3-45
Logical and Bit Manipulation Instructions 3-46
Program Transfer Instructions 3-48
Processor Control Instructions 3-49
Instruction Set Reference Information 3-51

PAGE

Addressing Modes 3-59
Register and Immediate Operands 3-59
Memory Addressing Modes 3-59
The Effective Address 3-60
Based Addressing 3-60
Offset Addressing 3-60
Indexed Addressing 3-60
Indexed Auto-Increment Addressing 3-61
Programming Facilities 3-63
ASM-89 3-63
Statements 3-63
Constants 3-66
Defining Data 3-66
Structures 3-67
Addressing Modes 3-68
Program Transfer Targets 3-68
Procedures 3-69
Segment Control 3-69
Intermodule Communication 3-70
Sample Program 3-73
Linking and Locating ASM-89 Modules 3-76
Programming Guidelines and Examples 3-79
Programming Guidelines 3-79
Segments 3-79
Self-Modifying Code 3-79
I/O System Design 3-79
Programming Examples 3-81
Initialization and Dispatch 3-81
Memory-to-Memory Transfer 3-85
Saving and Restoring Registers 3-85

CHAPTER 4
HARDWARE REFERENCE INFORMATION

Introduction 4-1
8086 and 8088 CPUs 4-1
CPU Architecture 4-1
Bus Operation 4-5
Clock Circuit 4-10
Minimum/Maximum Mode 4-10
Minimum Mode 4-11
Maximum Mode 4-11
External Memory Addressing 4-14
I/O Interfacing 4-15
Interrupts 4-16
Machine Instruction Encoding and Decoding 4-18
8086 Instruction Sequence 4-37
8089 I/O Processor 4-38
System Configuration 4-39
Local Mode 4-39
Remote Mode 4-40
Bus Operation 4-41
Initialization 4-44
I/O Dispatching 4-46

PAGE

DMA Transfers 4-47
DMA Termination 4-50
Peripheral Interfacing 4-50
Instruction Encoding 4-52

APPENDIX A
APPLICATION NOTES

AP-67 8086 System Design A-3
AP-61 Multitasking for the 8086 ..... A-67
AP-50 Debugging Strategies and Considerations for 8089 Systems A-85
AP-51 Designing 8086, 8088, 8089 Multiprocessing Systems with the 8289 Bus Arbiter A-111
AP-59 Using the 8259A Programmable Interrupt Controller A-135
AP-28A Intel®MultibusTM Interfacing A-175
AP-43 Using the iSBC-957TM Execution Vehicle for Executing 8086 Program Code A-209

APPENDIX B
DEVICE SPECIFICATIONS

8086 Family
8086/8086-2/8086-4 16-Bit HMOS Microprocessor B-1
M8086 16-Bit HMOS Microprocessor B-22
I8086 16-Bit HMOS Microprocessor B-23
8088 8-Bit HMOS Microprocessor B-24
8089 8/16-Bit HMOS I/O Processor B-46
8282/8283 Octal Latch B-59
8284 Clock Generator and Driver for 8086, 8088, 8089 Processors B-63
M8284 Clock Generator and Driver for 8086, 8088, 8089 Processors B-69
I8284 Clock Generator and Driver for 8086, 8088, 8089 Processors B-70
8286/8287 Octal Bus Transceiver B-71
8288 Bus Controller for 8086, 8088, 8089 Processors B-75
8289 Bus Arbiter B-81
8237/8237-2 High Performance Programmable DMA Controller B-92
8259A/8259A-2/8289A-8 Programmable Interrupt Controller B-106
8085 Peripherals
8155/8156/8155-2/8156-2 2048 Bit Static MOS RAM with I/O Ports and Timer ... B-124
8185/8185-2 1024 x 8-Bit Static RAM for MCS-85TM B-125
8355/8355-2 16,384-Bit ROM with I/O B-126
8755A/8755A-2 16,384-Bit EPROM with I/O B-127

PAGE

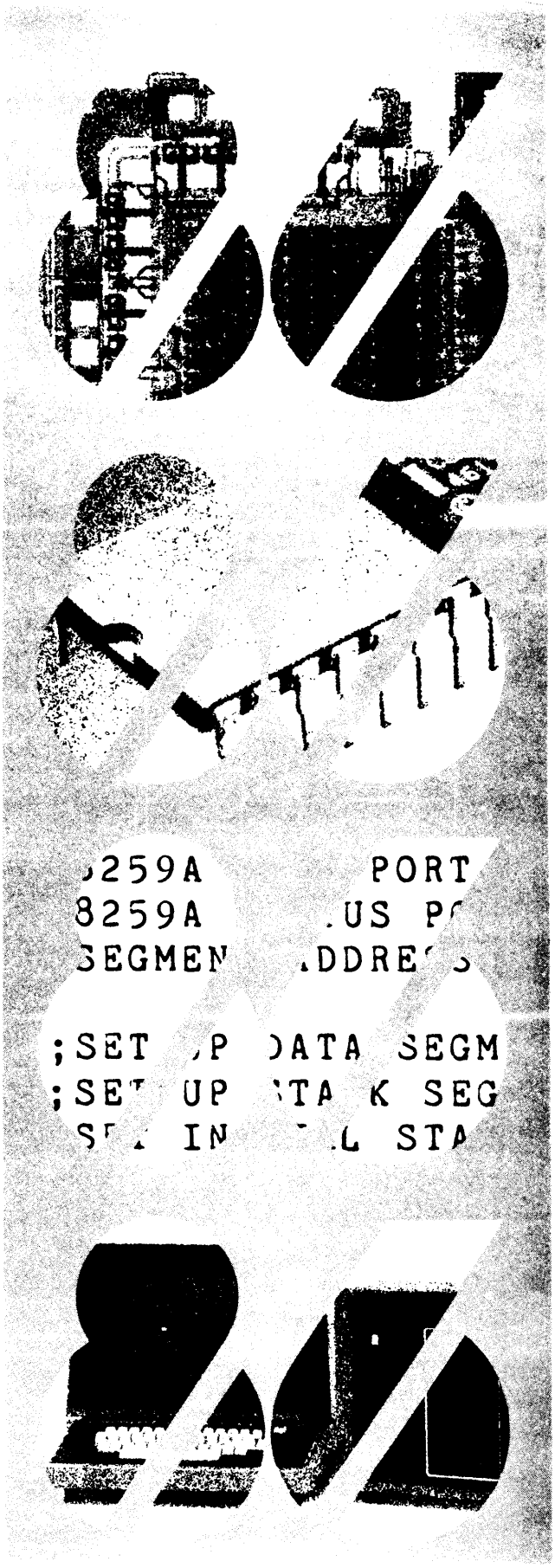
Standard Peripherals
8041A/8741A Universal Peripheral Interface 8-Bit Microcomputer B-128
8202 Dynamic RAM Controller..... B-129
8205 High Speed 1 Out of 8 Binary Decoder .. B-130
8251A Programmable Communication Interface..... B-131
8253/8253-5 Programmable Interval Timer... B-132
8255A/8255A-5 Programmable Peripheral Interface..... B-133
8271/8271-6/8271-8 Programmable Floppy Disk Controller B-134
8273 Programmable HDLC/SDLC Protocol Controller..... B-135
8275 Programmable CRT Controller B-136
8279/8279-5 Programmable Keyboard/Display Interface..... B-137
8291 GPIB Talker/Listener B-138
8292 GPIB Controller..... B-139
8293 GPIB Transceiver..... B-140
8294 Data Encryption Unit B-141
8295 Dot Matrix Printer Controller B-142

RAM Memories
2114A 1024 x 4 Bit Static RAM B-143
2142 1024 x 4 Bit Static RAM..... B-144
2148 1024 x 4 Bit Static RAM..... B-145

EPROM Memories
2716 16K (2K x 8) UV Erasable PROM..... B-146
2732 32K (4K x 8) UV Erasable PROM..... B-147
2758 8K (1K x 8) UV Erasable Low Power PROM B-148

Development Tools
Model 230 Intellec® Series II Microcomputer Development System.... B-149
8086/8088 Software Development Package B-153
8089 Assembler Support Package B-163
ICE-86™ 8086 In-Circuit Emulator B-165
iSBC 86/12A™ Single Board Computer B-171
iSBC 957™ Intellec® -iSBC 86/12A™ Interface and Execution Package..... B-179
iSBC 300/340™ iSBC 300™ 32K-Byte RAM Expansion Module iSBC 340™ 16K-Byte EPROM/ROM Expansion Module B-184
SDK-86 MCS-86™ System Design Kit..... B-188
SDK-C86 MCS-86™ System Design Kit B-194

Chapter 1
Introduction



CHAPTER 1
INTRODUCTION

This publication describes the Intel® 8086 family of microcomputing components, concentrating on the 8086, 8088 and 8089 microprocessors. It is written for hardware and software engineers and technicians who understand microcomputer operating principles. The manual is intended to introduce the product line and to serve as a reference during system design and implementation.

Recognizing that successful microcomputer-based products are judicious blends of hardware and software, the *User's Manual* addresses both subjects, although at different levels of detail. This publication is the definitive source for information describing the 8086 family components. Software topics, such as programming languages, utilities and examples, are given moderately detailed, but by no means complete, coverage. Additional references, available from Intel's Literature Department, are cited in the programming sections.

1.1 Manual Organization

The manual contains four chapters and three appendices. The remainder of this chapter describes the architecture of the 8086 family, and subsequent chapters cover the individual components in detail.

Chapter 2 describes the 8086 and 8088 Central Processing Units, and Chapter 3 covers the 8089 Input/Output Processor. These two chapters are identically organized and focus on providing a *functional* description of the 8086, 8088 and 8089, plus related Intel hardware and software products. Hardware reference information—electrical characteristics, timing and physical interfacing considerations—for all three processors is concentrated in Chapter 4.

Appendix A is a collection of 8086 family application notes; these provide design and debugging examples. Appendix B contains complete data sheets for all the 8086 family components and system development aids; summary data sheets covering compatible components from other Intel product lines are also reproduced in Appendix B.

1.2 8086 Family Architecture

Considered individually, the 8086, 8088 and 8089 are advanced third-generation microprocessors. Moreover, these processors are elements of a larger design, that of the 8086 family. This systems architecture specifies how the processors and other components relate to each other, and is the key to the exceptional versatility of these products.

The components in the 8086 family have been designed to operate together in diverse combinations within the systematic framework of the overall family architecture. In this way a single family of components can be used to solve a wide array of microcomputing problems. A component mix can be tailored to fit the performance needs of an application precisely, without having to pay for unneeded capabilities that may be bundled into more monolithic, CPU-centered architectures. Using the same family of components across multiple systems limits the learning curve problem and builds on past experience. Finally, the modular structure of the family architecture provides an orderly way for systems to grow and change.

The 8086 family architecture is characterized by three major principles:

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

Functional Distribution

Table 1-1 lists the components that constitute the 8086 microprocessor family. All components are contained in standard dual in-line packages and require single +5V power sources.

Table 1-1. 8086 Component Family

Microprocessor	Technology	Pins	Description
8086 Central Processing Unit (CPU)	HMOS	40	8/16 bit general-purpose microprocessor; 16-bit external data path.
8088 Central Processing Unit (CPU)	HMOS	40	8/16 bit general-purpose microprocessor; 8-bit external data path.
8089 Input/Output Processor (IOP)	HMOS	40	8/16 bit microprocessor optimized for high-speed I/O operations; 8-bit and 16-bit external data paths.

Support Component	Technology	Pins	Function
8259A Programmable Interrupt Controller (PIC)	NMOS	28	Identifies highest-priority interrupt request.
8282 Octal Latch 8283 Octal Latch (Inverting)	Bipolar	20	Demultiplexes and increases drive of address bus.
8284 Clock Generator and Driver	Bipolar	18	Provides time base.
8286 Octal Bus Transceiver 8287 Octal Bus Transceiver (Inverting)	Bipolar	20	Increases drive on data bus.
8288 Bus Controller	Bipolar	20	Generates bus command signals.
8289 Bus Arbiter	Bipolar	20	Controls access of microprocessors to multimaster system bus.

Microprocessors

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

- Standard operating speed is 5 MHz (200 ns cycle time); a selected 8 MHz version of the 8086 CPU is also available.
- Chips are housed in reliable 40-pin packages.
- Processors operate on both 8- and 16-bit data types; internal data paths are at least 16 bits wide.
- Up to 1 megabyte of memory can be addressed, along with a separate 64k byte I/O space.
- The address/data and status interfaces of the processors are compatible (the address and data buses are time-multiplexed at the processor, i.e., an address transmission is followed by a data transmission over a subset of the same physical lines).

The 8086 and 8088 are third-generation central processing units (CPUs) that differ primarily in their external data paths. The 8088 transfers data between itself and other system components 8 bits at a time. The 8086 can transfer either 8 or 16 bits in one bus cycle and is therefore capable of greater throughput. Both processors have two operating modes, selectable by a strapping pin. In minimum mode, the CPUs emit the bus control signals needed by memory and I/O peripheral components. In maximum mode, an 8288 Bus Controller assumes responsibility for controlling devices attached to the system bus. CPU pins no longer needed for bus control are then redefined to provide signals that support multiprocessing systems.

The 8089 Input/Output Processor (IOP) is an independent microprocessor whose design has been optimized for transferring data. The 8089

typically runs under the direction of a CPU, but it executes a separate instruction stream and can operate in parallel with other system processors. The IOP contains two independent I/O channels that combine attributes of both CPUs and advanced DMA (direct memory access) controllers. The channels can execute programs and perform programmed I/O operations similar to CPUs. They may also transfer data by DMA, at rates up to 1.25 megabytes per second (5 MHz version). The channels can support mixes of 8- and 16-bit I/O devices and memory. Combining speed with programmable intelligence, the 8089 can assume the bulk of I/O processing overhead and thereby free a CPU to perform other tasks.

Interrupt Controller

The 8259A Programmable Interrupt Controller (PIC) is a new, 8086 family-compatible version of the familiar 8259 that has been enhanced to operate with the advanced interrupt facilities of the 8086 and 8088 CPUs. The 8259A accepts interrupt requests from up to eight sources; up to 64 sources may be accommodated by "cascading" additional 8259As. Each interrupt source is assigned a priority number that typically reflects its "criticality" in the system. The 8259A has several built-in, priority-resolving mechanisms that are selectable by software commands from the CPU. These modes operate somewhat differently, but in general the 8259A continuously identifies the highest-priority active interrupt request and generates an interrupt request to the CPU if this request has higher priority than the request currently being processed. When the CPU recognizes the interrupt request, the 8259A transfers a code to the CPU that identifies the interrupt source.

Bus Interface Components

Components may be selected from this modular group to implement different system bus configurations. Except for the 8284, all components are optional; their inclusion in a system is based on the needs of the application. All of the bus interface components are implemented using bipolar technology to provide high-quality, high-drive signals and very fast internal switching.

The 8284 Clock Generator and Driver provides the time base for the 8086 family microprocessors. It divides the frequency signal from an external crystal or TTL signal by three and outputs the 5 MHz or 8 MHz processor clock signal. It also provides the microprocessors with reset and ready signals.

8282 or 8283 Octal Latches may be added to a system to demultiplex the combined address/data bus generated by the 8086 family microprocessors. A demultiplexed bus provides separate stable address and data lines required by many peripheral components. Two latches demultiplex 16 bits of the bus to provide an address space of up to 64k bytes, while three latches generate the full 20-bit (megabyte) address space. The latches also provide the high drive on the address lines needed in larger systems.

8286 and 8287 Octal Bus Transceivers are used to provide more drive on data lines than the processors themselves are capable of providing. One or two transceivers may be used depending on the width of the data bus (8 or 16 bits).

The 8288 Bus Controller decodes status signals output by an 8089, or a maximum mode 8086 or 8088. When these signals indicate that the processor is to run a bus cycle, the 8288 issues a bus command that identifies the bus cycle as memory read, memory write, I/O read, I/O write, etc. It also provides a signal that strobes the address into 8282/83 latches. The 8288 provides the drive levels needed for the bus control lines in medium to large systems.

The 8289 Bus Arbiter controls the access of a processor to a multimaster system bus. A multimaster bus is a path to system resources (typically memory) that is shared by two or more microprocessors (masters). Arbiters for each master may use one of several priority-resolving techniques to ensure that only one master drives the shared bus.

Multiprocessing

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

- system tasks may be allocated to special-purpose processors whose designs are optimized to perform certain types of tasks simply and efficiently;

- very high levels of performance can be attained when multiple processors can execute simultaneously (parallel processing);
- robustness can be improved by isolating system functions so that a failure or error in one part of the system has a limited effect on the rest of the system;
- the natural partitioning of the system promotes parallel development of subsystems, breaks the application into smaller, more manageable tasks, and helps isolate the effects of system modifications.

The 8086 family architecture is explicitly designed to simplify the development of multiple processor systems by providing facilities for coordinating the interaction of the processors.

The architecture supports two types of processors: independent processors and coprocessors. An independent processor is one that executes its own instruction stream. The 8086, 8088 and 8089 are examples of independent processors. An 8086 or 8088 typically executes a program in response to an interrupt. The 8089 starts its channels in response to an interrupt-like signal called a channel attention; this signal is typically issued by a CPU.

The 8086 architecture also supports a second type of processor, called a coprocessor. Coprocessor "hooks" have been designed into the 8086 and 8088 so that this type of processor can be accommodated in the future. A coprocessor differs from an independent processor in that it obtains its instructions from another processor, called a host. The coprocessor monitors instructions fetched by the host and recognizes certain of these as its own and executes them. A coprocessor, in effect, extends the instruction set of its host processor.

The 8086 family architecture provides built-in solutions to two classic multiprocessing coordination problems: bus arbitration and mutual exclusion. Bus arbitration may be performed by the bus request/grant logic contained in each of the processors, by 8289 Bus Arbiters, or by a combination of the two when processors have access to multiple shared buses. In all cases, the arbitration mechanism operates invisibly to software.

For mutual exclusion, each processor has a LOCK (bus lock) signal which a program may activate to prevent other processors from obtaining a shared system bus. The 8089 may lock the bus during a DMA transfer to ensure that both the transfer completes in the shortest possible time and that another processor does not access the target of the transfer (e.g., a buffer) while it is being updated. Each of the processors has an instruction that examines and updates a memory byte with the bus locked. This instruction can be used to implement a semaphore mechanism for controlling the access of multiple processors to shared resources. (A semaphore is a variable that indicates whether a resource, such as a buffer or a pointer, is "available" or "in use"; section 2.5 discusses semaphores in more detail).

Bus Organization

Figure 1-1 summarizes the 8086 family bus structure. There are two different types of buses: system and local. Both buses may be shared by multiple processors, i.e., both are multimaster buses. Microprocessors are always connected to a local bus, and memory and I/O components usually reside on a system bus. The 8086 family bus interface components link a local bus to a system bus.

Local Bus

The local bus is optimized for use by the 8086 family microprocessors. Since standard memory and I/O components are not attached to the local bus, information can be multiplexed and encoded to make very efficient use of processor pins (certain MCS-85™ peripheral components can be directly connected to the local bus). This allows several pins to be dedicated to coordinating the activity of multiple processors sharing the local bus. Multiple processors connected to the same local bus are said to be local to each other; processors on different local buses are said to be remote to each other, or configured remotely. Both independent processors and coprocessors may share a local bus; on-chip arbitration logic determines which processor drives the bus. Because the processors on the local bus share the same bus interface components, the local configuration of multiple processors provides a compact and inexpensive multiprocessing system.

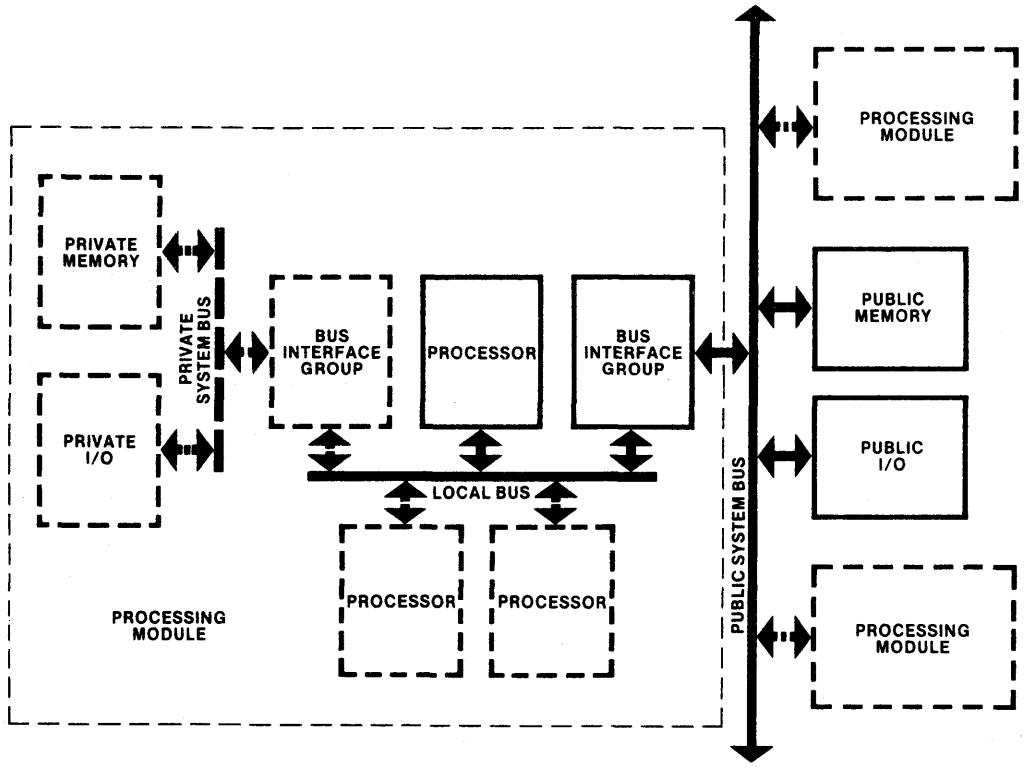


Figure 1-1. Generalized 8086 Family Bus Structure

System Bus

A full implementation of an 8086 system bus consists of the following five sets of signals:

1. address bus,
2. data bus,
3. control lines,
4. interrupt lines, and
5. arbitration lines.

These signals are designed to meet the needs of standard memory and I/O devices; the address and data buses are demultiplexed and traditional control signals (memory read/write, I/O read/write, etc.) are provided on the system bus.

The system bus design is modular and subsets may be implemented according to the needs of the application. For example, the arbitration lines are not needed in single-processor systems or in multiple-processor systems that perform arbitration at the local-bus level.

A group of bus interface components transforms the signals of a local bus into a system bus. The number of bus interface components required to generate a system bus depends on the size and complexity of the system; reduced application needs translate directly into reduced component counts. These main variables determine the configuration of a bus interface group: address space size (number of latches), data bus width (number of transceivers), and arbitration needs (presence of a bus arbiter).

The 8086 family system bus is functionally and electrically compatible with the Multibus™ multimaster system bus used in Intel's iSBC™ line of single board computing products. This compatability gives system designers access to a wide variety of computer, memory, communications and other modules that may be incorporated into products, used for evaluation or for test vehicles.

Processing Modules

The processor(s) and bus interface group(s) that are connected by a local bus constitute a processing module. A simple processing module could consist of a single CPU and one bus interface group. A more complex module would contain multiple processors, such as two IOPs, or a CPU and one or two IOPs. One bus interface group typically links the processors in the module to a public system bus. If there are multiple processing modules in the system, all memory or I/O connected to the public bus is accessible to all processing modules on the public bus. 8289 Bus Arbiters in each processing module control the access of the modules to the public bus and hence to the public memory and I/O.

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

Bus Implementation Examples

This section summarizes the 8086 family bus organization by showing how components from the family can be combined to implement diverse bus configurations. The first two examples illustrate special cases that extend the applicability of the 8086 family to smaller systems. The remaining examples add and recombine the same basic components to form progressively more complex bus configurations. Note that these examples are intended to be illustrative rather than exhaustive; many different combinations of components can be tailored to fit the needs of individual applications.

In its minimum mode configuration, the 8088 time-multiplexes its 8-bit data bus with the lower eight bits of its 20-bit address bus (figure 1-2). This multiplexed address/data bus, and the bus control signals emitted by the 8088, are directly compatible with the multiplexed bus components of Intel's 8085 family. These peripherals contain on-chip logic that demultiplexes a combined address/data bus. In addition, many of these devices are multifunctional, combining, for example, RAM, I/O ports and a timer on a single chip. By using these components, it is possible to build small (as few as four chips) economical systems that are nonetheless capable of performing significant computing tasks.

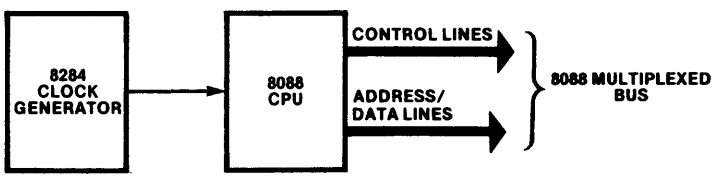


Figure 1-2. 8088 Multiplexed Bus

Combining 8282/83 latches with a minimum mode 8086 or 8088 produces a minimum mode system bus (figure 1-3). Two latches provide an address space of up to 64k bytes; adding a third latch provides access to the full megabyte of memory. An 8288 Bus Controller is not required for this implementation as the CPUs themselves emit the bus control signals when they are configured in the minimum mode. This demultiplexed bus structure is compatible with the wide array of memory and I/O components that have been developed for the industry-standard 8080A CPU. Eight-bit peripherals may be connected to both the upper and lower halves of the 8086's 16-bit data bus. 8286/87 transceivers may be added to provide additional drive on the data lines, where required. Including an 8259A gives the CPU the ability to respond to multiple interrupt sources without polling. The minimum mode system bus configuration is well-suited to a variety of systems whose computational requirements can be met by a single 8086 or 8088 CPU.

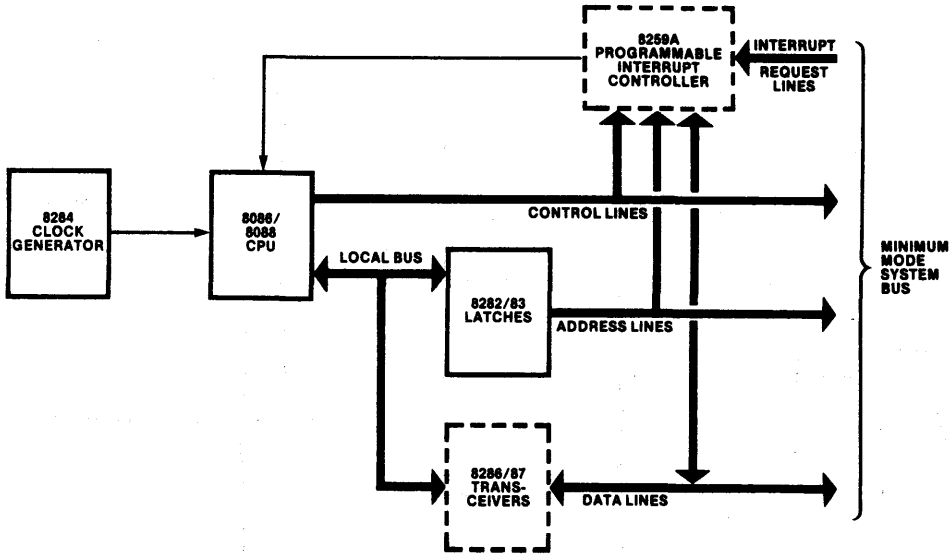


Figure 1-3. Minimum Mode System Bus

When an 8086 or 8088 is configured in maximum mode and an 8288 is added to control the system bus, one or two 8089s may be directly connected to the CPU (figure 1-4). The processors all share the same latches, transceivers, clock and bus controller, via the local bus. Arbitration logic built into the 8086, 8088 and 8089 coordinates use of the local bus, and thus of the system bus. This bus configuration enables the powerful I/O handling capabilities of the 8089 to be incorporated into systems of moderate size and cost.

The 8289 enables high-performance systems to be designed as a series of independent processing modules whose activities are coordinated via a shared system bus. Figure 1-5 shows the multimaster system bus interface; this bus structure is electrically compatible with the Multibus™ architecture used in Intel iSBC™ single-board computing systems.

Several different combinations of processors may be attached to the local bus of a multimaster computing module:

- a single 8086 or 8088
- a single 8089
- two 8089s
- an 8086 or 8088 and one 8089
- an 8086 or 8088 and two 8089s

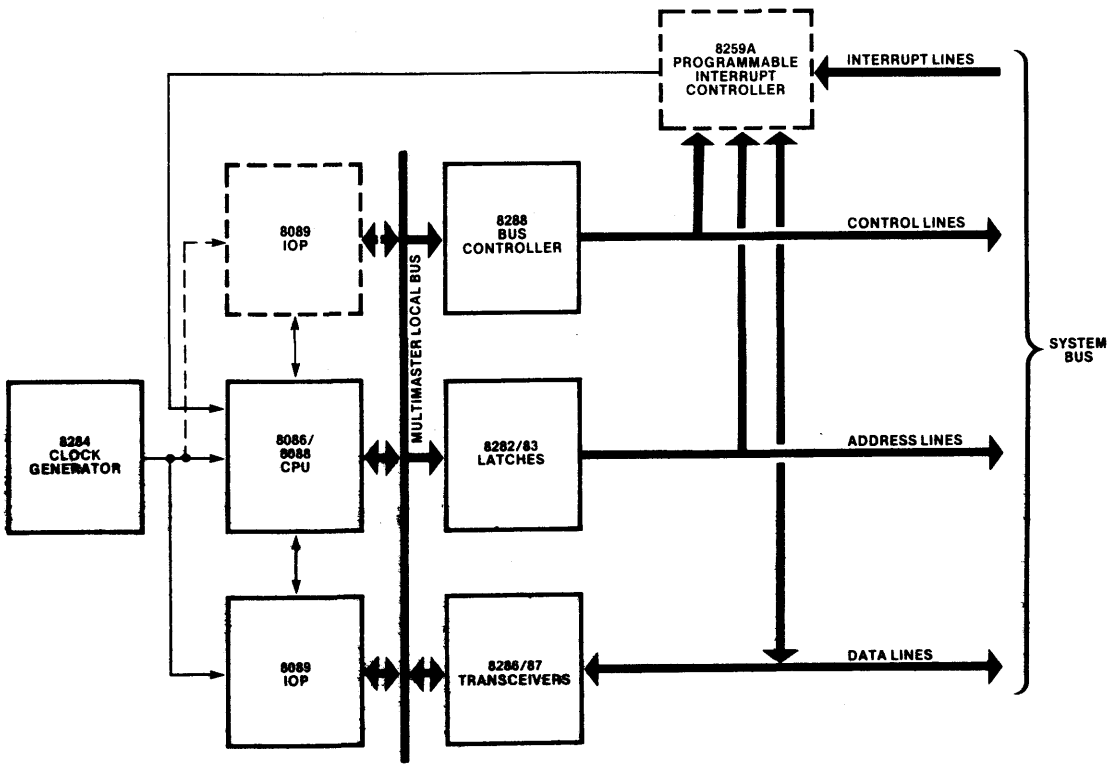


Figure 1-4. Multimaster Local Bus

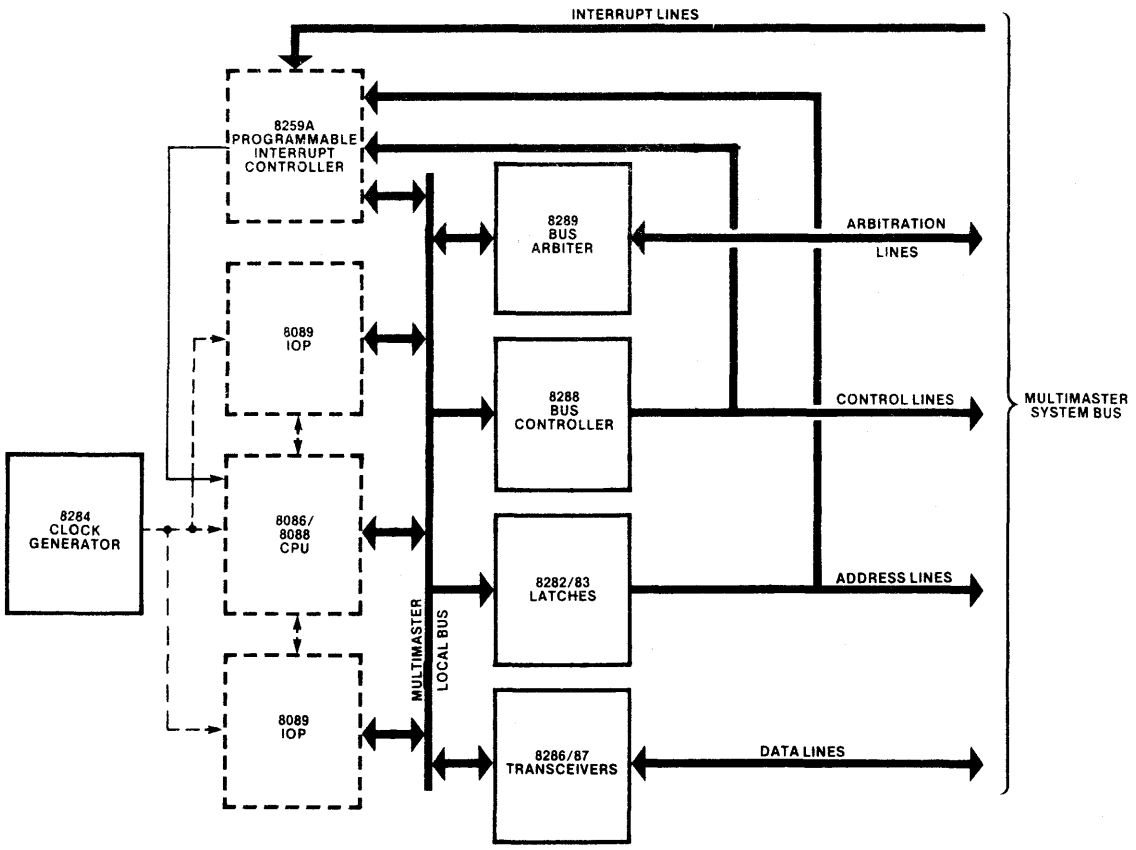


Figure 1-5. Basic Multimaster Processing Module

All of the processors on the local bus obtain access to the system bus through a single set of interface components.

One or two 8089s in a multimaster processing module may be configured with a private I/O bus as shown in figure 1-6. In this configuration, memory access commands are directed to the public multimaster system bus, while I/O commands use the private I/O bus. Memory, containing the 8089's programs, as well as I/O devices, may be connected to the private I/O bus. Taking this approach can greatly reduce the 8089's use of the system bus as most memory and I/O accesses can be made to the private address space. The system bus is thus made available for use by other processors, and the 8089 can execute in parallel with other processors for extended periods. A limited private I/O bus may be implemented using the 8-bit multiplexed peripherals of the 8085 family, eliminating the latches and transceivers shown in figure 1-6.

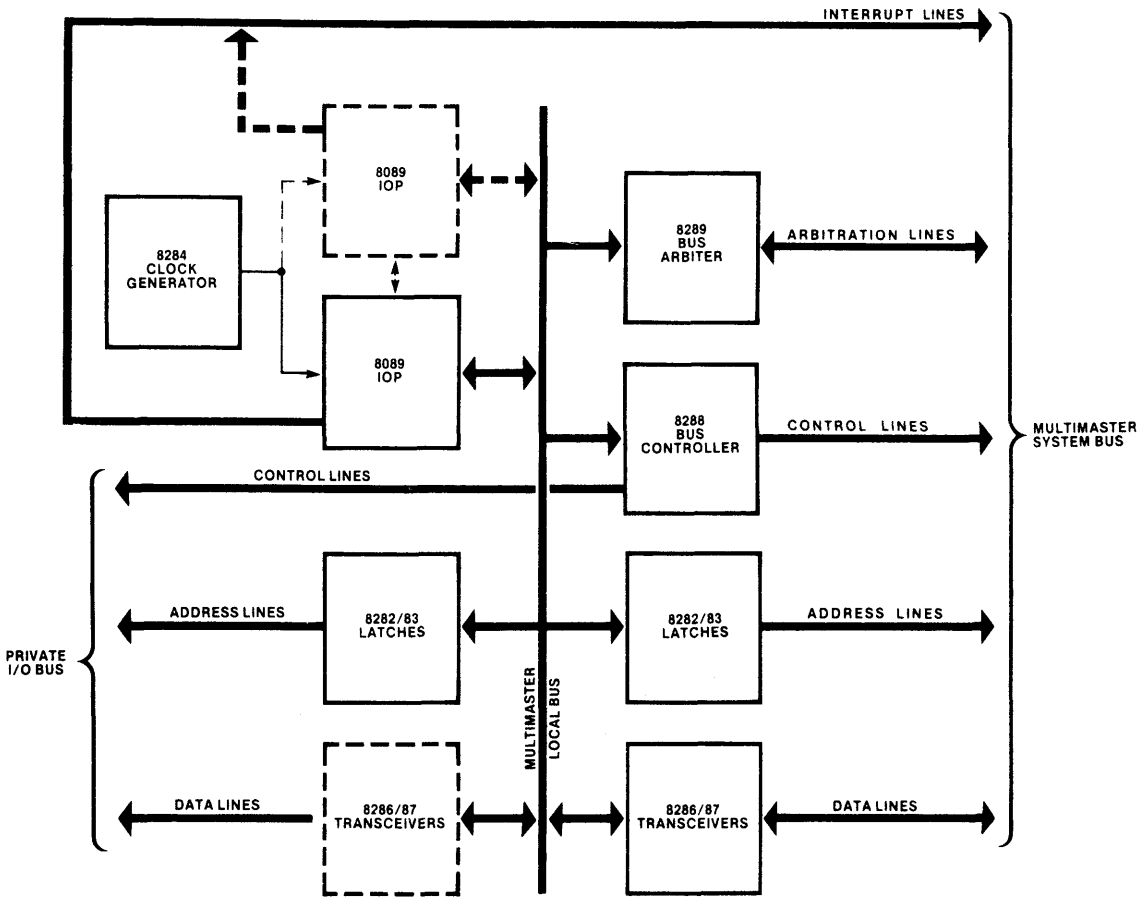


Figure 1-6. Private I/O Bus

Adding a second 8288 to the local bus allows an 8086 or 8088 in a processing module to divide its address space into system and resident sections (figure 1-7). A PROM or decoder is used to direct an address reference to the system bus or to the resident bus. The resident bus allows the CPU to run out of its own address space to minimize its use of the system bus. Since no other processors can access the private memory on the CPU's resident bus, operating system code and data in this space is protected from errors in other processor programs. If a second 8289 is added to a resident bus module, the resident bus becomes a second multimaster system bus.

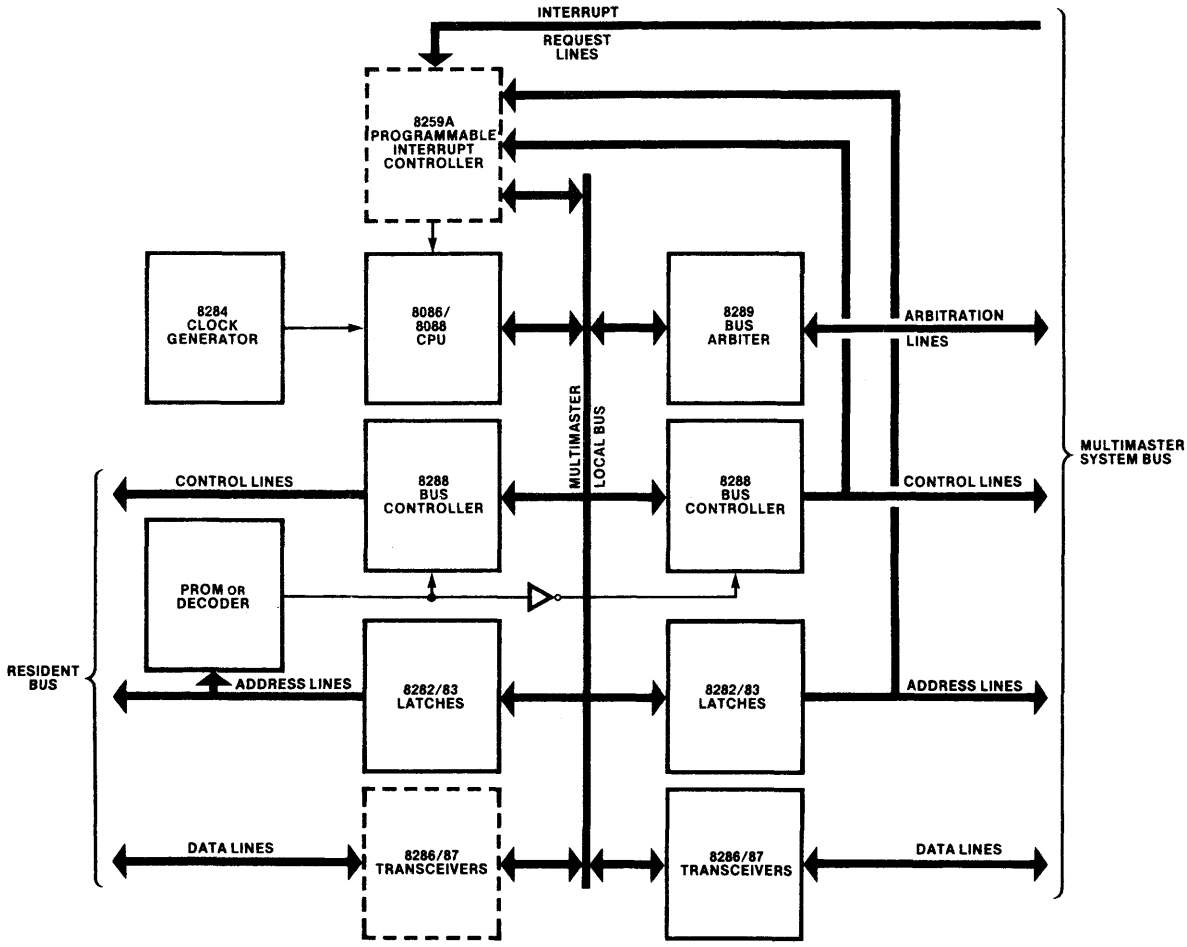


Figure 1-7. Resident Bus

As an alternative to the resident bus, a private read-only memory space can be implemented using the RD (read) signal provided by the CPUs in lieu of an 8288 Bus Controller.

Multiprocessing systems of widely varying complexity can be constructed from multimaster processing modules. Each module can be designed and implemented separately and can be optimized to perform a given task. The modules can communicate with each other by means of interrupts and messages placed in system memory. Additional functions can be added to a system by incorporating the new functions into modules and connecting the modules to the system bus.

Figure 1-8 illustrates a hypothetical system in which nine processors are distributed among five multimaster processing modules. (For clarity, bus interface components are not shown in figure 1-8.) A supervisor module controls the system, primarily responding to interrupts and dispatching other modules to perform tasks. The supervisor CPU, like the other processors in the system, executes code from private memory that is inaccessible to other modules. System memory, which is accessible to all the processors, is used only for messages, common buffers, etc. This helps to "protect" the processors from each other and to keep system bus contention at a minimum. The database module is responsible for maintaining all system files. Each of the three graphics modules supports a graphics CRT terminal. An 8089 in each module performs data transfers and CRT refresh and calls upon an 8088 for intensive computational routines.

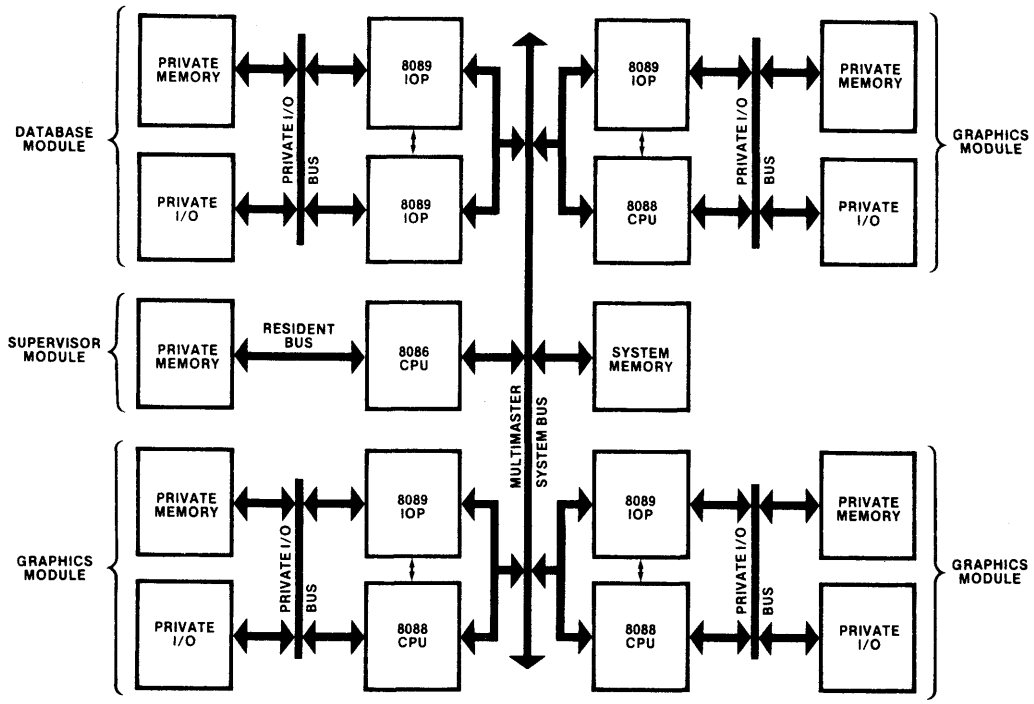


Figure 1-8. Multimaster Design Example

1.3 Development Aids

Intel provides the sophisticated tools needed for timely and economical development of products based on the 8086 family. The 8086 family system development environment is focused on the Intellec® Series II Microcomputer Development System (figure 1-9). The Intellec system is a multiple-microprocessor system that runs ISIS-II, a disk-based operating system that has been proven in thousands of installations. The Intellec has built-in interfaces for a printer, a PROM programmer and a paper tape reader/punch. This same hardware and operating system may be used to develop systems based on other Intel microprocessor families such as the 8085 and the 8048.

Three language translators support 8086 family programming. PL/M-86 is a high-level language for the 8086 and 8088 that supports structured programming techniques. It is upward-compatible with PL/M-80, the most widely used high-level microprocessor language. ASM-86 may be used to write assembly language programs for the 8086 and the 8088 CPUs and gives the programmer access to the full power of these CPUs. 8089 programs are written in ASM-89, the 8089 assembly language.

The language translators produce compatible outputs that can be manipulated by the software development utilities. LINK-86, for example, can combine programs written in ASM-86 with PL/M-86 programs. LIB-86 allows related programs to be stored in libraries to simplify storage and retrival. LOC-86 assigns absolute memory addresses to programs. OH-86 changes the format of an executable program for PROM programming or for loading into the RAM of a test vehicle.

The UPP-301 Universal PROM Programmer can burn programs into any of Intel's PROM memories; the UPP plugs into the Intellec® system and allows program data to be manipulated from the console before it is programmed into the PROM.

The SDK-86 is an (minimum mode) 8086-based prototyping and evaluation kit. It includes the CPU, RAM, I/O ports and a breadboard area for interfacing customer circuits. A ROM-based monitor program is supplied with the kit. Monitor commands may be entered from an on-board keypad or from a terminal; the monitor returns results to the SDK-86's on-board LED display or to a terminal. Monitor commands allow programs to be entered, run, stopped, and single-stepped; memory contents can be altered as well as displayed. The SDK-C86 Software and Cable Interface connects an SDK-86 to an Intellec® system. The software supplied with the cable enables programs to be transferred between the development system and the SDK-86 to allow users to develop programs using the text editor, translators and utilities of the Intellec system and then download the program to the SDK-86 for execution.

The iSBC 86/12™ board is a high-performance single board computer based on a maximum mode 8086 CPU. The board contains 32k of dual-port RAM that is accessible to the CPU via the on-board bus and to other processors via the built-in Multibus™ interface. The board also has an asynchronous serial port, parallel ports with sockets for drivers and terminators, two timers and sockets for 16k of ROM.

An iSBC 86/12™ can be linked to an Intellec® system using the iSBC 957™ Intellec-iSBC 86/12 Interface and Execution Package. The package includes a ROM-based monitor for the iSBC 86/12 board, software for the Intellec system and cabling to connect the two. The package supports data transfers between Intellec diskettes and iSBC 86/12 memory, full speed execution of customer programs on the iSBC 86/12 board, breakpoints, single-stepping, and data moves, replacements, searches and compares. All commands are entered from the Intellec console.

The ICE-86™ module is an in-circuit emulator for the 8086 microprocessor. A 40-pin probe replaces the 8086 in the system under test. This probe is connected to ICE-86 circuit boards that in turn plug into the Intellec® chassis. The ICE-86 module emulates the 8086 in the system under test in response to commands entered through the Intellec console. These commands allow the user to debug the system by setting breakpoints, tracing the flow of execution, single-stepping, examining and altering memory and I/O, etc. All references to program variables and labels are symbolic (i.e., their PL/M-86 or ASM-86 names). Software testing can also map "system under test" memory into the Intellec memory to permit software testing to begin before prototype hardware has been developed.

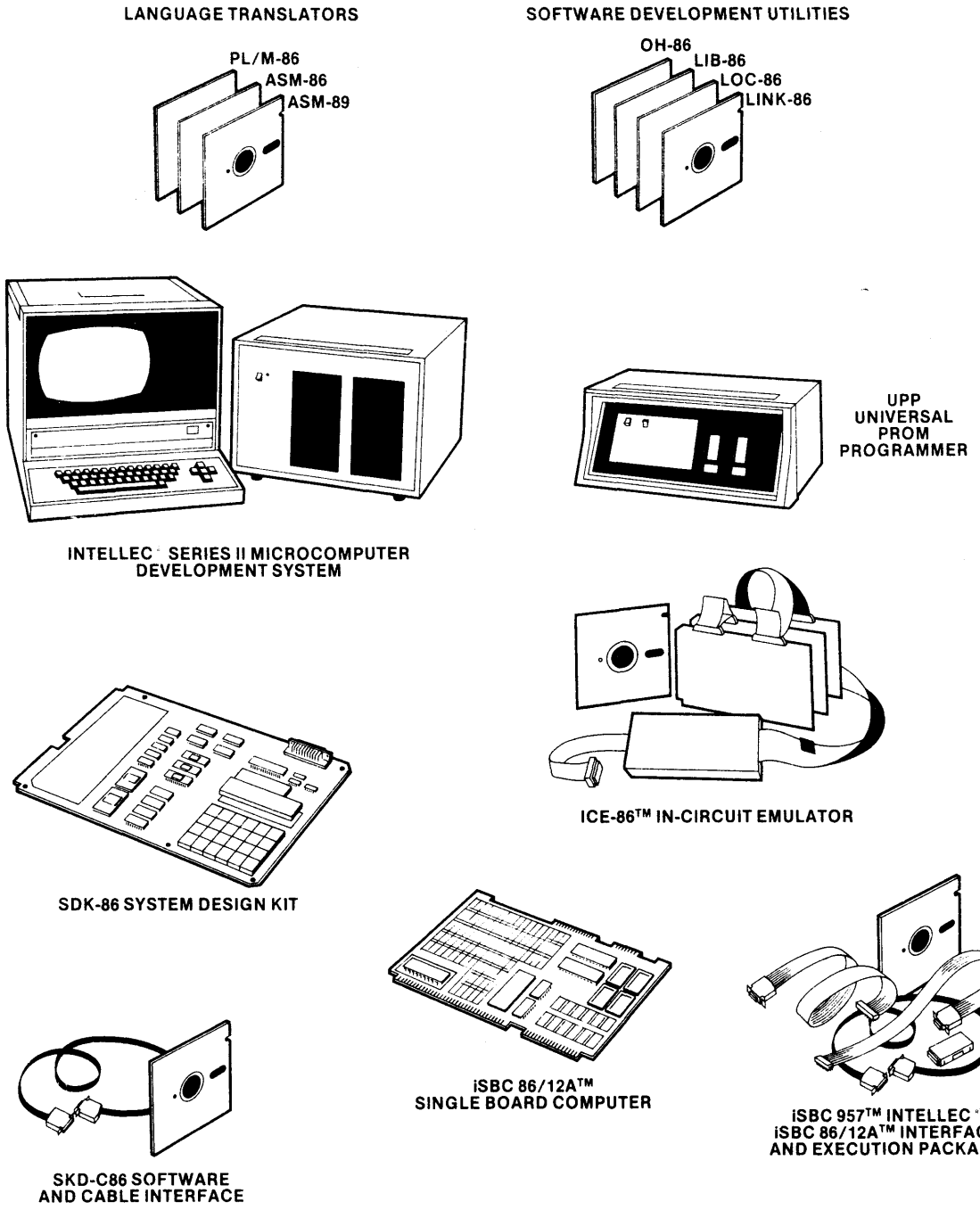
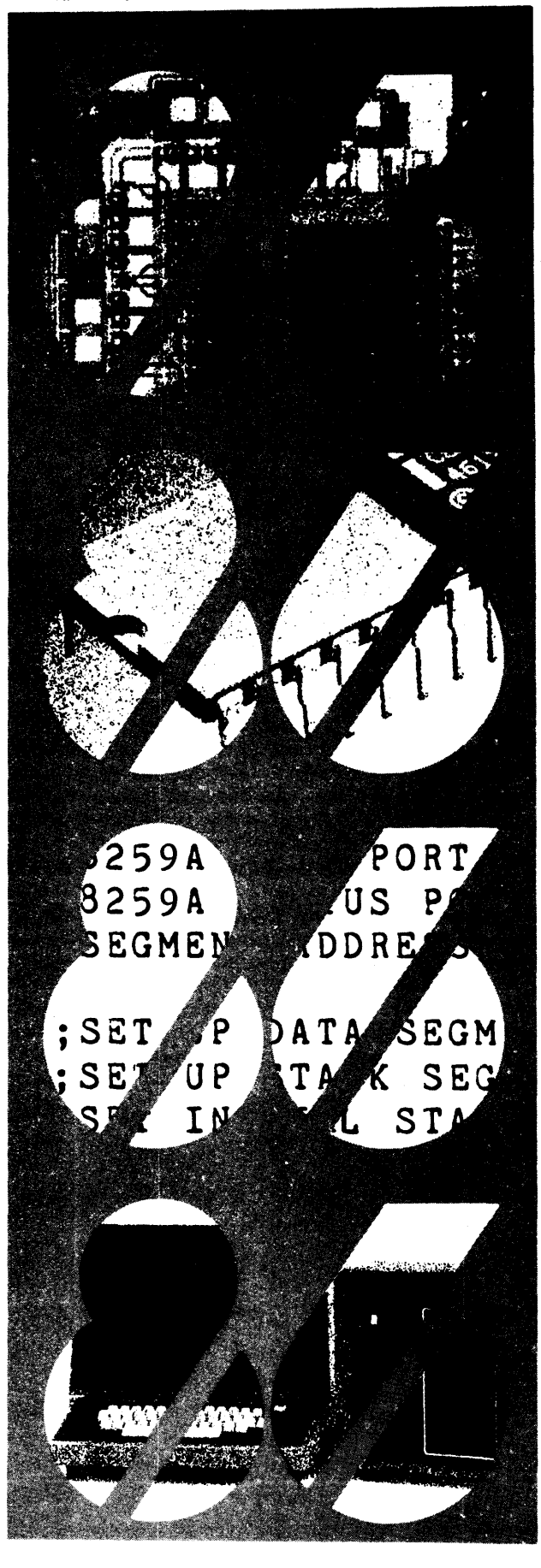


Figure 1-9. 8086 Family Development Aids

Chapter 2
The 8086 and 8088 Central Processing Units



CHAPTER 2
THE 8086 AND 8088 CENTRAL PROCESSING UNITS

This chapter describes the mainstays of the 8086 microprocessor family: the 8086 and 8088 central processing units (CPUs). The material is divided into ten sections and generally proceeds from hardware to software topics as follows:

1. Processor Overview
2. Processor Architecture
3. Memory
4. Input/Output
5. Multiprocessing Features
6. Processor Control and Monitoring
7. Instruction Set
8. Addressing Modes
9. Programming Facilities
10. Programming Guidelines and Examples

The chapter describes the internal operation of the CPUs in detail. The interaction of the processors with other devices is discussed in functional terms; electrical characteristics, timing, and other information needed to actually interface other devices with the 8086 and 8088 are provided in Chapter 4.

2.1 Processor Overview

The 8086 and 8088 are closely related third-generation microprocessors. The 8088 is designed with an 8-bit external data path to memory and I/O, while the 8086 can transfer 16 bits at a time. In almost every other respect the processors are identical; software written for one CPU will execute on the other without alteration. The chips are contained in standard 40-pin dual in-line packages (figure 2-1) and operate from a single +5V power source.

The 8086 and 8088 are suitable for an exceptionally wide spectrum of microcomputer applications, and this flexibility is one of their most outstanding characteristics. Systems can range from uniprocessor minimal-memory designs implemented with a handful of chips (figure 2-2), to multiprocessor systems with up to a megabyte of memory (figure 2-3).

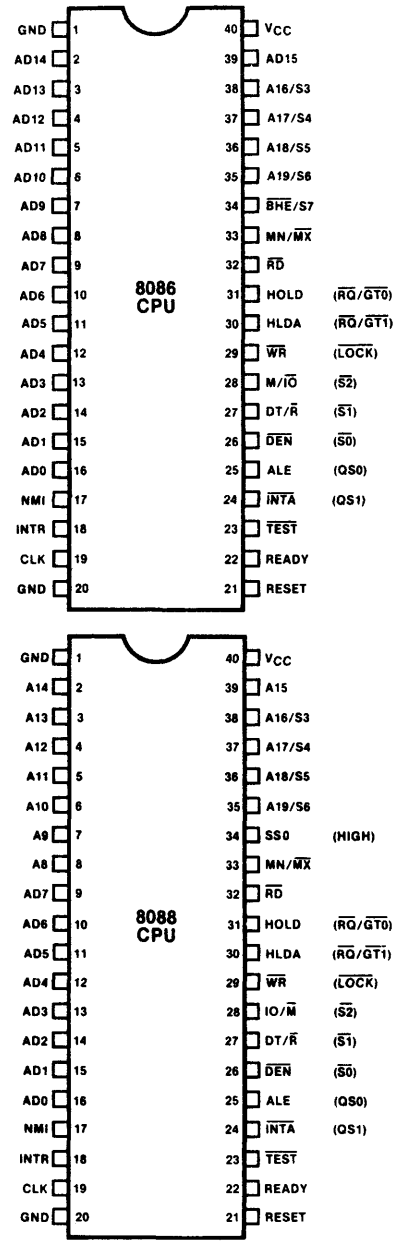


Figure 2-1. 8086 and 8088 Central Processing Units

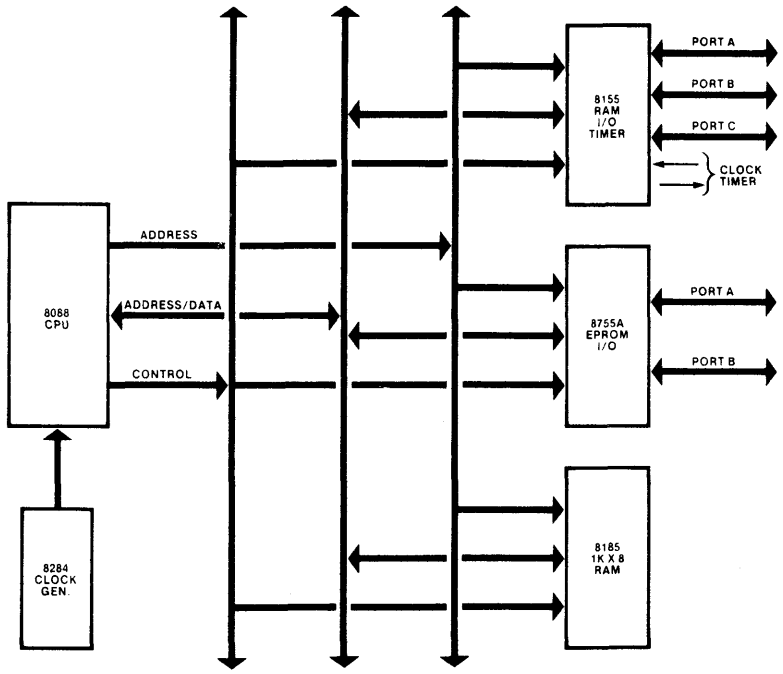


Figure 2-2. Small 8088-Based System

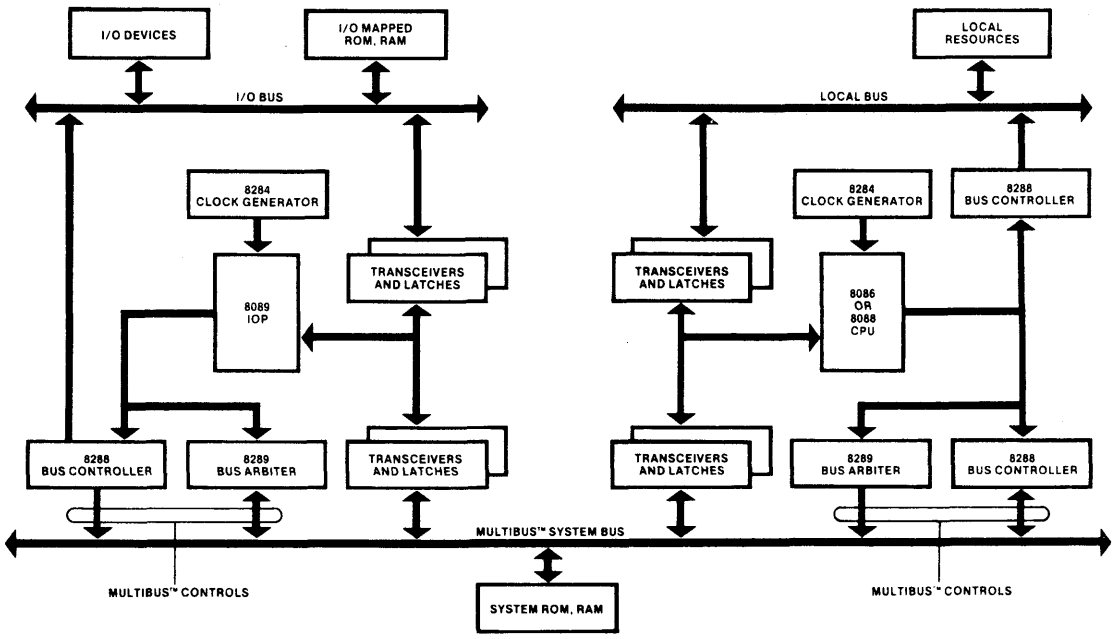


Figure 2-3. 8086/8088/8089 Multiprocessing System

The large application domain of the 8086 and 8088 is made possible primarily by the processors' dual operating modes (minimum and maximum mode) and built-in multiprocessing features. Several of the 40 CPU pins have dual functions that are selected by a strapping pin. Configured in minimum mode, these pins transfer control signals directly to memory and input/output devices. In maximum mode these same pins take on different functions that are helpful in medium to large ystems, especially systems with multiple processors. The control functions assigned to these pins in minimum mode are assumed by a support chip, the 8288 Bus Controller.

The CPUs are designed to operate with the 8089 Input/Output Processor (IOP) and other processors in multiprocessing and distributed processing systems. When used in conjunction with one or more 8089s, the 8086 and 8088 expand the applicability of microprocessors into I/O-intensive data processing systems. Built-in coordinating signals and instructions, and electrical compatibility with Intel's Multibus™ shared bus architecture, simplify and reduce the cost of developing multiple-processor designs.

Both CPUs are substantially more powerful than any microprocessor previously offered by Intel. Actual performance, of course, varies from application to application, but comparisons to the industry standard 2-MHz 8080A are instructive. The 8088 is from four to six times more powerful than the 8080A; the 8086 provides seven to ten times the 8080A's performance (see figure 2-4).

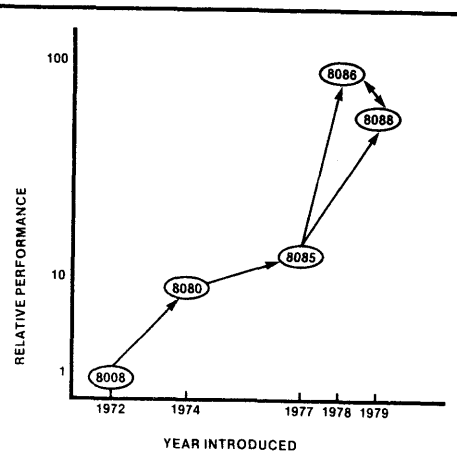


Figure 2-4. Relative Performance of the 8086 and 8088

The 8086's advantage over the 8088 is attributable to its 16-bit external data bus. In applications that manipulate 8-bit quantities extensively, or that are execution-bound, the 8088 can approach to within 10% of the 8086's processing throughput.

The high performance of the 8086 and 8088 is realized by combining a 16-bit internal data path with a pipelined architecture that allows instructions to be prefetched during spare bus cycles. Also contributing to performance is a compact instruction format that enables more instructions to be fetched in a given amount of time.

Software for high-performance 8086 and 8088 systems need not be written in assembly language. The CPUs are designed to provide direct hardware support for programs written in high-level languages such as Intel's PL/M-86. Most high-level languages store variables in memory; the 8086/8088 symmetrical instruction set supports direct operation on memory operands, including operands on the stack. The hardware addressing modes provide efficient, straightforward implementations of based variables, arrays, arrays of structures and other high-level language data constructs. A powerful set of memory-to-memory string operations is available for efficient character data manipulation. Finally, routines with critical performance requirements that cannot be met with PL/M-86 may be written in ASM-86 (the 8086/8088 assembly language) and linked with PL/M-86 code.

While the 8086 and 8088 are totally new designs, they make the most of users' existing investments in systems designed around the 8080/8085 microprocessors. Many of the standard Intel memory, peripheral control and communication chips are compatible with the 8086 and the 8088. Software is developed in the familiar Intellec® Microcomputer Development System environment, and most existing programs, whether written in ASM-80 or PL/M-80, can be directly converted to run on the 8086 and 8088.

2.2 Processor Architecture

Microprocessors generally execute a program by repeatedly cycling through the steps shown below (this description is somewhat simplified):

1. Fetch the next instruction from memory.
2. Read an operand (if required by the instruction).

3. Execute the instruction.
4. Write the result (if required by the instruction).

In previous CPUs, most of these steps have been performed serially, or with only a single bus cycle fetch overlap. The architecture of the 8086 and 8088 CPUs, while performing the same steps, allocates them to two separate processing units within the CPU. The execution unit (EU) executes instructions; the bus interface unit (BIU) fetches instructions, reads operands and writes results. The two units can operate independently of one another and are able, under most circumstances, to extensively overlap instruction fetch with execution. The result is that, in most cases, the time normally required to fetch instructions "disappears" because the EU executes instructions that have already been fetched by the BIU. Figure 2-5 illustrates this overlap and compares it with traditional microprocessor operation. In the example, overlapping reduces the elapsed time required to execute three instructions, and allows two additional instructions to be prefetched as well.

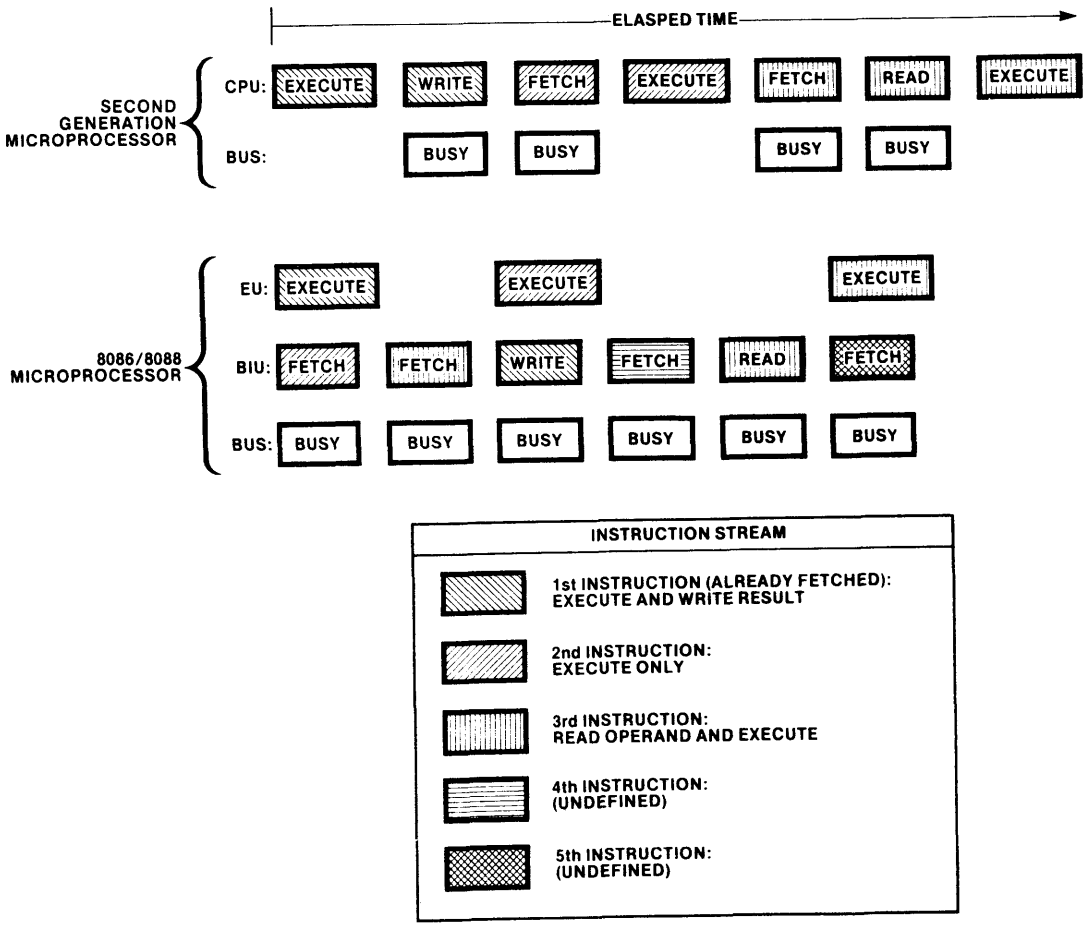


Figure 2-5. Overlapped Instruction Fetch and Execution

Execution Unit

The execution units of the 8086 and 8088 are identical (figure 2-6). A 16-bit arithmetic/logic unit (ALU) in the EU maintains the CPU status and control flags, and manipulates the general registers and instruction operands. All registers and data paths in the EU are 16 bits wide for fast internal transfers.

The EU has no connection to the system bus, the "outside world." It obtains instructions from a queue maintained by the BIU. Likewise, when an instruction requires access to memory or to a peripheral device, the EU requests the BIU to obtain or store the data. All addresses manipulated by the EU are 16 bits wide. The BIU, however, performs an address relocation that gives the EU access to the full megabyte of memory space (see section 2.3).

Bus Interface Unit

The BIUs of the 8086 and 8088 are functionally identical, but are implemented differently to match the structure and performance characteristics of their respective buses.

The BIU performs all bus operations for the EU. Data is transferred between the CPU and memory or I/O devices upon demand from the EU. Sections 2.3 and 2.4 describe the interaction of the BIU with memory and I/O devices.

In addition, during periods when the EU is busy executing instructions, the BIU "looks ahead" and fetches more instructions from memory. The instructions are stored in an internal RAM array called the instruction stream queue. The 8088 instruction queue holds up to four bytes of the instruction stream, while the 8086 queue can store

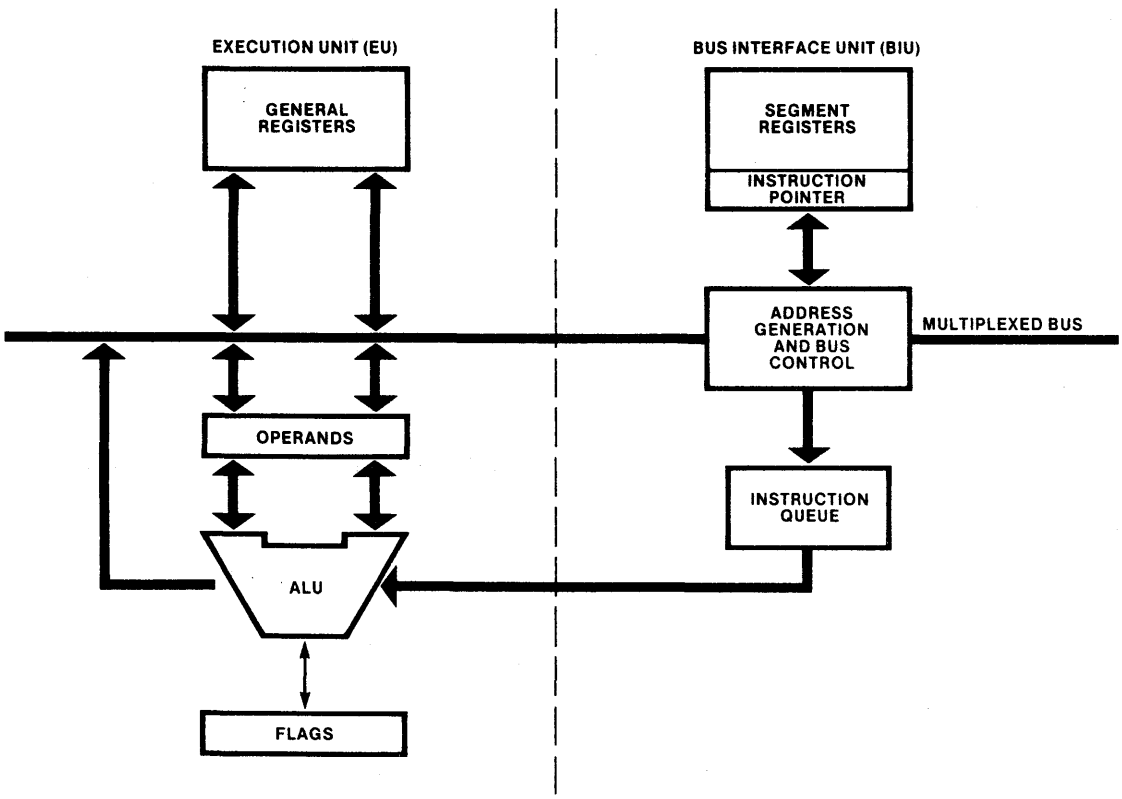


Figure 2-6. Execution and Bus Interface Units (EU and BIU)

up to six instruction bytes. These queue sizes allow the BIU to keep the EU supplied with prefetched instructions under most conditions without monopolizing the system bus. The 8088 BIU fetches another instruction byte whenever one byte in its queue is empty and there is no active request for bus access from the EU. The 8086 BIU operates similarly except that it does not initiate a fetch until there are two empty bytes in its queue. The 8086 BIU normally obtains two instruction bytes per fetch; if a program transfer forces fetching from an odd address, the 8086 BIU automatically reads one byte from the odd address and then resumes fetching two-byte words from the subsequent even addresses.

Under most circumstances the queues contain at least one byte of the instruction stream and the EU does not have to wait for instructions to be fetched. The instructions in the queue are those stored in the memory locations immediately adjacent to and higher than the instruction currently being executed. That is, they are the next logical instructions so long as execution proceeds serially. If the EU executes an instruction that transfers control to another location, the BIU resets the queue, fetches the instruction from the new address, passes it immediately to the EU, and then begins refilling the queue from the new location. In addition, the BIU suspends instruction fetching whenever the EU requests a memory or I/O read or write (except that a fetch already in progress is completed before executing the EU's bus request).

General Registers

Both CPUs have the same complement of eight 16-bit general registers (figure 2-7). The general registers are subdivided into two sets of four registers each: the data registers (sometimes called the H & L group for "high" and "low"), and the pointer and index registers (sometimes called the P & I group).

The data registers are unique in that their upper (high) and lower halves are separately addressable. This means that each data register can be used interchangeably as a 16-bit register, or as two 8-bit registers. The other CPU registers always are accessed as 16-bit units only. The data registers can be used without constraint in most arithmetic and logic operations. In addition,

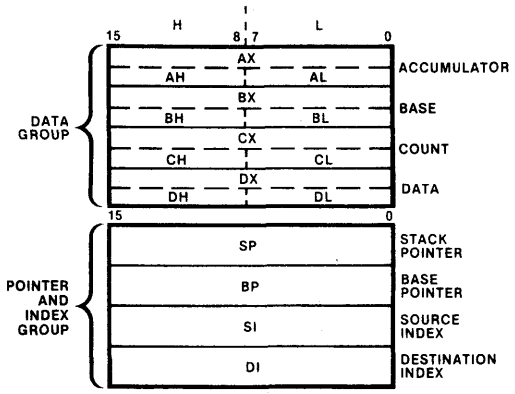


Figure 2-7. General Registers

some instructions use certain registers implicitly (see table 2-1) thus allowing compact yet powerful encoding.

Table 2-1. Implicit Use of General Registers

REGISTER	OPERATIONS
AX	Word Multiply, Word Divide, Word I/O
AL	Byte Multiply, Byte Divide, Byte I/O, Translate, Decimal Arithmetic
AH	Byte Multiply, Byte Divide
BX	Translate
CX	String Operations, Loops
CL	Variable Shift and Rotate
DX	Word Multiply, Word Divide, Indirect I/O
SP	Stack Operations
SI	String Operations
DI	String Operations

The pointer and index registers can also participate in most arithmetic and logic operations. In fact, all eight general registers fit the definition of "accumulator" as used in first and second generation microprocessors. The P & I registers (except for BP) also are used implicitly in some instructions as shown in table 2-1.

Segment Registers

The megabyte of 8086 and 8088 memory space is divided into logical segments of up to 64k bytes each. (Memory segmentation is described in section 2.3.) The CPU has direct access to four segments at a time; their base addresses (starting locations) are contained in the segment registers (see figure 2-8). The CS register points to the current code segment; instructions are fetched from this segment. The SS register points to the current stack segment; stack operations are performed on locations in this segment. The DS register points to the current data segment; it generally contains program variables. The ES register points to the current extra segment, which also is typically used for data storage.

The segment registers are accessible to programs and can be manipulated with several instructions. Good programming practice and consideration of compatibility with future Intel hardware and software products dictate that the segment registers be used in a disciplined fashion. Section 2.10 provides guidelines for segment register use.

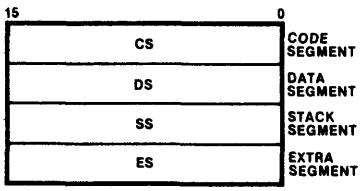


Figure 2-8. Segment Registers

Instruction Pointer

The 16-bit instruction pointer (IP) is analogous to the program counter (PC) in the 8080/8085 CPUs. The instruction pointer is updated by the BIU so that it contains the offset (distance in bytes) of the next instruction from the beginning of the current code segment; i.e., IP points to the next instruction. During normal execution, IP contains the offset of the next instruction to be *fetched* by the BIU; whenever IP is saved on the stack, however, it first is automatically adjusted to point to the next instruction to be *executed*. Programs do not have direct access to the instruction pointer, but instructions cause it to change and to be saved on and restored from the stack.

Flags

The 8086 and 8088 have six 1-bit status flags (figure 2-9) that the EU posts to reflect certain properties of the result of an arithmetic or logic

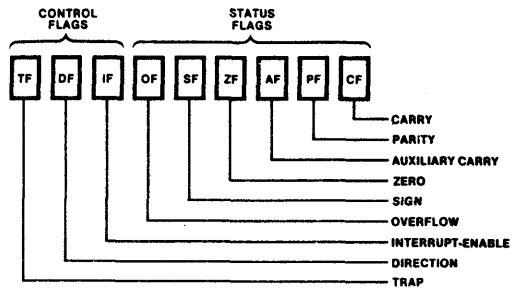


Figure 2-9. Flags

operation. A group of instructions is available that allows a program to alter its execution depending on the state of these flags, that is, on the result of a prior operation. Different instructions affect the status flags differently; in general, however, the flags reflect the following conditions:

1. If AF (the auxiliary carry flag) is set, there has been a carry out of the low nibble into the high nibble or a borrow from the high nibble into the low nibble of an 8-bit quantity (low-order byte of a 16-bit quantity). This flag is used by decimal arithmetic instructions.
2. If CF (the carry flag) is set, there has been a carry out of, or a borrow into, the high-order bit of the result (8- or 16-bit). The flag is used by instructions that add and subtract multibyte numbers. Rotate instructions can also isolate a bit in memory or a register by placing it in the carry flag.
3. If OF (the overflow flag) is set, an arithmetic overflow has occurred; that is, a significant digit has been lost because the size of the result exceeded the capacity of its destination location. An Interrupt On Overflow instruction is available that will generate an interrupt in this situation.

4. If SF (the sign flag) is set, the high-order bit of the result is a 1. Since negative binary numbers are represented in the 8086 and 8088 in standard two's complement notation, SF indicates the sign of the result (0 = positive, 1 = negative).
5. If PF (the parity flag) is set, the result has even parity, an even number of 1-bits. This flag can be used to check for data transmission errors.
6. If ZF (the zero flag) is set, the result of the operation is 0.

Three additional control flags (figure 2-9) can be set and cleared by programs to alter processor operations:

1. Setting DF (the direction flag) causes string instructions to auto-decrement; that is, to process strings from high addresses to low addresses, or from "right to left." Clearing DF causes string instructions to auto-increment, or to process strings from "left to right."
2. Setting IF (the interrupt-enable flag) allows the CPU to recognize external (maskable) interrupt requests. Clearing IF disables these interrupts. IF has no affect on either non-maskable external or internally generated interrupts.
3. Setting TF (the trap flag) puts the processor into single-step mode for debugging. In this mode, the CPU automatically generates an internal interrupt after each instruction, allowing a program to be inspected as it executes instruction by instruction. Section 2.10 contains an example showing the use of TF in a single-step and breakpoint routine.

8080/8085 Registers and Flag Correspondence

The registers, flags and program counter in the 8080/8085 CPUs all have counterparts in the 8086 and 8088 (see figure 2-10). The A register (accumulator) in the 8080/8085 corresponds to the AL register in the 8086 and 8088. The 8080/8085 H & L, B & C, and D & E registers correspond to registers BH, BL, CH, CL, DH and DL, respectively, in the 8086 and 8088. The 8080/8085 SP (stack pointer) and PC (program counter) have their counterparts in the 8086/8088 SP and IP. The AF, CF, PF, SF, and ZF flags are the same in both CPU families. The remaining flags and registers are unique to the 8086 and 8088. This 8080/8085 to 8086 mapping allows most existing 8080/8085 program code to be directly translated into 8086/8088 code.

Mode Selection

Both processors have a strap pin (MN/$\overline{MX}$) that defines the function of eight CPU pins in the 8086 and nine pins in the 8088. Connecting MN/$\overline{MX}$ to +5V places the CPU in minimum mode. In this configuration, which is designed for small systems (roughly one or two boards), the CPU itself provides the bus control signals needed by memory and peripherals. When MN/$\overline{MX}$ is strapped to ground, the CPU is configured in maximum mode. In this configuration the CPU encodes control signals on three lines. An 8288 Bus Controller is added to decode the signals from the CPU and to provide an expanded set of control signals to the rest of the system. The CPU uses the remaining free lines for a new set of signals designed to help coordinate the activities of other processors in the system. Sections 2.5 and 2.6 describe the functions of these signals.

2.3 Memory

The 8086 and 8088 can accommodate up to 1,048,576 bytes of memory in both minimum and maximum mode. This section describes how memory is functionally organized and used. There are substantial differences in the way memory components are actually accessed by the two processors; these differences, which are invisible to programs, are covered in section 4.2, External Memory Addressing.

Storage Organization

From a storage point of view, the 8086 and 8088 memory spaces are organized as identical arrays of 8-bit bytes (see figure 2-11). Instructions, byte data and word data may be freely stored at any byte address without regard for alignment thereby saving memory space by allowing code to be densely packed in memory (see figure 2-12). Odd-addressed (unaligned) word variables, however,

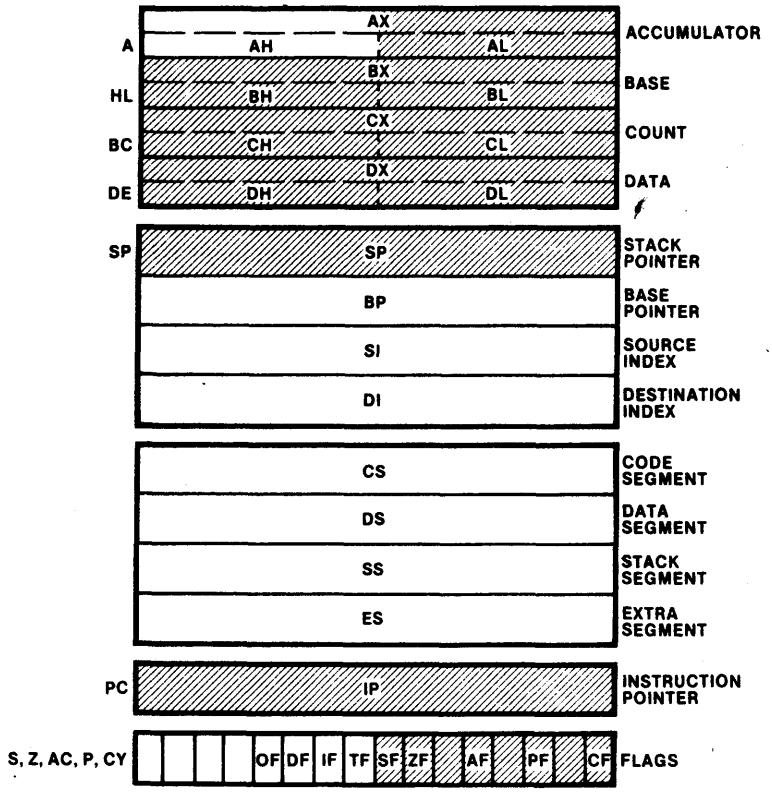


Figure 2-10. 8080/8085 Register Subset (Shaded)

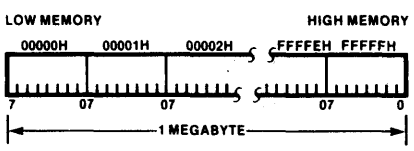


Figure 2-11. Storage Organization

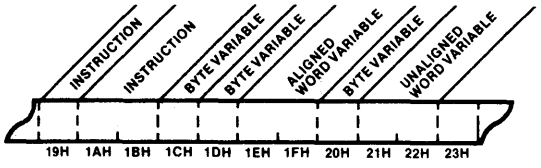
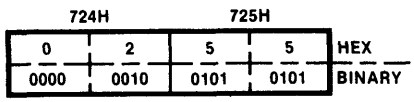


Figure 2-12. Instruction and Variable Storage

do not take advantage of the 8086's ability to transfer 16-bits at a time. Instruction alignment does not materially affect the performance of either processor.

Following Intel convention, word data always is stored with the most-significant byte in the higher memory location (see figure 2-13). Most of the time this storage convention is "invisible" to anyone working with the processors; exceptions may occur when monitoring the system bus or when reading memory dumps.

A special class of data is stored as doublewords; i.e., two consecutive words. These are called pointers and are used to address data and code that are outside the currently-addressable segments. The lower-addressed word of a pointer contains an offset value, and the higher-addressed word contains a segment base address. Each word is stored conventionally with the higher-addressed byte containing the most-significant eight bits of the word (see figure 2-14).



VALUE OF WORD STORED AT 724H: 5502H

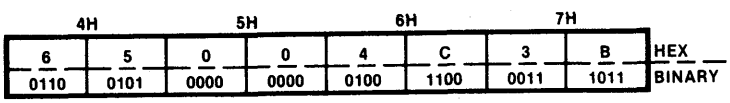
Figure 2-13. Storage of Word Variables

Segmentation

8086 and 8088 programs "view" the megabyte of memory space as a group of segments that are defined by the application. A segment is a logical unit of memory that may be up to 64k bytes long. Each segment is made up of contiguous memory locations and is an independent, separately-addressable unit. Every segment is assigned (by software) a base address, which is its starting location in the memory space. All segments begin on 16-byte memory boundaries. There are no other restrictions on segment locations; segments may be adjacent, disjoint, partially overlapped, or fully overlapped (see figure 2-15). A physical memory location may be mapped into (contained in) one or more logical segments.

The segment registers point to (contain the base address values of) the four currently addressable segments (see figure 2-16). Programs obtain access to code and data in other segments by changing the segment registers to point to the desired segments.

Every application will define and use segments differently. The currently addressable segments provide a generous work space: 64k bytes for code, a 64k byte stack and 128k bytes of data storage. Many applications can be written to simply initialize the segment registers and then forget them. Larger applications should be designed with careful consideration given to segment definition.



VALUE OF POINTER STORED AT 4H:
SEGMENT BASE ADDRESS: 3B4CH
OFFSET: 65H

Figure 2-14. Storage of Pointer Variables

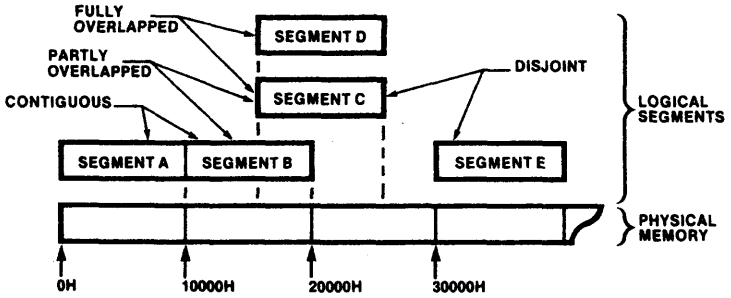


Figure 2-15. Segment Locations in Physical Memory

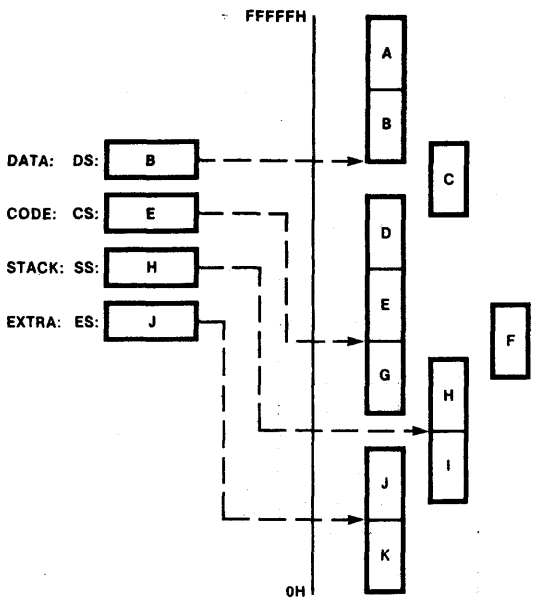


Figure 2-16. Currently Addressable Segments

The segmented structure of the 8086/8088 memory space supports modular software design by discouraging huge, monolithic programs. The segments also can be used to advantage in many programming situations. Take, for example, the case of an editor for several on-line terminals. A 64k text buffer (probably an extra segment) could be assigned to each terminal. A single program could maintain all the buffers by simply changing register ES to point to the buffer of the terminal requiring service.

Physical Address Generation

It is useful to think of every memory location as having two kinds of addresses, physical and logical. A physical address is the 20-bit value that uniquely identifies each byte location in the megabyte memory space. Physical addresses may range from 0H through FFFFFH. All exchanges between the CPU and memory components use this physical address.

Programs deal with logical, rather than physical addresses and allow code to be developed without prior knowledge of where the code is to be located in memory and facilitate dynamic management of memory resources. A logical address consists of a segment base value and an offset value. For any given memory location, the segment base value

locates the first byte of the containing segment and the offset value is the distance, in bytes, of the target location from the beginning of the segment. Segment base and offset values are unsigned 16-bit quantities; the lowest-addressed byte in a segment has an offset of 0. Many different logical addresses can map to the same physical location as shown in figure 2-17. In figure 2-17, physical memory location 2C3H is contained in two different overlapping segments, one beginning at 2B0H and the other at 2C0H.

Whenever the BIU accesses memory—to fetch an instruction or to obtain or store a variable—it generates a physical address from a logical address. This is done by shifting the segment base value four bit positions and adding the offset as illustrated in figure 2-18. Note that this addition process provides for modulo 64k addressing (addresses wrap around from the end of a segment to the beginning of the same segment).

The BIU obtains the logical address of a memory location from different sources depending on the type of reference that is being made (see table 2-2). Instructions always are fetched from the current code segment; IP contains the offset of the target instruction from the beginning of the segment. Stack instructions always operate on the current stack segment; SP contains the offset of the top of the stack. Most variables (memory operands) are assumed to reside in the current data segment, although a program can instruct the BIU to access a variable in one of the other currently addressable segments. The offset of a memory variable is calculated by the EU. This calculation is based on the addressing mode specified in the instruction; the result is called the operand's effective address (EA). Section 2.8 covers addressing modes and effective address calculation in detail.

Strings are addressed differently than other variables. The source operand of a string instruction is assumed to lie in the current data segment, but another currently addressable segment may be specified. Its offset is taken from register SI, the source index register. The destination operand of a string instruction always resides in the current

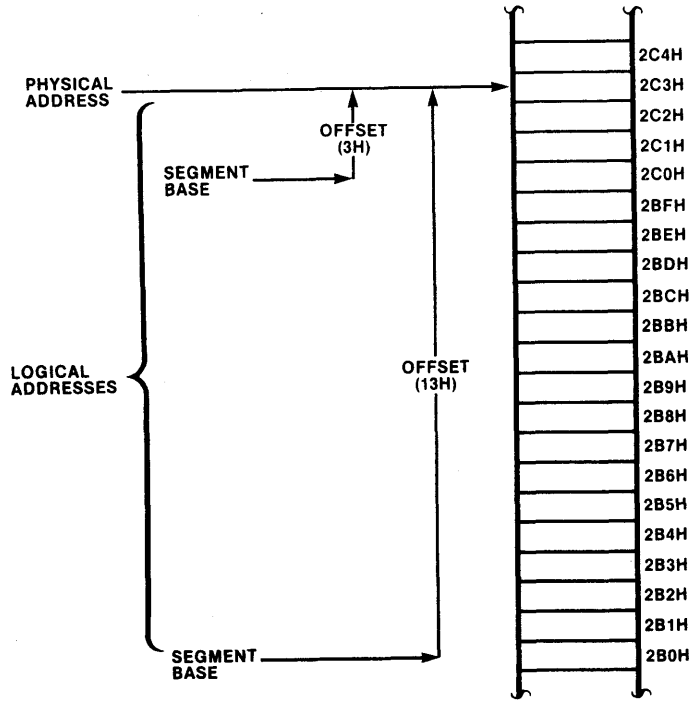


Figure 2-17. Logical and Physical Addresses

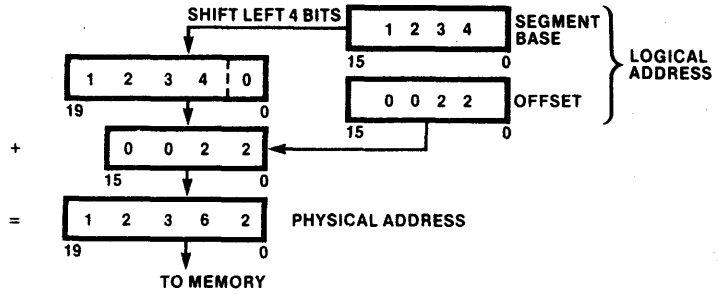


Figure 2-18. Physical Address Generation

Table 2-2. Logical Address Sources

TYPE OF MEMORY REFERENCE	DEFAULT SEGMENT BASE	ALTERNATE SEGMENT BASE	OFFSET
Instruction Fetch	CS	NONE	IP
Stack Operation	SS	NONE	SP
Variable (except following)	DS	CS,ES,SS	Effective Address
String Source	DS	CS,ES,SS	SI
String Destination	ES	NONE	DI
BP Used As Base Register	SS	CS,DS,ES	Effective Address

extra segment; its offset is taken from DI, the destination index register. The string instructions automatically adjust SI and DI as they process the strings one byte or word at a time.

When register BP, the base pointer register, is designated as a base register in an instruction, the variable is assumed to reside in the current stack segment. Register BP thus provides a convenient way to address data on the stack; BP can be used, however, to access data in any of the other currently addressable segments.

In most cases, the BIU's segment assumptions are a convenience to programmers. It is possible, however, for a programmer to explicitly direct the BIU to access a variable in any of the currently addressable segments (the only exception is the destination operand of a string instruction which must be in the extra segment). This is done by preceding an instruction with a segment override prefix. This one-byte machine instruction tells the BIU which segment register to use to access a variable referenced in the following instruction.

Dynamically Relocatable Code

The segmented memory structure of the 8086 and 8088 makes it possible to write programs that are position-independent, or dynamically relocatable. Dynamic relocation allows a multiprogramming or multitasking system to make particularly effective use of available memory. Inactive programs can be written to disk and the space they occupied allocated to other programs. If a disk-resident program is needed later, it can be read back into any available memory location and restarted. Similarly, if a program needs a large contiguous block of storage, and the total amount is available only in nonadjacent fragments, other program segments can be compacted to free up a continuous space. This process is shown graphically in figure 2-19.

In order to be dynamically relocatable, a program must not load or alter its segment registers and must not transfer directly to a location outside the current code segment. In other words, all offsets in the program must be relative to fixed values

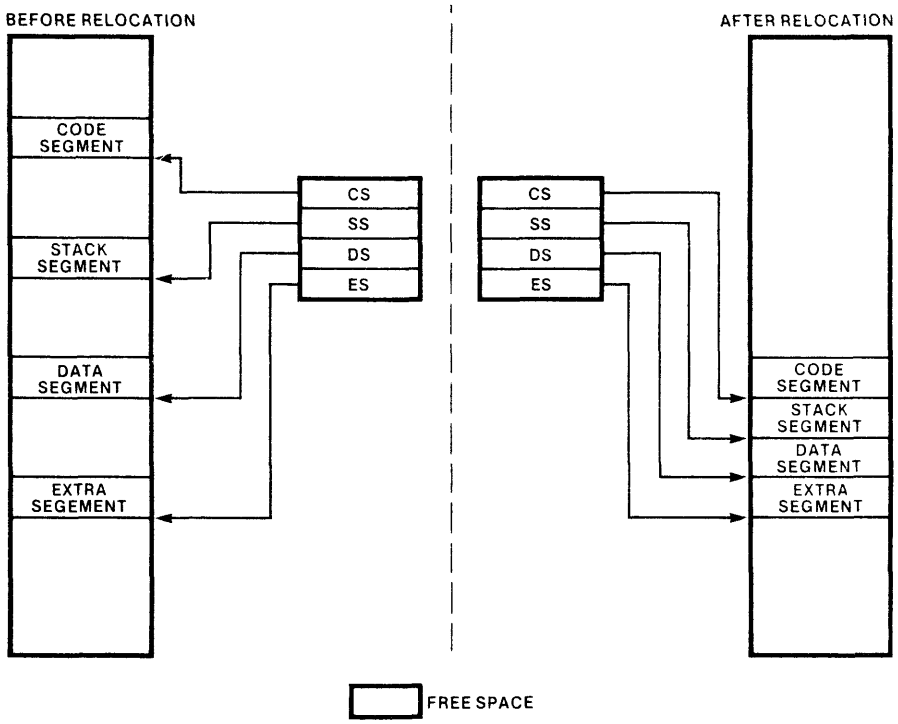


Figure 2-19. Dynamic Code Relocation

contained in the segment registers. This allows the program to be moved anywhere in memory as long as the segment registers are updated to point to the new base addresses. Section 2.10 contains an example that illustrates dynamic code relocation.

Stack Implementation

Stacks in the 8086 and 8088 are implemented in memory and are located by the stack segment register (SS) and the stack pointer register (SP). A system may have an unlimited number of stacks, and a stack may be up to 64k bytes long, the maximum length of a segment. (An attempt to expand a stack beyond 64k bytes overwrites the beginning of the stack.) One stack is directly addressable at a time; this is the current stack, often referred to simply as "the" stack. SS contains the base address of the current stack and SP points to the top of the stack (TOS). In other words, SP contains the offset of the top of the stack from the stack segment's base address. Note, however, that the stack's base address (contained in SS) is not the "bottom" of the stack.

8086 and 8088 stacks are 16 bits wide; instructions that operate on a stack add and remove stack items one word at a time. An item is pushed onto the stack (see figure 2-20) by *decrementing* SP by 2 and writing the item at the new TOS. An item is popped off the stack by copying it from TOS and then *incrementing* SP by 2. In other words, the stack grows *down* in memory toward its base address. Stack operations never move items on the stack, nor do they erase them. The top of the stack changes only as a result of updating the stack pointer.

Dedicated and Reserved Memory Locations

Two areas in extreme low and high memory are dedicated to specific processor functions or are reserved by Intel Corporation for use by Intel

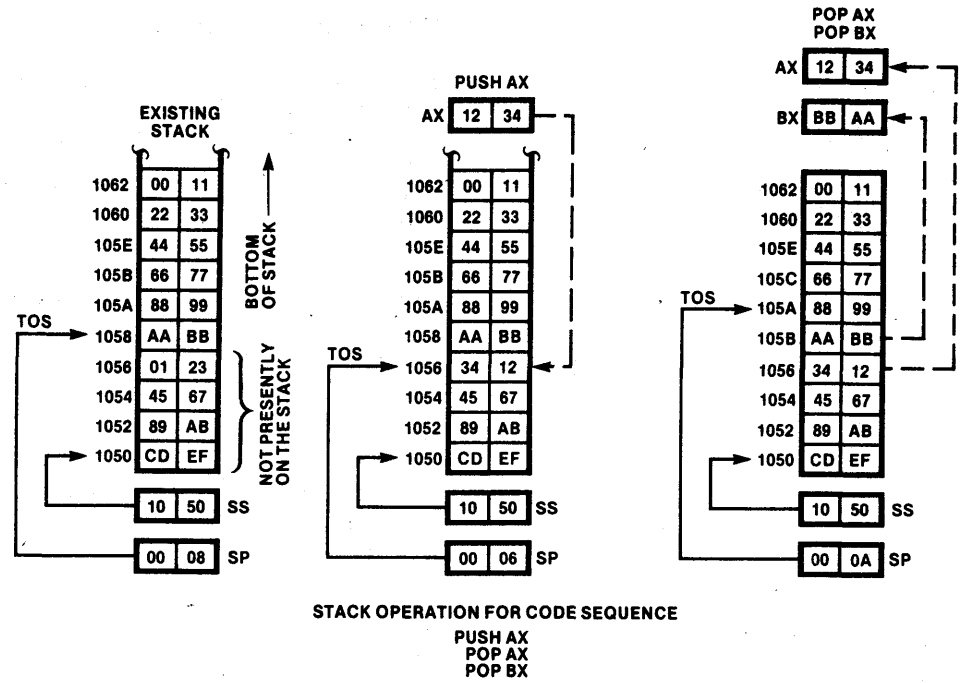


Figure 2-20. Stack Operation

hardware and software products. As shown in figure 2-21, the location are: 0H throgh 7FH (128 bytes) and FFFF0H through FFFFFH (16 bytes). These areas are used for interrupt and system reset processing 8086 and 8088 application systems should not use these areas for any other purpose. Doing so may make these systems incompatible with future Intel products.

8086/8088 Memory Access Differences

The 8086 can access either 8 or 16 bits of memory at a time. If an instruction refers to a word variable and that variable is located at an even-numbered address, the 8086 accesses the complete word in one bus cycle. If the word is located at an odd-numbered address, the 8086 accesses the word one byte at a time in two consecutive bus cycles.

To maximize throughput in 8086-based systems, 16-bit data should be stored at even addresses (should be word-aligned). This is particularly true of stacks. Unaligned stacks can slow a system's response to interrupts. Nevertheless, except for the performance penalty, word alignment is totally transparent to software. This allows maximum data packing where memory space is constrained.

The 8086 always fetches the instruction stream in words from even addresses except that the first fetch after a program transfer to an odd address obtains a byte. The instruction stream is disassembled inside the processor and instruction alignment will not materially affect the performance of most systems.

The 8088 always accesses memory in bytes. Word operands are accessed in two bus cycles regardless of their alignment. Instructions also are fetched one byte at a time. Although alignment of word operands does not affect the performance of the 8088, locating 16-bit data on even addresses will insure maximum throughput if the system is ever transferred to an 8086.

2.4 Input/Output

The 8086 and 8088 have a versatile set of input/output facilities. Both processors provide a large I/O space that is separate from the memory

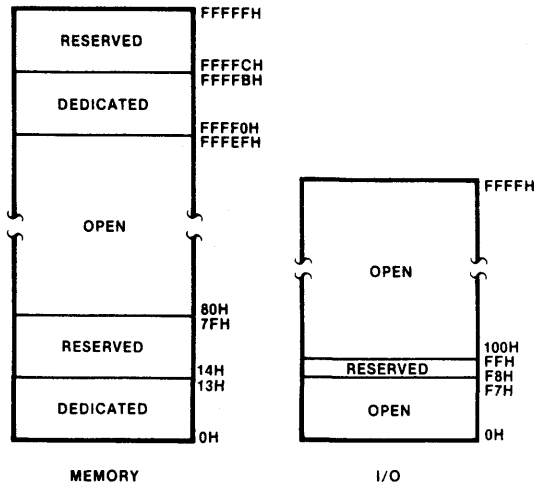


Figure 2-21. Reserved and Dedicated Memory and I/O Locations

space, and instructions that transfer data between the CPU and devices located in the I/O space. I/O devices also may be placed in the memory space to bring the power of the full instruction set and addressing modes to input/output processing. For high-speed transfers, the CPUs may be used with traditional direct memory access controllers or the 8089 Input/Output Processor.

Input/Output Space

The 8086/8088 I/O space can accommodate up to 64k 8-bit ports or up to 32k 16-bit ports. The IN and OUT (input and output) instructions transfer data between the accumulator (AL for byte transfers, AX for word transfers) and ports located in the I/O space.

The I/O space is not segmented; to access a port, the BIU simply places the port address (0-64k) on the lower 16 lines of the address bus. Different forms of the I/O instructions allow the address to be specified as a fixed value in the instruction or as a variable taken from register DX.

Restricted I/O Locations

Locations F8H through FFH (eight of the 64k locations) in the I/O space are reserved by Intel Corporation for use by future Intel hardware and software products. Using these locations for any other purpose may inhibit compatibility with future Intel products.

8086/8088 I/O Access Differences

The 8086 can transfer either 8 or 16 bits at a time to a device located in the I/O space. A 16-bit device should be located at an even address so that the word will be transferred in a single bus cycle. An 8-bit device may be located at either an even or odd address; however, the internal registers in a given device must be assigned all-even or all-odd addresses.

The 8088 transfers one byte per bus cycle. If a 16-bit device is used in the 8088 I/O space, it must be capable of transferring words in the same fashion, i.e., eight bits at a time in two bus cycles. (The 8089 Input/Output Processor can provide a straightforward interface between the 8088 and a 16-bit I/O device.) An 8-bit device may be located at odd or even addresses in the 8088 I/O space and internal registers may be assigned consecutive addresses (e.g., 1H, 2H, 3H). Assigning all-odd or all-even addresses to these registers, however, will simplify transferring the system to an 8086 CPU.

Memory-Mapped I/O

I/O devices also may be placed in the 8086/8088 memory space. As long as the devices respond like memory components, the CPU does not know the difference.

Memory-mapped I/O provides additional programming flexibility. Any instruction that references memory may be used to access an I/O port located in the memory space. For example, the MOV (move) instruction can transfer data between any 8086/8088 register and a port, or the AND, OR and TEST instructions may be used to manipulate bits in I/O device registers. In addition, memory-mapped I/O can take advantage of the 8086/8088 memory addressing modes. A group of terminals, for example, could be treated as an array in memory with an index register

selecting a terminal in the array. Section 2.10 provides examples of using the instruction set and addressing modes with memory-mapped I/O.

Of course, a price must be paid for the added programming flexibility that memory-mapped I/O provides. Dedicating part of the memory space to I/O devices reduces the number of addresses available for memory, although with a megabyte of memory space this should rarely be a constraint. Memory reference instructions also take longer to execute and are somewhat less compact than the simpler IN and OUT instructions.

Direct Memory Access

When configured in minimum mode, the 8086 and 8088 provide HOLD (hold) and HLDA (hold acknowledge) signals that are compatible with traditional DMA controllers such as the 8257 and 8237. A DMA controller can request use of the bus for direct transfer of data between an I/O device and memory by activating HOLD. The CPU will complete the current bus cycle, if one is in progress, and then issue HLDA, granting the bus to the DMA controller. The CPU will not attempt to use the bus until HOLD goes inactive.

The 8086 addresses memory that is physically organized in two separate banks, one containing even-addressed bytes and one containing odd-addressed bytes. An 8-bit DMA controller must alternately select these banks to access logically adjacent bytes in memory. The 8089 provides a simple way to interface a high-speed 8-bit device to an 8086-based system (see Chapter 3).

8089 Input/Output Processor (IOP)

The 8086 and 8088 are designed to be used with the 8089 in high-performance I/O applications. The 8089 conceptually resembles a microprocessor with two DMA channels and an instruction set specifically tailored for I/O operations. Unlike simple DMA controllers, the 8089 can service I/O devices directly, removing this task from the CPU. In addition, it can transfer data on its own bus or on the system bus, can match 8- or 16-bit peripherals to 8- or 16-bit buses, and can transfer data from memory to memory and from I/O device to I/O device. Chapter 3 describes the 8089 in detail.

2.5 Multiprocessing Features

As microprocessor prices have declined, multiprocessing (using two or more coordinated processors in a system) has become an increasingly attractive design alternative. Performance can be substantially improved by distributing system tasks among separate, concurrently executing processors. In addition, multiprocessing encourages a modular approach to design, usually resulting in systems that are more easily maintained and enhanced. For example, Figure 2-22 shows a multiprocessor system in which I/O activities have been delegated to an 8089 IOP. Should an I/O device in the system be changed (e.g., a hard disk substituted for a floppy), the impact of the modification is confined to the I/O subsystem and is transparent to the CPU and to the application software.

The 8086 and 8088 are designed for the multiprocessing environment. They have built-in features that help solve the coordination problems that have discouraged multiprocessing system development in the past.

Bus Lock

When configured in maximum mode, the 8086 and 8088 provide the $\overline{\text{LOCK}}$ (bus lock) signal. The BIU activates $\overline{\text{LOCK}}$ when the EU executes the one-byte LOCK prefix instruction. The $\overline{\text{LOCK}}$ signal remains active throughout execution of the instruction that follows the LOCK prefix. Interrupts are *not* affected by the LOCK prefix. If another processor requests use of the bus (via the request/grant lines, which are discussed shortly), the CPU records the request, but does not honor it until execution of the locked instruction has been completed.

Note that the $\overline{\text{LOCK}}$ signal remains active for the duration of a *single* instruction. If two consecutive instructions are each preceded by a LOCK prefix, there will still be an unlocked period between these instructions. In the case of a locked repeated string instruction, $\overline{\text{LOCK}}$ does remain active for the duration of the block operation.

When the 8086 or 8088 is configured in minimum mode, the $\overline{\text{LOCK}}$ signal is not available. The LOCK prefix can be used, however, to delay the

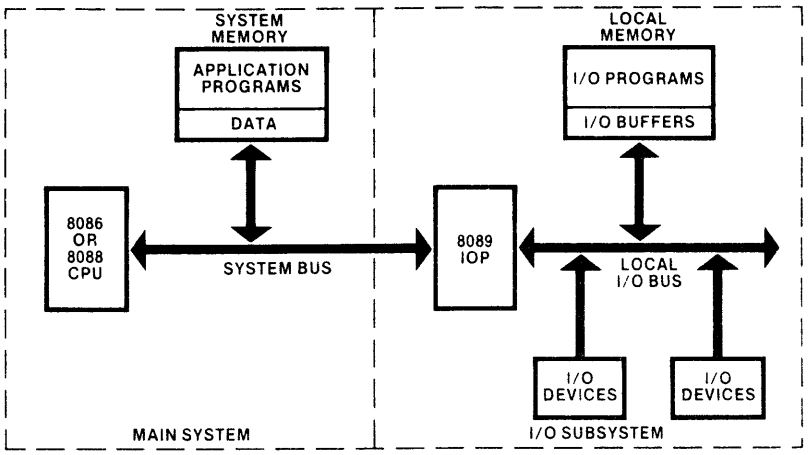


Figure 2-22. Multiprocessing System

generation of an HLDA response to a HOLD request until execution of the locked instruction is completed.

The $\overline{\text{LOCK}}$ signal provides information only. It is the responsibility of other processors on the shared bus to not attempt to obtain the bus while $\overline{\text{LOCK}}$ is active. If the system uses 8289 Bus Arbiters to control access to the shared bus, the 8289's accept $\overline{\text{LOCK}}$ as an input and do not relinquish the bus while this signal is active.

$\overline{\text{LOCK}}$ may be used in multiprocessing systems to coordinate access to a common resource, such as a buffer or a pointer. If access to the resource is not controlled, one processor can read an erroneous value from the resource when another processor is updating it (see figure 2-23).

Access can be controlled (see figure 2-24) by using the LOCK prefix in conjunction with the XCHG (exchange register with memory) instruction. The basis for controlling access to a given resource is a semaphore, a software-settable flag or switch that indicates whether the resource is "available" (semaphore=0) or "busy" (semaphore=1). Processors that share the bus agree by convention not to use the resource unless the semaphore indicates that it is available. They likewise agree to set the semaphore when they are using the resource and to clear it when they are finished.

The XCHG instruction can obtain the current value of the semaphore and set it to "busy" in a single instruction. The instruction, however, requires two bus cycles to swap 8-bit values. It is possible for another processor to obtain the bus between these two cycles and to gain access to the partially-updated semaphore. This can be prevented by preceding the XCHG instruction with a LOCK prefix, as illustrated in figure 2-25. The bus lock establishes control over access to the semaphore and thus to the shared resource.

WAIT and $\overline{\text{TEST}}$

The 8086 and 8088 (in either maximum or minimum mode) can be synchronized to an external event with the WAIT (wait for $\overline{\text{TEST}}$) instruction and the $\overline{\text{TEST}}$ input signal. When the EU executes a WAIT instruction, the result depends on the state of the $\overline{\text{TEST}}$ input line. If $\overline{\text{TEST}}$ is inactive, the processor enters an idle state and repeatedly retests the $\overline{\text{TEST}}$ line at five-clock intervals. If $\overline{\text{TEST}}$ is active, execution continues with the instruction following the WAIT.

BUS CYCLE	SHARED POINTER IN MEMORY	PROCESSOR ACTIVITIES
0	05,22 | 4C,1B	
1	C2,59 | 4C,1B	"A" UPDATES 1 WORD
2	C2,59 | 4C,1B	"B" READS PARTIALLY UPDATED VALUE
3	C2,59 | 31,05	"A" COMPLETES UPDATE

Figure 2-23. Uncontrolled Access to Shared Resource

Escape

The ESC (escape) instruction provides a way for another processor to obtain an instruction and/or a memory operand from an 8086/8088 program. When used in conjunction with WAIT and TEST, ESC can initiate a "subroutine" that executes concurrently in another processor (see figure 2-26).

Six bits in the ESC instruction may be specified by the programmer when the instruction is written. By monitoring the 8086/8088 bus and control lines, another processor can capture the ESC instruction when it is fetched by the BIU. The six bits may then direct the external processor to perform some predefined activity.

If the 8086/8088 is configured in maximum mode, the external processor, having determined that an ESC has been fetched, can monitor QS0

BUS CYCLE	SEMAPHORE	SHARED POINTER IN MEMORY	PROCESSOR ACTIVITIES
0	0	05,22 | 4C,1B	
1	1	05,22 | 4C,1B	"A" OBTAINS EXCLUSIVE USE
2	1	C2,59 | 4C,1B	"A" UPDATES 1 WORD
3	1	C2,59 | 4C,1B	"B" TESTS SEMAPHORE AND WAITS
4	1	C2,59 | 31,05	"A" COMPLETES UPDATE
5	1	C2,59 | 31,05	"B" TESTS SEMAPHORE AND WAITS
6	0	C2,59 | 31,05	"A" RELEASES RESOURCE
7	1	C2,59 | 31,05	"B" OBTAINS EXCLUSIVE USE
8	1	C2,59 | 31,05	"B" READS UPDATED VALUE
9	0	C2,59 | 31,05	"B" RELEASES RESOURCE

Figure 2-24. Controlled Access to Shared Resource

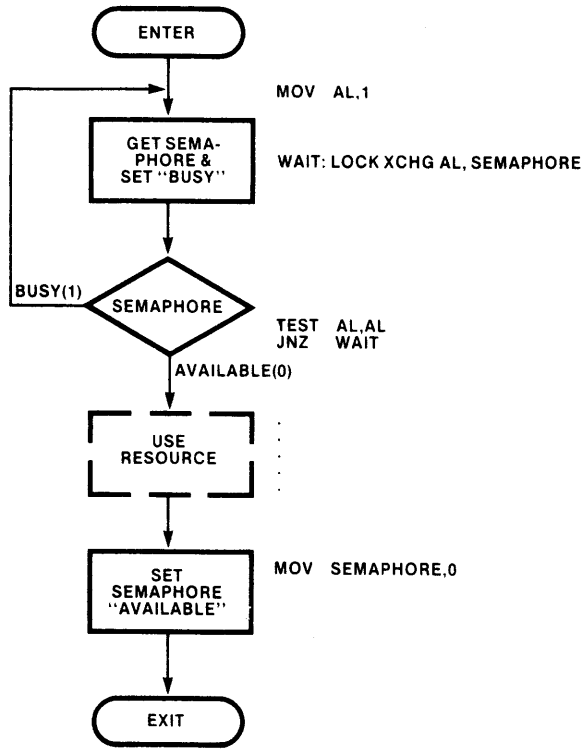


Figure 2-25. Using XCHG and LOCK

and QS1 (the queue status lines, discussed in section 2.6) and determine when the ESC instruction is executed. If the instruction references memory the external processor can then monitor the bus and capture the operand's physical address and/or the operand itself.

Note that fetching an ESC instruction is not tantamount to executing it. The ESC may be preceded by a jump that causes the queue to be reinitialized. This event also can be determined from the queue status lines.

Request/Grant Lines

When the 8086 or 8088 is configured in maximum mode, the HOLD and HLDA lines evolve into two more sophisticated signals called $\overline{RQ}/\overline{GT0}$ and $\overline{RQ}/\overline{GT1}$. These are bidirectional lines that can be used to share a local bus between an 8086 or 8088 and two other processors via a handshake sequence.

The request/grant sequence is a three-phase cycle: request, grant and release. First, the processor desiring the bus pulses a request/grant line. The CPU returns a pulse on the same line indicating that it is entering the "hold acknowledge" state and is relinquishing the bus. The BIU is logically disconnected from the bus during this period. The

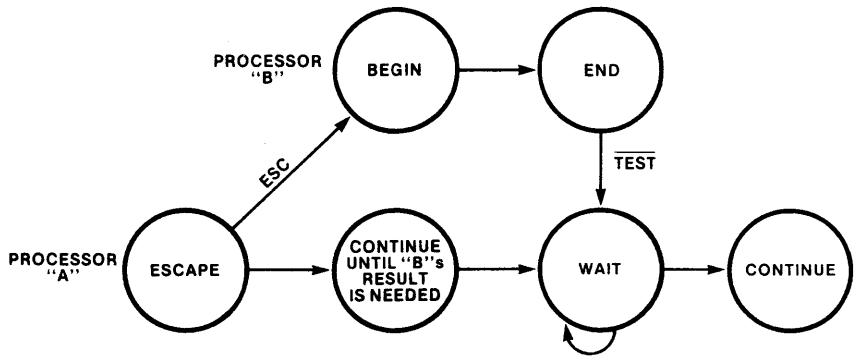


Figure 2-26. Using ESC with WAIT and TEST

EU, however, will continue to execute instructions until an instruction requires bus access or the queue is emptied, whichever occurs first. When the other processor has finished with the bus, it sends a final pulse to the 8086/8088 indicating that the request has ended and that the CPU may reclaim the bus.

$\overline{RQ}/\overline{GT0}$ has higher priority than $\overline{RQ}/\overline{GT1}$. If requests arrive simultaneously on both lines, the grant goes to the processor on $\overline{RQ}/\overline{GT0}$ and $\overline{RQ}/\overline{GT1}$ is acknowledged after the bus has been returned to the CPU. If, however, a request arrives on $\overline{RQ}/\overline{GT0}$ while the CPU is processing a prior request on $\overline{RQ}/\overline{GT1}$, the second request is not honored until the processor on $\overline{RQ}/\overline{GT1}$ releases the bus.

Multibus™ Architecture

Intel has designed a general-purpose multiprocessing bus called the Multibus. This is the standard design used in iSBC™ single-board microcomputer products. Many other manufacturers offer products that are compatible with the Multibus architecture as well. When the 8086 and 8088 are configured in maximum mode, the 8288 Bus Controller outputs signals that are electrically compatible with the Multibus protocol. Designers of multiprocessing systems may want to consider using the Multibus architecture in the design of their products to reduce development cost and time, and to obtain compatibility with the wide variety of boards available in the iSBC product line.

The Multibus architecture provides a versatile communications channel that can be used to coordinate a wide variety of computing modules (see figure 2-27). Modules in a Multibus system are designated as masters or slaves. Masters may obtain use of the bus and initiate data transfers on it. Slaves are the objects of data transfers only. The Multibus architecture allows both 8- and 16-bit masters to be intermixed in a system. In addition to 16 data lines, the bus design provides 20 address lines, eight multilevel interrupt lines, and control and arbitration lines. An auxiliary power bus also is provided to route standby power to memories if the normal supply fails.

The Multibus architecture maintains its own clock, independent of the clocks of the modules it links together. This allows different speed masters to share the bus and allows masters to operate asynchronously with respect to each other. The arbitration logic of the bus permit slow-speed masters to compete equably for use of the bus. Once a module has obtained the bus, however, transfer speeds are dependent only on the capabilities of the transmitting and receiving modules. Finally, the Multibus standard defines the form factors and physical requirements of modules that communicate on this bus. For a complete description of the Multibus architec-

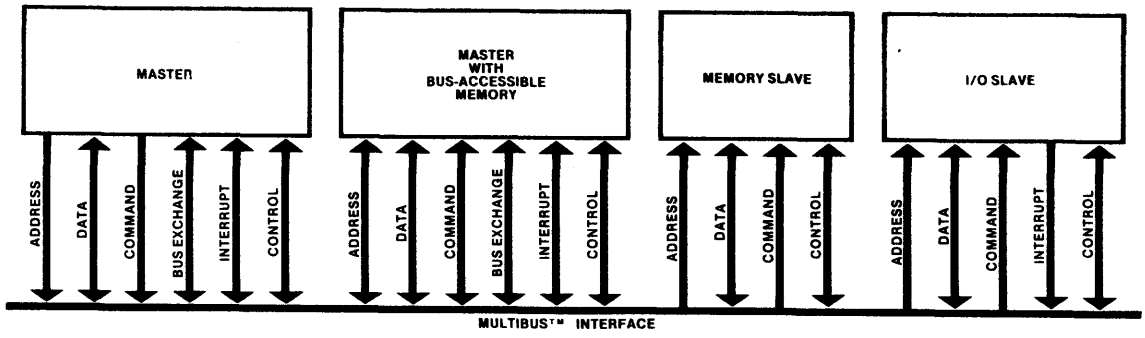


Figure 2-27. Multibus™-Based System

ture, refer to the Intel Multibus Specification (document number 9800683) and Application Note 28A, "Intel Multibus Interfacing."

8289 Bus Arbiter

Multiprocessor systems require a means of coordinating the processors' use of the shared bus. The 8289 Bus Arbiter works in conjunction with the 8288 Bus Controller to provide this control for 8086- and 8088-based systems. It is compatible with the Multibus architecture and can be used in other shared-bus designs as well.

The 8289 eliminates race conditions, resolves bus contention and matches processors operating asynchronously with respect to each other. Each processor on the bus is assigned a different priority. When simultaneous requests for the bus arrive, the 8289 resolves the contention and grants the bus to the processor with the highest priority; three different prioritizing techniques may be used. Chapter 4 discusses the 8289 in more detail.

2.6 Processor Control and Monitoring

Interrupts

The 8086 and 8088 have a simple and versatile interrupt system. Every interrupt is assigned a type code that identifies it to the CPU. The 8086 and 8088 can handle up to 256 different interrupt types. Interrupts may be initiated by devices external to the CPU; in addition, they also may be triggered by software interrupt instructions and, under certain conditions, by the CPU itself (see figure 2-28). Figure 2-29 illustrates the basic response of the 8086 and 8088 to an interrupt. The next sections elaborate on the information presented in this drawing.

External Interrupts

The 8086 and 8088 have two lines that external devices may use to signal interrupts (INTR and NMI). The INTR (Interrupt Request) line is usually driven by an Intel® 8259A Programmable Interrupt Controller (PIC), which is in turn connected to the devices that need interrupt services. The 8259A is a very flexible circuit that is controlled by software commands from the 8086 or 8088 (the PIC appears as a set of I/O ports to the software). Its main job is to accept interrupt requests from the devices attached to it, determine which requesting device has the highest priority, and then activate the 8086/8088 INTR line if the selected device has higher priority than the device currently being serviced (if there is one).

When INTR is active, the CPU takes different action depending on the state of the interrupt-enable flag (IF). No action takes place, however, until the currently-executing instruction has been

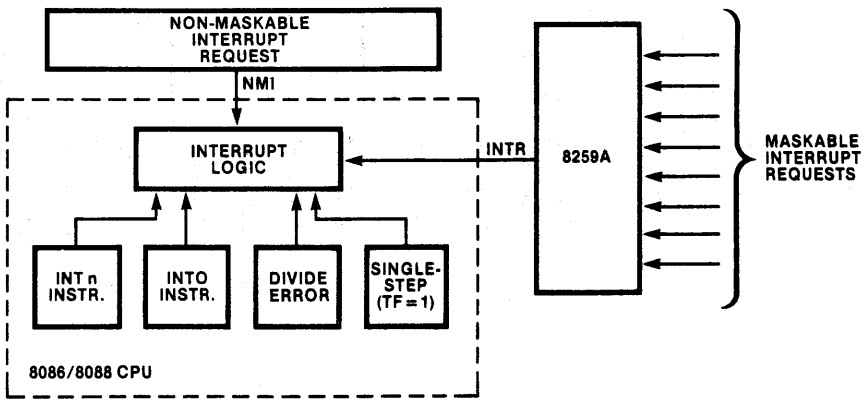


Figure 2-28. Interrupt Sources

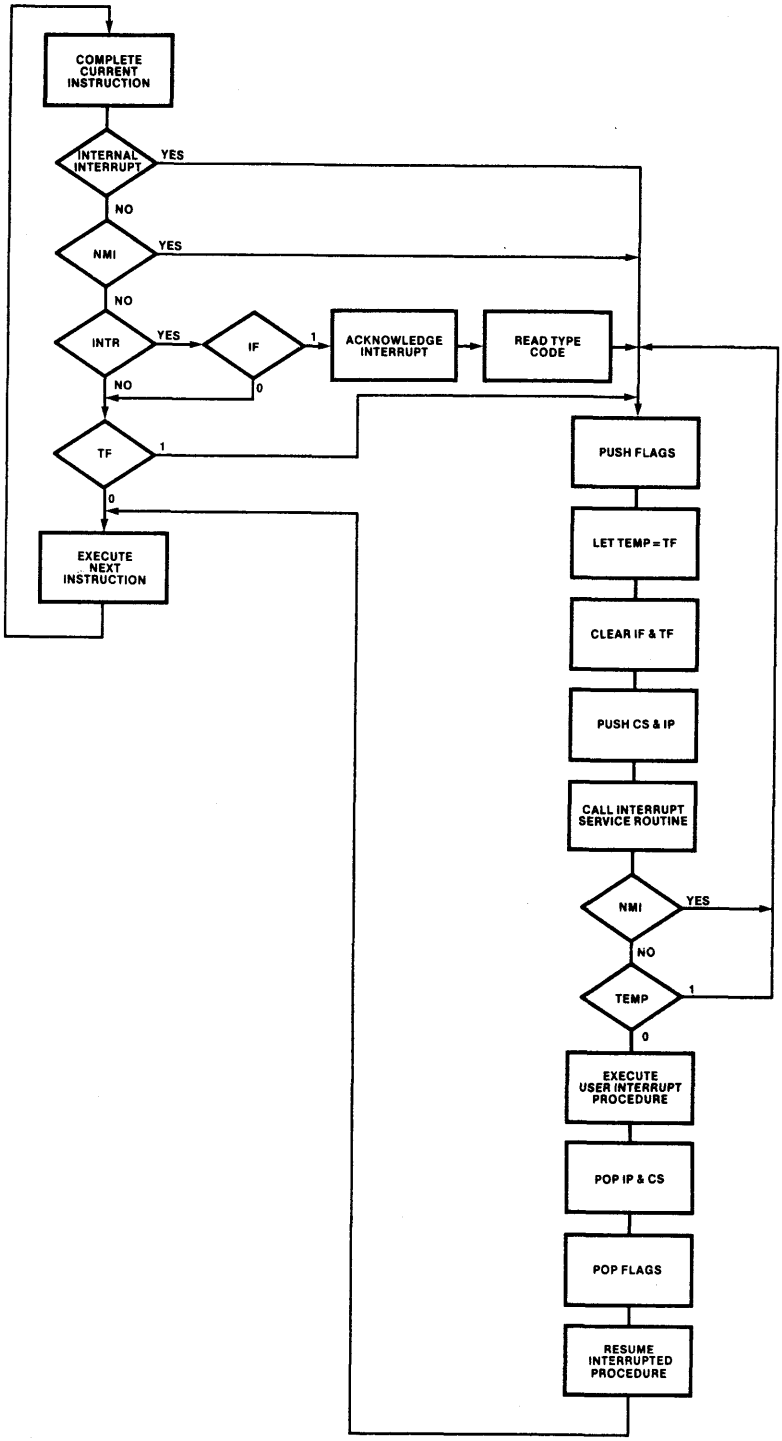


Figure 2-29. Interrupt Processing Sequence

completed.* Then, if IF is clear (meaning that interrupts signaled on INTR are masked or disabled), the CPU ignores the interrupt request and processes the next instruction. The INTR signal is not latched by the CPU, so it must be held active until a response is received or the request is withdrawn. If interrupts on INTR are enabled (if IF is set), then the CPU recognizes the interrupt request and processes it. Interrupt requests arriving on INTR can be enabled by executing an STI (set interrupt-enable flag) instruction, and disabled by executing a CLI (clear interrupt-enable flag) instruction. They also may be selectively masked (some types enabled, some disabled) by writing commands to the 8259A. It should be noted that in order to reduce the likelihood of excessive stack buildup, the STI and IRET instructions will reenable interrupts only after the end of the following instruction.

The CPU acknowledges the interrupt request by executing two consecutive interrupt acknowledge (INTA) bus cycles. If a bus hold request arrives (via the HOLD or request/grant lines) during the INTA cycles, it is not honored until the cycles have been completed. In addition, if the CPU is configured in maximum mode, it activates the $\overline{\text{LOCK}}$ signal during these cycles to indicate to other processors that they should not attempt to obtain the bus. The first cycle signals the 8259A that the request has been honored. During the second INTA cycle, the 8259A responds by placing a byte on the data bus that contains the interrupt type (0-255) associated with the device requesting service. (The type assignment is made when the 8259A is initialized by software in the 8086 or 8088.) The CPU reads this type code and uses it to call the corresponding interrupt procedure.

An external interrupt request also may arrive on another CPU line, NMI (non-maskable interrupt). This line is edge-triggered (INTR is level-triggered) and is generally used to signal the CPU of a "catastrophic" event, such as the imminent loss of power, memory error detection or bus parity error. Interrupt requests arriving on NMI cannot be disabled, are latched by the CPU, and have higher priority than an interrupt request on INTR. If an interrupt request arrives on both lines during the execution of an instruction, NMI will be recognized first. Non-maskable interrupts are predefined as type 2; the processor does not need to be supplied with a type code to call the NMI procedure, and it does not run the INTA bus cycles in response to a request on NMI.

The time required for the CPU to recognize an external interrupt request (interrupt latency) depends on how many clock periods remain in the execution of the current instruction. On the average, the longest latency occurs when a multiplication, division or variable-bit shift or rotate instruction is executing when the interrupt request arrives (see section 2.7 for detailed instruction timing data). As mentioned previously, in a few cases, worst-case latency will span two instructions rather than one.

Internal Interrupts

An INT (interrupt) instruction generates an interrupt immediately upon completion of its execution. The interrupt type coded into the instruction supplies the CPU with the type code needed to call the procedure to process the interrupt. Since any type code may be specified, software interrupts may be used to test interrupt procedures written to service external devices.

*There are a few cases in which an interrupt request is not recognized until after the *following* instruction. Repeat, LOCK and segment override prefixes are considered "part of" the instructions they prefix; no interrupt is recognized between execution of a prefix and an instruction. A MOV (move) to segment register instruction and a POP segment register instruction are treated similarly: no interrupt is recognized until after the following instruction. This mechanism protects a program that is changing to a new stack (by updating SS and SP). If an interrupt were recognized after SS had been changed, but before SP had been altered, the processor would push the flags, CS and IP into the wrong area of memory. It follows from this that whenever a segment register and another value must be updated together, the segment register should be changed first, followed immediately by the instruction that changes the other value. There are also two cases, WAIT and repeated string instructions, where an interrupt request is recognized in the middle of an instruction. In these cases, interrupts are accepted after any completed primitive operation or wait test cycle.

If the overflow flag (OF) is set, an INTO (interrupt on overflow) instruction generates a type 4 interrupt immediately upon completion of its execution.

The CPU itself generates a type 0 interrupt immediately following execution of a DIV or IDIV (divide, integer divide) instruction if the calculated quotient is larger than the specified destination.

If the trap flag (TF) is set, the CPU automatically generates a type 1 interrupt following *every* instruction. This is called single-step execution and is a powerful debugging tool that is discussed in more detail shortly.

All internal interrupts (INT, INTO, divide error, and single-step) share these characteristics:

1. The interrupt type code is either contained in the instruction or is predefined.
2. No INTA bus cycles are run.
3. Internal interrupts cannot be disabled, except for single-step.
4. Any internal interrupt (except single-step) has higher priority than any external interrupt (see table 2-3). If interrupt requests arrive on NMI and/or INTR during execution of an instruction that causes an internal interrupt (e.g., divide error), the internal interrupt is processed first.

Interrupt Pointer Table

The interrupt pointer (or interrupt vector) table (figure 2-30) is the link between an interrupt type code and the procedure that has been designated to service interrupts associated with that code. The interrupt pointer table occupies up to the first 1k bytes of low memory. There may be up to 256 entries in the table, one for each interrupt type

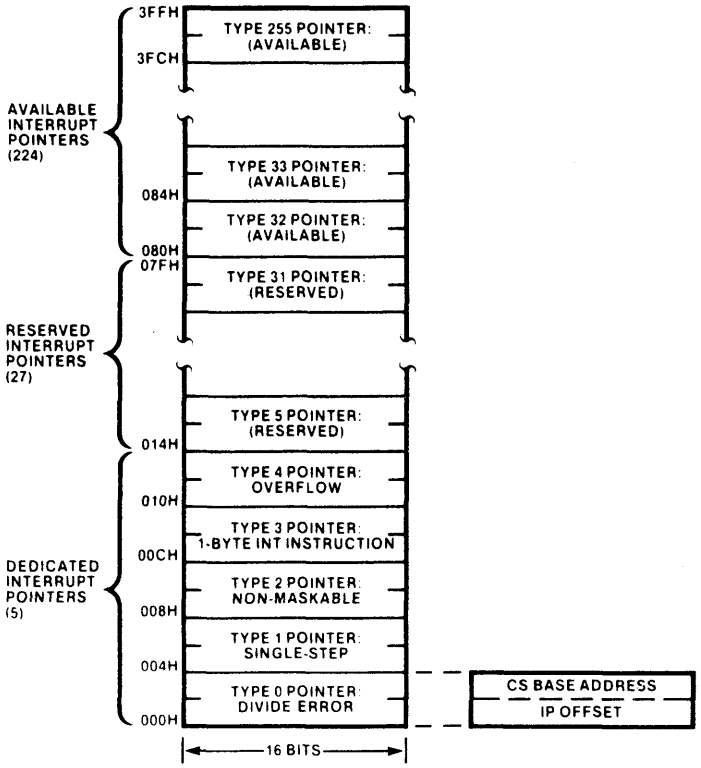


Figure 2-30. Interrupt Pointer Table

that can occur in the system. Each entry in the table is a doubleword pointer containing the address of the procedure that is to service interrupts of that type. The higher-addressed word of the pointer contains the base address of the segment containing the procedure. The lower-addressed word contains the procedure's offset from the beginning of the segment. Since each entry is four bytes long, the CPU can calculate the location of the correct entry for a given interrupt type by simply multiplying (type*4).

Table 2-3. Interrupt Priorities

INTERRUPT	PRIORITY
Divide error, INT n, INTO	highest
NMI	
INTR	
Single-step	lowest

Space at the high end of the table that would be occupied by entries for interrupt types that cannot occur in a given application may be used for other purposes. The dedicated and reserved portions of the interrupt pointer table (locations 0H through 7FH), however, should not be used for any other purpose to insure proper system operation and to preserve compatibility with future Intel hardware and software products.

After pushing the flags onto the stack, the 8086 or 8088 activates an interrupt procedure by executing the equivalent of an intersegment indirect CALL instruction. The target of the "CALL" is the address contained in the interrupt pointer table element located at (type*4). The CPU saves the address of the next instruction by pushing CS and IP onto the stack. These are then replaced by the second and first words of the table element, thus transferring control to the procedure.

If multiple interrupt requests arrive simultaneously, the processor activates the interrupt procedures in priority order. Figure 2-31 shows how procedures would be activated in an extreme case. The processor is running in single-step mode with external interrupts enabled. During execution of a divide instruction, INTR is activated. Furthermore the instruction generates a divide error interrupt. Figure 2-31 shows that the interrupts are recognized in turn, in the order of their priorities except for INTR. INTR is not recognized until after the following instruction because recognition of the earlier interrupts cleared IF. Of couse interrupts could be reenabled in any of the interrupt response routines if earlier response to INTR is desired.

As figure 2-31 shows, all main-line code is executed in single-step mode. Also, because of the order of interrupt processing, the opportunity exists in each occurrence of the single-step routine to select whether pending interrupt routines (divide error and INTR routines in this example) are executed at full speed or in single-step mode.

Interrupt Procedures

When an interrupt service procedure is entered, the flags, CS, and IP are pushed onto the stack and TF and IF are cleared. The procedure may reenable external interrupts with the STI (set interrupt-enable flag) instruction, thus allowing itself to be interrupted by a request on INTR. (Note, however, that interrupts are not actually enabled until the instruction *following* STI has executed.) An interrupt procedure always may be interrupted by a request arriving on NMI. Software- or processor-initiated interrupts occurring within the procedure also will interrupt the procedure. Care must be taken in interrupt procedures that the type of interrupt being serviced by the procedure does not itself inadvertently occur within the procedure. For example, an attempt to divide by 0 in the divide error (type 0) interrupt procedure may result in the procedure being reentered endlessly. Enough stack space must be available to accommodate the maximum depth of interrupt nesting that can occur in the system.

Like all procedures, interrupt procedures should save any registers they use before updating them, and restore them before terminating. It is good practice for an interrupt procedure to enable external interrupts for all but "critical sections" of code (those sections that cannot be interrupted without risking erroneous results). If external interrupts are disabled for too long in a procedure, interrupt requests on INTR can potentially be lost.

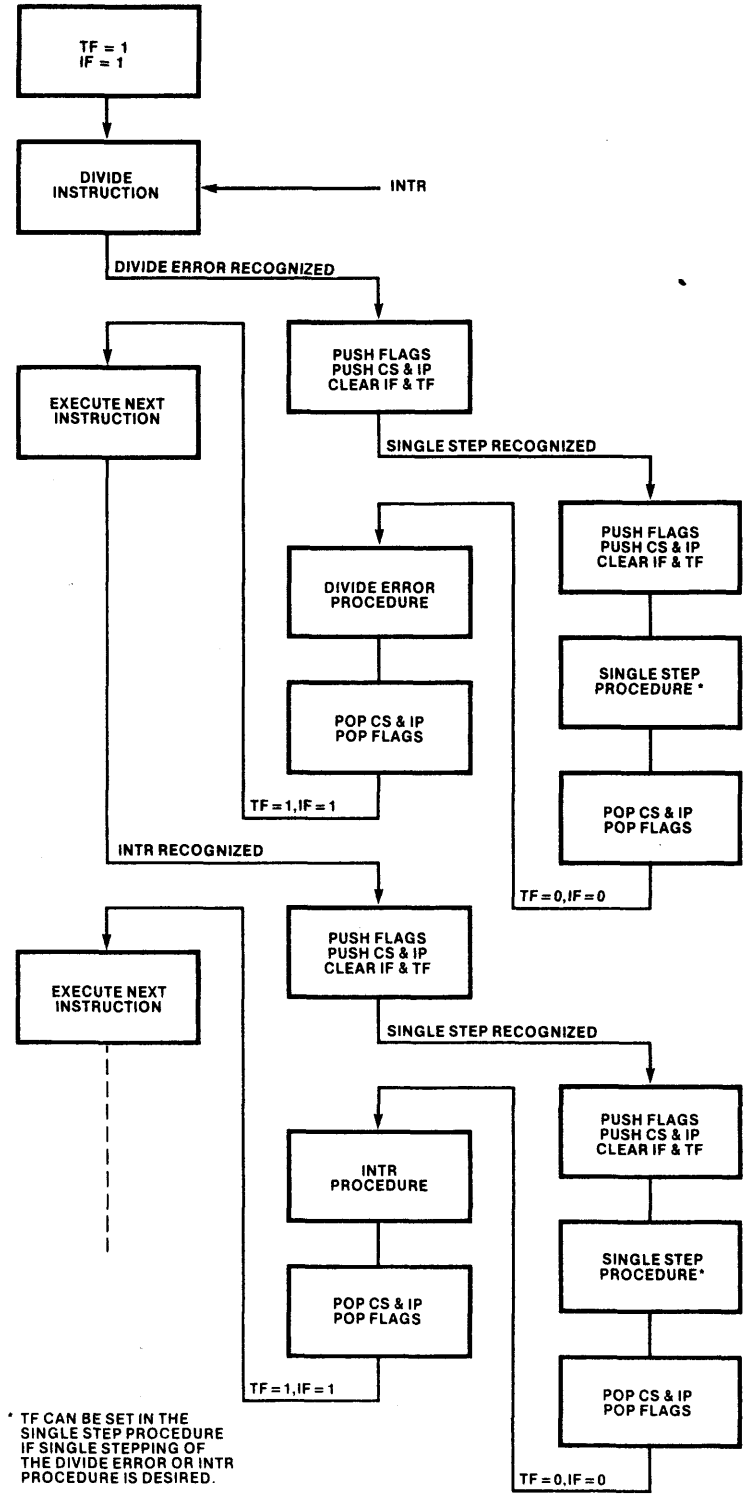


Figure 2-31. Processing Simultaneous Interrupts

All interrupt procedures should be terminated with an IRET (interrupt return) instruction. The IRET instruction assumes that the stack is in the same condition as it was when the procedure was entered. It pops the top three stack words into IP, CS and the flags, thus returning to the instruction that was about to be executed when the interrupt procedure was activated.

The actual processing done by the procedure is dependent upon the application. If the procedure is servicing an external device, it should output a command to the device instructing it to remove its interrupt request. It might then read status information from the device, determine the cause of the interrupt and then take action accordingly. Section 2.10 contains three typical interrupt procedure examples.

Software-initiated interrupt procedures may be used as service routines ("supervisor calls") for other programs in the system. In this case, the interrupt procedure is activated when a program, rather than an external device, needs attention. (The "attention" might be to search a file for a record, send a message to another program, request an allocation of free memory, etc.) Software interrupt procedures can be advantageous in systems that dynamically relocate programs during execution. Since the interrupt pointer table is at a fixed storage location, procedures may "call" each other through the table by issuing software interrupt instructions. This provides a stable communication "exchange" that is independent of procedure addresses. The interrupt procedures may themselves be moved so long as the interrupt pointer table always is updated to provide the linkage from the "calling" program via the interrupt type code.

Single-Step (Trap) Interrupt

When TF (the trap flag) is set, the 8086 or 8088 is said to be in single-step mode. In this mode, the processor automatically generates a type 1 interrupt after each instruction. Recall that as part of its interrupt processing, the CPU automatically pushes the flags onto the stack and then clears TF and IF. Thus the processor is *not* in single-step mode when the single-step interrupt procedure is entered; it runs normally. When the single-step procedure terminates, the old flag image is restored from the stack, placing the CPU back into single-step mode.

Single-stepping is a valuable debugging tool. It allows the single-step procedure to act as a "window" into the system through which operation can be observed instruction-by-instruction. A single-step interrupt procedure, for example, can print or display register contents, the value of the instruction pointer (it is on the stack), key memory variables, etc., as they change after each instruction. In this way the exact flow of a program can be traced in detail, and the point at which discrepancies occur can be determined. Other possible services that could be provided by a single-step routine include:

- Writing a message when a specified memory location or I/O port changes value (or equals a specified value).
- Providing diagnostics selectively (only for certain instruction addresses for instance).
- Letting a routine execute a number of times before providing diagnostics.

The 8086 and 8088 do not have instructions for setting or clearing TF directly. Rather, TF can be changed by modifying the flag-image on the stack. The PUSHF and POPF instructions are available for pushing and popping the flags directly (TF can be set by ORing the flag-image with 0100H and cleared by ANDing it with FEFFH). After TF is set in this manner, the first single-step interrupt occurs after the first instruction following the IRET from the single-step procedure.

If the processor is single-stepping, it processes an interrupt (either internal or external) as follows. Control is passed normally (flags, CS and IP are pushed) to the procedure designated to handle the type of interrupt that has occurred. However, before the first instruction of that procedure is executed, the single-step interrupt is "recognized" and control is passed normally (flags, CS and IP are pushed) to the type 1 interrupt procedure. When single-step procedure terminates, control returns to the previous interrupt procedure. Figure 2-31 illustrates this process in a case where two interrupts occur when the processor is in single-step mode.

Breakpoint Interrupt

A type 3 interrupt is dedicated to the breakpoint interrupt. A breakpoint is generally any place in a program where normal execution is arrested so

that some sort of special processing may be performed. Breakpoints typically are inserted into programs during debugging as a way of displaying registers, memory locations, etc., at crucial points in the program.

The INT 3 (breakpoint) instruction is one byte long. This makes it easy to "plant" a breakpoint anywhere in a program. Section 2.10 contains an example that shows how a breakpoint may be set and how a breakpoint procedure may be used to place the processor into single-step mode.

The breakpoint instruction also may be used to "patch" a program (insert new instructions) without recompiling or reassembling it. This may be done by saving an instruction byte, and replacing it with an INT 3 (CCH) machine instruction. The breakpoint procedure would contain the new machine instructions, plus code to restore the saved instruction byte and decrement IP on the stack before returning, so that the displaced instruction would be executed after the patch instructions. The breakpoint example in section 2.10 illustrates these principles.

Note that patching a program requires machine-instruction programming and should be undertaken with considerable caution; it is easy to add new bugs to a program in an attempt to correct existing ones. Note also that a patch is only a temporary measure to be used in exceptional conditions. The affected code should be updated and retranslated as soon as possible.

System Reset

The 8086/8088 RESET line provides an orderly way to start or restart an executing system. When the processor detects the positive-going edge of a pulse on RESET, it terminates all activities until the signal goes low, at which time it initializes the system as shown in table 2-4.

Since the code segment register contains FFFFH and the instruction pointer contains 0H, the processor executes its first instruction following system reset from absolute memory location FFFF0H. This location normally contains an intersegment direct JMP instruction whose target is the actual beginning of the system program. The LOC-86 utility supplies this JMP instruction from information in the program that identifies its first instruction. As external (maskable) interrupts are disabled by system reset, the system software should reenable interrupts as soon as the system is initialized to the point where they can be processed.

Table 2-4. CPU State Following RESET

CPU COMPONENT	CONTENT
Flags	Clear
Instruction Pointer	0000H
CS Register	FFFFH
DS Register	0000H
SS Register	0000H
ES Register	0000H
Queue	Empty

Instruction Queue Status

When configured in maximum mode, the 8086 and 8088 provide information about instruction queue operations on lines QS0 and QS1. Table 2-5 interprets the four states that these lines can represent.

The queue status lines are provided for external processors that receive instructions and/or operands via the 8086/8088 ESC (escape) instruction (see sections 2.5 and 2.8). Such a processor may monitor the bus to see when an ESC instruction is fetched and then track the instruction through the queue to determine when (and if) the instruction is executed.

Table 2-5. Queue Status Signals (Maximum Mode Only)

QS_0	QS_1	QUEUE OPERATION IN LAST CLK CYCLE
0	0	No operation; default value
0	1	First byte of an instruction was taken from the queue
1	0	Queue was reinitialized
1	1	Subsequent byte of an instruction was taken from the queue

Processor Halt

When the HLT (halt) instruction (see section 2.7) is executed, the 8086 or 8088 enters the halt state. This condition may be interpreted as "stop all

operations until an external interrupt occurs or the system is reset." No signals are floated during the halt state, and the content of the address and data buses is undefined. A bus hold request arriving on the HOLD line (minimum mode) or either request/grant line (maximum mode) is acknowledged normally while the processor is halted.

The halt state can be used when an event prevents the system from functioning correctly. An example might be a power-fail interrupt. After recognizing that loss of power is imminent, the CPU could use the remaining time to move registers, flags and vital variables to (for example) a battery-powered CMOS RAM area and then halt until the return of power was signaled by an interrupt or system reset.

Status Lines

When configured in maximum mode, the 8086 and 8088 emit eight status signals that can be used by external devices. Lines $\overline{S0}$, $\overline{S1}$ and $\overline{S2}$ identify the type of bus cycle that the CPU is starting to execute (table 2-6). These lines are typically decoded by the 8288 Bus Controller. S3 and S4 indicate which segment register was used to construct the physical address being used in this bus cycle (see table 2-7). Line S5 reflects the state of the interrupt-enable flag. S6 is always 0. S7 is a spare line whose content is undefined.

Table 2-6. Bus Cycle Status Signals

$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	TYPES OF BUS CYCLE
0	0	0	Interrupt Acknowledge
0	0	1	Read I/O
0	1	0	Write I/O
0	1	1	HALT
1	0	0	Instruction Fetch
1	0	1	Read Memory
1	1	0	Write Memory
1	1	1	Passive; no bus cycle

Table 2-7. Segment Register Status Lines

S_4	S_3	SEGMENT REGISTER
0	0	ES
0	1	SS
1	0	CS or none (I/O or Interrupt Vector)
1	1	DS

2.7 Instruction Set

The 8086 and 8088 execute exactly the same instructions. This instruction set includes equivalents to the instructions typically found in previous microprocessors, such as the 8080/8085. Significant new operations include:

- multiplication and division of signed and unsigned binary numbers as well as unpacked decimal numbers,
- move, scan and compare operations for strings up to 64k bytes in length,
- non-destructive bit testing,
- byte translation from one code to another,
- software-generated interrupts, and
- a group of instructions that can help coordinate the activities of multiprocessor systems.

These instructions treat different types of operands uniformly. Nearly every instruction can operate on either byte or word data. Register, memory and immediate operands may be specified interchangeably in most instructions (except, of course, that immediate values may only serve as "source" and not "destination" operands). In particular, memory variables can be added to, subtracted from, shifted, compared, and so on, in place, without moving them in and out of registers. This saves instructions, registers, and execution time in assembly language programs. In high-level languages, where most variables are memory based, compilers, such as PL/M-86, can produce faster and shorter object programs.

The 8086/8088 instruction set can be viewed as existing at two levels: the assembly level and the machine level. To the assembly language programmer, the 8086 and 8088 appear to have a repertoire of about 100 instructions. One MOV (move) instruction, for example, transfers a byte or a word from a register or a memory location or an immediate value to either a register or a memory location. The 8086 and 8088 CPUs, however, recognize 28 different MOV machine instructions ("move byte register to memory," "move word immediate to register," etc.). The ASM-86 assembler translates the assembly-level instructions written by a programmer into the

machine-level instructions that are actually executed by the 8086 or 8088. Compilers such as PL/M-86 translate high-level language statements directly into machine-level instructions.

The two levels of the instruction set address two different requirements: efficiency and simplicity. The numerous—there are about 300 in all—forms of machine-level instructions allow these instructions to make very efficient use of storage. For example, the machine instruction that increments a memory operand is three or four bytes long because the address of the operand must be encoded in the instruction. To increment a register, however, does not require as much information, so the instruction can be shorter. In fact, the 8086 and 8088 have eight different machine-level instructions that increment a different 16-bit register; these instructions are only one byte long.

If a programmer had to write one instruction to increment a register, another to increment a memory variable, etc., the benefit of compact instructions would be offset by the difficulty of programming. The assembly-level instructions simplify the programmer's view of the instruction set. The programmer writes one form of the INC (increment) instruction and the ASM-86 assembler examines the operand to determine which machine-level instruction to generate.

This section presents the 8086/8088 instruction set from two perspectives. First, the assembly-level instructions are described in functional terms. The assembly-level instructions are then presented in a reference table that breaks out all permissible operand combinations with execution times and machine instruction length, plus the effect that the instruction has on the CPU flags. Machine-level instruction encoding and decoding are covered in section 4.2.

Data Transfer Instructions

The 14 data transfer instructions (table 2-8) move single bytes and words between memory and registers as well as between register AL or AX and I/O ports. The stack manipulation instructions are included in this group as are instructions for transferring flag contents and for loading segment registers.

Table 2-8. Data Transfer Instructions

GENERAL PURPOSE	
MOV	Move byte or word
PUSH	Push word onto stack
POP	Pop word off stack
XCHG	Exchange byte or word
XLAT	Translate byte
INPUT/OUTPUT	
IN	Input byte or word
OUT	Output byte or word
ADDRESS OBJECT	
LEA	Load effective address
LDS	Load pointer using DS
LES	Load pointer using ES
FLAG TRANSFER	
LAHF	Load AH register from flags
SAHF	Store AH register in flags
PUSHF	Push flags onto stack
POPF	Pop flags off stack

General Purpose Data Transfers

MOV *destination,source*

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

PUSH *source*

PUSH decrements SP (the stack pointer) by two and then transfers a word from the source operand to the top of stack now pointed to by SP. PUSH often is used to place parameters on the stack before calling a procedure; more generally, it is the basic means of storing temporary data on the stack.

POP *destination*

POP transfers the word at the current top of stack (pointed to by SP) to the destination operand, and then increments SP by two to point to the new top of stack. POP can be used to move temporary variables from the stack to registers or memory.

XCHG *destination,source*

XCHG (exchange) switches the contents of the source and destination (byte or word) operands. When used in conjunction with the LOCK prefix, XCHG can test and set a semaphore that controls access to a resource shared by multiple processors (see section 2.5).

XLAT *translate-table*

XLAT (translate) replaces a byte in the AL register with a byte from a 256-byte, user-coded translation table. Register BX is assumed to point to the beginning of the table. The byte in AL is used as an index into the table and is replaced by the byte at the offset in the table corresponding to AL's binary value. The first byte in the table has an offset of 0. For example, if AL contains 5H, and the sixth element of the translation table contains 33H, then AL will contain 33H following the instruction. XLAT is useful for translating characters from one code to another, the classic example being ASCII to EBCDIC or the reverse.

IN *accumulator,port*

IN transfers a byte or a word from an input port to the AL register or the AX register, respectively. The port number may be specified either with an immediate byte constant, allowing access to ports numbered 0 through 255, or with a number previously placed in the DX register, allowing variable access (by changing the value in DX) to ports numbered from 0 through 65,535.

OUT *port,accumulator*

OUT transfers a byte or a word from the AL register or the AX register, respectively, to an output port. The port number may be specified either with an immediate byte constant, allowing access to ports numbered 0 through 255, or with a number previously placed in register DX, allowing variable access (by changing the value in DX) to ports numbered from 0 through 65,535.

Address Object Transfers

These instructions manipulate the *addresses* of variables rather than the contents or values of variables. They are most useful for list processing, based variables, and string operations.

LEA *destination,source*

LEA (load effective address) 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. LEA does not affect any flags. The XLAT and string instructions assume that certain registers point to operands; LEA can be used to load these registers (e.g., loading BX with the address of the translate table used by the XLAT instruction).

LDS *destination,source*

LDS (load pointer using DS) transfers a 32-bit pointer variable from the source operand, which must be a memory operand, to the destination operand and register DS. The offset word of the pointer is transferred to the destination operand, which may be any 16-bit general register. The segment word of the pointer is transferred to register DS. Specifying SI as the destination operand is a convenient way to prepare to process a source string that is not in the current data segment (string instructions assume that the source string is located in the current data segment and that SI contains the offset of the string).

LES *destination,source*

LES (load pointer using ES) transfers a 32-bit pointer variable from the source operand, which must be a memory operand, to the destination operand and register ES. The offset word of the pointer is transferred to the destination operand, which may be any 16-bit general register. The segment word of the pointer is transferred to register ES. Specifying DI as the destination operand is a convenient way to prepare to process a destination string that is not in the current extra segment. (The destination string must be located in the extra segment, and DI must contain the offset of the string.)

Flag Transfers

LAHF

LAHF (load register AH from flags) copies SF, ZF, AF, PF and CF (the 8080/8085 flags) into bits 7, 6, 4, 2 and 0, respectively, of register AH

(see figure 2-32). The content of bits 5, 3 and 1 is undefined; the flags themselves are not affected. LAHF is provided primarily for converting 8080/8085 assembly language programs to run on an 8086 or 8088.

SAHF

SAHF (store register AH into flags) transfers bits 7, 6, 4, 2 and 0 from register AH into SF, ZF, AF, PF and CF, respectively, replacing whatever values these flags previously had. OF, DF, IF and TF are not affected. This instruction is provided for 8080/8085 compatibility.

PUSHF

PUSHF decrements SP (the stack pointer) by two and then transfers all flags to the word at the top of stack pointed to by SP (see figure 2-32). The flags themselves are not affected.

POPF

POPF transfers specific bits from the word at the current top of stack (pointed to by register SP) into the 8086/8088 flags, replacing whatever values the flags previously contained (see figure 2-32). SP is then incremented by two to point to the new top of stack. PUSHF and POPF allow a procedure to save and restore a calling program's flags. They also allow a program to change the setting of TF (there is no instruction for updating this flag directly). The change is accomplished by pushing the flags, altering bit 8 of the memory-image and then popping the flags.

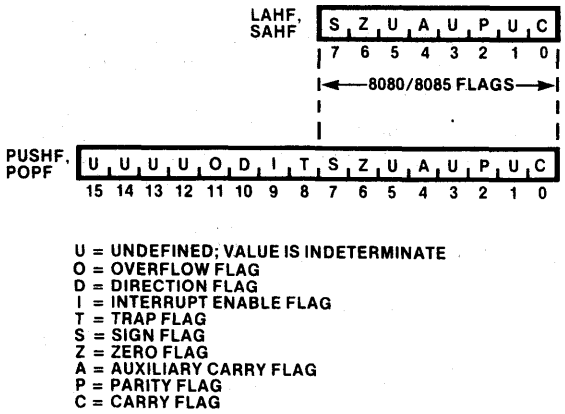


Figure 2-32. Flag Storage Formats

Arithmetic Instructions

Arithmetic Data Formats

8086 and 8088 arithmetic operations (table 2-9) may be performed on four types of numbers: unsigned binary, signed binary (integers), unsigned packed decimal and unsigned unpacked decimal (see table 2-10). Binary numbers may be 8 or 16 bits long. Decimal numbers are stored in bytes, two digits per byte for packed decimal and one digit per byte for unpacked decimal. The processor always assumes that the operands specified in arithmetic instructions contain data that represent valid numbers for the type of instruction being performed. Invalid data may produce unpredictable results.

Table 2-9. Arithmetic Instructions

ADDITION	
ADD	Add byte or word
ADC	Add byte or word with carry
INC	Increment byte or word by 1
AAA	ASCII adjust for addition
DAA	Decimal adjust for addition
SUBTRACTION	
SUB	Subtract byte or word
SBB	Subtract byte or word with borrow
DEC	Decrement byte or word by 1
NEG	Negate byte or word
CMP	Compare byte or word
AAS	ASCII adjust for subtraction
DAS	Decimal adjust for subtraction
MULTIPLICATION	
MUL	Multiply byte or word unsigned
IMUL	Integer multiply byte or word
AAM	ASCII adjust for multiply
DIVISION	
DIV	Divide byte or word unsigned
IDIV	Integer divide byte or word
AAD	ASCII adjust for division
CBW	Convert byte to word
CWD	Convert word to doubleword

Table 2-10. Arithmetic Interpretation of 8-Bit Numbers

HEX	BIT PATTERN	UNSIGNED BINARY	SIGNED BINARY	UNPACKED DECIMAL	PACKED DECIMAL
07	0 0 0 0 0 1 1 1	7	+7	7	7
89	1 0 0 0 1 0 0 1	137	−119	invalid	89
C5	1 1 0 0 0 1 0 1	197	−59	invalid	invalid

Unsigned binary numbers may be either 8 or 16 bits long; all bits are considered in determining a number's magnitude. The value range of an 8-bit unsigned binary number is 0-255; 16 bits can represent values from 0 through 65,535. Addition, subtraction, multiplication and division operations are available for unsigned binary numbers.

Signed binary numbers (integers) may be either 8 or 16 bits long. The high-order (leftmost) bit is interpreted as the number's sign: 0 = positive and 1 = negative. Negative numbers are represented in standard two's complement notation. Since the high-order bit is used for a sign, the range of an 8-bit integer is −128 through +127; 16-bit integers may range from −32,768 through +32,767. The value zero has a positive sign. Multiplication and division operations are provided for signed binary numbers. Addition and subtraction are performed with the unsigned binary instructions. Conditional jump instructions, as well as an "interrupt on overflow" instruction, can be used following an unsigned operation on an integer to detect overflow into the sign bit.

Packed decimal numbers are stored as unsigned byte quantities. The byte is treated as having one decimal digit in each half-byte (nibble); the digit in the high-order half-byte is the most significant. Hexadecimal values 0-9 are valid in each half-byte, and the range of a packed decimal number is 0-99. Addition and subtraction are performed in two steps. First an unsigned binary instruction is used to produce an intermediate result in register AL. Then an adjustment operation is performed which changes the intermediate value in AL to a final correct packed decimal result. Multiplication and division adjustments are not available for packed decimal numbers.

Unpacked decimal numbers are stored as unsigned byte quantities. The magnitude of the number is determined from the low-order half-byte; hexadecimal values 0-9 are valid and are interpreted as decimal numbers. The high-order half-byte must be zero for multiplication and division; it may contain any value for addition and subtraction. Arithmetic on unpacked decimal numbers is performed in two steps. The unsigned binary addition, subtraction and multiplication operations are used to produce an intermediate result in register AL. An adjustment instruction then changes the value in AL to a final correct unpacked decimal number. Division is performed similarly, except that the adjustment is carried out on the numerator operand in register AL first, then a following unsigned binary division instruction produces a correct result.

Unpacked decimal numbers are similar to the ASCII character representations of the digits 0-9. Note, however, that the high-order half-byte of an ASCII numeral is always 3H. Unpacked decimal arithmetic may be performed on ASCII numeric characters under the following conditions:

- the high-order half-byte of an ASCII numeral must be set to 0H prior to multiplication or division.
- unpacked decimal arithmetic leaves the high-order half-byte set to 0H; it must be set to 3H to produce a valid ASCII numeral.

Arithmetic Instructions and Flags

The 8086/8088 arithmetic instructions post certain characteristics of the result of the operation to six flags. Most of these flags can be tested by following the arithmetic instruction with a conditional jump instruction; the INTO (interrupt on overflow) instruction also may be used. The

various instructions affect the flags differently, as explained in the instruction descriptions. However, they follow these general rules:

- 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. Note that a *signed* carry is indicated by CF ≠ OF. CF can be used to detect an unsigned overflow. Two instructions, ADC (add with carry) and SBB (subtract with borrow), incorporate the carry flag in their operations and can be used to perform multibyte (e.g., 32-bit, 64-bit) addition and subtraction.

- AF (auxiliary carry flag): If an addition results in a carry out of the low-order half-byte of the result, then AF is set; otherwise AF is cleared. If a subtraction results in a borrow into the low-order half-byte of the result, then AF is set; otherwise AF is cleared. The auxiliary carry flag is provided for the decimal adjust instructions and ordinarily is not used for any other purpose.

- SF (sign flag): Arithmetic and logical instructions set the sign flag equal to the high-order bit (bit 7 or 15) of the result. For signed binary numbers, the sign flag will be 0 for positive results and 1 for negative results (so long as overflow does not occur). A conditional jump instruction can be used following addition or subtraction to alter the flow of the program depending on the sign of the result. Programs performing unsigned operations typically ignore SF since the high-order bit of the result is interpreted as a digit rather than a sign.

- ZF (zero flag): If the result of an arithmetic or logical operation is zero, then ZF is set; otherwise ZF is cleared. A conditional jump instruction can be used to alter the flow of the program if the result is or is not zero.

- 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. PF is provided for 8080/8085 compatibility; it also can be used to check ASCII characters for correct parity.

- OF (overflow flag): If the result of an operation is too large a positive number, or too small a negative number to fit in the destination operand (excluding the sign bit), then OF is set; otherwise OF is cleared. OF thus indicates signed arithmetic overflow; it can be tested with a conditional jump or the INTO (interrupt on overflow) instruction. OF may be ignored when performing unsigned arithmetic.

Addition

ADD *destination,source*

The sum of the two operands, which may be bytes or words, replaces the destination operand. Both operands may be signed or unsigned binary numbers (see AAA and DAA). ADD updates AF, CF, OF, PF, SF and ZF.

ADC *destination,source*

ADC (Add with Carry) sums the operands, which may be bytes or words, adds one if CF is set and replaces the destination operand with the result. Both operands may be signed or unsigned binary numbers (see AAA and DAA). ADC updates AF, CF, OF, PF, SF and ZF. Since ADC incorporates a carry from a previous operation, it can be used to write routines to add numbers longer than 16 bits.

INC *destination*

INC (Increment) adds one to the destination operand. The operand may be a byte or a word and is treated as an unsigned binary number (see AAA and DAA). INC updates AF, OF, PF, SF and ZF; it does not affect CF.

AAA

AAA (ASCII Adjust for Addition) changes the contents of register AL to a valid unpacked decimal number; the high-order half-byte is zeroed. AAA updates AF and CF; the content of OF, PF, SF and ZF is undefined following execution of AAA.

DAA

DAA (Decimal Adjust for Addition) corrects the result of previously adding two valid packed decimal operands (the destination operand must have been register AL). DAA changes the content of AL to a pair of valid packed decimal digits. It updates AF, CF, PF, SF and ZF; the content of OF is undefined following execution of DAA.

Subtraction

SUB *destination,source*

The source operand is subtracted from the destination operand, and the result replaces the destination operand. The operands may be bytes or words. Both operands may be signed or unsigned binary numbers (see AAS and DAS). SUB updates AF, CF, OF, PF, SF and ZF.

SBB *destination,source*

SBB (Subtract with Borrow) subtracts the source from the destination, subtracts one if CF is set, and returns the result to the destination operand. Both operands may be bytes or words. Both operands may be signed or unsigned binary numbers (see AAS and DAS). SBB updates AF, CF, OF, PF, SF and ZF. Since it incorporates a borrow from a previous operation, SBB may be used to write routines that subtract numbers longer than 16 bits.

DEC *destination*

DEC (Decrement) subtracts one from the destination, which may be a byte or a word. DEC updates AF, OF, PF, SF, and ZF; it does not affect CF.

NEG *destination*

NEG (Negate) subtracts the destination operand, which may be a byte or a word, from 0 and returns the result to the destination. This forms the two's complement of the number, effectively reversing the sign of an integer. If the operand is zero, its sign is not changed. Attempting to negate a byte containing −128 or a word containing −32,768 causes no change to the operand and sets OF. NEG updates AF, CF, OF, PF, SF and ZF. CF is always set except when the operand is zero, in which case it is cleared.

CMP *destination,source*

CMP (Compare) subtracts the source from the destination, which may be bytes or words, but does not return the result. The operands are unchanged, but the flags are updated and can be tested by a subsequent conditional jump instruction. CMP updates AF, CF, OF, PF, SF and ZF. The comparison reflected in the flags is that of the destination to the source. If a CMP instruction is followed by a JG (jump if greater) instruction, for example, the jump is taken if the destination operand is greater than the source operand.

AAS

AAS (ASCII Adjust for Subtraction) corrects the result of a previous subtraction of two valid unpacked decimal operands (the destination operand must have been specified as register AL). AAS changes the content of AL to a valid unpacked decimal number; the high-order half-byte is zeroed. AAS updates AF and CF; the content of OF, PF, SF and ZF is undefined following execution of AAS.

DAS

DAS (Decimal Adjust for Subtraction) corrects the result of a previous subtraction of two valid packed decimal operands (the destination operand must have been specified as register AL). DAS changes the content of AL to a pair of valid packed decimal digits. DAS updates AF, CF, PF, SF and ZF; the content of OF is undefined following execution of DAS.

Multiplication

MUL *source*

MUL (Multiply) performs an unsigned multiplication of the source operand and the accumulator. If the source is a byte, then it is multiplied by register AL, and the double-length

result is returned in AH and AL. If the source operand is a word, then it is multiplied by register AX, and the double-length result is returned in registers DX and AX. The operands are treated as unsigned binary numbers (see AAM). If the upper half of the result (AH for byte source, DX for word source) is nonzero, CF and OF are set; otherwise they are cleared. When CF and OF are set, they indicate that AH or DX contains significant digits of the result. The content of AF, PF, SF and ZF is undefined following execution of MUL.

IMUL *source*

IMUL (Integer Multiply) performs a signed multiplication of the source operand and the accumulator. If the source is a byte, then it is multiplied by register AL, and the double-length result is returned in AH and AL. If the source is a word, then it is multiplied by register AX, and the double-length result is returned in registers DX and AX. If the upper half of the result (AH for byte source, DX for word source) is not the sign extension of the lower half of the result, CF and OF are set; otherwise they are cleared. When CF and OF are set, they indicate that AH or DX contains significant digits of the result. The content of AF, PF, SF and ZF is undefined following execution of IMUL.

AAM

AAM (ASCII Adjust for Multiply) corrects the result of a previous multiplication of two valid unpacked decimal operands. A valid 2-digit unpacked decimal number is derived from the content of AH and AL and is returned to AH and AL. The high-order half-bytes of the multiplied operands must have been 0H for AAM to produce a correct result. AAM updates PF, SF and ZF; the content of AF, CF and OF is undefined following execution of AAM.

Division

DIV *source*

DIV (divide) performs an unsigned division of the accumulator (and its extension) by the source operand. If the source operand is a byte, it is divided into the double-length dividend assumed to be in registers AL and AH. The single-length quotient is returned in AL, and the single-length remainder is returned in AH. If the source operand is a word, it is divided into the double-length dividend in registers AX and DX. The single-length quotient is returned in AX, and the single-length remainder is returned in DX. If the quotient exceeds the capacity of its destination register (FFH for byte source, FFFFH for word source), as when division by zero is attempted, a type 0 interrupt is generated, and the quotient and remainder are undefined. Nonintegral quotients are truncated to integers. The content of AF, CF, OF, PF, SF and ZF is undefined following execution of DIV.

IDIV *source*

IDIV (Integer Divide) performs a signed division of the accumulator (and its extension) by the source operand. If the source operand is a byte, it is divided into the double-length dividend assumed to be in registers AL and AH; the single-length quotient is returned in AL, and the single-length remainder is returned in AH. For byte integer division, the maximum positive quotient is +127 (7FH) and the minimum negative quotient is −127 (81H). If the source operand is a word, it is divided into the double-length dividend in registers AX and DX; the single-length quotient is returned in AX, and the single-length remainder is returned in DX. For word integer division, the maximum positive quotient is +32,767 (7FFFH) and the minimum negative quotient is −32,767 (8001H). If the quotient is positive and exceeds the maximum, or is negative and is less than the minimum, the quotient and remainder are undefined, and a type 0 interrupt is generated. In particular, this occurs if division by 0 is attempted. Nonintegral quotients are truncated (toward 0) to integers, and the remainder has the same sign as the dividend. The content of AF, CF, OF, PF, SF and ZF is undefined following IDIV.

AAD

AAD (ASCII Adjust for Division) 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 subse-

quent 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. AAD updates PF, SF and ZF; the content of AF, CF and OF is undefined following execution of AAD.

CBW

CBW (Convert Byte to Word) 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 prior to performing byte division.

CWD

CWD (Convert Word to Doubleword) 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 (doubleword) dividend from a word prior to performing word division.

Bit Manipulation Instructions

The 8086 and 8088 provide three groups of instructions (table 2-11) for manipulating bits within both bytes and words: logical, shifts and rotates.

Table 2-11. Bit Manipulation Instructions

LOGICALS	
NOT	"Not" byte or word
AND	"And" byte or word
OR	"Inclusive or" byte or word
XOR	"Exclusive or" byte or word
TEST	"Test" byte or word
SHIFTS	
SHL/SAL	Shift logical/arithmetic left byte or word
SHR	Shift logical right byte or word
SAR	Shift arithmetic right byte or word
ROTATES	
ROL	Rotate left byte or word
ROR	Rotate right byte or word
RCL	Rotate through carry left byte or word
RCR	Rotate through carry right byte or word

Logical

The logical instructions include the boolean operators "not," "and," "inclusive or," and "exclusive or," plus a TEST instruction that sets the flags, but does not alter either of its operands.

AND, OR, XOR and TEST affect the flags as follows: The overflow (OF) and carry (CF) flags are always cleared by logical instructions, and the content of the auxiliary carry (AF) flag is always undefined following execution of a logical instruction. The sign (SF), zero (ZF) and parity (PF) flags are always posted to reflect the result of the operation and can be tested by conditional jump instructions. The interpretation of these flags is the same as for arithmetic instructions. SF is set if the result is negative (high-order bit is 1), and is cleared if the result is positive (high-order bit is 0). ZF is set if the result is zero, cleared otherwise. PF is set if the result contains an even number of 1-bits (has even parity) and is cleared if the number of 1-bits is odd (the result has odd parity). Note that NOT has no effect on the flags.

NOT *destination*

NOT inverts the bits (forms the one's complement) of the byte or word operand.

AND *destination,source*

AND performs the logical "and" of the two operands (byte or word) and returns the result to the destination operand. A bit in the result is set if both corresponding bits of the original operands are set; otherwise the bit is cleared.

OR *destination,source*

OR performs the logical "inclusive or" of the two operands (byte or word) and returns the result to the destination operand. A bit in the result is set if either or both corresponding bits in the original operands are set; otherwise the result bit is cleared.

XOR *destination,source*

XOR (Exclusive Or) performs the logical "exclusive or" of the two operands and returns the result to the destination operand. A bit in the

result is set if the corresponding bits of the original operands contain opposite values (one is set, the other is cleared); otherwise the result bit is cleared.

TEST *destination,source*

TEST performs the logical "and" of the two operands (byte or word), updates the flags, but does not return the result, i.e., neither operand is changed. If a TEST instruction is followed by a JNZ (jump if not zero) instruction, the jump will be taken if there are any corresponding 1-bits in both operands.

Shifts

The bits in bytes and words may be shifted arithmetically or logically. Up to 255 shifts may be performed, according to the value of the count operand coded in the instruction. The count may be specified as the constant 1, or as register CL, allowing the shift count to be a variable supplied at execution time. Arithmetic shifts may be used to multiply and divide binary numbers by powers of two (see note in description of SAR). Logical shifts can be used to isolate bits in bytes or words.

Shift instructions affect the flags as follows. AF is always undefined following a shift operation. PF, SF and ZF are updated normally, as in the logical instructions. CF always contains the value of the last bit shifted out of the destination operand. The content of OF is always undefined following a multibit shift. In a single-bit shift, OF is set if the value of the high-order (sign) bit was changed by the operation; if the sign bit retains its original value, OF is cleared.

SHL/SAL *destination,count*

SHL and SAL (Shift Logical Left and Shift Arithmetic Left) perform the same operation and are physically the same instruction. The destination byte or word is shifted left by the number of bits specified in the count operand. Zeros are shifted in on the right. If the sign bit retains its original value, then OF is cleared.

SHR *destination,source*

SHR (Shift Logical Right) shifts the bits in the destination operand (byte or word) to the right by the number of bits specified in the count operand. Zeros are shifted in on the left. If the sign bit retains its original value, then OF is cleared.

SAR *destination,count*

SAR (Shift Arithmetic Right) 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 on the left, preserving the sign of the original value. Note that SAR does not produce the same result as the dividend of an "equivalent" IDIV instruction if the destination operand is negative and 1-bits are shifted out. For example, shifting −5 right by one bit yields −3, while integer division of −5 by 2 yields −2. The difference in the instructions is that IDIV truncates all numbers toward zero, while SAR truncates positive numbers toward zero and negative numbers toward negative infinity.

Rotates

Bits in bytes and words also may be rotated. Bits rotated out of an operand are not lost as in a shift, but are "circled" back into the other "end" of the operand. As in the shift instructions, the number of bits to be rotated is taken from the count operand, which may specify either a constant of 1, or the CL register. The carry flag may act as an extension of the operand in two of the rotate instructions, allowing a bit to be isolated in CF and then tested by a JC (jump if carry) or JNC (jump if not carry) instruction.

Rotates affect only the carry and overflow flags. CF always contains the value of the last bit rotated out. On multibit rotates, the value of OF is always undefined. In single-bit rotates, OF is set if the operation changes the high-order (sign) bit of the destination operand. If the sign bit retains its original value, OF is cleared.

ROL *destination,count*

ROL (Rotate Left) rotates the destination byte or word left by the number of bits specified in the count operand.

ROR *destination,count*

ROR (Rotate Right) operates similar to ROL except that the bits in the destination byte or word are rotated right instead of left.

RCL *destination,count*

RCL (Rotate through Carry Left) rotates the bits in the byte or word destination operand to the left by the number of bits specified in the count operand. The carry flag (CF) is treated as "part of" the destination operand; that is, its value is rotated into the low-order bit of the destination, and itself is replaced by the high-order bit of the destination.

RCR *destination,count*

RCR (Rotate through Carry Right) operates exactly like RCL except that the bits are rotated right instead of left.

String Instructions

Five basic string operations, called primitives, allow strings of bytes or words to be operated on, one element (byte or word) at a time. Strings of up to 64k bytes may be manipulated with these instructions. Instructions are available to move, compare and scan for a value, as well as for moving string elements to and from the accumulator (see table 2-12). These basic operations may be preceded by a special one-byte prefix that causes the instruction to be repeated by the hardware, allowing long strings to be processed much faster than would be possible with a software loop. The repetitions can be terminated by a variety of conditions, and a repeated operation may be interrupted and resumed.

The string instructions operate quite similarly in many respects; the common characteristics are covered here and in table 2-13 and figure 2-33 rather than in the descriptions of the individual instructions. A string instruction may have a source operand, a destination operand, or both. The hardware assumes that a source string resides in the current data segment; a segment prefix byte may be used to override this assumption. A destination string must be in the current extra segment. The assembler checks the attributes of the operands to determine if the elements of the strings are bytes or words. The assembler does not, however, use the operand names to address the strings. Rather, the content of register SI (source index) is used as an offset to address the current element of the source string, and the content of register DI (destination index) is taken as the offset of the current destination string element. These registers must be initialized to point to the source/destination strings before executing the string instruction; the LDS, LES and LEA instructions are useful in this regard.

Table 2-12. String Instructions

REP	Repeat
REPE/REPZ	Repeat while equal/zero
REPNE/REPNZ	Repeat while not equal/not zero
MOVS	Move byte or word string
MOVSB/MOVSW	Move byte or word string
CMPS	Compare byte or word string
SCAS	Scan byte or word string
LODS	Load byte or word string
STOS	Store byte or word string

Table 2-13. String Instruction Register and Flag Use

SI	Index (offset) for source string
DI	Index (offset) for destination string
CX	Repetition counter
AL/AX	Scan value Destination for LODS Source for STOS
DF	0 = auto-increment SI, DI 1 = auto-decrement SI, DI
ZF	Scan/compare terminator

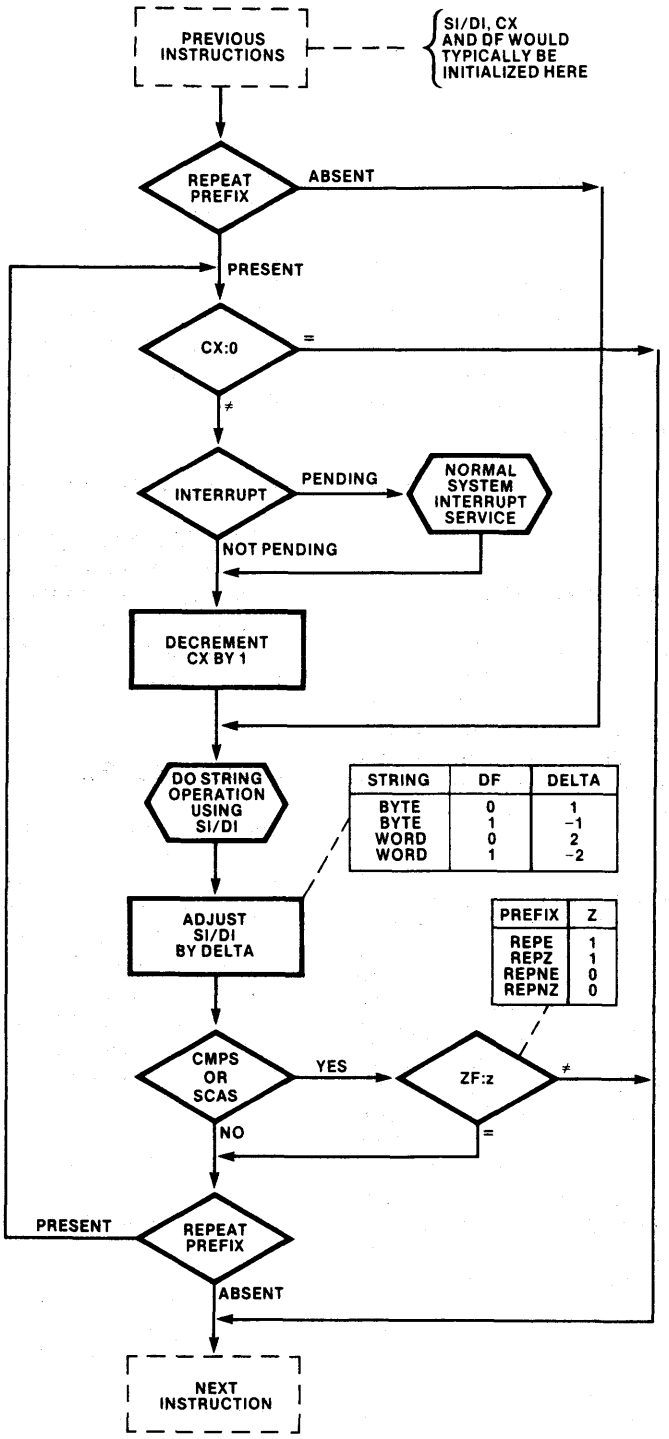


Figure 2-33. String Operation Flow

The string instructions automatically update SI and/or DI in anticipation of processing the next string element. The setting of DF (the direction flag) determines whether the index registers are auto-incremented (DF = 0) or auto-decremented (DF = 1). If byte strings are being processed, SI and/or DI is adjusted by1; the adjustment is 2 for word strings.

If a Repeat prefix has been coded, then register CX (count register) is decremented by 1 after each repetition of the string instruction; therefore, CX must be initialized to the number of repetitions desired before the string instruction is executed. If CX is 0, the string instruction is not executed, and control goes to the following instruction.

Section 2.10 contains examples that illustrate the use of all the string instructions.

REP/REPE/REPZ/REPNE/REPNZ

Repeat, Repeat While Equal, Repeat While Zero, Repeat While Not Equal and Repeat While Not Zero are five mnemonics for two forms of the prefix byte that controls repetition of a subsequent string instruction. The different mnemonics are provided to improve program clarity. The repeat prefixes do not affect the flags.

REP is used in conjunction with the MOVS (Move String) and STOS (Store String) instructions and is interpreted as "repeat while not end-of-string" (CX not 0). REPE and REPZ operate identically and are physically the same prefix byte as REP. These instructions are used with the CMPS (Compare String) and SCAS (Scan String) instructions and require ZF (posted by these instructions) to be set before initiating the next repetition. REPNE and REPNZ are two mnemonics for the same prefix byte. These instructions function the same as REPE and REPZ except that the zero flag must be cleared or the repetition is terminated. Note that ZF does not need to be initialized before executing the repeated string instruction.

Repeated string sequences are interruptable; the processor will recognize the interrupt before processing the next string element. System interrupt processing is not affected in any way. Upon return from the interrupt, the repeated operation is resumed from the point of interruption. Note, however, that execution does *not* resume properly if a second or third prefix (i.e., segment override or LOCK) has been specified in addition to any of the repeat prefixes. The processor "remembers" only one prefix in effect at the time of the interrupt, the prefix that immediately precedes the string instruction. After returning from the interrupt, processing resumes at this point, but any additional prefixes specified are not in effect. If more than one prefix must be used with a string instruction, interrupts may be disabled for the duration of the repeated execution. However, this will not prevent a non-maskable interrupt from being recognized. Also, the time that the system is unable to respond to interrupts may be unacceptable if long strings are being processed.

MOVS *destination-string,source-string*

MOVS (Move String) transfers a byte or a word from the source string (addressed by SI) to the destination string (addressed by DI) and updates SI and DI to point to the next string element. When used in conjunction with REP, MOVS performs a memory-to-memory block transfer.

MOVSB/MOVSW

These are alternate mnemonics for the move string instruction. These mnemonics are coded without operands; they explicitly tell the assembler that a byte string (MOVSB) or a word string (MOVSW) is to be moved (when MOVS is coded, the assembler determines the string type from the attributes of the operands). These mnemonics are useful when the assembler cannot determine the attributes of a string, e.g., a section of code is being moved.

CMPS *destination-string,source-string*

CMPS (Compare String) subtracts the destination byte or word (addressed by DI) from the source byte or word (addressed by SI). CMPS affects the flags but does not alter either operand, updates SI and DI to point to the next string element and updates AF, CF, OF, PF, SF and ZF to reflect the relationship of the destination element to the source element. For example, if a JG (Jump if Greater) instruction follows CMPS, the jump is taken if the destination element is greater than the source element. If CMPS is prefixed with REPE

or REPZ, the operation is interpreted as "compare while not end-of-string (CX not zero) and strings are equal (ZF = 1)." If CMPS is preceded by REPNE or REPNZ, the operation is interpreted as "compare while not end-of-string (CX not zero) and strings are not equal (ZF = 0)." Thus, CMPS can be used to find matching or differing string elements.

SCAS *destination-string*

SCAS (Scan String) subtracts the destination string element (byte or word) addressed by DI from the content of AL (byte string) or AX (word string) and updates the flags, but does not alter the destination string or the accumulator. SCAS also updates DI to point to the next string element and AF, CF, OF, PF, SF and ZF to reflect the relationship of the scan value in AL/AX to the string element. If SCAS is prefixed with REPE or REPZ, the operation is interpreted as "scan while not end-of-string (CX not 0) and string-element = scan-value (ZF = 1)." This form may be used to scan for departure from a given value. If SCAS is prefixed with REPNE or REPNZ, the operation is interpreted as "scan while not end-of-string (CX not 0) and string-element is not equal to scan-value (ZF = 0)." This form may be used to locate a value in a string.

LODS *source-string*

LODS (Load String) transfers the byte or word string element addressed by SI to register AL or AX, and updates SI to point to the next element in the string. This instruction is not ordinarily repeated since the accumulator would be overwritten by each repetition, and only the last element would be retained. However, LODS is very useful in software loops as part of a more complex string function built up from string primitives and other instructions.

STOS *destination-string*

STOS (Store String) transfers a byte or word from register AL or AX to the string element addressed by DI and updates DI to point to the next location in the string. As a repeated operation, STOS provides a convenient way to initialize a string to a constant value (e.g., to blank out a print line).

Program Transfer Instructions

The sequence of execution of instructions in an 8086/8088 program is determined by the content of the code segment register (CS) and the instruction pointer (IP). The CS register contains the base address of the current code segment, the 64k portion of memory from which instructions are presently being fetched. The IP is used as an offset from the beginning of the code segment; the combination of CS and IP points to the memory location from which the next instruction is to be fetched. (Recall that under most operating conditions, the next instruction to be *executed* has already been fetched from memory and is waiting in the CPU instruction queue.) The program transfer instructions operate on the instruction pointer and on the CS register; changing the content of these causes normal sequential execution to be altered. When a program transfer occurs, the queue no longer contains the correct instruction, and the BIU obtains the next instruction from memory using the new IP and CS values, passes the instruction directly to the EU, and then begins refilling the queue from the new location.

Four groups of program transfers are available in the 8086/8088 (see table 2-14): unconditional transfers, conditional transfers, iteration control instructions and interrupt-related instructions. Only the interrupt-related instructions affect any CPU flags. As will be seen, however, the execution of many of the program transfer instructions is affected by the states of the flags.

Unconditional Transfers

The unconditional transfer instructions may transfer control to a target instruction within the current code segment (intrasegment transfer) or to a different code segment (intersegment transfer). (The ASM-86 assembler terms an intrasegment target NEAR and an intersegment target FAR.) The transfer is made unconditionally any time the instruction is executed.

CALL *procedure-name*

CALL activates an out-of-line procedure, saving information on the stack to permit a RET (return) instruction in the procedure to transfer control back to the instruction following the CALL. The

Table 2-14. Program Transfer Instructions

UNCONDITIONAL TRANSFERS	
CALL	Call procedure
RET	Return from procedure
JMP	Jump
CONDITIONAL TRANSFERS	
JA/JNBE	Jump if above/not below nor equal
JAE/JNB	Jump if above or equal/not below
JB/JNAE	Jump if below/not above nor equal
JBE/JNA	Jump if below or equal/not above
JC	Jump if carry
JE/JZ	Jump if equal/zero
JG/JNLE	Jump if greater/not less nor equal
JGE/JNL	Jump if greater or equal/not less
JL/JNGE	Jump if less/not greater nor equal
JLE/JNG	Jump if less or equal/not greater
JNC	Jump if not carry
JNE/JNZ	Jump if not equal/not zero
JNO	Jump if not overflow
JNP/JPO	Jump if not parity/parity odd
JNS	Jump if not sign
JO	Jump if overflow
JP/JPE	Jump if parity/parity even
JS	Jump if sign
ITERATION CONTROLS	
LOOP	Loop
LOOPE/LOOPZ	Loop if equal/zero
LOOPNE/LOOPNZ	Loop if not equal/not zero
JCXZ	Jump if register CX = 0
INTERRUPTS	
INT	Interrupt
INTO	Interrupt if overflow
IRET	Interrupt return

assembler generates a different type of CALL instruction depending on whether the programmer has defined the procedure name as NEAR or FAR. For control to return properly, the type of CALL instruction must match the type of RET instruction that exits from the procedure. (The potential for a mismatch exists if the procedure and the CALL are contained in separately assembled programs.) Different forms of the CALL instruction allow the address of the target procedure to be obtained from the instruction itself (direct CALL) or from a memory location or register referenced by the instruction (indirect CALL). In the following descriptions, bear in mind that the processor automatically adjusts IP to point to the next instruction to be *executed* before saving it on the stack.

For an intrasegment direct CALL, SP (the stack pointer) is decremented by two and IP is pushed onto the stack. The relative displacement (up to ±32k) of the target procedure from the CALL instruction is then added to the instruction pointer. This form of the CALL instruction is "self-relative" and is appropriate for position-independent (dynamically relocatable) routines in which the CALL and its target are in the same segment and are moved together.

An intrasegment indirect CALL may be made through memory or through a register. SP is decremented by two and IP is pushed onto the stack. The offset of the target procedure is obtained from the memory word or 16-bit general register referenced in the instruction and replaces IP.

For an intersegment direct CALL, SP is decremented by two, and CS is pushed onto the stack. CS is replaced by the segment word contained in the instruction. SP again is decremented by two. IP is pushed onto the stack and is replaced by the offset word contained in the instruction.

For an intersegment indirect CALL (which only may be made through memory), SP is decremented by two, and CS is pushed onto the stack. CS is then replaced by the content of the second word of the doubleword memory pointer referenced by the instruction. SP again is decremented by two, and IP is pushed onto the stack and is replaced by the content of the first word of the doubleword pointer referenced by the instruction.

RET *optional-pop-value*

RET (Return) transfers control from a procedure back to the instruction following the CALL that activated the procedure. The assembler generates an intrasegment RET if the programmer has defined the procedure NEAR, or an intersegment RET if the procedure has been defined as FAR. RET pops the word at the top of the stack (pointed to by register SP) into the instruction pointer and increments SP by two. If RET is intersegment, the word at the new top of stack is popped into the CS register, and SP is again incremented by two. If an optional pop value has been specified, RET adds that value to SP. This feature may be used to discard parameters pushed onto the stack before the execution of the CALL instruction.

JMP *target*

JMP unconditionally transfers control to the target location. Unlike a CALL instruction, JMP does not save any information on the stack, and no return to the instruction following the JMP is expected. Like CALL, the address of the target operand may be obtained from the instruction itself (direct JMP) or from memory or a register referenced by the instruction (indirect JMP).

An intrasegment direct JMP changes the instruction pointer by adding the relative displacement of the target from the JMP instruction. If the assembler can determine that the target is within 127 bytes of the JMP, it automatically generates a two-byte form of this instruction called a SHORT JMP; otherwise, it generates a NEAR JMP that can address a target within ±32k. Intrasegment direct JMPS are self-relative and are appropriate in position-independent (dynamically relocatable) routines in which the JMP and its target are in the same segment and are moved together.

An intrasegment indirect JMP may be made either through memory or through a 16-bit general register. In the first case, the content of the word referenced by the instruction replaces the instruction pointer. In the second case, the new IP value is taken from the register named in the instruction.

An intersegment direct JMP replaces IP and CS with values contained in the instruction.

An intersegment indirect JMP may be made only through memory. The first word of the doubleword pointer referenced by the instruction replaces IP, and the second word replaces CS.

Conditional Transfers

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 is executed. These 18 instructions (see table 2-15) each test a different combination of flags for a condition. If the condition is "true," then control is transferred to the target specified in the instruction. If the condition is "false," then control passes to the instruction that follows the conditional jump. All conditional jumps are SHORT, that is, the target must be in the current code segment and within −128 to +127 bytes of the first byte of the next instruction (JMP 00H jumps to the first byte of the next instruction). Since the jump is made by adding the relative displacement of the target to the instruction pointer, all conditional jumps are self-relative and are appropriate for position-independent routines.

Iteration Control

The iteration control instructions can be used to regulate the repetition of software loops. These instructions use the CX register as a counter. Like the conditional transfers, the iteration control instructions are self-relative and may only transfer to targets that are within −128 to +127 bytes of themselves, i.e., they are SHORT transfers.

LOOP *short-label*

LOOP decrements CX by 1 and transfers control to the target operand if CX is not 0; otherwise the instruction following LOOP is executed.

LOOPE/LOOPZ *short-label*

LOOPE and LOOPZ (Loop While Equal and Loop While Zero) are different mnemonics for the same instruction (similar to the REPE and

Table 2-15. Interpretation of 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
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
JNC	CF=0	not carry
JNE/JNZ	ZF=0	not equal/not zero
JNO	OF=0	not overflow
JNP/JPO	PF=0	not parity/parity odd
JNS	SF=0	not sign
JO	OF=1	overflow
JP/JPE	PF=1	parity/parity equal
JS	SF=1	sign

Note: "above" and "below" refer to the relationship of two unsigned values; "greater" and "less" refer to the relationship of two signed values.

REPZ repeat prefixes). CX is decremented by 1, and control is transferred to the target operand if CX is not 0 and if ZF is set; otherwise the instruction following LOOPE/LOOPZ is executed.

LOOPNE/LOOPNZ *short-label*

LOOPNE and LOOPNZ (Loop While Not Equal and Loop While Not Zero) are also synonyms for the same instruction. CX is decremented by 1, and control is transferred to the target operand if CX is not 0 and if ZF is clear; otherwise the next sequential instruction is executed.

JCXZ *short-label*

JCXZ (Jump If CX Zero) transfers control to the target operand if CX is 0. This instruction is useful at the beginning of a loop to bypass the loop if CX has a zero value, i.e., to execute the loop zero times.

Interrupt Instructions

The interrupt instructions allow interrupt service routines to be activated by programs as well as by external hardware devices. The effect of software interrupts is similar to hardware-initiated interrupts. However, the processor does not execute an interrupt acknowledge bus cycle it the interrupt originates in software or with an NMI. The effect of the interrupt instructions on the flags is covered in the description of each instruction.

INT *interrupt-type*

INT (Interrupt) activates the interrupt procedure specified by the interrupt-type operand. INT decrements the stack pointer by two, pushes the flags onto the stack, and clears the trap (TF) and interrupt-enable (IF) flags to disable single-step and maskable interrupts. The flags are stored in the format used by the PUSHF instruction. SP is decremented again by two, and the CS register is pushed onto the stack. The address of the interrupt pointer is calculated by multiplying interrupt-type by four; the second word of the interrupt pointer replaces CS. SP again is decremented by two, and IP is pushed onto the stack and is replaced by the first word of the interrupt pointer. If interrupt-type = 3, the assembler generates a short (1 byte) form of the instruction, known as the breakpoint interrupt.

Software interrupts can be used as "supervisor calls," i.e., requests for service from an operating system. A different interrupt-type can be used for each type of service that the operating system could supply for an application program. Software interrupts also may be used to check out interrupt service procedures written for hardware-initiated interrupts.

INTO

INTO (Interrupt on Overflow) generates a software interrupt if the overflow flag (OF) is set; otherwise control proceeds to the following instruction without activating an interrupt procedure. INTO addresses the target interrupt procedure (its type is 4) through the interrupt pointer at location 10H; it clears the TF and IF flags and otherwise operates like INT. INTO may be written following an arithmetic or logical operation to activate an interrupt procedure if overflow occurs.

IRET

IRET (Interrupt Return) transfers control back to the point of interruption by popping IP, CS and the flags from the stack. IRET thus affects all flags by restoring them to previously saved values. IRET is used to exit any interrupt procedure, whether activated by hardware or software.

Processor Control Instructions

These instructions (see table 2-16) allow programs to control various CPU functions. One group of instructions updates flags, and another group is used primarily for synchronizing the 8086 or 8088 with external events. A final instruction causes the CPU to do nothing. Except for the flag operations, none of the processor control instructions affect the flags.

Flag Operations

CLC

CLC (Clear Carry flag) zeroes the carry flag (CF) and affects no other flags. It (and CMC and STC) is useful in conjunction with the RCL and RCR instructions.

Table 2-16. Processor Control Instructions

FLAG OPERATIONS	
STC	Set carry flag
CLC	Clear carry flag
CMC	Complement carry flag
STD	Set direction flag
CLD	Clear direction flag
STI	Set interrupt enable flag
CLI	Clear interrupt enable flag
EXTERNAL SYNCHRONIZATION	
HLT	Halt until interrupt or reset
WAIT	Wait for $\overline{\text{TEST}}$ pin active
ESC	Escape to external processor
LOCK	Lock bus during next instruction
NO OPERATION	
NOP	No operation

CMC

CMC (Complement Carry flag) "toggles" CF to its opposite state and affects no other flags.

STC

STC (Set Carry flag) sets CF to 1 and affects no other flags.

CLD

CLD (Clear Direction flag) zeroes DF causing the string instructions to auto-increment the SI and/or DI index registers. CLD does not affect any other flags.

STD

STD (Set Direction flag) sets DF to 1 causing the string instructions to auto-decrement the SI and/or DI index registers. STD does not affect any other flags.

CLI

CLI (Clear Interrupt-enable flag) zeroes IF. When the interrupt-enable flag is cleared, the 8086 and 8088 do not recognize an external interrupt request that appears on the INTR line; in other words maskable interrupts are disabled. A non-maskable interrupt appearing on the NMI line, however, is honored, as is a software interrupt. CLI does not affect any other flags.

STI

STI (Set Interrupt-enable flag) sets IF to 1, enabling processor recognition of maskable interrupt requests appearing on the INTR line. Note however, that a pending interrupt will not actually be recognized until the instruction following STI has executed. STI does not affect any other flags.

External Synchronization

HLT

HLT (Halt) causes the 8086/8088 to enter the halt state. The processor leaves the halt state upon activation of the RESET line, upon receipt of a non-maskable interrupt request on NMI, or, if interrupts are enabled, upon receipt of a maskable interrupt request on INTR. HLT does not affect any flags. It may be used as an alternative to an endless software loop in situations where a program must wait for an interrupt.

WAIT

WAIT causes the CPU to enter the wait state while its $\overline{\text{TEST}}$ line is not active. WAIT does not affect any flags. This instruction is described more completely in section 2.5.

ESC *external-opcode, source*

ESC (Escape) provides a means for an external processor to obtain an opcode and possibly a memory operand from the 8086 or 8088. The external opcode is a 6-bit immediate constant that the assembler encodes in the machine instruction it builds (see table 2-26). An external processor may monitor the system bus and capture this opcode when the ESC is fetched. If the source operand is a register, the processor does nothing. If the source operand is a memory variable, the processor obtains the operand from memory and discards it. An external processor may capture the memory operand when the processor reads it from memory.

LOCK

LOCK is a one-byte prefix that causes the 8086/8088 (configured in maximum mode) to assert its bus $\overline{\text{LOCK}}$ signal while the following instruction executes. LOCK does not affect any flags. See section 2.5 for more information on LOCK.

No Operation

NOP

NOP (No Operation) causes the CPU to do nothing. NOP does not affect any flags.

Instruction Set Reference Information

Table 2-21 provides detailed operational information for the 8086/8088 instruction set. The information is presented from the point of view of utility to the assembly language programmer. Tables 2-17, 2-18 and 2-19 explain the symbols used in table 2-21. Machine language instruction encoding and decoding information is given in Chapter 4.

Instruction timings are presented as the number of clock periods required to execute a particular form (register-to-register, immediate-to-memory, etc.) of the instruction. If a system is running with a 5 MHz maximum clock, the maximum clock period is 200 ns; at 8 MHz, the clock period is 125 ns. Where memory operands are used, "+EA" denotes a variable number of additional clock periods needed to calculate the operand's effective address (discussed in section 2.8). Table 2-20 lists all effective address calculation times.

Table 2-17. Key to Instruction Coding Formats

IDENTIFIER	USED IN	EXPLANATION
destination	data transfer, bit manipulation	A register or memory location that may contain data operated on by the instruction, and which receives (is replaced by) the result of the operation.
source	data transfer, arithmetic, bit manipulation	A register, memory location or immediate value that is used in the operation, but is not altered by the instruction.
source-table	XLAT	Name of memory translation table addressed by register BX.
target	JMP, CALL	A label to which control is to be transferred directly, or a register or memory location whose *content* is the address of the location to which control is to be transferred indirectly.
short-label	cond. transfer, iteration control	A label to which control is to be conditionally transferred; must lie within −128 to +127 bytes of the first byte of the next instruction.
accumulator	IN, OUT	Register AX for word transfers, AL for bytes.
port	IN, OUT	An I/O port number; specified as an immediate value of 0-255, or register DX (which contains port number in range 0-64k).
source-string	string ops.	Name of a string in memory that is addressed by register SI; used only to identify string as byte or word and specify segment override, if any. This string is used in the operation, but is not altered.
dest-string	string ops.	Name of string in memory that is addressed by register DI; used only to identify string as byte or word. This string receives (is replaced by) the result of the operation.
count	shifts, rotates	Specifies number of bits to shift or rotate; written as immediate value 1 or register CL (which contains the count in the range 0-255).
interrupt-type	INT	Immediate value of 0-255 identifying interrupt pointer number.
optional-pop-value	RET	Number of bytes (0-64k, ordinarily an even number) to discard from stack.
external-opcode	ESC	Immediate value (0-63) that is encoded in the instruction for use by an external processor.

Table 2-18. Key to Flag Effects

IDENTIFIER	EXPLANATION
(blank)	not altered
0	cleared to 0
1	set to 1
X	set or cleared according to result
U	undefined—contains no reliable value
R	restored from previously-saved value

For control transfer instructions, the timings given include any additional clocks required to reinitialize the instruction queue as well as the time required to fetch the target instruction. For instructions executing on an 8086, four clocks should be added for each instruction reference to a word operand located at an odd memory address to reflect any additional operand bus cycles required. Similarly for instructions executing on an 8088, four clocks should be added to each instruction reference to a 16-bit memory operand; this includes all stack operations. The required number of data references is listed in table 2-21 for each instruction to aid in this calculation.

Several additional factors can increase actual execution time over the figures shown in table 2-21. The time provided assumes that the instruction has already been prefetched and that it is waiting in the instruction queue, an assumption that is valid under most, but not all, operating conditions. A series of fast executing (fewer than two clocks per opcode byte) instructions can drain the queue and increase execution time. Execution time also is slightly impacted by the interaction of the EU and BIU when memory operands must be read or written. If the EU needs access to memory, it may have to wait for up to one clock if the BIU has already started an instruction fetch bus cycle. (The EU can detect the need for a memory operand and post a bus request far enough in advance of its need for this operand to avoid waiting a full 4-clock bus cycle). Of course the EU does not have to wait if the queue is full, because the BIU is idle. (This discussion assumes

Table 2-19. Key to Operand Types

IDENTIFIER	EXPLANATION
(no operands)	No operands are written
register	An 8- or 16-bit general register
reg 16	A 16-bit general register
seg-reg	A segment register
accumulator	Register AX or AL
immediate	A constant in the range 0-FFFFH
immed8	A constant in the range 0-FFH
memory	An 8- or 16-bit memory location(1)
mem8	An 8-bit memory location(1)
mem16	A 16-bit memory location(1)
source-table	Name of 256-byte translate table
source-string	Name of string addressed by register SI
dest-string	Name of string addressed by register DI
DX	Register DX
short-label	A label within −128 to +127 bytes of the end of the instruction
near-label	A label in current code segment
far-label	A label in another code segment
near-proc	A procedure in current code segment
far-proc	A procedure in another code segment
memptr16	A word containing the offset of the location in the current code segment to which control is to be transferred(1)
memptr32	A doubleword containing the offset and the segment base address of the location in another code segment to which control is to be transferred(1)
regptr16	A 16-bit general register containing the offset of the location in the current code segment to which control is to be transferred
repeat	A string instruction repeat prefix

(1)Any addressing mode—direct, register indirect, based, indexed, or based indexed—may be used (see section 2.8).

Table 2-20. Effective Address Calculation Time

EA COMPONENTS		CLOCKS*
Displacement Only		6
Base or Index Only	(BX,BP,SI,DI)	5
Displacement + Base or Index	(BX,BP,SI,DI)	9
Base + Index	BP + DI, BX + SI	7
	BP + SI, BX + DI	8
Displacement + Base + Index	BP + DI + DISP BX + SI + DISP	11
	BP + SI + DISP BX + DI + DISP	12

*Add 2 clocks for segment override

that the BIU can obtain the bus on demand, i.e., that no other processors are competing for the bus.)

With typical instruction mixes, the time actually required to execute a sequence of instructions will typically be within 5-10% of the sum of the individual timings given in table 2-21. Cases can be constructed, however, in which execution time may be much higher than the sum of the figures provided in the table. The execution time for a given sequence of instructions, however, is always repeatable, assuming comparable external conditions (interrupts, coprocessor activity, etc.). If the execution time for a given series of instructions must be determined exactly, the instructions should be run on an execution vehicle such as the SDK-86 or the iSBC 86/12™ board.

Table 2-21. Instruction Set Reference Data

AAA	**AAA** (no operands) ASCII adjust for addition			**Flags** O D I T S Z A P C U U U X U X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	4	—	1	AAA

AAD	**AAD** (no operands) ASCII adjust for division			**Flags** O D I T S Z A P C U X X U X U
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	60	—	2	AAD

AAM	**AAM** (no operands) ASCII adjust for multiply			**Flags** O D I T S Z A P C U X X U X U
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	83	—	1	AAM

AAS	**AAS** (no operands) ASCII adjust for subtraction			**Flags** O D I T S Z A P C U U U X U X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	4	—	1	AAS

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Table 2-21. Instruction Set Reference Data (Cont'd.)

ADC	**ADC** destination,source Add with carry			**Flags** O D I T S Z A P C X X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, register	3	—	2	ADC AX, SI
register, memory	9+EA	1	2-4	ADC DX, BETA [SI]
memory, register	16+EA	2	2-4	ADC ALPHA [BX] [SI], DI
register, immediate	4	—	3-4	ADC BX, 256
memory, immediate	17+EA	2	3-6	ADC GAMMA, 30H
accumulator, immediate	4	—	2-3	ADC AL, 5

ADD	**ADD** destination,source Addition			**Flags** O D I T S Z A P C X X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, register	3	—	2	ADD CX, DX
register, memory	9+EA	1	2-4	ADD DI, [BX].ALPHA
memory, register	16+EA	2	2-4	ADD TEMP, CL
register, immediate	4	—	3-4	ADD CL, 2
memory, immediate	17+EA	2	3-6	ADD ALPHA, 2
accumulator, immediate	4	—	2-3	ADD AX, 200

AND	**AND** destination,source Logical and			**Flags** O D I T S Z A P C 0 X X U X 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, register	3	—	2	AND AL,BL
register, memory	9+EA	1	2-4	AND CX,FLAG_WORD
memory, register	16+EA	2	2-4	AND ASCII [DI],AL
register, immediate	4	—	3-4	AND CX,0F0H
memory, immediate	17+EA	2	3-6	AND BETA, 01H
accumulator, immediate	4	—	2-3	AND AX, 01010000B

CALL	**CALL** target Call a procedure			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Examples**
near-proc	19	1	3	CALL NEAR_PROC
far-proc	28	2	5	CALL FAR_PROC
memptr 16	21+EA	2	2-4	CALL PROC_TABLE [SI]
regptr 16	16	1	2	CALL AX
memptr 32	37+EA	4	2-4	CALL [BX].TASK [SI]

CBW	**CBW** (no operands) Convert byte to word			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	CBW

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Table 2-21. Instruction Set Reference Data (Cont'd.)

CLC	CLC (no operands) Clear carry flag			Flags O D I T S Z A P C 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	CLC

CLD	CLD (no operands) Clear direction flag			Flags O D I T S Z A P C 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	CLD

CLI	CLI (no operands) Clear interrupt flag			Flags O D I T S Z A P C 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	CLI

CMC	CMC (no operands) Complement carry flag			Flags O D I T S Z A P C X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	CMC

CMP	CMP destination,source Compare destination to source			Flags O D I T S Z A P C X X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, register	3	—	2	CMP BX, CX
register, memory	9+EA	1	2-4	CMP DH, ALPHA
memory, register	9+EA	1	2-4	CMP [BP+2], SI
register, immediate	4	—	3-4	CMP BL, 02H
memory, immediate	10+EA	1	3-6	CMP [BX].RADAR [DI], 3420H
accumulator, immediate	4	—	2-3	CMP AL, 00010000B

CMPS	CMPS dest-string,source-string Compare string			Flags O D I T S Z A P C X X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
dest-string, source-string	22	2	1	CMPS BUFF1, BUFF2
(repeat) dest-string, source-string	9+22/rep	2/rep	1	REPE CMPS ID, KEY

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Table 2-21. Instruction Set Reference Data (Cont'd.)

CWD	CWD (no operands) Convert word to doubleword			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	5	—	1	CWD

DAA	DAA (no operands) Decimal adjust for addition			Flags O D I T S Z A P C X X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	4	—	1	DAA

DAS	DAS (no operands) Decimal adjust for subtraction			Flags O D I T S Z A P C U X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	4	—	1	DAS

DEC	DEC destination Decrement by 1			Flags O D I T S Z A P C X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
reg16	2	—	1	DEC AX
reg8	3	—	2	DEC AL
memory	15+EA	2	2-4	DEC ARRAY [SI]

DIV	DIV source Division, unsigned			Flags O D I T S Z A P C U U U U U U
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
reg8	80-90	—	2	DIV CL
reg16	144-162	—	2	DIV BX
mem8	(86-96) +EA	1	2-4	DIV ALPHA
mem16	(150-168) +EA	1	2-4	DIV TABLE [SI]

ESC	ESC external-opcode,source Escape			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
immediate, memory	8+EA	1	2-4	ESC 6,ARRAY [SI]
immediate, register	2	—	2	ESC 20,AL

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Table 2-21. Instruction Set Reference Data (Cont'd.)

HLT	HLT (no operands) Halt			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	HLT

IDIV	IDIV source Integer division			Flags O D I T S Z A P C U U U U U U
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
reg8	101-112	—	2	IDIV BL
reg16	165-184	—	2	IDIV CX
mem8	(107-118) +EA	1	2-4	IDIV DIVISOR_BYTE [SI]
mem16	(171-190) +EA	1	2-4	IDIV [BX].DIVISOR_WORD

IMUL	IMUL source Integer multiplication			Flags O D I T S Z A P C X U U U U X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
reg8	80-98	—	2	IMUL CL
reg16	128-154	—	2	IMUL BX
mem8	(86-104) +EA	1	2-4	IMUL RATE_BYTE
mem16	(134-160) +EA	1	2-4	IMUL RATE_WORD [BP] [DI]

IN	IN accumulator,port Input byte or word			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
accumulator, immed8	10	1	2	IN AL, 0FFEAH
accumulator, DX	8	1	1	IN AX, DX

INC	INC destination Increment by 1			Flags O D I T S Z A P C X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
reg16	2	—	1	INC CX
reg8	3	—	2	INC BL
memory	15+EA	2	2-4	INC ALPHA [DI] [BX]

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Table 2-21. Instruction Set Reference Data (Cont'd.)

INT	**INT** interrupt-type Interrupt			**Flags** O D I T S Z A P C 0 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
immed8 (type = 3) immed8 (type ≠ 3)	52 51	5 5	1 2	INT 3 INT 67

INTR†	**INTR** (external maskable interrupt) Interrupt if INTR and IF=1			**Flags** O D I T S Z A P C 0 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	61	7	N/A	N/A

INTO	**INTO** (no operands) Interrupt if overflow			**Flags** O D I T S Z A P C 0 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	53 or 4	5	1	INTO

IRET	**IRET** (no operands) Interrupt Return			**Flags** O D I T S Z A P C R R R R R R R R
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	24	3	1	IRET

JA/JNBE	**JA/JNBE** short-label Jump if above/Jump if not below nor equal			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JA ABOVE

JAE/JNB	**JAE/JNB** short-label Jump if above or equal/Jump if not below			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JAE ABOVE_EQUAL

JB/JNAE	**JB/JNAE** short-label Jump if below/Jump if not above nor equal			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JB BELOW

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

†INTR is not an instruction; it is included in table 2-21 only for timing information.

Table 2-21. Instruction Set Reference Data (Cont'd.)

JBE/JNA	JBE/JNA short-label Jump if below or equal/Jump if not above			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JNA NOT_ABOVE

JC	JC short-label Jump if carry			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JC CARRY_SET

JCXZ	JCXZ short-label Jump if CX is zero			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	18 or 6	—	2	JCXZ COUNT_DONE

JE/JZ	JE/JZ short-label Jump if equal/Jump if zero			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JZ ZERO

JG/JNLE	JG/JNLE short-label Jump if greater/Jump if not less nor equal			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JG GREATER

JGE/JNL	JGE/JNL short-label Jump if greater or equal/Jump if not less			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JGE GREATER_EQUAL

JL/JNGE	JL/JNGE short-label Jump if less/Jump if not greater nor equal			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JL LESS

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Table 2-21. Instruction Set Reference Data (Cont'd.)

JLE/JNG	JLE/JNG short-label Jump if less or equal/Jump if not greater			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JNG NOT_GREATER

JMP	JMP target Jump			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	15	—	2	JMP SHORT
near-label	15	—	3	JMP WITHIN_SEGMENT
far-label	15	—	5	JMP FAR_LABEL
memptr16	18+EA	1	2-4	JMP [BX].TARGET
regptr16	11	—	2	JMP CX
memptr32	24+EA	2	2-4	JMP OTHER.SEG [SI]

JNC	JNC short-label Jump if not carry			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JNC NOT_CARRY

JNE/JNZ	JNE/JNZ short-label Jump if not equal/Jump if not zero			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JNE NOT_EQUAL

JNO	JNO short-label Jump if not overflow			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JNO NO_OVERFLOW

JNP/JPO	JNP/JPO short-label Jump if not parity/Jump if parity odd			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JPO ODD_PARITY

JNS	JNS short-label Jump if not sign			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JNS POSITIVE

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Table 2-21. Instruction Set Reference Data (Cont'd.)

JO	JO short-label Jump if overflow			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JO SIGNED_OVRFLW

JP/JPE	JP/JPE short-label Jump if parity/Jump if parity even			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JPE EVEN_PARITY

JS	JS short-label Jump if sign			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JS NEGATIVE

LAHF	LAHF (no operands) Load AH from flags			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	4	—	1	LAHF

LDS	LDS destination,source Load pointer using DS			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers**	**Bytes**	**Coding Example**
reg16, mem32	16+EA	2	2-4	LDS SI,DATA.SEG [DI]

LEA	LEA destination,source Load effective address			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
reg16, mem16	2+EA	—	2-4	LEA BX, [BP] [DI]

LES	LES destination,source Load pointer using ES			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
reg16, mem32	16+EA	2	2-4	LES DI, [BX].TEXT_BUFF

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Table 2-21. Instruction Set Reference Data (Cont'd.)

LOCK	LOCK (no operands) Lock bus			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	LOCK XCHG FLAG,AL

LODS	LODS source-string Load string			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
source-string (repeat) source-string	12 9+13/rep	1 1/rep	1 1	LODS CUSTOMER_NAME REP LODS NAME

LOOP	LOOP short-label Loop			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	17/5	—	2	LOOP AGAIN

LOOPE/LOOPZ	LOOPE/LOOPZ short-label Loop if equal/Loop if zero			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	18 or 6	—	2	LOOPE AGAIN

LOOPNE/LOOPNZ	LOOPNE/LOOPNZ short-label Loop if not equal/Loop if not zero			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	19 or 5	—	2	LOOPNE AGAIN

NMI†	NMI (external nonmaskable interrupt) Interrupt if NMI = 1			Flags O S I T S Z A P C 0 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	50	5	N/A	N/A

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

†NMI is not an instruction; it is included in table 2-21 only for timing information.

Table 2-21. Instruction Set Reference Data (Cont'd.)

MOV	**MOV** destination,source Move			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
memory, accumulator	10	1	3	MOV ARRAY [SI], AL
accumulator, memory	10	1	3	MOV AX, TEMP_RESULT
register, register	2	—	2	MOV AX,CX
register, memory	8+EA	1	2-4	MOV BP, STACK_TOP
memory, register	9+EA	1	2-4	MOV COUNT [DI], CX
register, immediate	4	—	2-3	MOV CL, 2
memory, immediate	10+EA	1	3-6	MOV MASK [BX] [SI], 2CH
seg-reg, reg16	2	—	2	MOV ES, CX
seg-reg, mem16	8+EA	1	2-4	MOV DS, SEGMENT_BASE
reg16, seg-reg	2	—	2	MOV BP, SS
memory, seg-reg	9+EA	1	2-4	MOV [BX].SEG_SAVE, CS

MOVS	**MOVS** dest-string,source-string Move string			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
dest-string, source-string	18	2	1	MOVS LINE EDIT_DATA
(repeat) dest-string, source-string	9+17/rep	2/rep	1	REP MOVS SCREEN, BUFFER

MOVSB/MOVSW	**MOVSB/MOVSW** (no operands) Move string (byte/word)			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	18	2	1	MOVSB
(repeat) (no operands)	9+17/rep	2/rep	1	REP MOVSW

MUL	**MUL** source Multiplication, unsigned			Flags O D I T S Z A P C X U U U U X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
reg8	70-77	—	2	MUL BL
reg16	118-133	—	2	MUL CX
mem8	(76-83) +EA	1	2-4	MUL MONTH [SI]
mem16	(124-139) +EA	1	2-4	MUL BAUD_RATE

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Table 2-21. Instruction Set Reference Data (Cont'd.)

NEG	**NEG** destination Negate			**Flags** O D I T S Z A P C X X X X X 1*
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register	3	—	2	NEG AL
memory	16+EA	2	2-4	NEG MULTIPLIER

*0 if destination = 0

NOP	**NOP** (no operands) No Operation			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	3	—	1	NOP

NOT	**NOT** destination Logical not			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register	3	—	2	NOT AX
memory	16+EA	2	2-4	NOT CHARACTER

OR	**OR** destination,source Logical inclusive or			**Flags** O D I T S Z A P C 0 X X U X 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, register	3	—	2	OR AL, BL
register, memory	9+EA	1	2-4	OR DX, PORT_ID [DI]
memory, register	16+EA	2	2-4	OR FLAG_BYTE, CL
accumulator, immediate	4	—	2-3	OR AL, 01101100B
register, immediate	4	—	3-4	OR CX,01H
memory, immediate	17+EA	2	3-6	OR [BX].CMD_WORD,0CFH

OUT	**OUT** port,accumulator Output byte or word			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
immed8, accumulator	10	1	2	OUT 44, AX
DX, accumulator	8	1	1	OUT DX, AL

POP	**POP** destination Pop word off stack			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register	8	1	1	POP DX
seg-reg (CS illegal)	8	1	1	POP DS
memory	17+EA	2	2-4	POP PARAMETER

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Table 2-21. Instruction Set Reference Data (Cont'd.)

POPF	POPF (no operands) Pop flags off stack			Flags O D I T S Z A P C R R R R R R R R
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	8	1	1	POPF

PUSH	PUSH source Push word onto stack			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register	11	1	1	PUSH SI
seg-reg (CS legal)	10	1	1	PUSH ES
memory	16+EA	2	2-4	PUSH RETURN_CODE [SI]

PUSHF	PUSHF (no operands) Push flags onto stack			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	10	1	1	PUSHF

RCL	RCL destination,count Rotate left through carry			Flags O D I T S Z A P C X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, 1	2	—	2	RCL CX, 1
register, CL	8+4/bit	—	2	RCL AL, CL
memory, 1	15+EA	2	2-4	RCL ALPHA, 1
memory, CL	20+EA+4/bit	2	2-4	RCL [BP].PARM, CL

RCR	RCR designation,count Rotate right through carry			Flags O D I T S Z A P C X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, 1	2	—	2	RCR BX, 1
register, CL	8+4/bit	—	2	RCR BL, CL
memory, 1	15+EA	2	2-4	RCR [BX].STATUS, 1
memory, CL	20+EA+4/bit	2	2-4	RCR ARRAY [DI], CL

REP	REP (no operands) Repeat string operation			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	REP MOVS DEST, SRCE

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Table 2-21. Instruction Set Reference Data (Cont'd.)

REPE/REPZ	**REPE/REPZ** (no operands) Repeat string operation while equal/while zero			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	REPE CMPS DATA, KEY

REPNE/REPNZ	**REPNE/REPNZ** (no operands) Repeat string operation while not equal/not zero			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	REPNE SCAS INPUT_LINE

RET	**RET** optional-pop-value Return from procedure			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(intra-segment, no pop)	8	1	1	RET
(intra-segment, pop)	12	1	3	RET 4
(inter-segment, no pop)	18	2	1	RET
(inter-segment, pop)	17	2	3	RET 2

ROL	**ROL** destination,count Rotate left			**Flags** O D I T S Z A P C X X
Operands	**Clocks**	**Transfers**	**Bytes**	**Coding Examples**
register, 1	2	—	2	ROL BX, 1
register, CL	8+4/bit	—	2	ROL DI, CL
memory, 1	15+EA	2	2-4	ROL FLAG_BYTE [DI],1
memory, CL	20+EA+4/bit	2	2-4	ROL ALPHA , CL

ROR	**ROR** destination,count Rotate right			**Flags** O D I T S Z A P C X X
Operand	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, 1	2	—	2	ROR AL, 1
register, CL	8+4/bit	—	2	ROR BX, CL
memory, 1	15+EA	2	2-4	ROR PORT_STATUS, 1
memory, CL	20+EA+4/bit	2	2-4	ROR CMD_WORD, CL

SAHF	**SAHF** (no operands) Store AH into flags			**Flags** O D I T S Z A P C R R R R R
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	4	—	1	SAHF

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Table 2-21. Instruction Set Reference Data (Cont'd.)

SAL/SHL	**SAL/SHL** destination,count Shift arithmetic left/Shift logical left			**Flags** O D I T S Z A P C X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Examples**
register,1	2	—	2	SAL AL,1
register, CL	8+4/bit	—	2	SHL DI, CL
memory,1	15+EA	2	2-4	SHL [BX].OVERDRAW, 1
memory, CL	20+EA+4/bit	2	2-4	SAL STORE_COUNT, CL

SAR	**SAR** destination,source Shift arithmetic right			**Flags** O D I T S Z A P C X X X U X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, 1	2	—	2	SAR DX, 1
register, CL	8+4/bit	—	2	SAR DI, CL
memory, 1	15+EA	2	2-4	SAR N_BLOCKS, 1
memory, CL	20+EA+4/bit	2	2-4	SAR N_BLOCKS, CL

SBB	**SBB** destination,source Subtract with borrow			**Flags** O D I T S Z A P C X X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, register	3	—	2	SBB BX, CX
register, memory	9+EA	1	2-4	SBB DI, [BX].PAYMENT
memory, register	16+EA	2	2-4	SBB BALANCE, AX
accumulator, immediate	4	—	2-3	SBB AX, 2
register, immediate	4	—	3-4	SBB CL, 1
memory, immediate	17+EA	2	3-6	SBB COUNT [SI], 10

SCAS	**SCAS** dest-string Scan string			**Flags** O D I T S Z A P C X X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
dest-string	15	1	1	SCAS INPUT_LINE
(repeat) dest-string	9+15/rep	1/rep	1	REPNE SCAS BUFFER

SEGMENT†	**SEGMENT** override prefix Override to specified segment			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	MOV SS:PARAMETER, AX

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

†ASM-86 incorporates the segment override prefix into the operand specification and not as a separate instruction. SEGMENT is included in table 2-21 only for timing information.

Table 2-21. Instruction Set Reference Data (Cont'd.)

SHR	**SHR** destination,count Shift logical right			**Flags** O D I T S Z A P C X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, 1	2	—	2	SHR SI, 1
register, CL	8+4/bit	—	2	SHR SI, CL
memory, 1	15+EA	2	2-4	SHR ID_BYTE [SI] [BX], 1
memory, CL	20+EA+4/bit	2	2-4	SHR INPUT_WORD, CL

SINGLE STEP†	**SINGLE STEP** (Trap flag interrupt) Interrupt if TF = 1			**Flags** O D I T S Z A P C 0 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	50	5	N/A	N/A

STC	**STC** (no operands) Set carry flag			**Flags** O D I T S Z A P C 1
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	STC

STD	**STD** (no operands) Set direction flag			**Flags** O D I T S Z A P C 1
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	STD

STI	**STI** (no operands) Set interrupt enable flag			**Flags** O D I T S Z A P C 1
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	STI

STOS	**STOS** dest-string Store byte or word string			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
dest-string	11	1	1	STOS PRINT_LINE
(repeat) dest-string	9+10/rep	1/rep	1	REP STOS DISPLAY

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

†SINGLE STEP is not an instruction; it is included in table 2-21 only for timing information.

Table 2-21. Instruction Set Reference Data (Cont'd.)

SUB	SUB destination,source Subtraction			Flags O D I T S Z A P C X X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, register	3	—	2	SUB CX, BX
register, memory	9+EA	1	2-4	SUB DX, MATH_TOTAL [SI]
memory, register	16+EA	2	2-4	SUB [BP+2], CL
accumulator, immediate	4	—	2-3	SUB AL, 10
register, immediate	4	—	3-4	SUB SI, 5280
memory, immediate	17+EA	2	3-6	SUB [BP].BALANCE, 1000

TEST	TEST destination,source Test or non-destructive logical and			Flags O D I T S Z A P C 0 X X U X 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, register	3	—	2	TEST SI, DI
register, memory	9+EA	1	2-4	TEST SI, END_COUNT
accumulator, immediate	4	—	2-3	TEST AL, 00100000B
register, immediate	5	—	3-4	TEST BX, 0CC4H
memory, immediate	11+EA	—	3-6	TEST RETURN_CODE, 01H

WAIT	WAIT (no operands) Wait while $\overline{\text{TEST}}$ pin not asserted			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	3 + 5n	—	1	WAIT

XCHG	XCHG destination,source Exchange			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
accumulator, reg16	3	—	1	XCHG AX, BX
memory, register	17+EA	2	2-4	XCHG SEMAPHORE, AX
register, register	4	—	2	XCHG AL, BL

XLAT	XLAT source-table Translate			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
source-table	11	1	1	XLAT ASCII_TAB

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Table 2-21. Instruction Set Reference Data (Cont'd.)

XOR	XOR destination,source Logical exclusive or			Flags O D I T S Z A P C 0 X X U X 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, register	3	—	2	XOR CX, BX
register, memory	9+EA	1	2-4	XOR CL, MASK_BYTE
memory, register	16+EA	2	2-4	XOR ALPHA [SI], DX
accumulator, immediate	4	—	2-3	XOR AL, 01000010B
register, immediate	4	—	3-4	XOR SI, 00C2H
memory, immediate	17+EA	2	3-6	XOR RETURN_CODE, 0D2H

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

2.8 Addressing Modes

The 8086 and 8088 provide many different ways to access instruction operands. Operands may be contained in registers, within the instruction itself, in memory or in I/O ports. In addition, the addresses of memory and I/O port operands can be calculated in several different ways. These addressing modes greatly extend the flexibility and convenience of the instruction set. This section briefly describes register and immediate operands and then covers the 8086/8088 memory and I/O addressing modes in detail.

Register and Immediate Operands

Instructions that specify only register operands are generally the most compact and fastest executing of all instruction forms. This is because the register "addresses" are encoded in instructions in just a few bits, and because these operations are performed entirely within the CPU (no bus cycles are run). Registers may serve as source operands, destination operands, or both.

Immediate operands are constant data contained in an instruction. The data may be either 8 or 16 bits in length. Immediate operands can be accessed quickly because they are available directly from the instruction queue; like a register operand, no bus cycles need to be run to obtain an immediate operand. The limitations of immediate operands are that they may only serve as source operands and that they are constant values.

Memory Addressing Modes

Whereas the EU has direct access to register and immediate operands, memory operands must be transferred to or from the CPU over the bus. When the EU needs to read or write a memory operand, it must pass an offset value to the BIU. The BIU adds the offset to the (shifted) content of a segment register producing a 20-bit physical address and then executes the bus cycle(s) needed to access the operand.

The Effective Address

The offset that the EU calculates for a memory operand is called the operand's effective address or EA. It is an unsigned 16-bit number that expresses the operand's distance in bytes from the beginning of the segment in which it resides. The EU can calculate the effective address in several different ways. Information encoded in the second byte of the instruction tells the EU how to calculate the effective address of each memory operand. A compiler or assembler derives this information from the statement or instruction written by the programmer. Assembly language programmers have access to all addressing modes.

Figure 2-34 shows that the execution unit calculates the EA by summing a displacement, the content of a base register and the content of an index register. The fact that any combination of these three components may be present in a given instruction gives rise to the variety of 8086/8088 memory addressing modes.

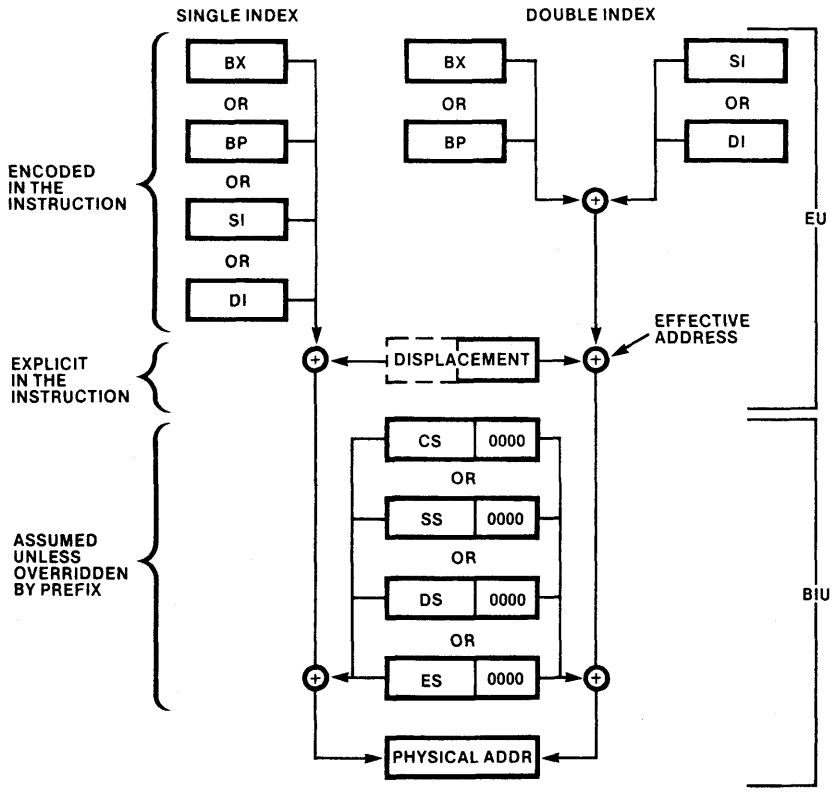


Figure 2-34. Memory Address Computation

The displacement element is an 8- or 16-bit number that is contained in the instruction. The displacement generally is derived from the position of the operand name (a variable or label) in the program. It also is possible for a programmer to modify this value or to specify the displacement explicitly.

A programmer may specify that either BX or BP is to serve as a base register whose content is to be used in the EA computation. Similarly, either SI or DI may be specified as an index register. Whereas the displacement value is a constant, the contents of the base and index registers may change during execution. This makes it possible for one instruction to access different memory locations as determined by the current values in the base and/or index registers.

It takes time for the EU to calculate a memory operand's effective address. In general, the more elements in the calculation, the longer it takes. Table 2-20 shows how much time is required to compute an effective address for any combination of displacement, base register and index register.

Direct Addressing

Direct addressing (see figure 2-35) is the simplest memory addressing mode. No registers are involved; the EA is taken directly from the displacement field of the instruction. Direct addressing typically is used to access simple variables (scalars).

Register Indirect Addressing

The effective address of a memory operand may be taken directly from one of the base or index registers as shown in figure 2-36. One instruction can operate on many different memory locations if the value in the base or index register is updated

appropriately. The LEA (load effective address) and arithmetic instructions might be used to change the register value.

Note that *any* 16-bit general register may be used for register indirect addressing with the JMP or CALL instructions.

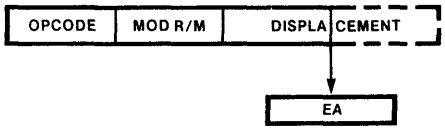


Figure 2-35. Direct Addressing

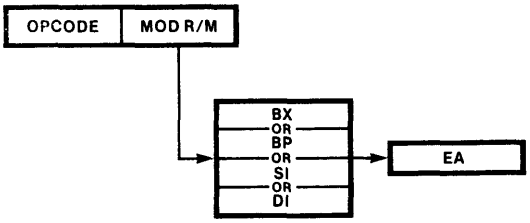


Figure 2-36. Register Indirect Addressing

Based Addressing

In based addressing (figure 2-37), the effective address is the sum of a displacement value and the content of register BX or register BP. Recall that specifying BP as a base register directs the BIU to obtain the operand from the current stack segment (unless a segment override prefix is present). This makes based addressing with BP a very convenient way to access stack data (see section 2.10 for examples).

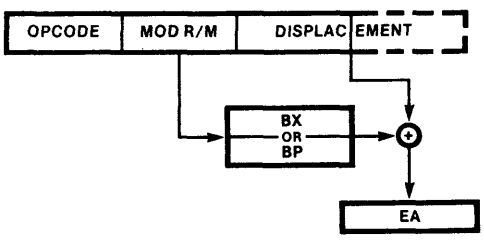


Figure 2-37. Based Addressing

Based addressing also provides a straightforward way to address structures which may be located at different places in memory (see figure 2-38). A base register can be pointed at the base of the structure and elements of the structure addressed by their displacements from the base. Different copies of the same structure can be accessed by simply changing the base register.

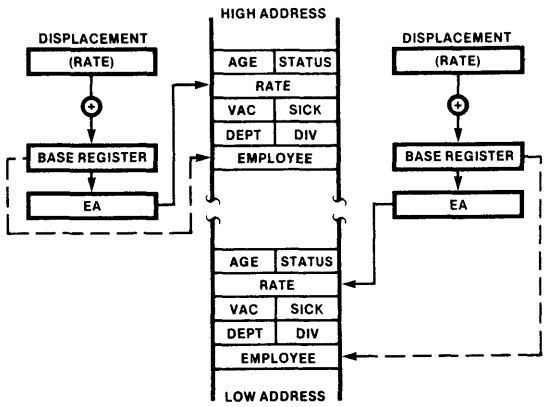


Figure 2-38. Accessing a Structure With Based Addressing

Indexed Addressing

In indexed addressing, the effective address is calculated from the sum of a displacement plus the content of an index register (SI or DI) as shown in figure 2-39. Indexed addressing often is

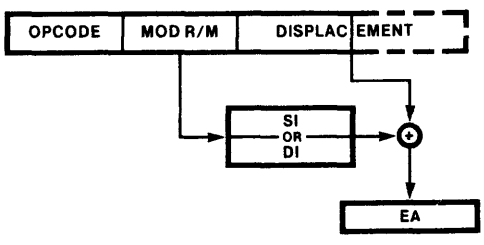


Figure 2-39. Indexed Addressing

used to access elements in an array (see figure 2-40). The displacement locates the beginning of the array, and the value of the index register selects one element (the first element is selected if the index register contains 0). Since all array elements are the same length, simple arithmetic on the index register will select any element.

Based Indexed Addressing

Based indexed addressing generates an effective address that is the sum of a base register, an index register and a displacement (see figure 2-41). Based indexed addressing is a very flexible mode because two address components can be varied at execution time.

Based indexed addressing provides a convenient way for a procedure to address an array allocated on a stack (see figure 2-42). Register BP can contain the offset of a reference point on the stack, typically the top of the stack after the procedure has saved registers and allocated local storage. The offset of the beginning of the array from the reference point can be expressed by a displacement value, and an index register can be used to access individual array elements.

Arrays contained in structures and matrices (two-dimension arrays) also could be accessed with based indexed addressing.

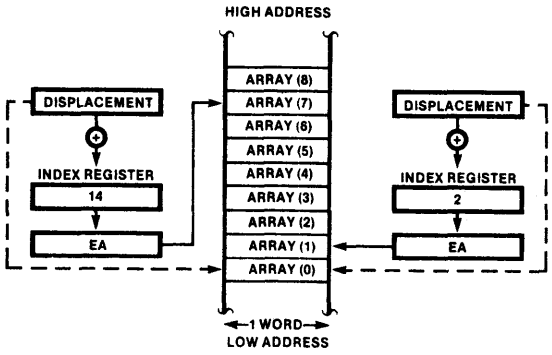


Figure 2-40. Accessing an Array With Indexed Addressing

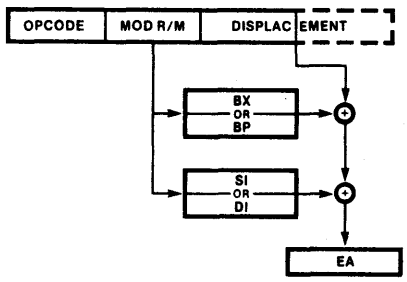


Figure 2-41. Based Indexed Addressing

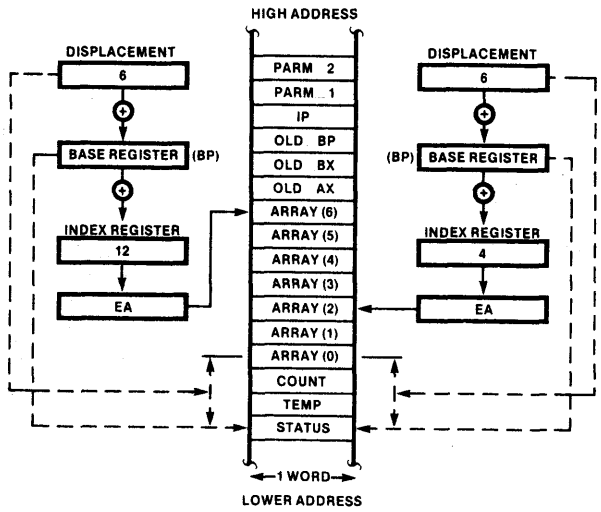


Figure 2-42. Accessing a Stack Array With Based Indexed Addressing

String Addressing

String instructions do not use the normal memory addressing modes to access their operands. Instead, the index registers are used implicitly as shown in figure 2-43. When a string instruction is executed, SI is assumed to point to the first byte or word of the source string, and DI is assumed to point to the first byte or word of the destination string. In a repeated string operation, the CPUs automatically adjust SI and DI to obtain subsequent bytes or words.

I/O Port Addressing

If an I/O port is memory mapped, any of the memory operand addressing modes may be used to access the port. For example, a group of terminals can be accessed as an "array." String instructions also can be used to transfer data to memory-mapped ports with an appropriate hardware interface. Section 2.10 contains examples of addressing memory-mapped I/O ports.

Two different addressing modes can be used to access ports located in the I/O space; these are illustrated in figure 2-44. In direct port addressing, the port number is an 8-bit immediate operand. This allows fixed access to ports numbered 0-255. Indirect port addressing is similar to register indirect addressing of memory operands. The port number is taken from register DX and can range from 0 to 65,535. By previously adjusting the content of register DX, one instruction can access any port in the I/O space. A group of adjacent ports can be accessed using a simple software loop that adjusts the value in DX.

2.9 Programming Facilities

A comprehensive integrated set of tools supports 8086/8088 software development. These tools are programs that run on Intellec® 800 or Series II Microcomputer Development Systems under the ISIS-II operating system, the same hardware and operating system used to develop software for the 8080 and the 8085. Since the 8086 and 8088 are software-compatible with one another, the same tools are used for both processors to provide programmers with a uniform development environment.

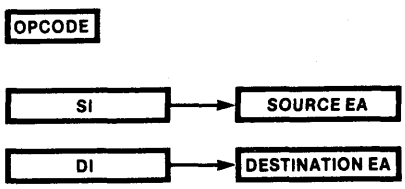


Figure 2-43. String Operand Addressing

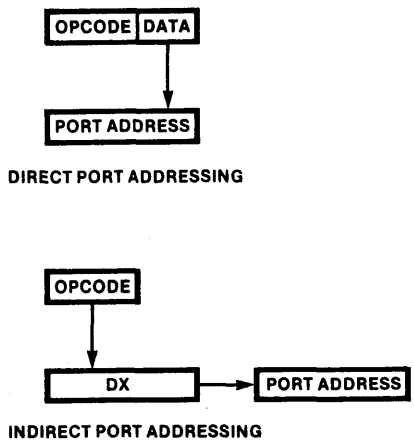


Figure 2-44. I/O Port Addressing

Software Development Overview

A program that will ultimately execute on an 8086- or 8088-based system is developed in steps (see figure 2-45). The overall program is composed of functional units called modules. For purposes of this discussion, a module is a section of code that is separately created, edited, and compiled or assembled. A very small program might consist of a single module; a large program could be comprised of 100 or more modules. The 8086/8088 LINK-86 utility binds modules together into a single program. (The module structure of a program is critical to its successful development and maintenance; see section 2.10 for guidelines.)

8086 and 8088 modules can be written in either PL/M-86 or ASM-86 (see table 2-22). PL/M-86 is a high-level language suitable for most microprocessor applications. It is easy to use, even by programmers who have little experience with microprocessors. Because it reduces software development time, PL/M-86 is ideal for most of the programming in any application, especially applications that must get to market quickly.

ASM-86 is the 8086/8088 assembly language. ASM-86 provides the programmer who is familiar with the CPU architecture, access to all processor features. For critical code segments within programs that make sophisticated use of the hardware, have extremely demanding performance or memory constraints, ASM-86 is the best choice.

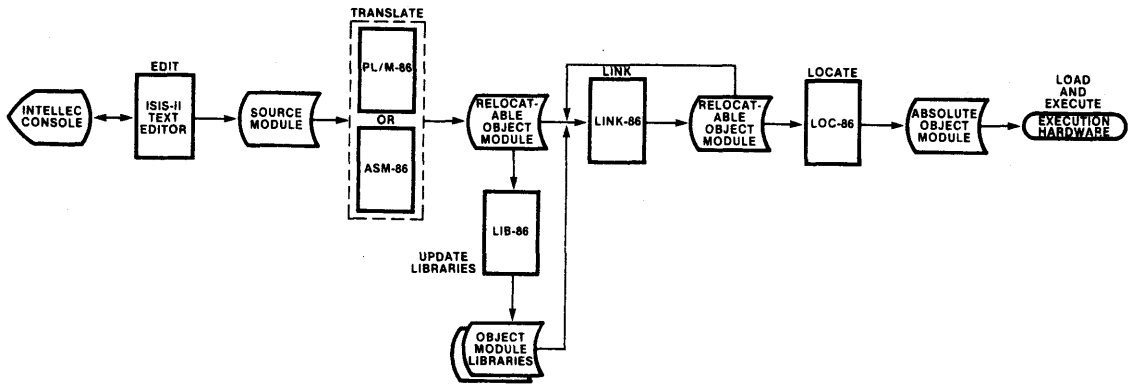


Figure 2-45. Software Development Process

Table 2-22. PL/M-86/ASM-86 Characteristics

PL/M-86	ASM-86
• Fast Development	• Fastest Execution Speed
• Less Programmer Training	• Smallest Memory Requirements
• Detailed Hardware Knowledge Not Required	• Access To All Processor Facilities

The languages are completely compatible, and a judicious combination of the two often makes good sense. Prototype software can be developed rapidly with PL/M-86. When the system is operating correctly, it can be analyzed to see which sections can best profit from being written in ASM-86. Since the logic of these sections already has been debugged, selective rewriting can be done quickly and with low risk.

Each PL/M-86 or ASM-86 module (called a source moduel) is keyed into the Intellec® system using the ISIS-II text editor and is stored as a diskette file. This source file is then input to the appropriate language translator (ASM-86 assembler or PL/M-86 compiler). The language translator creates a diskette file from the source file, which is called a relocatable object module. The translator also lists the program and flags any errors detected during the translation. The relocatable object module contains the 8086/8088 machine instructions that the translator created from the statements in the source module. The term "relocatable" refers to the fact that all references to memory locations in the module are relative, rather than being absolute memory addresses. The module generally is not executable until the relative references are changed to the actual memory locations where the module will reside in the execution system's memory. The process of changing the relative references to absolute memory locations is called locating.

There are very good reasons for not locating modules when they are translated. First, the execution system's physical memory configuration (where RAM and ROM/PROM segments are actually located in the megabyte memory space) may not be known at the time the modules are written. Second, it is desirable to be able to use a common module (e.g., a square root routine) in more than one system. If absolute addresses were assigned at translation time, the common module would either have to occupy the same physical addresses in every system, or separate versions with different addresses would have to be maintained for each system. When locating is deferred, a single version of a common routine can be used by any number of systems. Finally, the locations of modules typically change as a system is developed, maintained and enhanced. Separating the location process from the translation process means that as modifications are made, unchanged modules only need to be relocated, not retranslated.

Relocatable object modules may be placed into special files called libraries, using the LIB-86 library manager program. Libraries provide a convenient means of collecting groups of related modules so that they can be accessed automatically by the LINK-86 program.

When enough relocatable object modules have been created to test the system, or part of it, the modules are linked and located. Linking combines all the separate modules into a single program. Locating changes the relative memory references in the program to the actual memory locations where the program will be loaded in the execution system. The link and locate process also is referred to as R & L, for relocation and linkage.

Two other programs round out the software development tools available for the 8086 and 8088. OH-86 converts an absolute object file into a hexadecimal format used by some PROM programmers and system loaders (for example, the SDK-86 and iSBC 957™ loaders). CONV-86 can do most of the conversion work required to translate 8080/8085 assembly language source modules into ASM-86 source modules.

The 8086/8088 software development facilities are covered in more detail in the remainder of this section. However, these are only introductions to

the use of these tools. Complete documentation is available in the following publications available from Intel's Literature Department:

ISIS-II:

ISIS-II System User's Guide, Order No. 9800306

ASM-86:

MCS-86 Assembly Language Reference Manual, Order No. 9800640

MCS-86 Assembler Operating Instructions for ISIS-II Users, Order No. 9800641

PL/M-86:

PL/M-86 Programming Manual, Order No. 9800466

ISIS-II PL/M-86 Compiler Operator's Manual, Order No. 9800478

LINK-86, LOC-86, LIB-86, OH-86:

MCS-86 Software Development Utilities Operating Instructions for ISIS-II Users, Order No. 9800639

CONV-86:

MCS-86 Assembly Language Converter Operating Instructions for ISIS-II Users, Order No. 9800642

PL/M-86

PL/M-86 is a general-purpose, high-level language for programming the 8086 and 8088 microprocessors. It is an extension of PL/M-80, the most widely-used, high-level programming language for microprocessors. (PL/M-80 source programs can be processed by the PL/M-86 compiler; the resulting object program is generally reduced by 15-30% in size.) PL/M-86 is suitable for all types of microprocessor software from operating systems to application programs.

PL/M-86's purpose is simple: to reduce the time and cost of developing and maintaining software for the 8086 and 8088. It accomplishes this by creating a programming environment that, for the most part, is distinct from the architecture of the CPUs. Registers, segments, addressing modes, stacks, etc., are effectively "invisible" to the PL/M-86 programmer. Instead, the processors appear to respond to simple commands and familiar algebraic expressions. The responsibility for translating these source statements into the machine instructions ultimately required to execute on the 8086/8088 is assumed by the PL/M-86 compiler. By "hiding" the details of the machine architecture, PL/M-86 encourages programmers to concentrate on solving the problem at hand. Furthermore, because PL/M-86 is closer to natural language, it is easier to "think in PL/M-86" than it is to "think in assembly language." This speeds up the expression of a program solution, and, equally important, makes that solution easier for someone other than the original programmer to understand. PL/M-86 also contains all the constructs necessary for structured programming.

Statements and Comments

A programmer builds a PL/M-86 program by writing statements and comments (see figure 2-46). There are several different types of statements in PL/M-86; they always end with a semicolon. Blanks can be used freely before, within, and after statements to improve readability. A statement also may span more than one line.

The characters "/*" start a comment, and the characters "*/" end it; any characters may be used in between. Comments do not affect the execution of a PL/M-86 program, but all good programs are thoughtfully commented. Comments are notes that document and clarify the program's operation; they may be written virtually anywhere in a PL/M-86 program.

Data Definition

Most PL/M-86 programs begin by defining the data items (variables) with which they are going to work. An individual PL/M-86 data element is called a scalar. Every scalar variable has a programmer-supplied name up to 31 characters long, and a type. PL/M-86 supports five types of scalars: byte, word, integer, real, and pointer. Table 2-23 lists the characteristics of these PL/M-86 data types.

```
/*TRAFFIC DATA RECORDER CONTROL PROGRAM*
 *VERSION 2.2, RELEASE 5, 23APR79.*
 *THIS RELEASE FIXES THREE BUGS*
 *DOCUMENTED IN PROBLEM REPORT #16.*/

/*COMPUTE TOTAL PAYMENT DUE*/
TOTAL = PRINCIPAL + INTEREST;

IF TERMINAL$READY
    THEN CALL FILL$BUFFER;
    ELSE CALL WAIT (50);      /*WAIT 50 MS FOR RESPONSE*/
```

Figure 2-46. PL/M-86 Statements and Comments

Table 2-23. PL/M-86 Data Types

TYPE	BYTES	RANGE	USAGE
BYTE	1	0 to 255	Unsigned Integer, Character
WORD	2	0 to 65,535	Unsigned Integer
INTEGER	2	−32,768 to +32,767	Signed Integer
REAL	4	1×10^{-38} to $3.37 \times 10^{+38}$	Floating Point
POINTER	2/4	N/A	Address Manipulation

Variables are defined by writing a DECLARE statement of this form:

```
DECLARE scalar-name type;
```

Options of the DECLARE statement can be used to specify an initial value for the scalar and to define a series of items in a shorthand form.

Besides scalar variables, scalar constants may be used in PL/M-86 programs (see figure 2-47). Constants may be written "as is" or may be given names to improve program clarity.

Scalars can be aggregated into named collections of data such as arrays and structures. An array is a collection of scalars of the same type (all integer, all real, etc.). Arrays are useful for representing data that has a repetitive nature. For example, monthly rainfall samples could be represented as an array of 12 elements, one for each month:

```
DECLARE RAINFALL (12) REAL;
```

Each element in an array is accessible by a number called a subscript which is the element's relative location in the array. In PL/M-86, the first element in an array has a subscript of 0; it is considered the "0th" element. Thus, RAINFALL (11) refers to December's sample. The subscript need not be a constant; variables and expressions also may be used as subscripts.

Strings of character data are typically defined as byte arrays. Characters can be accessed with subscripts or with powerful string-handling functions built into PL/M-86.

```
        10  /*DECIMAL NUMBER*/
       0AH  /*HEXADECIMAL NUMBER*/
       12Q  /*OCTAL NUMBER*/
  00001010B  /*BINARY NUMBER*/
      10.0  /*FLOATING POINT NUMBER*/
     1.0E1  /*FLOATING POINT NUMBER*/
       'A'  /*CHARACTER*/

/*CONSTANTS MAY BE GIVEN NAMES*/
DECLARE STATUS$PORT LITERALLY '0FFEH';
DECLARE THRESHOLD LITERALLY '98.6';
```

Figure 2-47. PL/M-86 Constants

A structure is a collection of related data elements that do not necessarily have the same type. The elements are related by virtue of "belonging" to the entity represented by the structure. Here is a simple structure declaration:

```
DECLARE BRIDGE STRUCTURE
        (SPAN          WORD,
        YR$BUILT       BYTE,
        AVG$TRAFFIC REAL);
```

The year the bridge was built could be accessed by writing BRIDGE.YR$BUILT; the structure element name is "qualified" by the dot and the structure name. This allows structures with the same element names to be distinguished from each other (e.g., HIGHWAY.YR$BUILT).

Arrays and structures can be combined into more complex data aggregates:

- array elements may be structures rather than scalars,
- a structure element may be an array,
- structures in arrays may themselves contain arrays.

Figure 2-48 provides sample PL/M-86 data declarations.

Assignment Statement

Data that has been defined can be operated on with PL/M-86 executable statements. The fundamental executable statement is the assignment statement, written in this form:

```
variable-name = expression;
```

This means "evaluate the expression and assign (move) the result to the variable."

There are three basic classes of expressions in PL/M-86; arithmetic, relational and logical (see table 2-24 and figure 2-49). All expressions are combinations of operands and operators, although an expression can consist of a single operand. Operands are variables and constants; operators vary according to the type of expression. Evaluation of an expression always yields a single result; different classes of expressions yield different types of results.

Table 2-24. Characteristics of PL/M-86 Expressions

EXPRESSION	OPERATORS	RESULT
ARITHMETIC	+, −, *, /, MOD	NUMBER
RELATIONAL	>, <, =, >=, <=	"TRUE" - FFH "FALSE" - 0H
LOGICAL	AND, OR, XOR, NOT	8/16-BIT STRING

```
    /****SCALARS****/
DECLARE SWITCH          BYTE;
DECLARE COUNT           WORD,                   /*1 SCALAR*/
        INDEX           INTEGER;                /*1 SCALAR*/
DECLARE (NET, GROSS,    TOTAL)   REAL;          /*3 SCALARS*/

    /****ARRAYS****/
DECLARE MONTH (12)      BYTE;
DECLARE TERMINAL_LINE (80)          BYTE;

    /****STRUCTURE****/
DECLARE EMPLOYEE STRUCTURE
        (ID_NUMBER                  WORD,
        DEPARTMENT                  BYTE
        RATE                        REAL);

    /****ARRAY OF STRUCTURES****/
DECLARE INVENTORY_ITEM (100)        STRUCTURE
        (PART_NUMBER                WORD,
        ON_HAND                     WORD,
        RE_ORDER                    BYTE);

    /****ARRAY WITHIN STRUCTURE****/
DECLARE COUNTY_DATA                 STRUCTURE
        (NAME (20)                  BYTE,
        TEN_YR_RAINFALL(10)         BYTE,
        PER CAPITA_INCOME           REAL);
```

Figure 2-48. PL/M-86 Data Declarations

```
/*ARITHMETIC*/
A = 2; B = 3;
B = B+ 1;                   /*B CONTAINS 4*/
C = (A*B) -2;               /*C CONTAINS 6*/
C = ((A*B) + 3) MOD 3;      /*C CONTAINS 2*/

/*RELATIONAL*/
A =2; B= 3
C = B > A;                  /*C CONTAINS 0FFH*/
C = B < > A;                /*C CONTAINS 0FFH*/
C = B = (A+1);              /*C CONTAINS 0FFH*/

/*LOGICAL*/
A = 0011$0001B;             /*$ IS FOR READABILITY*/
B = 1000$0001B;
C = NOT B;                  /*C CONTAINS 0111$1110B*/
C = A AND B;                /*C CONTAINS 0000$0001B*/
C = A OR B;                 /*C CONTAINS 1011$0001B*/
C = B XOR A;                /*C CONTAINS 1011$0000B*/
C = (A AND B) OR 0F0H;      /*C CONTAINS 1111$0001B*/
```

Figure 2-49. Expressions in PL/M-86 Assignment Statements

Program Flow Statements

Simple PL/M-86 programs can be written with just DECLARE and assignment statements. Such programs, however, execute exactly the same sequence of statements every time they are run and would not prove very useful. PL/M-86 provides statements that change the flow of control through a program. These statements allow sections of the program to be executed selectively, repeated, skipped entirely, etc.

The IF statement (figure 2-50) selects one or the other of two statements for execution depending on the result of a relational expression. The IF statement is written:

```
IF relational-expression

   THEN statement1;

   ELSE statement2;
```

Statement1 is executed if the expression is "true"; statement2 is not executed in this case. If the relation is "false," statement1 is skipped and statement2 is executed. In determining the "truth" of an expression, the IF statement only examines the low-order bit of the result (1="true"). Therefore, arithmetic and logical expressions also may be used in an IF statement.

```
A = 3; B = 5;
IF A < B
  THEN MINIMUM = 1;      /*EXECUTED*/
  ELSE MINIMUM = 2;      /*SKIPPED*/

MORE_DATA = 0FFH;
IF NOT MORE_DATA
  THEN DONE = 1;         /*SKIPPED*/
  ELSE DONE = 0;         /*EXECUTED*/

/*NESTED IF STATEMENTS*/
CLOCK_ON = 1; HOUR=24; ALARM=OFF;
IF CLOCK_ON
  THEN IF HOUR = 24
     THEN IF ALARM = OFF
        THEN HOUR = 0; /*EXECUTED*/
```

Figure 2-50. PL/M-86 IF Statements

A DO block begins with a DO statement and ends with an END statement. All intervening statements are part of the block. A DO block can appear anywhere in a program that an executable statement can appear. There are four kinds of DO statements in PL/M-86: simple DO, DO CASE, interative DO, and DO WHILE.

A simple DO statement (figure 2-51) causes all the statements in the block to be treated as though they were a single statement. Simple DOs enable a single IF statement to cause multiple statements to be executed (the alternative would be to repeat the IF statement for every statement to be executed).

```
/*SIMPLE DO*/
A=5; B=9;
IF (A+2) < B THEN DO;
                  X=X-1;      /*EXECUTED*/
                  Y(X)=0;     /*EXECUTED*/
                  END;
          ELSE    DO;
                  X=X+1;      /*SKIPPED*/
                  Y(X)=1;     /*SKIPPED*/
                  END;


/*DO CASE*/
A = 2;
DO CASE (A);
    X = X+1;      /*SKIPPED*/
    X = X+2;      /*SKIPPED*/
    X = X+3;      /*EXECUTED*/
    X = X+4;      /*SKIPPED*/
    END;
```

Figure 2-51. PL/M-86 Simple DO and DO CASE

DO CASE (figure 2-51) causes one statement in the DO block to be selected and executed depending on the result of the expression (usually arithmetic) written immediately following DO CASE:

```
DO CASE arithmetic-expression;
```

If the expression yields 0, the first statement in the DO block is executed; if the expression yields 1, the second statement is executed, etc. A statement in the DO block may be null (consist of only a semicolon) to cause no action for selected cases. DO CASE provides a rapid and easily-understood way to respond to data like "transaction codes"

where a different action is required for each of many values a code might assume (an alternative would be an IF statement for every value the code could assume).

An iterative DO block (figures 2-52 and 2-53) is executed from 0 to an infinite number of times based on the relationship of an index variable to an expression that terminates execution. The general form is:

```
DO index = start-expr TO stop-expr BY step-expr;
```

The "BY step-expr" is optional, and the step is assumed to be 1 if not supplied (the typical case). When control first reaches the DO statement, start-expr is evaluated and is assigned to index. Then index is compared to stop-expr; if index exceeds stop-expr, control goes to the statement following the DO block, otherwise the block is executed. At the end of the block, the result of step-expr is added to index, and it is compared to stop-expr again, etc. (The iterative DO is quite flexible—this is a simplified explanation.) Iterative DOs are handy for "stepping through" an array. For example, an array of 10 elements could be zeroed by:

```
DO I = 0 TO 9;
    ARRAY(I) = 0;
    END;
```

In a DO WHILE (figures 2-52 and 2-54), the statements are executed repeatedly as long as the expression following WHILE evaluates to "true." DO WHILE often can be applied in situations where an interative DO will not work, or is clumsy, such as where repetition must be controlled by a non-integer value. Like an iterative DO, DO WHILE may be executed from 0 times to an infinite number of times.

```
/*ITERATIVE DO*/
DO I = 0 TO 5;
    ARRAY (I) = I;                  /*EXECUTED 6 TIMES*/
    TOTAL = TOTAL+1;                /*EXECUTED 6 TIMES*/
    END;
/*I = 6 AT THIS POINT*/

/*DO WHILE*/
MORE = 0; SPACE_OK =1;
DO WHILE (MORE AND SPACE_OK);
    ITEMS = ITEMS + 1;              /*SKIPPED*/
    N_TRACKS =
    N_TRACKS + 10;                  /*SKIPPED*/
    IF N_TRACKS >= 999              /*SKIPPED*/
        THEN SPACE_OK = 0;
    END;


/*DO WHILE*/
CODE = 'A';
DO WHILE (CODE = 'A');
    TEMP = TEMP * STEP;             /*EXECUTION STOPS*/
    IF TEMP > 98.6                  /*AFTER TEMP*/
        THEN CODE = 'B';            /*EXCEEDS 98.6*/
        N_STEPS = N_STEPS + 1;
    END;
```

Figure 2-52. PL/M-86 Iterative DO and DO WHILE

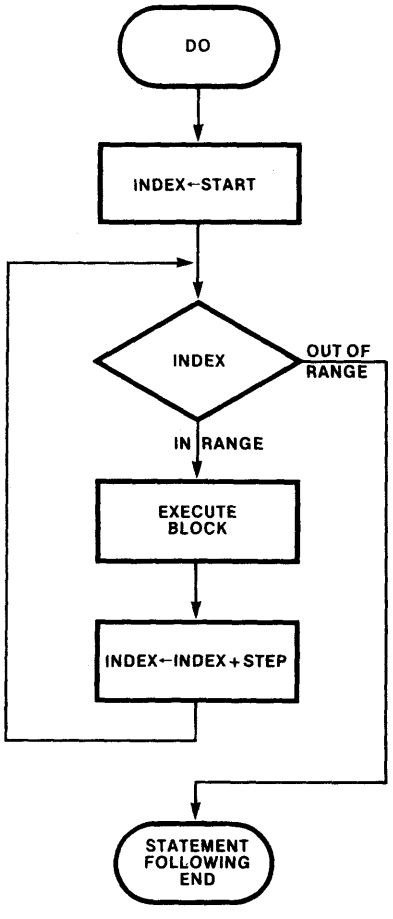


Figure 2-53. PL/M-86 Iterative DO Flowchart

A GOTO written in the form

```
GOTO target;
```

causes an unconditional transfer (branch) to another statement in the program. The statement receiving control would be written

```
target: statement;
```

where "target" is a label identifying the statement.

A CALL statement written in the form

```
CALL proc-name (parm-list);
```

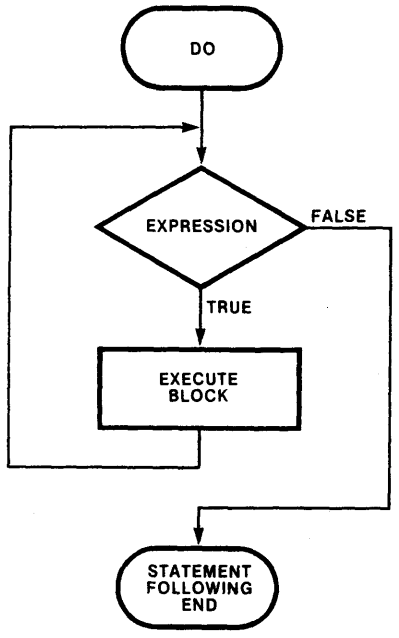


Figure 2-54. PL/M-86 DO WHILE Flowchart

activates a procedure defined earlier in the program. The variables listed in "parm-list" are passed to the procedure, the procedure is executed, and then control returns to the statement following the CALL. Thus, unlike a GOTO, a CALL brings control back to the point of departure.

Procedures

Procedures are "subprograms" that make it possible to simplify the design of complex programs and to share a single copy of a routine among programs. A procedure usually is designed to perform one function; i.e., to solve one part of the total problem with which the program is dealing. For example, a program to calculate paychecks could be broken down into separate procedures for calculating gross pay, income tax, Social Security and net pay. The organization of the "main" program then could be understood at a glance:

```
CALL GROSS_PAY;
CALL INCOME_TAX;
CALL SOCIAL_SECURITY;
CALL NET_PAY;
```

Furthermore, the income tax procedure could be divided into separate procedures for calculating state and federal taxes. Procedures, then, provide a mechanism by which a large, complex problem can be attacked with a "divide and conquer" strategy.

A procedure usually is defined early in a program, but it is only executed when it is referred to by name in a later PL/M-86 statement. A procedure can accept a list of variables, called parameters, that it will use in performing its function. These parameters may assume different values each time the procedure is executed.

PL/M-86 provides two classes of procedures, typed and untyped. A typed procedure returns a value to the statement that activates it and, in addition, may accept parameters from that statement. A typed procedure is activated whenever its name appears in a statement; the value it returns effectively takes the place of the procedure name in the statement. Typed procedures can be used in all kinds of PL/M-86 expressions. Untyped procedures may accept parameters, but do not return a value. Untyped procedures are activated by CALL statements. Figure 2-55 shows how simple typed and untyped procedures may be declared and then activated.

The statements forming the body of a procedure need not exist within the module that activates the procedure. The activating module can declare the procedure EXTERNAL, and the LINK-86 utility will connect the two modules.

PL/M-86 procedures can be written to handle interrupts. Procedures also may be declared REENTRANT, making them concurrently usable by different tasks in a multitasking system. PL/M-86 also has about 50 procedures built into the language, including facilities for:

- converting variables from one type to another
- shifting and rotating bits
- performing input and output
- manipulating strings
- activating the CPU $\overline{\text{LOCK}}$ signal.

```
/*DECLARATION OF A TYPED PROCEDURE THAT
  ACCEPTS TWO REAL PARAMETERS AND RETURNS A REAL VALUE*/
  AVG: PROCEDURE (X,Y) REAL;
       DECLARE (X,Y) REAL;
       RETURN (X+Y)/2.0;
       END AVG;

/*ACTIVATING A TYPED PROCEDURE*/
LOW = 2.0;
HIGH = 3.0;
TOTAL = TOTAL + AVG (LOW,HIGH);  /*2.5 IS ADDED TO TOTAL*/

/*DECLARATION OF AN UNTYPED PROCEDURE
  THAT ACCEPTS ONE PARAMETER*/
TEST: PROCEDURE (X);
  DECLARE X BYTE;
  IF X = 0H THEN
      COUNT = COUNT + 1;
  END TEST;

/*ACTIVATING AN UNTYPED PROCEDURE*/
CALL TEST (ALPHA);  /*COUNT IS INCREMENTED
                      IF ALPHA = 0*/
```

Figure 2-55. PL/M-86 Procedures

ASM-86

Programmers who are familiar with the CPU architecture can obtain complete access to all processor facilities with ASM-86. Since the execution unit on both the 8086 and the 8088 is identical, both processors use the same assembly language. Examples of processor features not accessible through PL/M-86 that can be utilized in ASM-86 programs include: software interrupts, the WAIT and ESC instructions and explicit control of the segment registers.

An ASM-86 program often can be written to execute faster and/or to use less memory than the same program written in PL/M-86. This is because the compiler has a limited "knowledge" of the entire program and must generate a generalized set of machine instructions that will work in all situations, but may not be optimal in a particular situation. For example, assume that the elements of an array are to be summed and the result placed in a variable in memory. The machine instructions generated by the PL/M-86 compiler would move the next array element to a register and then add the register to the sum variable in memory. An ASM-86 programmer, knowing that a register will be "safe" while the array is summed, could instead add all the array elements to a register and then move the register to the sum variable, saving one instruction execution per array element.

It is easier to write assembly language programs in ASM-86 than it is in many assembly languages. ASM-86 contains powerful data structuring facilities that are usually found only in high-level languages. ASM-86 also simplifies the programmer's "view" of the 8086/8088 machine instruction set. For example, although there are 28 different types of MOV machine instructions, the programmer always writes a single form of the instruction:

```
MOV destination-operand, source-operand
```

The assembler generates the correct machine-instruction form based on the attributes of the source and destination operands (attributes are covered later in this section). Finally, the ASM-86 assembler performs extensive checks on the consistency of operand definition versus operand use in instructions, catching many common types of clerical errors.

Statements

Compared to many assemblers, ASM-86 accepts a relaxed statement format (see figure 2-56). This helps to reduce clerical errors and allows programmers to format their programs for better readability. Variable and label names may be up to 31 characters long and are not restricted to alphabetic and numeric characters. In particular, the underscore (_) may be used to improve the readability of long names. Blanks may be inserted freely between identifiers (there are no "column" requirements), and statements also may span multiple lines.

All ASM-86 statements are classified as instructions or directives. A clear distinction must be made here between ASM-86 instructions and

```
; THIS STATEMENT CONTAINS A COMMENT ONLY

MOV     AX, [BX + 3]                    ; TYPICAL ASM-86 INSTRUCTION
   MOV AX,        [BX + 3]              ; BLANKS NOT SIGNIFICANT
MOV     AX,
&       [BX + 3]                        ; CONTINUED STATEMENTS

ZERO    EQU    0                        ; SIMPLE ASM-86 DIRECTIVE
CUR_PROJ EQU   PROJECT [BX] [SI]        ; MORE COMPLEX DIRECTIVE
THE_STACK_STARTS_HERE SEGMENT           ; LONG IDENTIFIER
TIGHT_LOOP: JMP TIGHT_LOOP              ; LABELLED STATEMENT
MOV ES: DATA_STRING [SI], AL            ; SEGMENT OVERRIDE PREFIX
WAIT: LOCK XCHG AX,SEMAPHORE            ; LABEL & LOCK PREFIX
```

Figure 2-56. ASM-86 Statements

8086/8088 machine instructions. The assembler generates machine instructions from ASM-86 instructions written by a programmer. Each ASM-86 instruction produces one machine instruction, but the form of the generated machine instruction will vary according to the operands written in the ASM-86 instruction. For example, writing

```
MOV BL,1
```

produces a byte-immediate-to-register MOV, while writing

```
MOV TERMINAL_NO,BX
```

produces a word-register-to-memory MOV. To the programmer, though, there is simply a MOV source-to-destination instruction.

ASM-86 instructions are written in the form:

```
(label:) (prefix) mnemonic (operand(s)) (;comment)
```

where parentheses denote optional fields (the parentheses are not actually written by programmers). The label field names the storage location containing the machine instruction so that it can be referred to symbolically as the target of a JMP instruction elsewhere in the program. Writing a prefix causes ASM-86 to generate one of the special prefix bytes (segment override, bus lock or repeat) immediately preceding the machine instruction. The mnemonic identifies the type of instruction (MOV for move, ADD for add, etc.) that is to be generated. Zero, one or two operands may be written next, separated by commas, according to the requirements of the instruction. Finally, writing a semicolon signifies that what follows is a comment. Comments do not affect the execution of a program, but they can greatly improve its clarity; all good ASM-86 programs are thoughtfully commented.

Writing a directive gives ASM-86 information to use in generating instructions, but does not itself produce a machine instruction. About 20 different directives are available in ASM-86. Directives are written like this:

```
(name) mnemonic (operand(s)) (;comment)
```

Some directives require a name to be present, while others prohibit a name. ASM-86 recognizes the directive from the mnemonic keyword written in the next field. Any operands required by the directive are written next, separated by commas. A comment may be written as the last field of a directive.

Some of the more commonly used directives define procedures (PROC), allocate storage for variables (DB, DW, DD) give a descriptive name to a number or an expression (EQU), define the bounds of segments (SEGMENT and ENDS), and force instructions and data to be aligned at word boundaries (EVEN).

Constants

Binary, decimal, octal and hexadecimal numeric constants (see figure 2-57) may be written in ASM-86 statements; the assembler can perform basic arithmetic operations on these as well. All numbers must, however, be integers and must be representable in 16 bits including a sign bit. Negative numbers are assembled in standard two's complement notation.

Character constants are enclosed in single quotes and may be up to 255 characters long when used

```
MOV         STRING [SI], 'A'      ; CHARACTER
MOV         STRING [SI], 41H      ; EQUIVALENT IN HEX
ADD         AX, 0C4H              ; HEX CONSTANT MUST START WITH NUMERAL
OCTAL_8     EQU  100              ; OCTAL
OCTAL_9     EQU  10Q              ; OCTAL ALTERNATE
ALL_ONES    EQU  11111111B        ; BINARY
MINUS_5     EQU  -5               ; DECIMAL
MINUS_6     EQU  -6D              ; DECIMAL ALTERNATE
```

Figure 2-57. ASM-86 Constants

to initialize storage. When used as immediate operands, character constants may be one or two bytes long to match the length of the destination operand.

Defining Data

Most ASM-86 programs begin by defining the variables with which they will work. Three directives, DB, DW and DD, are used to allocate and name data storage locations in ASM-86 (see figure 2-58). The directives are used to define storage in three different units: DB means "define byte," DW means "define word," and DD means "define doubleword." The operands of these directives tell the assembler how many storage units to allocate and what initial values, if any, with which to fill the locations.

```
A_SEG    SEGMENT
ALPHA    DB    ?          ; NOT INITIALIZED
BETA     DW    ?          ; NOT INITIALIZED
GAMMA    DD    ?          ; NOT INITIALIZED
DELTA    DB    ?          ; NOT INITIALIZED
EPSILON  DW    5          ; CONTAINS 05H
A_SEG    ENDS

B_SEG    SEGMENT AT 55H ; SPECIFYING BASE ADDRESS
IOTA     DB    'HELLO'    ; CONTAINS 48 45 4C 4C 4F H
KAPPA    DW    'AB'       ; CONTAINS 42 41 H
LAMBDA   DD    B_SEG      ; CONTAINS 0000 5500 H
MU       DB    100 DUP 0  ; CONTAINS (100 X) 00H
B_SEG    ENDS
```

	ATTRIBUTES			OPERATORS	
VARIABLE	SEGMENT	OFFSET	TYPE	LENGTH	SIZE
ALPHA	A_SEG	0	1	1	1
BETA	A_SEG	1	2	1	2
GAMMA	A_SEG	3	4	1	4
DELTA	A_SEG	7	1	1	1
EPSILON	A_SEG	8	2	1	2
IOTA	B_SEG	0	1	5	5
KAPPA	B_SEG	5	2	1	2
LAMBDA	B_SEG	7	4	1	4
MU	B_SEG	11	1	100	100

Figure 2-58. ASM-86 Data Definitions

For every variable in an ASM-86 program, the assembler keeps track of three attributes: segment, offset and type. Segment identifies the segment that contains the variable (segment control is covered shortly). Offset is the distance in bytes of the variable from the beginning of its containing segment. Type identifies the variable's allocation unit (1 = byte, 2 = word, 4 = doubleword). When a variable is referenced in an instruction, ASM-86 uses these attributes to determine what form of the instruction to generate. If the variable's attributes conflict with its usage in an instruction, ASM-86 produces an error message. For example, attempting to add a variable defined as a word to a byte register is an error. There are cases where the assembler must be explicitly told an operand's type. For example, writing MOVE [BX],5 will produce an error message because the assembler does not know if [BX] refers to a byte, a word or a doubleword. The following operators can be used to provide this information: BYTE PTR, WORD PTR and DWORD PTR. In the previous example, a word could be moved to the location referenced by [BX] by writing MOVE WORD PTR [BX],5.

ASM-86 also provides two built-in operators, LENGTH and SIZE, that can be written in ASM-86 instructions along with attribute information. LENGTH causes the assembler to return the number of storage units (bytes, words or doublewords) occupied by an array. SIZE causes ASM-86 to return the total number of bytes occupied by a variable or an array. These operators and attributes make it possible to write generalized instruction sequences that need not be changed (only reassembled) if the attributes of the variables change (e.g., a byte array is changed to a word array). See figure 2-59 for an example of using the attributes and attribute operators.

Records

ASM-86 provides a means of symbolically defining individual bits and strings of bits within a byte or a word. Such a definition is called a record, and each named bit string (which may consist of a single bit) in a record is called a field. Records promote efficient use of storage while at the same time improving the readability of the program and reducing the likelihood of clerical errors. Defining a record does not allocate storage; rather, a record is a template that tells the assembler the name and location of each bit field within the byte or word. When a field name is written later in an instruction, ASM-86 uses the record to generate an immediate mask for instructions like TEST, AND, OR, etc., or an immediate count for shifts and rotates. See figure 2-60 for an example of using a record.

```
; SUM THE CONTENTS OF TABLE INTO AX
TABLE       DW        50  DUP(?)
; NOTE SAME INSTRUCTIONS WOULD WORK FOR
; TABLE     DB        25  DUP(?)
; TABLE     DW       118  DUP(?),  ETC.

            SUB       AX,AX              ; CLEAR SUM
            MOV       CX, LENGTH TABLE   ; LOOP TERMINATOR
            MOV       SI, SIZE TABLE     ;POINT SUBSCRIPT
                                         ; TO END OF TABLE
ADD_NEXT:   SUB       SI, TYPE TABLE     ; BACK UP ONE ELEMENT
            ADD       AX, TABLE [SI]     ; ADD ELEMENT
            LOOP      ADD_NEXT           ; UNTIL CX = 0
            ; AX CONTAINS SUM
```

Figure 2-59. Using ASM-86 Attributes and Attribute Operators

```
            EMP_BYTE  DB  ?              ; 1 BYTE, UNINITIALIZED
            ; BIT DEFINITIONS:
            ;    7-2    : YEARS EMPLOYED
                 1      : SEX (1 = FEMALE)
                 0      : STATUS (1 = EXEMPT)
            EMP_BITSRECORD               ;RECORD DEFINED HERE
            &          YRS_EMP : 6,
            &          SEX : 1,
            &          STATUS : 1
            .
            .
            .
            ; SELECT NONEXEMPT FEMALES EMPLOYED 10 + YEARS

            MOV        AL, EMP_BYTE      ; KEEP ORIGINAL INTACT
            TEST       AL, MASK SEX      ; FEMALE ?
            JZ         REJECT            ; NO, QUITE
            TEST       AL, MASK STATUS   ; NONEXEMPT?
            JNZ        REJECT            ;NO, QUIT
            SHR        AL, CL            ; ISOLATE YEARS
            CMP        AL, 11            ; >=10 YEARS?
            JL         REJECT            ; NO, QUIT
            ; PROCESS SELECTED EMPLOYEE
            .
            .
            .
REJECT:     ; PROCESS REJECTED EMPLOYEE
            .
            .
            .                            ; RECORD USED HERE
            MOV        CL, YRS_EMP       ; GET SHIFT COUNT
```

Figure 2-60. Using an ASM-86 RECORD Definition

Structures

An ASM-86 structure is a map, or template, that gives names and attributes (length, type, etc.) to a collection of fields. Each field in a structure is defined using DB, DW and DD directives; however, no storage is allocated to the structure. Instead, the structure becomes associated with a particular area of memory when a field name is referenced in an instruction along with a base value. The base value "locates" the structure; it may be a variable name or a base register (BX or BP). The structure may be associated with another area of memory by specifying a different base value. Figure 2-61 shows how a simple structure may be defined and used. Note that a structure field may itself be a structure, allowing much more complex organizations to be laid out.

Structures are particularly useful in situations where the same storage format is at multiple locations, where the location of a collection of variables is not known at assembly-time, and where the location of a collection of variables changes during execution. Applications include multiple buffers for a single file, list processing and stack addressing.

Addressing Modes

Figure 2-62 provides sample ASM-86 coding for each of the 8086/8088 addressing modes. The assembler interprets a bracketed reference to BX, BP, SI or DI as a base or index register to be used to construct the effective address of a memory operand. An unbracketed reference means the register itself is the operand.

The following cases illustrate typical ASM-86 coding for accessing arrays and structures, and show which addressing mode the assembler specifies in the machine instruction it generates:

- If ALPHA is an array, then ALPHA [SI] is the element indexed by SI, and ALPHA [SI + 1] is the following byte (indexed).
- If ALPHA is the base address of a structure and BETA is a field in the structure, then ALPHA.BETA selects the BETA field (direct).
- If register BX contains the base address of a structure and BETA is a field in the structure, then [BX].BETA refers to the BETA field (based).

```
EMPLOYEE        STRUC
   SSN          DB  9     DUP(?)
   RATE         DB  1     DUP(?)
   DEPT         DW  1     DUP(?)
   YR_HIRED     DB  1     DUP(?)
   EMPLOYEE     ENDS

MASTER          DB  12    DUP(?)
TXN             DB  12    DUP(?)

; CHANGE RATE IN MASTER TO VALUE IN TXN.
                MOV       AL, TXN.RATE
                MOV       MASTER.RATE, AL

; ASSUME BX POINTS TO AN AREA CONTAINING
;       DATA IN THE SAME FORMAT AS THE EMPLOYEE
;       STRUCTURE. ZERO THE SECOND DIGIT
;       OF SSN
                MOV       SI, 1   ; INDEX VALUE OF 2ND DIGIT
                MOV       [BX].SSN[SI],0
```

Figure 2-61. Using an ASM-86 Structure

```
ADD     AX, BX              ; REGISTER ← REGISTER
ADD     AL, 5               ; REGISTER ← IMMEDIATE
ADD     CX, ALPHA           ; REGISTER ← MEMORY (DIRECT)
ADD     ALPHA, 6            ; MEMORY (DIRECT) ← IMMEDIATE
ADD     ALPHA, DX           ; MEMORY (DIRECT) ← REGISTER
ADD     BL, [BX]            ; REGISTER ← MEMORY (REGISTER INDIRECT)
ADD     [SI], BH            ; MEMORY (REGISTER INDIRECT) ← IMMEDIATE
ADD     [PP].ALPHA, AH      ; MEMORY (BASED) ← REGISTER
ADD     CX, ALPHA [SI]      ; REGISTER ← MEMORY (INDEXED)
ADD     ALPHA [DI+2], 10    ; MEMORY (INDEXED) ← IMMEDIATE
ADD     [BX].ALPHA [SI], AL ; MEMORY (BASED INDEXED) ← REGISTER
ADD     SI, [BP+4] [DI]     ; REGISTER ← MEMORY (BASED INDEXED)
IN      AL, 30              ; DIRECT PORT
OUT     DX, AX              ; INDIRECT PORT
```

Figure 2-62. ASM-86 Addressing Mode Examples

- If register BX contains the address of an array, then [BX] [SI] refers to the element indexed by SI (based indexed).
- If register BX points to a structure whose ALPHA field is an array, then [BX] .ALPHA [SI] selects the element indexed by SI (based indexed).
- If register BX points to a structure whose ALPHA field is itself a structure, then [BX].ALPHA.BETA refers to the BETA field of the ALPHA substructure (based).
- If register BX points to a structure and the ALPHA field of the structure is an array and each element of ALPHA is a structure, then [BX].ALPHA[SI + 3].BETA refers to the field BETA in the element of ALPHA indexed by [SI + 3] (based indexed).

Note that DI may be used in place of SI in these cases and that BP may be substituted for BX. Without a segment override prefix, expressions containing BP refer to the current stack segment, and expressions containing BX refer to the current data segment.

Segment Control

An ASM-86 program is organized into a series of named segments. These are "logical" segments; they are eventually mapped into 8086/8088 memory segments, but this usually is not done until the program is located. A SEGMENT directive starts a segment, and an ENDS directive ends the segment (see figure 2-63). All data and instructions written between SEGMENT and ENDS are part of the named segment. In small programs, variables often are defined in one or two segment(s), stack space is allocated in another segment, and instructions are written in a third or fourth segment. It is perfectly possible, however, to write a complete program in one segment; if this is done, all the segment registers will contain the same base address; that is, the memory segments will completely overlap. Large programs may be divided into dozens of segments.

The first instructions in a program usually establish the correspondence between segment names and segment registers, and then load each segment register with the base address of its corresponding segment. The ASSUME directive tells the assembler what addresses will be in the segment registers at execution time. The assembler checks each memory instruction operand, determines which segment it is in and which segment register contains the address of that segment. If the assumed register is the register expected by the hardware for that instruction type, then the assembler generates the machine instruction normally. If, however, the hardware expects one segment register to be used, and the operand is *not* in the segment pointed to by that register, then the assembler automatically precedes the machine instruction with a segment override prefix byte. (If the segment cannot be overridden, the assembler produces an error message.) An example may clarify this. If register BP is used in an instruction, the 8086 and 8088 CPUs expect, as a default, that the memory operand will be located in the segment pointed to by SS—in the current

```
DATA_SEG     SEGMENT
   ; DATA DEFINITIONS GO HERE
DATA_SEG     ENDS

STACK_SEG    SEGMENT
   ; ALLOCATE 100 WORDS FOR A STACK AND
   ;    LABEL THE INITIAL TOS FOR LOADING SP.
         DW 100 DUP(?)
STACK TOP LABEL WORD
STACK_SEG    ENDS

CODE_SEG     SEGMENT
   ; GIVE ASSEMBLER INITIAL REGISTER-TO-SEGMENT
   ;   CORRESPONDENCE. NOTE THAT IN THIS
   ;   PROGRAM THE EXTRA SEGMENT INITIALLY
   ;   OVERLAPS THE DATA SEGMENT ENTIRELY.
ASSUME  CS: CODE_SEG,
&          DS: DATA_SEG,
&          ES: DATA_SEG,
&          SS: STACK_SEG

START:   ; THIS IS THE BEGINNING OF THE PROGRAM.
         ; LOC-86 WILL PLACE A JMP TO THIS
         ; LOCATION AT ADDRESS FFFF0H.

   ; LOAD THE SEGMENT REGISTERS. CS DOES NOT
   ;    HAVE TO BE LOADED BECAUSE SYSTEM
   ;    RESET SETS IT TO FFFFH, AND THE
   ;    LONG JMP INSTRUCTION AT THAT ADDRESS
   ;    UPDATES IT TO THE ADDRESS OF CODE_SEG.
   ;    SEGMENT REGISTERS ARE LOADED FROM AX
   ;    BECAUSE THERE IS NO IMMEDIATE-TO-
   ;    SEGMENT_REGISTER FORM OF THE MOV
   ;    INSTRUCTION.

                     MOV   AX, DATA_SEG
                     MOV   DS, AX
                     MOV   ES, AX
                     MOV   AX, STACK_SEG
                     MOV   SS, AX
; SET STACK POINTER TO INITIAL TOS.
                     MOV   SP, OFFSET STACK_TOP

; SEGMENTS ARE NOW ADDRESSABLE.
; MAIN PROGRAM CODE GOES HERE.
CODE_SEG                   ENDS

; NEXT STATEMENT ENDS ASSEMBLY AND TELLS
;    LOC-86 THE PROGRAMS STARTING ADDRESS.

                     END   START
```

Figure 2-63. Setting Up ASM-86 Segments

stack segment. A programmer may, however, choose to use BP to address a variable in the current data segment—the segment pointed to by DS. The ASSUME directive enables the assembler to detect this situation and to automatically generate the needed override prefix.

It also is possible for a programmer to explicitly code segment override prefixes rather than relying on the assembler. This may result in a somewhat better-documented program since attention is called to the override. The disadvantage of explicit segment overrides is that the assembler does not check whether the operand is in fact addressable through the overriding segment register.

ASM-86, in conjunction with the relocation and linkage facilities, provides much more sophisticated segment handling capabilities than have been described in this introduction. For example, different logical segments may be combined into the same physical segment, and segments may be assigned the same physical locations (allowing a "common" area to be accessed by different programs using different variable and label names).

Procedures

Procedures may be written in ASM-86 as well as in PL/M-86. In fact, procedures written in one language are callable from the other, provided that a few simple conventions are observed in the ASM-86 program. The purpose of ASM-86 procedures is the same as in PL/M-86: to simplify the design of complex programs and to make a single copy of a commonly-used routine accessible from anywhere in the program.

An ASM-86 program activates a procedure with a CALL instruction. The procedure terminates with a RET instruction, which transfers control to the instruction following the CALL. Parameters may be passed in registers or pushed onto the stack before calling the procedure. The RET instruction can discard stack parameters before returning to the caller.

Unlike PL/M-86 procedures, ASM-86 procedures are executable where they are coded, as well as by a CALL instruction. Therefore, ASM-86 procedures often are defined following the main program logic, rather than preceding it as in PL/M-86. Figure 2-64 shows how procedures may be defined and called in ASM-86. Section 2-10 contains examples of procedures that accept parameters on the stack.

LINK-86

Fundamentally, LINK-86 combines separate relocatable object modules into a single program. This process consists primarily of combining (logical) segments of the same name into single segments, adjusting relative addresses when segments are combined, and resolving external references.

A programmer can use a procedure that is actually contained in another module by naming the procedure in an ASM-86 EXTRN directive, or declaring the procedure to be EXTERNAL in PL/M-86. The procedure is defined or declared PUBLIC in the module where it actually resides, meaning that it can be used by other modules. When LINK-86 encounters such an external reference, it searches through the other modules in its input, trying to find the matching PUBLIC declaration. If it finds the referenced object, it links it to the reference, "satisfying" the external reference. If it cannot satisfy the reference, LINK-86 prints a diagnostic message. LINK-86 also checks PL/M-86 procedure calls and function references to insure that the parameters passed to a procedure are the type expected by the procedure.

LINK-86 gives the programmer, particularly the ASM-86 programmer, great control over segments (segments may be combined end to end, renamed, assigned the same locations, etc.). LINK-86 also produces a map that summarizes the link process and lists any unusual conditions encountered. While the output of LINK-86 is generally input to LOC-86, it also may again be input to LINK-86 to permit modules to be linked in incremental groups.

LOC-86

LOC-86 accepts the single relocatable object module produced by LINK-86 and binds the memory references in the module to actual memory addresses. Its output is an absolute object module ready for loading into the memory of an execution vehicle. LOC-86 also inserts a

```
FREQUENCY       DB      256 DUP (0)
.
.
USART_DATA      EQU     0FF0H           ; DATA PORT ADDRESS
USART_STAT      EQU     0FF2H           ; STATUS PORT ADDRESS
.
.
NEXT:           CALL    CHAR_IN
                CALL    COUNT_IT
                JMP     NEXT

CHAR_IN         PROC
    ; THIS PROCEDURE DOES NOT TAKE PARAMETERS.
    ;   IT SAMPLES THE USART STATUS PORT
    ;   UNTIL A CHARACTER IS READY, AND
    ;   THEN READS THE CHARACTER INTO AL
                MOV     DX, USART_STAT
    AGAIN:      IN      AL, DX          ; READ STATUS
                AND     AL, 2           ; CHARACTER PRESENT?
                JZ      AGAIN           ; NO, TRY AGAIN
                MOV     DX, USART_DATA
                IN      AL, DX          ; YES, READ CHARACTER
                RET
CHAR_IN         ENDP

COUNT_IT        PROC
    ; THIS PROCEDURE EXPECTS A CHARACTER IN AL.
    ;   IT INCREMENTS A COUNTER IN A FREQUENCY
    ;   TABLE BASED ON THE BINARY VALUE OF
    ;   THE CHARACTER.
                XOR     AH, AH          ; CLEAR HIGH BYTE
                MOV     SI, AL          ; INDEX INTO TABLE
                INC     FREQUENCY [S]; BUMP THE COUNTER
                RET
COUNT_IT        ENDP
```

Figure 2-64. ASM-86 Procedures

direct intersegment JMP instruction at location FFFF0H. The target of the JMP instruction is the logical beginning of the program. When the 8086 or 8088 is reset, this instruction is automatically executed to restart the system. LOC-86 produces a memory map of the absolute object module and a table showing the address of every symbol defined in the program.

LIB-86

LIB-86 is a valuable adjunct to the R & L programs. It is used to maintain relocatable object modules in special files called libraries. Libraries are a convenient way to make collections of modules available to LINK-86. When a module being linked refers to "external" data or instructions, LINK-86 can automatically search a series of libraries, find the referenced module, and include it in the program being created.

OH-86

OH-86 converts an absolute object module into Intel's standard hexadecimal format. This format is used by some PROM programmers and system loaders, such as the iSBC 957™ and SDK-86 loaders.

CONV-86

Users who have developed substantial, fully-tested assembly language programs for the 8080/8085 microprocessors may want to use CONV-86 to automatically convert large amounts of this code into ASM-86 source code (see figure 2-65). CONV-86 accepts an ASM-80 source program as input and produces an ASM-86 source program as output, plus a print file that documents the conversion and lists any diagnostic messages.

Some programs cannot be completely converted by CONV-86. Exceptions include:

- self-modifying code,
- software timing loops,
- 8085 RIM and SIM instructions,
- interrupt code, and
- macros.

By using the diagnostic messages produced by CONV-86, the converted ASM-86 source file can be manually edited to clean up any sections not converted. A converted program is typically 10-20% larger than the ASM-80 version and does not take full advantage of the 8086/8088 architecture. However, the development time saved by using CONV-86 can make it an attractive alternative to rewriting working programs from scratch.

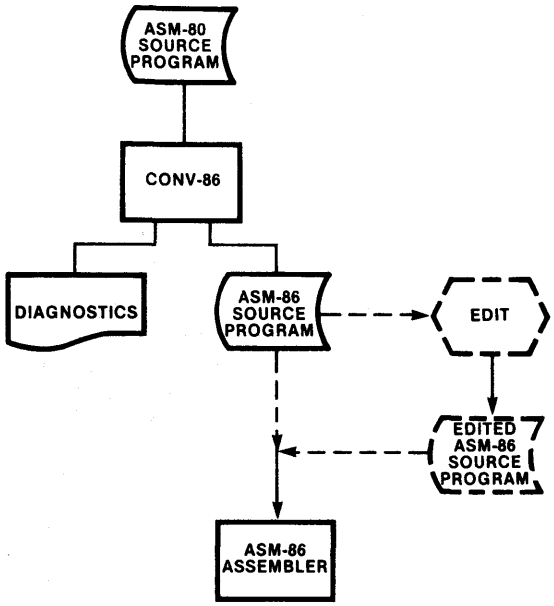


Figure 2-65. ASM-80/ASM-86 Conversion

Sample Programs

Figures 2-66 and 2-67 show how a simple program might be written in PL/M-86 and ASM-86. The program simulates a pair of rolling dice and executes on an Intel SDK-86 System Design Kit. The SDK-86 is an 8086-based computer with memory, parallel and serial I/O ports, a keypad and a display. The SDK-86 is implemented on a single PC board which includes a large prototype area for system expansion and experimentation. A ROM-based monitor program provides a user interface to the system; commands are entered through the keypad and monitor responses are written on the display. With the addition of a cable and software interface (called SDK-C86), the SDK-86 may be connected to an Intellec® Microcomputer Development System. In this mode, the user enters monitor commands from the Intellec keyboard and receives replies on the Intellec CRT display.

The dice program runs on an SDK-86 that is connected to an Intellec® Microcomputer Development System. The program displays two continuously changing digits in the upper left corner of the Intellec display. The digits are random numbers in the range 1-6. A roll is started by entering a monitor GO command. Pressing the INTR key on the SDK-86 keypad stops the roll.

There are two procedures in the PL/M-86 version of the dice program. The first is called CO for console output. This is an untyped PUBLIC procedure that is supplied on an SDK-C86 diskette. CO is written in PL/M-86 and outputs one character to the Intellec console. It is declared EXTERNAL in the dice program because it exists in another module. LINK-86 searches the SDK-C86 library for CO and includes it in the single relocatable object module it builds.

RANDOM is an internal typed procedure; it is contained in the dice module and returns a word value that is a random number between 1 and 6. RANDOM does not use any parameters and is activated in the parameter list passed to CO. When CO is called like this, first RANDOM is activated, then 30 is added to the number it returns and the sum is passed to CO.

```
PL/M-86 COMPILER    DICE

ISIS-II PL/M-86 V1.2 COMPILATION OF MODULE DICE
OBJECT MODULE PLACED IN :F1:DICE.OBJ
COMPILER INVOKED BY:  PLM86 :F1:DICE.P86 XREF

   1          DICE:  DO;
              /* THIS PROGRAM SIMULATES THE ROLL OF A PAIR OF DICE */

              /* GIVE NAMES TO CONSTANTS */
   2    1     DECLARE CLEAR$CRT1      LITERALLY '01BH';  /* INTELLEC  */
   3    1     DECLARE CLEAR$CRT2      LITERALLY '045H';  /*  CRT      */
   4    1     DECLARE HOME$CURSOR1    LITERALLY '01BH';  /*  CONTROL  */
   5    1     DECLARE HOME$CURSOR2    LITERALLY '048H';  /*  CODES    */
   6    1     DECLARE SPACE           LITERALLY '020H';  /*ASCII BLANK*/

              /* PROGRAM VARIABLES */
   7    1     DECLARE (RANDOM$NUMBER,SAVE)  WORD;

              /* CONSOLE OUTPUT PROCEDURE */
   8    1     CO:  PROCEDURE(X) EXTERNAL;
   9    2          DECLARE X    BYTE;
  10    2          END CO;

              /* RANDOM NUMBER GENERATOR PROCEDURE              */
              /* ALGORITHM FOR 16-BIT RANDOM NUMBER FROM:       */
              /*    "A GUIDE TO PL/M PROGRAMMING FOR            */
              /*     MICROCOMPUTER APPLICATIONS,"               */
              /*    DANIEL D. MCCRACKEN,                        */
              /*    ADDISON-WESLEY, 1978                        */
  11    1     RANDOM:  PROCEDURE WORD;
  12    2        RANDOM$NUMBER = SAVE;        /*START WITH OLD NUMBER*/
  13    2        RANDOM$NUMBER = 2053 * RANDOM$NUMBER + 13849;
  14    2        SAVE = RANDOM$NUMBER;        /*SAVE FOR NEXT TIME*/
                 /*FORCE 16-BIT NUMBER INTO RANGE 1-6*/
  15    2        RANDOM$NUMBER = RANDOM$NUMBER MOD 6  + 1;
  16    2        RETURN RANDOM$NUMBER;
  17    2        END RANDOM;

              /* MAIN ROUTINE */
              /* CLEAR THE SCREEN*/
  18    1     CALL CO(CLEAR$CRT1);
  19    1     CALL CO(CLEAR$CRT2);

              /* ROLL THE DICE UNTIL INTERRUPTED */
  20    1     DO WHILE 1;   /*"DO FOREVER"*/
                 /*NOTE THAT ADDING 30 TO THE DIE VALUE */
                 /*  CONVERTS IT TO ASCII.              */
  21    2        CALL CO(RANDOM + 030H);      /*1ST DIE*/
  22    2        CALL CO(SPACE);              /*BLANK*/
  23    2        CALL CO(RANDOM + 030H);      /*2ND DIE*/
                 /* HOME THE CURSOR */
  24    2        CALL CO(HOME$CURSOR1);
  25    2        CALL CO(HOME$CURSOR2);
  26    2        END;

  27    1     END DICE;


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

  DEFN  ADDR   SIZE  NAME, ATTRIBUTES, AND REFERENCES
 ----- ------ -----  --------------------------------
     2               CLEARCRT1              LITERALLY
                       18

     3               CLEARCRT2              LITERALLY
                       19

     8  0000H        CO                     PROCEDURE EXTERNAL(0) STACK=0000H
                       18   19   21   22   23   24   25

     1  0002H    71  DICE                   PROCEDURE STACK=0004H

     4               HOMECURSOR1            LITERALLY
                       24

     5               HOMECURSOR2            LITERALLY
                       25

    11  0049H    44  RANDOM                 PROCEDURE WORD STACK=0002H
                       21   23
```

Figure 2-66. Sample PL/M-86 Program

```
  7  0000H     2  RANDOMNUMBER          WORD
                                          12   13   14   15   16

  7  0002H     2  SAVE                  WORD
                                          12   14

  6               SPACE                 LITERALLY
                                          22

  8  0000H     1  X                     BYTE PARAMETER
                                           9

MODULE INFORMATION:

     CODE AREA SIZE     = 0075H    117D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 0004H      4D
     MAXIMUM STACK SIZE = 0004H      4D
     51 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

Figure 2-66. Sample PL/M-86 Program (Cont'd.)

```
MCS-86 MACRO ASSEMBLER    DICE

ISIS-II MCS-86 MACRO ASSEMBLER V2.0 ASSEMBLY OF MODULE DICE
OBJECT MODULE PLACED IN :F1:DICE.OBJ
ASSEMBLER INVOKED BY: ASM86 :F1:DICE.A86 XREF

LOC  OBJ                  LINE     SOURCE

                             1     ; THIS PROGRAM SIMULATES THE ROLL OF A PAIR OF DICE
                             2
                             3     ; CONSOLE OUTPUT PROCEDURE
                             4             EXTRN   CO:NEAR
                             5
                             6     ; SEGMENT GROUP DEFINITIONS NEEDED FOR PL/M-86 COMPATIBILITY
                             7     CGROUP  GROUP   CODE
                             8     DGROUP  GROUP   DATA,STACK
                             9
                            10     ; INFORM ASSEMBLER OF SEGMENT REGISTER CONTENTS.
                            11             ASSUME  CS:CGROUP,DS:DGROUP,SS:DGROUP,ES:NOTHING
                            12
                            13     ; ALLOCATE DATA
----                        14     DATA    SEGMENT PUBLIC  'DATA'
                            15     ; NOTE THAT THE FOLLOWING ARE PASSED ON THE STACK TO THE PL/M-86
                            16     ;  PROCEDURE 'CO'.  BY CONVENTION, A BYTE PARAMETER IS PASSED IN
                            17     ;  THE LOW-ORDER 8-BITS OF A WORD ON THE STACK.  HENCE, THESE ARE
                            18     ;  DEFINED AS WORD VALUES, THOUGH THEY OCCUPY 1 BYTE ONLY.
0000 1B00                   19     CLEAR_CRT1      DW      01BH    ; INTELLEC
0002 4500                   20     CLEAR_CRT2      DW      045H    ;   CRT
0004 1B00                   21     HOME_CURSOR1    DW      01BH    ;   CONTROL
0006 4800                   22     HOME_CURSOR2    DW      048H    ;   CODES
0008 2000                   23     SPACE           DW      020H    ; ASCII BLANK
000A ????                   24     SAVE            DW      ?       ; HOLDS LAST 16-BIT RANDOM NUMBER
----                        25     DATA    ENDS
                            26
                            27
                            28     ; ALLOCATE STACK SPACE
----                        29     STACK   SEGMENT STACK   'STACK'
0000 (20                    30             DW      20 DUP (?)
     ????
     )
                            31     ; LABEL INITIAL TOS: FOR LATER USE.
0028                        32     STACK_TOP       LABEL   WORD
----                        33     STACK   ENDS
                            34
                            35
                            36     ; PROGRAM CODE
----                        37     CODE    SEGMENT PUBLIC  'CODE'
                            38
                            39
                            40     ; RANDOM NUMBER GENERATOR PROCEDURE
                            41     ; ALGORITHM FOR 16-BIT RANDOM NUMBER FROM:
                            42     ;   "A GUIDE TO PL/M PROGRAMMING FOR
                            43     ;   MICROCOMPUTER APPLICATIONS,"
                            44     ;   DANIEL D. MCCRACKEN
                            45     ;   ADDISON-WESLEY, 1978
0000                        46     RANDOM  PROC
0000 A10A00          R      47             MOV     AX,SAVE         ; NEW NUMBER =
```

Figure 2-67. ASM-86 Sample Program

```
MCS-86 MACRO ASSEMBLER    DICE

LOC  OBJ                  LINE     SOURCE

0003 B90508                 48             MOV     CX,2053         ;   OLD NUMBER * 2053
0006 F7E1                   49             MUL     CX              ;   * 13849
0008 051936                 50             ADD     AX,13849        ;
000B A30A00          R      51             MOV     SAVE,AX         ; SAVE FOR NEXT TIME
                            52             ; FORCE 16-BIT NUMBER INTO RANGE 1 - 6
                            53             ;   BY MODULO 6 DIVISION + 1
000E 2BD2                   54             SUB     DX,DX           ; CLEAR UPPER DIVIDEND
0010 B90600                 55             MOV     CX,6            ; SET DIVISOR
0013 F7F1                   56             DIV     CX              ; DIVIDE BY 6
0015 8BC2                   57             MOV     AX,DX           ; REMAINDER TO AX
0017 40                     58             INC     AX              ; ADD 1
0018 C3                     59             RET                     ; RESULT IN AX
                            60     RANDOM  ENDP
                            61
                            62
                            63     ; MAIN PROGRAM
                            64
                            65     ; LOAD SEGMENT REGISTERS
                            66     ; NOTE PROGRAM DOES NOT USE ES; CS IS INITIALIZED BY HARDWARE RESET;
                            67     ;  DATA & STACK ARE MEMBERS OF SAME GROUP, SO ARE TREATED AS A SINGLE
                            68     ;  MEMORY SEGMENT POINTED TO BY BOTH DS & SS.
0019 B8----          R      69     START:  MOV     AX,DGROUP
001C 8ED8                   70             MOV     DS,AX
001E 8ED0                   71             MOV     SS,AX
                            72
                            73     ; INITIALIZE STACK POINTER
0020 BC2800          R      74             MOV     SP,OFFSET DGROUP:STACK_TOP
                            75
                            76     ; CLEAR THE SCREEN
0023 FF360000        R      77             PUSH    CLEAR_CRT1
0027 E80000          E      78             CALL    CO
002A FF360200        R      79             PUSH    CLEAR_CRT2
002E E80000          E      80             CALL    CO
                            81
                            82     ; ROLL THE DICE UNTIL INTERRUPTED
0031 E8CCFF                 83     ROLL:   CALL    RANDOM          ; GET 1ST DIE IN AL
0034 0430                   84             ADD     AL,030H         ; CONVERT TO ASCII
0036 50                     85             PUSH    AX              ; PASS IT TO
0037 E80000          E      86             CALL    CO              ;   CONSOLE OUTPUT
003A FF360800        R      87             PUSH    SPACE           ; OUTPUT
003E E80000          E      88             CALL    CO              ;   A BLANK
0041 E8BCFF                 89             CALL    RANDOM          ; GET 2ND DIE IN AL
0044 0430                   90             ADD     AL,030H         ; CONVERT TO ASCII
0046 50                     91             PUSH    AX              ; PASS IT TO
0047 E80000          E      92             CALL    CO              ;   CONSOLE OUTPUT
                            93     ; HOME THE CURSOR
004A FF360400        R      94             PUSH    HOME_CURSOR1
004E E80000          E      95             CALL    CO
0051 FF360600        R      96             PUSH    HOME_CURSOR2
0055 E80000          E      97             CALL    CO
                            98     ; CONTINUE FOREVER
0058 EBD7                   99             JMP     ROLL
----                       100     CODE    ENDS
                           101


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

NAME            TYPE     VALUE  ATTRIBUTES, XREFS

??SEG . . . . SEGMENT           SIZE=0000H PARA PUBLIC
CGROUP. . . . GROUP             CODE   7# 11
CLEAR_CRT1. . V WORD    0000H   DATA  19# 77
CLEAR_CRT2. . V WORD    0002H   DATA  20# 79
CO. . . . . . L NEAR    0000H   EXTRN  4# 78 80 86 88 92 95 97
CODE. . . . . SEGMENT           SIZE=005AH PARA PUBLIC 'CODE'  7# 37 100
DATA. . . . . SEGMENT           SIZE=000CH PARA PUBLIC 'DATA'  8# 14 25
DGROUP. . . . GROUP             DATA STACK   8# 11 11 69 74
HOME_CURSOR1. V WORD    0004H   DATA  21# 94
HOME_CURSOR2. V WORD    0006H   DATA  22# 96
RANDOM. . . . L NEAR    0000H   CODE  46# 60 83 89
ROLL. . . . . L NEAR    0031H   CODE  83# 99
SAVE. . . . . V WORD    000AH   DATA  24# 47 51
SPACE . . . . V WORD    0008H   DATA  23# 87
STACK . . . . SEGMENT           SIZE=0028H PARA STACK 'STACK'
STACK_TOP . . V WORD    0028H   STACK  32# 74
START . . . . L NEAR    0019H   CODE  69# 104


ASSEMBLY COMPLETE, NO ERRORS FOUND
```

Figure 2-67. ASM-86 Sample Program (Cont'd.)

The ASM-86 version of the dice program operates like the PL/M-86 version. Since the program uses the PL/M-86 CO procedure for writing data to the Intellec console, it adheres to certain conventions established by the PL/M-86 compiler. The program's logical segments (called CODE, DATA and STACK—the program does not use an extra segment) are organized into two groups called CGROUP and DGROUP. All the members of a group of logical segments are located in the same 64k byte physical memory segment. Physically, the program's DATA and STACK segments can be viewed as "subsegments" of DGROUP.

PL/M-86 procedures expect parameters to be passed on the stack, so the program pushes each character before calling CO. Note that the stack will be "cleaned up" by the PL/M-86 procedure before returning (i.e., the parameter will be removed from the stack by CO).

2.10 Programming Guidelines and Examples

This section addresses 8086/8088 programming from two different perspectives. A series of general guidelines is presented first. These guidelines apply to all types of systems and are intended to make software easier to write, and particularly, easier to maintain and enhance. The second part contains a number of specific programming examples. Written primarily in ASM-86, these examples illustrate how the instruction set and addressing modes may be utilized in various, commonly encountered programming situations.

Programming Guidelines

These guidelines encourage the development of 8086/8088 software that is adaptable to change. Some of the guidelines refer to specific processor features and others suggest approaches to general software design issues. PL/M-86 programmers need not be concerned with the discussions that deal with specific hardware topics; they should, however, give careful attention to the system design subjects. **Systems that are designed in accordance with these recommendations should be less costly to modify or extend. In addition, they should be better-positioned to take advantage of new hardware and software products that are constantly being introduced by Intel.**

Segments and Segment Registers

Segments should be considered as independent logical units whose physical locations in memory *happen* to be defined by the contents of the segment registers. Programs should be independent of the actual contents of the segment registers and of the physical locations of segments in memory. For example, a program should not take advantage of the "knowledge" that two segments are physically adjacent to each other in memory. The single exception to this fully-independent treatment of segments is that a program may set up more than one segment register to point to the same segment in memory, thereby obtaining addressability through more than one segment register. For example, if both DS and ES point to the same segment, a string located in that segment may be used as a source operand in one string instruction and as a destination string in another instruction (recall that a destination string must be located in the extra segment).

Any data aggregate or construct such as an array, a structure, a string or a stack should be restricted to 64k bytes in length and should be wholly contained in one segment (i.e., should not cross a segment boundary).

Segment registers should only contain values supplied by the relocation and linkage facilities. Segment register values may be moved to and from memory, pushed onto the stack and popped from the stack. Segment registers should never be used to hold temporary variables nor should they be altered in any other way.

As an additional guideline, code should *not* be written within six bytes of the end of physical memory (or the end of the code segment if this segment is dynamically relocatable). Failure to observe this guideline could result in an attempted opcode prefetch from non-existent memory, hanging the CPU if READY is not returned.

Self-Modifying Code

It is possible to write a program that deliberately changes some of its own machine instructions

during execution. While this technique may save a few bytes or machine cycles, it does so at the expense of program clarity. This is particularly true if the program is being examined at the machine instruction level; the machine instructions shown in the assembly listing may not match those found in memory or monitored from the bus. It also precludes executing the code from ROM. Also, because of the prefetch queue within the 8086 and 8088, code that is self-modified within six bytes of the current point of execution cannot be guaranteed to execute as intended. (This code may already have been fetched.) Finally, a self-modifying program may prove incompatible with future Intel products that assume that the content of a code segment remains constant during execution.

A corrollary to this requirement is that variable data should not be placed in a code segment. Constant data may be written in a code segment, but this is not recommended for two reasons. First, programs are simpler to understand if they are uniformly subdivided into segments of code, data and stack. Second, placing data in a code segment can restrict the segment's position independence. This is because, in general, the segment base address of a data item may be changed, but the offset (displacement) of the data item may not. This means that the entire segment must be moved as a unit to avoid changing the offset of the constant data. If the constant data were located in a data segment or an extra segment, individual procedures within the code segment could be moved independently.

Input/Output

Since I/O devices vary so widely in their capabilities and their interface designs, I/O software is inevitably device dependent. Substituting a hard disk for a floppy disk, for example, necessitates software changes even though the disks are functionally identical. I/O software can, however, be designed to minimize the effect of device changes on programs.

Figure 2-68 illustrates a design concept that structures an I/O system into a hierarchy of separately compiled/assembled modules. This approach isolates application modules that use the input/output devices from all physical characteristics of the hardware with which they ultimately communicate. An application module that reads a disk file, for example, should have no knowledge of where the file is located on the disk, what size the disk sectors are, etc. This allows these characteristics to change without affecting the application module. To an application module, the I/O system appears to be a series of file-oriented commands (e.g., Open, Close, Read, Write). An application module would typically issue a command by calling a file system procedure.

The file system processes I/O command requests, perhaps checking for gross errors, and calls a procedure in the I/O supervisor. The I/O supervisor is a bridge between the functional I/O request of the application module and the physical I/O performed by the lowest-level modules in the hierarchy. There should be separate modules in the supervisor for different types of devices and some device-dependent code may be unavoidable at this level. The I/O supervisor would typically perform overhead activities such as maintaining disk directories.

The modules that actually communicate with the I/O devices (or their controllers) are at the lowest level in the hierarchy. These modules contain the bulk of the system's device-dependent code that will have to be modified in the event that a device is changed.

The 8089 Input/Output Processor is specifically designed to encourage the development of modular, hierarchical I/O systems. The 8089 allows knowledge of device characteristics to be "hidden" from not only application programs, but also from the operating system that controls the CPU. The CPU's I/O supervisor can simply prepare a message in memory that describes the nature of the operation to be performed, and then activate the 8089. The 8089 independently performs all physical I/O and notifies the CPU when the operation has been completed.

Operating Systems

Operating systems also should be organized in a hierarchy similar to the concept illustrated in figure 2-69. Application modules should "see" only the upper level of the operating system. This level might provide services like sending messages between application modules, providing time delays, etc. An intermediate level might consist of housekeeping routines that dispatch tasks, alter

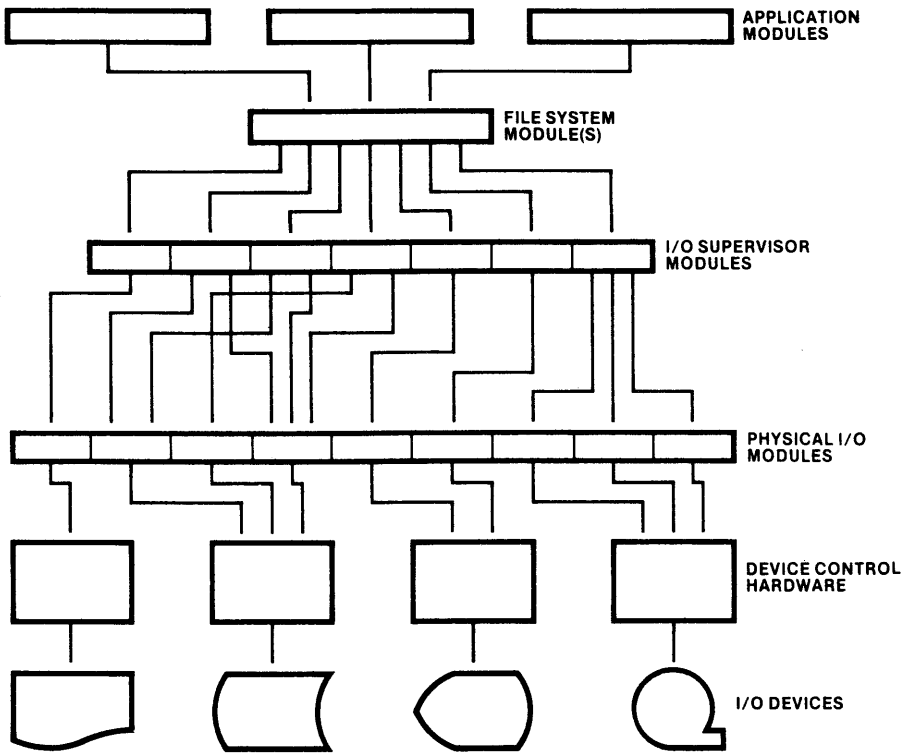


Figure 2-68. I/O System Hierarchy Concept

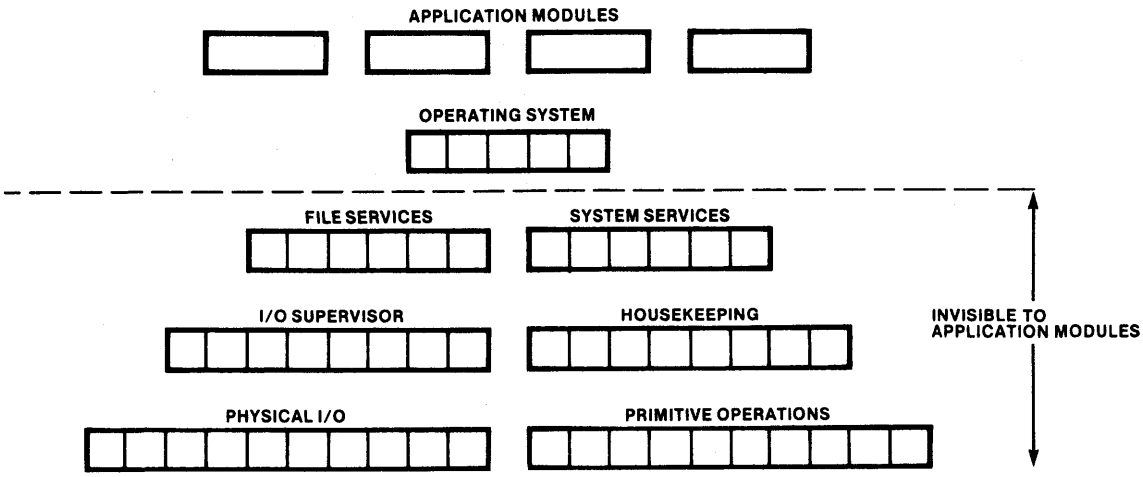


Figure 2-69. Operating System Hierarchy

priorities, manage memory, etc. At the lowest level would be the modules that implement primitive operations such as adding and removing tasks or messages from lists, servicing timer interrupts, etc.

Interrupt Service Procedures

Procedures that service external interrupts should be considered differently than those that service internal interrupts. A service procedure that is activated by an internal interrupt, may, and often should, be made reentrant. External interrupt procedures, on the other hand, should be viewed as temporary tasks. In this sense, a task is a single sequential thread of execution; it should not be reentered. The processor's response to an external interrupt may be viewed as the following sequence of events:

- the running (active) task is suspended,
- a new task, the interrupt service procedure, is created and becomes the running task,
- the interrupt task ends, and is deleted,
- the suspended task is reactived and becomes the running task from the point where it was suspended.

An external interrupt procedure should only be interruptable by a request that activates a different interrupt procedure. When the number of interrupt sources is not too large, this can be accomplished by assigning a different type code and corresponding service procedure to each source. In systems where a large number of similar sources can generate closely spaced interrupts (e.g., 500 communication lines), an approach similar to that illustrated in figure 2-70, may be used to insure that the interrupt service procedure is not reentered, and yet, interrupts arriving in bursts are not missed. The basic technique is to divide the code required to service an interrupt into two parts. The interrupt service procedure itself is kept as short as possible; it performs the absolute minimum amount of processing necessary to service the device. It then builds a message that contains enough information to permit another task, the interrupt message processor, to complete the interrupt service. It adds the message to a queue (which might be implemented as a linked list), and terminates so that it is available to service the next interrupt. The interrupt message processor, which is not reentrant, obtains a message from the queue, finishes processing the interrupt associated with that message, obtains the next message (if there is one), etc. When a burst of interrupts occurs, the queue will lengthen, but interrupts will not be missed so long as there is time for the interrupt service procedure to be activated and run between requests.

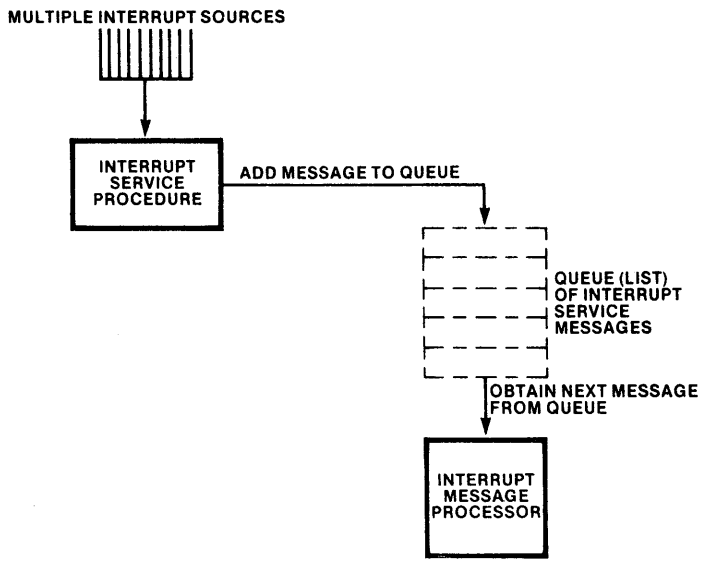


Figure 2-70. Interrupt Message Processor

Stack-Based Parameters

Parameters are frequently passed to procedures on a stack. Results produced by the procedure, however, should be returned in other memory locations or in registers. In other words, the called procedure should "clean up" the stack by discarding the parameters before returning. The RET instruction can perform this function. PL/M-86 procedures always follow this convention.

Flag-Images

Programs should make no assumptions about the contents of the undefined bits in the flag-images stored in memory by the PUSHF and SAHF instructions. These bits always should be masked out of any comparisons or tests that use these flag-images. The undefined bits of the word flag-image can be cleared by ANDing the word with FD5H. The undefined bits of the byte flag-image can be cleared by ANDing the byte with D5H.

Programming Examples

These examples demonstrate the 8086/8088 instruction set and addressing modes in common programming situations. The following topics are addressed:

- procedures (parameters, reentrancy)
- various forms of JMP and CALL instructions
- bit manipulation with the ASM-86 RECORD facility
- dynamic code relocation
- memory mapped I/O
- breakpoints
- interrupt handling
- string operations

These examples are written primarily in ASM-86 and will be of most interest to assembly language programmers. The PL/M-86 compiler generates code that handles many of these situations automatically for PL/M-86 programs. For example, the compiler takes care of the stack in PL/M-86 procedures, allowing the programmer to concentrate on solving the application problem. PL/M-86 programmers, however, may want to examine the memory mapped I/O and interrupt handling examples, since the concepts illustrated are generally applicable; one of the interrupt procedures is written in PL/M-86.

The examples are intended to show one way to use the instruction set, addressing modes and features of ASM-86. They do not demonstrate the "best" way to solve any particular problem. The flexibility of the 8086 and 8088, application differences plus variations in programming style usually add up to a number of ways to implement a programming solution.

Procedures

The code in figure 2-71 illustrates several techniques that are typically used in writing ASM-86 procedures. In this example a calling program invokes a procedure (called EXAMPLE) twice, passing it a different byte array each time. Two parameters are passed on the stack; the first contains the number of elements in the array, and the second contains the address (offset in DATA_SEG) of the first array element. This same technique can be used to pass a variable-length parameter list to a procedure (the "array" could be any series of parameters or parameter addresses). Thus, although the procedure always receives two parameters, these can be used to indirectly access any number of variables in memory.

Any results returned by a procedure should be placed in registers or in memory, but not on the stack. AX or AL is often used to hold a single word or byte result. Alternatively, the calling program can pass the address (or addresses) of a result area to the procedure as a parameter. It is good practice for ASM-86 programs to follow the calling conventions used by PL/M-86; these are documented in *MCS-86 Assembler Operating Instructions For ISIS-II Users*, Order No. 9800641.

EXAMPLE is defined as a FAR procedure, meaning it is in a different segment than the calling program. The calling program must use an intersegment CALL to activate the procedure. Note that this type of CALL saves CS and IP on the stack. If EXAMPLE were defined as NEAR (in the same segment as the caller) then an intra-segment CALL would be used, and only IP would be saved on the stack. It is the responsibility of the calling program to know how the procedure is defined and to issue the correct type of CALL.

```
STACK_SEG       SEGMENT
                DW          20 DUP (?)      ; ALLOCATE 20-WORD STACK

STACK_TOP       LABEL       WORD            ; LABEL INITIAL TOS
STACK_SEG       ENDS

DATA_SEG        SEGMENT
ARRAY_1         DB          10 DUP (?)      ; 10-ELEMENT BYTE ARRAY

ARRAY_2         DB          5 DUP (?)       ; 5-ELEMENT BYTE ARRAY

DATA_SEG        ENDS


PROC_SEG        SEGMENT
ASSUME CS:PROC_SEG,DS:DATA_SEG,SS:STACK_SEG,ES:NOTHING

EXAMPLE         PROC        FAR             ; MUST BE ACTIVATED BY
                                            ;    INTERSEGMENT CALL

; PROCEDURE PROLOG
                PUSH        BP              ; SAVE BP
                MOV         BP, SP          ; ESTABLISH BASE POINTER
                PUSH        CX              ; SAVE CALLER'S
                PUSH        BX              ;      REGISTERS
                PUSHF                       ;      AND FLAGS
                SUB         SP, 6           ; ALLOCATE 3 WORDS LOCAL STORAGE
                ; END OF PROLOG
; PROCEDURE BODY
                MOV         CX, [BP+8]      ; GET ELEMENT COUNT
                MOV         BX, [BP+6]      ; GET OFFSET OF 1ST ELEMENT
                ; PROCEDURE CODE GOES HERE
                ; FIRST PARAMETER CAN BE ADDRESSED:
                ;   [BX]
                ; LOCAL STORAGE CAN BE ADDRESSED:
                ;       [BP-8], [BP-10], [BP-12]
                ; END OF PROCEDURE BODY
; PROCEDURE EPILOG
                ADD         SP, 6           ; DE-ALLOCATE LOCAL STORAGE
                POPF                        ; RESTORE CALLER'S
                POP         BX              ;    REGISTERS
                POP         CX              ;    AND
                POP         BP              ;    FLAGS
                ; END OF EPILOG
; PROCEDURE RETURN
                RET         4               ; DISCARD 2 PARAMETERS

EXAMPLE         ENDP                        ; END OF PROCEDURE "EXAMPLE"

PROC_SEG        ENDS
```

Figure 2-71. Procedure Example 1

```
CALLER_SEG      SEGMENT
; GIVE ASSEMBLER SEGMENT/REGISTER CORRESPONDENCE
ASSUME          CS:CALLER_SEG,
&               DS:DATA_SEG,
&               SS:STACK_SEG,
&               ES:NOTHING              ; NO EXTRA SEGMENT IN THIS PROGRAM

; INITIALIZE SEGMENT REGISTERS
START:          MOV     AX,DATA_SEG
                MOV     DS,AX
                MOV     AX,STACK_SEG
                MOV     SS,AX
                MOV     SP,OFFSET STACK_TOP  ; POINT SP TO TOS

; ASSUME ARRAY_1 IS INITIALIZED
;
; CALL "EXAMPLE", PASSING ARRAY_1, THAT IS, THE NUMBER OF ELEMENTS
;   IN THE ARRAY, AND THE LOCATION OF THE FIRST ELEMENT.
                MOV     AX,SIZE ARRAY_1
                PUSH    AX
                MOV     AX,OFFSET ARRAY_1
                PUSH    AX
                CALL    EXAMPLE

; ASSUME ARRAY_2 IS INITIALIZED
;
; CALL "EXAMPLE" AGAIN WITH DIFFERENT SIZE ARRAY.
                MOV     AX,SIZE ARRAY_2
                PUSH    AX
                MOV     AX,OFFSET ARRAY_2
                PUSH    AX
                CALL    EXAMPLE
CALLER_SEG              ENDS

                END     START
```

Figure 2-71. Procedure Example 1 (Cont'd.)

Figure 2-72 shows the stack before the caller pushes the parameters onto it. Figure 2-73 shows the stack as the procedure receives it after the CALL has been executed.

EXAMPLE is divided into four sections. The "prolog" sets up register BP so it can be used to address data on the stack (recall that specifying BP as a base register in an instruction automatically refers to the stack segment unless a segment override prefix is coded). The next step in the prolog is to save the "state of the machine" as it existed when the procedure was activated. This is done by pushing any registers used by the procedure (only CX and BP in this case) onto the stack. If the procedure changes the flags, and the caller expects the flags to be unchanged following execution of the procedure, they also may be saved on the stack. The last instruction in the prolog allocates three words on the stack for the procedure to use as local temporary storage. Figure 2-74 shows the stack at the end of the prolog. Note that PL/M-86 procedures assume that all registers except SP and BP can be used without saving and restoring.

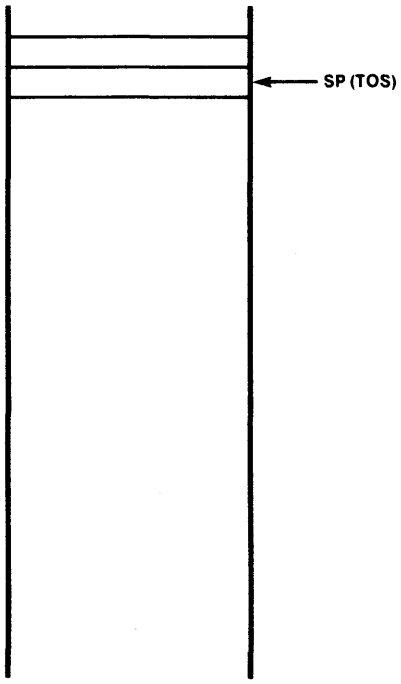


Figure 2-72. Stack Before Pushing Parameters

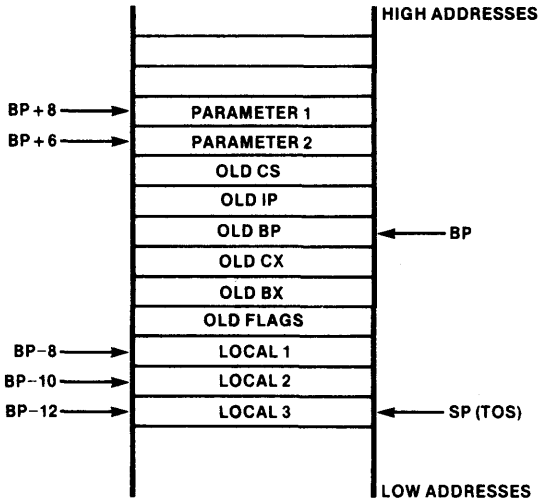


Figure 2-74. Stack Following Procedure Prolog

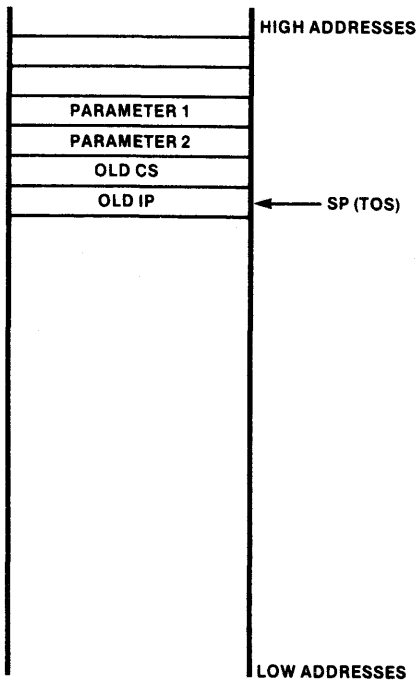


Figure 2-73. Stack at Procedure Entry

The procedure "body" does the actual processing (none in the example). The parameters on the stack are addressed relative to BP. Note that if EXAMPLE were a NEAR procedure, CS would not be on the stack and the parameters would be two bytes "closer" to BP. BP also is used to address the local variables on the stack. Local constants are best stored in a data or extra segment.

The procedure "epilog" reverses the activities of the prolog, leaving the stack as it was when the procedure was entered (see figure 2-75).

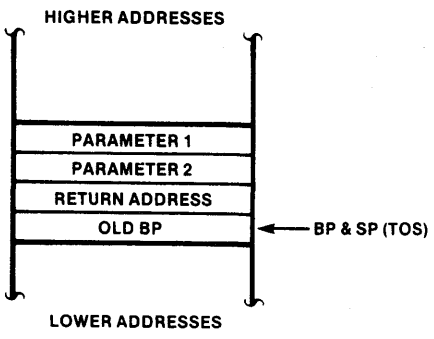


Figure 2-75. Stack Following Procedure Epilog

The procedure "return" restores CS and IP from the stack and discards the parameters. As figure 2-76 shows, when the calling program is resumed, the stack is in the same state as it was before any parameters were pushed onto it.

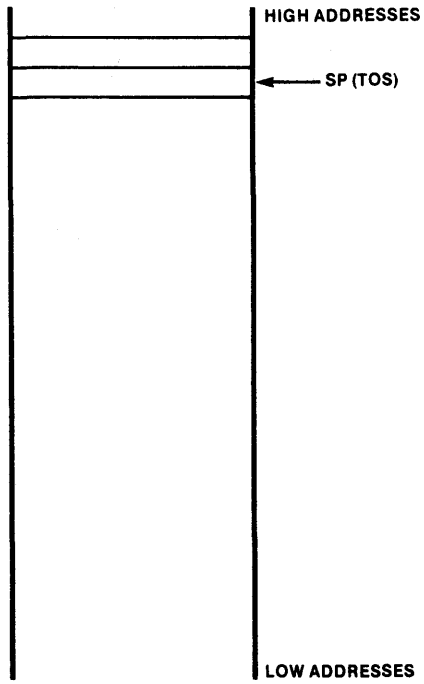


Figure 2-76. Stack Following Procedure Return

Figure 2-77 shows a simple procedure that uses an ASM-86 structure to address the stack. Register BP is pointed to the base of the structure, which is the top of the stack since the stack grows toward lower addresses (see figure 2-78). Any structure element can then be addressed by specifying BP as a base register:

```
[BP].structure_element.
```

Figure 2-79 shows a different approach to using an ASM-86 structure to define the stack layout. As shown in figure 2-80, register BP is pointed at the middle of the structure (at OLD_BP) rather than at the base of the structure. Parameters and the return address are thus located at positive displacements (high addresses) from BP, while local variables are at negative displacements (lower addresses) from BP. This means that the local variables will be "closer" to the beginning of the stack segment and increases the likelihood that the assembler will be able to produce shorter instructions to access these variables, i.e., their offsets from SS may be 255 bytes or less and can be expressed as a 1-byte value rather than a 2-byte value. Exit from the subroutine also is slightly faster because a MOV instruction can be used to deallocate the local storage instead of an ADD (compare figure 2-71).

It is possible for a procedure to be activated a second time before it has returned from its first activation. For example, procedure A may call procedure B, and an interrupt may occur while procedure B is executing. If the interrupt service procedure calls B, then procedure B is *reentered* and must be written to handle this situation correctly, i.e., the procedure must be made reentrant.

In PL/M-86 this can be done by simply writing:

```
B: PROCEDURE (PARM1, PARM2) REENTRANT;
```

An ASM-86 procedure will be reentrant if it uses the stack for storing all local variables. When the procedure is reentered, a new "generation" of variables will be allocated on the stack. The stack will grow, but the sets of variables (and the parameters and return addresses as well) will automatically be kept straight. The stack must be large enough to accommodate the maximum "depth" of procedure activation that can occur under actual running conditions. In addition, any procedure called by a reentrant procedure must itself be reentrant.

A related situation that also requires reentrant procedures is recursion. The following are examples of recursion:

- A calls A (direct recursion),
- A calls B, B calls A (indirect recursion),
- A calls B, B calls C, C calls A (indirect recursion).

```
CODE            SEGMENT
                ASSUME CS:CODE
MAX             PROC
; THIS PROCEDURE IS CALLED BY THE FOLLOWING
;       SEQUENCE:
;           PUSH PARM1
;           PUSH PARM2
;           CALL MAX
; IT RETURNS THE MAXIMUM OF THE TWO WORD
;   PARAMETERS IN AX.

; DEFINE THE STACK LAYOUT AS A STRUCTURE.
STACK_LAYOUT STRUC
OLD_BP          DW ?        ; SAVED BP VALUE—BASE OF STRUCTURE
RETURN_ADDR     DW ?        ; RETURN ADDRESS
PARM_2          DW ?        ; SECOND PARAMETER
PARM_1          DW ?        ; FIRST PARAMETER
STACK_LAYOUT    ENDS

; PROLOG
                PUSH    BP              ; SAVE IN OLD_BP
                MOV     BP, SP          ; POINT TO OLD_BP
; BODY
                MOV     AX, [BP].PARM_1 ; IF FIRST
                CMP     AX, [BP].PARM_2 ; > SECOND
                JG      FIRST_IS_MAX    ; THEN RETURN FIRST
                MOV     AX, [BP].PARM_2 ; ELSE RETURN SECOND
; EPILOG
FIRST_IS_MAX:   POP     BP              ; RESTORE BP (& SP)
; RETURN
                RET     4               ; DISCARD PARAMETERS
MAX             ENDP

CODE            ENDS
                END
```

Figure 2-77. Procedure Example 2

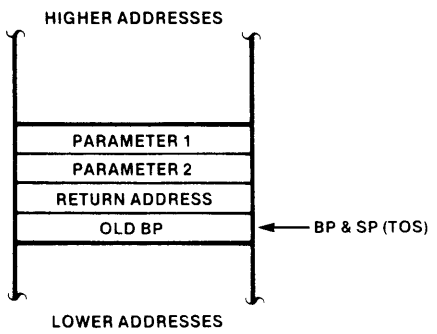


Figure 2-78. Procedure Example 2 Stack Layout

Jumps and Calls

The 8086/8088 instruction set contains many different types of JMP and CALL instructions (e.g., direct, indirect through register, indirect through memory, etc.). These varying types of transfer provide efficient use of space and execution time in different programming situations. Figure 2-81 illustrates typical use of the different forms of these instructions. Note that the ASM-86 assembler uses the terms "NEAR" and "FAR" to denote intrasegment and intersegment transfers, respectively.

```
EXTRA           SEGMENT
; CONTAINS STRUCTURE TEMPLATE THAT "NEARPROC"
;    USES TO ADDRESS AN ARRAY PASSED BY ADDRESS.
DUMMY           STRUC
PARM_ARRAY      DB          256 DUP ?
DUMMY           ENDS
EXTRA           ENDS

CODE            SEGMENT
                ASSUME CS:CODE,ES:EXTRA
NEARPROC        PROC
; LAY OUT THE STACK (THE DYNAMIC STORAGE AREA OR DSA).
DSASTRUC        STRUC
I               DW          ?               ; LOCAL VARIABLES FIRST
LOC_ARRAY       DW          10 DUP (?)      ;
OLD_BP          DW          ?               ; ORIGINAL BP VALUE
RETADDR         DW          ?               ; RETURN ADDRESS
POINTER         DD          ?               ; 2ND PARM—POINTER TO "PARM_ARRAY"
COUNT           DB          ?               ; 1ST PARM—A BYTE OCCUPIES
                DB          ?               ;    A WORD ON THE STACK
DSASTRUC        ENDS

; USE AN EQU TO DEFINE THE BASE ADDRESS OF THE
;    DSA. CANNOT SIMPLY USE BP BECAUSE IT WILL
;    BE POINTING TO "OLD_BP" IN THE MIDDLE OF
;    THE DSA.
DSA             EQU         [BP - OFFSET OLD_BP]

; PROCEDURE ENTRY
                PUSH        BP              ; SAVE BP
                MOV         BP, SP          ; POINT BP AT OLD_BP
                SUB         SP, OFFSET OLD_BP ; ALLOCATE LOC_ARRAY & I

; PROCEDURE BODY
                ; ACCESS LOCAL VARIABLE I
                MOV         AX,DSA.I

                ; ACCESS LOCAL ARRAY (3) I.E., 4TH ELEMENT
                MOV         SI,6            ; WORD ARRAY-INDEX IS 3*2
                MOV         AX,DSA.LOC_ARRAY [SI]

                ; LOAD POINTER TO ARRAY PASSED BY ADDRESS
                LES         BX,DSA.POINTER

                ; ES:BX NOW POINTS TO PARM_ARRAY (0)
                ; ACCESS SI'TH ELEMENT OF PARM_ARRAY
                MOV         AL,ES:[BX].PARM_ARRAY [SI]

                ; ACCESS THE BYTE PARAMETER
                MOV         AL,DSA.COUNT
```

Figure 2-79. Procedure Example 3

```
; PROCEDURE EXIT
                MOV     SP,BP           ; DE-ALLOCATE LOCALS
                POP     BP              ; RESTORE BP
                ; STACK NOW AS RECEIVED FROM CALLER
                RET     6               ; DISCARD PARAMETERS

NEARPROC        ENDP
CODE            ENDS
                END
```

Figure 2-79. Procedure Example 3 (Cont'd.)

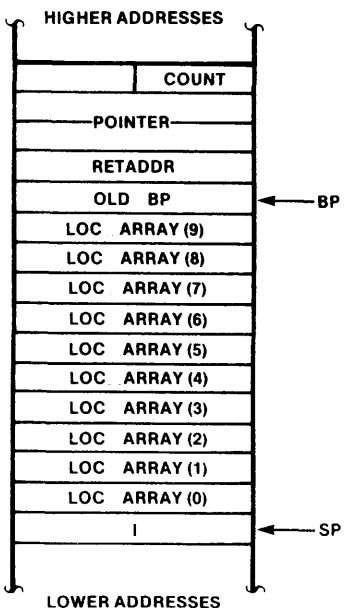


Figure 2-80. Procedure Example 3 Stack Layout

The procedure in figure 2-81 illustrates how a PL/M-86 DO CASE construction may be implemented in ASM-86. It also shows:

- an indirect CALL through memory to a procedure located in another segment,
- a direct JMP to a label in another segment,
- an indirect JMP though memory to a label in the same segment,
- an indirect JMP through a register to a label in the same segment,
- a direct CALL to a procedure in another segment,
- a direct CALL to a procedure in the same segment,
- direct JMPs to labels in the same segment, within −128 to +127 bytes ("SHORT") and farther than −128 to +127 bytes ("NEAR").

```
DATA            SEGMENT
; DEFINE THE CASE TABLE (JUMP TABLE) USED BY PROCEDURE
;       "DO_CASE." THE OFFSET OF EACH LABEL WILL
;       BE PLACED IN THE TABLE BY THE ASSEMBLER.
CASE_TABLE      DW          ACTION0, ACTION1, ACTION2,
&                           ACTION3, ACTION4, ACTION5
DATA            ENDS

; DEFINE TWO EXTERNAL (NOT PRESENT IN THIS
;       ASSEMBLY BUT SUPPLIED BY R & L FACILITY)
;       PROCEDURES. ONE IS IN THIS CODE SEGMENT
;       (NEAR) AND ONE IS IN ANOTHER SEGMENT (FAR).
                EXTRN       NEAR_PROC: NEAR, FAR_PROC: FAR

; DEFINE AN EXTERNAL LABEL (JUMP TARGET) THAT
;       IS IN ANOTHER SEGMENT.
                EXTRN       ERR_EXIT: FAR

CODE            SEGMENT
                ASSUME      CS: CODE, DS: DATA
; ASSUME DS HAS BEEN SET UP
;       BY CALLER TO POINT TO "DATA" SEGMENT.

DO_CASE         PROC        NEAR
; THIS EXAMPLE PROCEDURE RECEIVES TWO
;     PARAMETERS ON THE STACK. THE FIRST
;     PARAMETER IS THE "CASE NUMBER" OF
;     A ROUTINE TO BE EXECUTED (0-5). THE SECOND
;     PARAMETER IS A POINTER TO AN ERROR
;     PROCEDURE THAT IS EXECUTED IF AN INVALID
;     CASE NUMBER (>5) IS RECEIVED.

; LAY OUT THE STACK.
STACK_LAYOUT    STRUC
OLD_BP          DW      ?
RETADDR         DW      ?
ERR_PROC_ADDR   DD      ?
CASE_NO         DB      ?
                DB      ?
STACK_LAYOUT    ENDS

; SET UP PARAMETER ADDRESSING
                PUSH        BP
                MOV         BP, SP

; CODE TO SAVE CALLER'S REGISTERS COULD GO HERE.

; CHECK THE CASE NUMBER
                MOV         BH, 0
                MOV         BL, [BP].CASE_NO
                CMP         BX, LENGTH CASE_TABLE
                JLE         OK              ; ALL CONDITIONAL JUMPS
                                            ; ARE SHORT DIRECT
```

Figure 2-81. JMP and CALL Examples

```
; CALL THE ERROR ROUTINE WITH A FAR
;    INDIRECT CALL. A FAR INDIRECT CALL
;    IS INDICATED SINCE THE OPERAND HAS
;    TYPE "DOUBLEWORD."
                    CALL        [BP].ERR_PROC_ADDR

; JUMP DIRECTLY TO A LABEL IN ANOTHER SEGMENT.
;    A FAR DIRECT JUMP IS INDICATED SINCE
;    THE OPERAND HAS TYPE "FAR."
                    JMP         ERR_EXIT

OK:
; MULTIPLY CASE NUMBER BY 2 TO GET OFFSET
;    INTO CASE_TABLE (EACH ENTRY IS 2 BYTES).
                    SHL         BX, 1
; NEAR INDIRECT JUMP THROUGH SELECTED
;    ELEMENT OF CASE_TABLE. A NEAR
;    INDIRECT JUMP IS INDICATED SINCE THE
;    OPERAND HAS TYPE "WORD."
                    JMP         CASE_TABLE [BX]

ACTION0:            ; EXECUTED IF CASE_NO = 0
    ; CODE TO PROCESS THE ZERO CASE GOES HERE.
    ; FOR ILLUSTRATION PURPOSES, USE A
    ;    NEAR INDIRECT JUMP THROUGH A
    ;    REGISTER TO BRANCH TO THE POINT
    ;    WHERE ALL CASES CONVERGE.
    ;    A DIRECT JUMP (JMP ENDCASE) IS
    ;    ACTUALLY MORE APPROPRIATE HERE.
                    MOV         AX, OFFSET ENDCASE
                    JMP         AX

ACTION1:            ; EXECUTED IF CASE_NO = 1
    ; CALL A FAR EXTERNAL PROCEDURE. A FAR
    ;    DIRECT CALL IS INDICATED SINCE OPERAND
    ;    HAS TYPE "FAR."
                    CALL        FAR_PROC
    ; CALL A NEAR EXTERNAL PROCEDURE.
                    CALL        NEAR_PROC
    ; BRANCH TO CONVERGENCE POINT USING NEAR
    ;    DIRECT JUMP. NOTE THAT "ENDCASE"
    ;    IS MORE THAN 127 BYTES AWAY
    ;    SO A NEAR DIRECT JUMP WILL BE USED.
                    JMP         ENDCASE

ACTION2:            ; EXECUTED IF CASE_NO = 2
    ; CODE GOES HERE
                    JMP         ENDCASE     ; NEAR DIRECT JUMP
```

Figure 2-81. JMP and CALL Examples (Cont'd.)

```
ACTION3:        ; EXECUTED IF CASE_NO = 3
    ; CODE GOES HERE
                JMP         ENDCASE     ; NEAR DIRECT JMP

; ARTIFICIALLY FORCE "ENDCASE" FURTHER AWAY
;   SO THAT ABOVE JUMPS CANNOT BE "SHORT."
                ORG         500

ACTION4:        ; EXECUTED IF CASE_NO = 4
    ; CODE GOES HERE
                JMP         ENDCASE     ; NEAR DIRECT JUMP

ACTION5:        ; EXECUTED IF CASE_NO = 5
    ; CODE GOES HERE.
    ; BRANCH TO CONVERGENCE POINT USING
    ;   SHORT DIRECT JUMP SINCE TARGET IS
    ;   WITHIN 127 BYTES. MACHINE INSTRUCTION
    ;   HAS 1-BYTE DISPLACEMENT RATHER THAN
    ;   2-BYTE DISPLACEMENT REQUIRED FOR
    ;   NEAR DIRECT JUMPS. "SHORT" IS
    ;   WRITTEN BECAUSE "ENDCASE" IS A FORWARD
    ;   REFERENCE, WHICH ASSEMBLER ASSUMES IS
    ;   "NEAR." IF "ENDCASE" APPEARED PRIOR
    ;   TO THE JUMP, THE ASSEMBLER WOULD
    ;   AUTOMATICALLY DETERMINE IF IT WERE REACHABLE
    ;   WITH A SHORT JUMP.
                JMP         SHORT ENDCASE

ENDCASE:        ; ALL CASES CONVERGE HERE.

    ; POP CALLER'S REGISTERS HERE.
    ; RESTORE BP & SP, DISCARD PARAMETERS
    ;   AND RETURN TO CALLER.
                MOV         SP, BP
                POP         BP
                RET         6

DO_CASE         ENDP
CODE            ENDS
                END         ; OF ASSEMBLY
```

Figure 2-81. JMP and CALL Examples (Cont'd.)

Records

Figure 2-82 shows how the ASM-86 RECORD facility may be used to manipulate bit data. The example shows how to:

- right-justify a bit field,
- test for a value,
- assign a constant known at assembly time,
- assign a variable,
- set or clear a bit field.

```
DATA            SEGMENT
; DEFINE A WORD ARRAY
XREF            DW 3000 DUP (?)
; EACH ELEMENT OF XREF CONSISTS OF 3 FIELDS:
;       A 2-BIT TYPE CODE,
;       A 1-BIT FLAG,
;       A 13-BIT NUMBER.
; DEFINE A RECORD TO LAY OUT THIS ORGANIZATION.
LINE_REC        RECORD    LINE_TYPE: 2,
&                         VISIBLE: 1,
&                         LINE_NUM: 13
DATA            ENDS

CODE            SEGMENT
                ASSUME CS: CODE,   DS: DATA
; ASSUME SEGMENT REGISTERS ARE SET UP PROPERLY
;     AND THAT SI INDEXES AN ELEMENT OF XREF.

; A RECORD FIELD-NAME USED BY ITSELF RETURNS
;   THE SHIFT COUNT REQUIRED TO RIGHT-JUSTIFY
;   THE FIELD. ISOLATE "LINE_TYPE" IN THIS
;   MANNER.
                MOV       AL, XREF [SI]
                MOV       CL, LINE_TYPE
                SHR       AX, CL

; THE "MASK " OPERATOR APPLIED TO A RECORD
;    FIELD-NAME RETURNS THE BIT MASK
;    REQUIRED TO ISOLATE THE FIELD WITHIN
;    THE RECORD. CLEAR ALL BITS EXCEPT
;    "LINE_NUM."
                MOV       DX, XREF[SI]
                AND       DX, MASK LINE_NUM

; DETERMINE THE VALUE OF THE "VISIBLE" FIELD
                TEST      XREF[SI], MASK VISIBLE
                JZ        NOT_VISIBLE
; NO JUMP IF VISIBLE = 1
NOT_VISIBLE:    ; JUMP HERE IF VISIBLE = 0

; ASSIGN A CONSTANT KNOWN AT ASSEMBLY-TIME
;     TO A FIELD, BY FIRST CLEARING THE BITS
;     AND THEN OR'ING IN THE VALUE. IN
;     THIS CASE "LINE_TYPE" IS SET TO 2 (10B).
                AND       XREF[SI], NOT MASK LINE_TYPE
                OR        XREF[SI],2 SHL LINE_TYPE
; THE ASSEMBLER DOES THE MASKING AND SHIFTING.
; THE RESULT IS THE SAME AS:
                AND       XREF[SI], 3FFFH
                OR        XREF[SI], 8000H
;    BUT IS MORE READABLE AND LESS SUBJECT
;    TO CLERICAL ERROR.
```

Figure 2-82. RECORD Example

```
; ASSIGN A VARIABLE (THE CONTENT OF AX)
;     TO LINE_TYPE.
                MOV     CL, LINE_TYPE   ; SHIFT COUNT
                SHL     AX, CL   ; SHIFT TO "LINE UP" BITS
                AND     XREF[SI], NOT MASK LINE_TYPE   ; CLEAR BITS
                OR      XREF[SI], AX   ; OR IN NEW VALUE

; NO SHIFT IS REQUIRED TO ASSIGN TO THE
;     RIGHT-MOST FIELD. ASSUMING AX CONTAINS
;     A VALID NUMBER (HIGH 3 BITS ARE 0),
;     ASSIGN AX TO "LINE_NUM."
                AND     XREF[SI], NOT MASK LINE_NUM
                OR      XREF[SI], AX

; A FIELD MAY BE SET OR CLEARED WITH
;   ONE INSTRUCTION. CLEAR THE "VISIBLE"
;   FLAG AND THEN SET IT.
                AND     XREF[SI], NOT MASK VISIBLE
                OR      XREF[SI], MASK VISIBLE

CODE            ENDS
                END     ; OF ASSEMBLY
```

Figure 2-82. RECORD Example (Cont'd.)

The following considerations apply to position-independent code sequences:

- A label that is referenced by a direct FAR (intersegment) transfer is not moveable.
- A label that is referenced by an indirect transfer (either NEAR or FAR) is moveable so long as the register or memory pointer to the label contains the label's current address.
- A label that is referenced by a SHORT (e.g., conditional jump) or a direct NEAR (intrasegment) transfer is moveable so long as the referencing instruction is moved with the label as a unit. These transfers are self-relative; that is they require only that the label maintain the same distance from the referencing instruction, and actual addresses are immaterial.
- Data is segment-independent, but not offset-independent. That is, a data item may be moved to a different segment, but it must maintain the same offset from the beginning of the segment. Placing constants in a unit of code also effectively makes the code offset-dependent, and therefore is not recommended.
- A procedure should not be moved while it is active or while any procedure it has called is active.
- A section of code that has been interrupted should not be moved.

The segment that is receiving a section of code must have "room" for the code. If the MOVS (or MOVSB or MOVSW) instruction attempts to auto-increment DI past 64k, it wraps around to 0 and causes the beginning of the segment to be overwritten. If a segment override is needed for the source operand, code similar to the following can be used to properly resume the instruction if it is interrupted:

```
RESUME:  REP   MOVS     DESTINATION, ES:SOURCE
  ;IF CX NOT = 0 THEN INTERRUPT HAS OCCURRED
           AND  CX,CX    ; CX=0?
           JNZ  RESUME   ;NO, FINISH EXECUTION
  ;CONTROL COMES HERE WHEN STRING HAS BEEN MOVED.
```

If the MOVS is interrupted, the CPU "remembers" the segment override, but "forgets" the presence of the REP prefix when execution resumes. Testing CX indicates whether the instruction is completed or not. Jumping back to the instruction resumes it where it left off. Note that a segment override cannot be specified with MOVSB or MOVSW.

Dynamic Code Relocation

Figure 2-83 illustrates one approach to moving programs in memory at execution time. A "supervisor" program (which is not moved) keeps a pointer variable that contains the current location (offset and segment base) of a position-independent procedure. The supervisor always calls the procedure through this pointer. The supervisor also has access to the procedure's length in bytes. The procedure is moved with the MOVSB instruction. After the procedure is moved, its pointer is updated with the new location. The ASM-86 WORD PTR operator is written to inform the assembler that one word of the doubleword pointer is being updated at a time.

```
MAIN_DATA       SEGMENT
; SET UP POINTERS TO POSITION-INDEPENDENT PROCEDURE
;    AND FREE SPACE.
PIP_PTR         DD          EXAMPLE
FREE_PTR        DD          TARGET_SEG
; SET UP SIZE OF PROCEDURE IN BYTES
PIP_SIZE        DW          EXAMPLE_LEN
MAIN_DATA       ENDS

STACK           SEGMENT
                DW          20 DUP (?)          ; 20 WORDS FOR STACK

STACK_TOP       LABEL       WORD                ; TOS BEGINS HERE
STACK           ENDS

SOURCE_SEG      SEGMENT
; THE POSITION-INDEPENDENT PROCEDURE IS INITIALLY IN THIS SEGMENT.
; OTHER CODE MAY PRECEDE IT, I.E., ITS OFFSET NEED NOT BE ZERO.
ASSUME          CS:SOURCE_SEG
EXAMPLE         PROC        FAR
   ; THIS PROCEDURE READS AN 8-BIT PORT UNTIL
   ; BIT 3 OF THE VALUE READ IS FOUND SET. IT
   ; THEN READS ANOTHER PORT. IF THE VALUE READ
   ; IS GREATER THAN 10H IT WRITES THE VALUE TO
   ; A THIRD PORT AND RETURNS; OTHERWISE IT STARTS
   ; OVER.
STATUS_PORT     EQU         0D0H
PORT_READY      EQU         008H
INPUT_PORT      EQU         0D2H
THRESHOLD       EQU         010H
OUTPUT_PORT     EQU         0D4H
CHECK_AGAIN:    IN          AL,STATUS_PORT      ; GET STATUS
                TEST        AL,PORT_READY       ; DATA READY?
                JNE         CHECK_AGAIN         ; NO, TRY AGAIN
                IN          AL,INPUT_PORT       ; YES, GET DATA
                CMP         AL,THRESHOLD        ; > 10H?
                JLE         CHECK_AGAIN         ; NO, TRY AGAIN
                OUT         OUTPUT_PORT,AL      ; YES, WRITE IT
```

Figure 2-83. Dynamic Code Relocation Example

```
                RET         ; RETURN TO CALLER
; GET PROCEDURE LENGTH
EXAMPLE_LEN     EQU         (OFFSET THIS BYTE)—(OFFSET CHECK_AGAIN)
                ENDP        EXAMPLE  ENDP
SOURCE_SEG      ENDS

TARGET_SEG      SEGMENT
; THE POSITION-INDEPENDENT PROCEDURE
;    IS MOVED TO THIS SEGMENT, WHICH IS
;    INITIALLY "EMPTY."
; IN TYPICAL SYSTEMS, A "FREE SPACE MANAGER" WOULD
;   MAINTAIN A POOL OF AVAILABLE MEMORY SPACE
; FOR ILLUSTRATION PURPOSES, ALLOCATE ENOUGH
;    SPACE TO HOLD IT
                DB          EXAMPLE_LEN DUP (?)

TARGET_SEG      ENDS

MAIN_CODE       SEGMENT
; THIS ROUTINE CALLS THE EXAMPLE PROCEDURE
; AT ITS INITIAL LOCATION, MOVES IT, AND
; CALLS IT AGAIN AT THE NEW LOCATION.

ASSUME          CS:MAIN_CODE,SS:STACK,
&               DS:MAIN_DATA,ES:NOTHING

; INITIALIZE SEGMENT REGISTERS & STACK POINTER.
START:          MOV         AX,MAIN_DATA
                MOV         DS,AX
                MOV         AX,STACK
                MOV         SS,AX
                MOV         SP,OFFSET STACK_TOP

; CALL EXAMPLE AT INITIAL LOCATION.
                CALL        PIP_PTR

; SET UP CX WITH COUNT OF BYTES TO MOV
                MOV         CX,PIP_SIZE
; SAVE DS, SET UP DS/SI AND ES/DI TO
;    POINT TO THE SOURCE AND DESTINATION
;    ADDRESSES.
                PUSH        DS
                LES         DI,FREE_PTR
                LDS         SI,PIP_PTR
; MOVE THE PROCEDURE.
                CLD                                 ; AUTO INCREMENT
                REP MOVSB

; RESTORE OLD ADDRESSABILITY.
                MOV         AX,DS                   ; HOLD TEMPORARILY
                POP         DS
; UPDATE POINTER TO POSITION-INDEPENDENT PROCEDURE
                MOV         WORD PTR PIP_PTR+2,ES
                SUB         DI,PIP_SIZE             ; PRODUCES OFFSET
                MOV         WORD PTR PIP_PTR,DI
```

Figure 2-83. Dynamic Code Relocation Example (Cont'd.)

```
; UPDATE POINTER TO FREE SPACE
                MOV     WORD PTR FREE_PTR+2,AX
                SUB     SI,PIP_SIZE          ; PRODUCES OFFSET
                MOV     WORD PTR FREE_PTR,SI

; CALL POSITION-INDEPENDENT PROCEDURE AT
;   NEW LOCATION AND STOP
                CALL    PIP_PTR
MAIN_CODE       ENDS
                END     START
```

Figure 2-83. Dynamic Code Relocation Example (Cont'd.)

Memory-Mapped I/O

Figure 2-84 shows how memory-mapped I/O can be used to address a group of communication lines as an "array." In the example, indexed addressing is used to poll the array of status ports, one port at a time. Any of the other 8086/8088 memory addressing modes may be used in conjunction with memory-mapped I/O devices as well.

In figure 2-85 a MOVS instruction is used to perform a high-speed transfer to a memory-mapped line printer. Using this technique requires the hardware to be set up as follows. Since the MOVS instruction transfers characters to successive memory addresses, the decoding logic must select the line printer if any of these locations is written. One way of accomplishing this is to have the chip select logic decode only the upper 12 lines of the address bus (A19-A8), ignoring the contents of the lower eight lines (A7-A0). When data is written to any address in this 256-byte block, the upper 12 lines will not change, so the printer will be selected.

If an 8086 is being used with an 8-bit printer, the 8086's 16-bit data bus must be mapped into 8-bits by external hardware. Using an 8088 provides a more direct interface.

```
COM_LINES       SEGMENT  AT 800H
; THE FOLLOWING IS A MEMORY MAPPED "ARRAY"
;       OF EIGHT 8-BIT COMMUNICATIONS CONTROLLERS
;       (E.G., 8251 USARTS). PORTS HAVE ALL-ODD
;       OR ALL-EVEN ADDRESSES (EVERY OTHER BYTE
;       IS SKIPPED) FOR 8086-COMPATIBILITY.

COM_DATA        DB    ?
                DB    ?                    ; SKIP THIS ADDRESS
COM_STATUS      DB    ?
                DB    ?                    ; SKIP THIS ADDRESS
                DB    28   DUP (?)         ; REST OF "ARRAY"
COM_LINES       ENDS

CODE            SEGMENT
; ASSUME STACK IS SET UP, AS ARE SEGMENT
;       REGISTERS (DS POINTING TO COM_LINES).
;       FOLLOWING CODE POLLS THE LINES.

CHAR_RDY        EQU     00000010B    ; CHARACTER PRESENT
START_POLL:     MOV     CX, 8        ; POLL 8 LINES ZERO
                SUB     SI, SI       ; ARRAY INDEX
```

Figure 2-84. Memory Mapped I/O "Array"

```
POLL_NEXT:    TEST    COM_STATUS [SI], CHAR_RDY
              JE      READ_CHAR ; READ IF PRESENT
              ADD     SI, 4     ; ELSE BUMP TO NEXT LINE
              LOOP    POLL_NEXT ; CONTINUE POLLING UNTIL
                                ;    ALL 8 HAVE BEEN CHECKED
              JMP     START_POLL; START OVER

READ_CHAR:    MOV     AL,COM_DATA [SI]   ;GET THE DATA
; ETC.
CODE          ENDS
              END
```

Figure 2-84. Memory Mapped I/O "Array" (Cont'd.)

```
PRINTER       SEGMENT
; THIS SEGMENT CONTAINS A "STRING" THAT
;    IS ACTUALLY A MEMORY-MAPPED LINE PRINTER.
;    THE SEGMENT (PRINTER) MUST BE ASSIGNED (LOCATED)
;    TO A BLOCK OF THE ADDRESS SPACE SUCH
;    THAT WRITING TO ANY ADDRESS IN THE
;    BLOCK SELECTS THE PRINTER.

PRINT_SELECT  DB 133    DUP (?)          ; "STRING" REPRESENTING PRINTER
              DB 123    DUP (?)          ; REST OF 256-BYTE BLOCK
PRINTER       ENDS

DATA          SEGMENT
PRINT_BUF     DB 133    DUP (?)          ; LINE TO BE PRINTED
PRINT_COUNT   DB 1      ?                ; LINE LENGTH
; OTHER PROGRAM DATA
DATA          ENDS

CODE          SEGMENT
; ASSUME STACK AND SEGMENT REGISTERS HAVE
;      BEEN SET UP (DS POINTS TO DATA SEGMENT).
;      FOLLOWING CODE TRANSFERS A LINE TO
;      THE PRINTER.

              ASSUME  ES: PRINTER
              MOV     AX, PRINTER      ; PREVENT SEGMENT OVERRIDE
              MOV     ES, AX
              SUB     DI, DI           ; CLEAR SOURCE AND
              SUB     SI, SI           ;    DESTINATION POINTERS
              MOV     CX, PRINT_COUNT
              CLD     ; AUTO-INCREMENT
    REP       MOVS    PRINT_SELECT, PRINT_BUF
              ; ETC.
CODE          ENDS
              END
```

Figure 2-85. Memory Mapped Block Transfer Example

Breakpoints

Figure 2-86 illustrates how a program may set a breakpoint. In the example, the breakpoint routine puts the processor into single-step mode, but the same general approach could be used for other purposes as well. A program passes the address where the break is to occur to a procedure that saves the byte located at that address and replaces it with an INT 3 (breakpoint) instruction. When the CPU encounters the breakpoint instruction, it calls the type 3 interrupt procedure. In the example, this procedure places the processor into single-step mode starting with the instruction where the breakpoint was placed.

```
INT_PTR_TAB   SEGMENT
; INTERRUPT POINTER TABLE-LOCATE AT 0H
TYPE_0        DD        ?                ; NOT DEFINED IN EXAMPLE
TYPE_1        DD        SINGLE_STEP
TYPE_2        DD        ?                ; NOT DEFINED IN EXAMPLE
TYPE_3        DD        BREAKPOINT
INT_PTR_TAB   ENDS

SAVE_SEG      SEGMENT
SAVE_INSTR    DB 1      DUP (?)          ; INSTRUCTION REPLACED
                                         ; BY BREAKPOINT
SAVE_SEG      ENDS

MAIN_CODE     SEGMENT
; ASSUME STACK AND SEGMENT REGISTERS ARE SET UP.

; ENABLE SINGLE-STEPPING WITH INSTRUCTION AT
;    LABEL "NEXT" BY PASSING SEGMENT AND
;    OFFSET OF "NEXT" TO "SET_BREAK" PROCEDURE
              PUSH      CS
              LEA       AX, CS: NEXT
              PUSH      AX
              CALL      FAR SET_BREAK
; ETC.

NEXT:         IN        AL, 0FFFH        ; BREAKPOINT SET HERE
              ; ETC.

MAIN_CODE     ENDS

BREAK         SEGMENT
SET_BREAK     PROC      FAR
; THIS PROCEDURE SAVES AN INSTRUCTION BYTE (WHOSE
;    ADDRESS IS PASSED BY THE CALLER) AND WRITES
;    AN INT 3 (BREAKPOINT) MACHINE INSTRUCTION
:    AT THE TARGET ADDRESS.

TARGET        EQU       DWORD PTR [BP+6]
```

Figure 2-86. Breakpoint Example

```
; SET UP BP FOR PARM ADDRESSING & SAVE REGISTERS
                PUSH    BP
                MOV     BP, SP
                PUSH    DS
                PUSH    ES
                PUSH    AX
                PUSH    BX
; POINT DS/BX TO THE TARGET INSTRUCTION
                LDS     BX, TARGET
; POINT ES TO THE SAVE AREA
                MOV     AX, SAVE_SEG
                MOV     ES, AX
; SWAP THE TARGET INSTRUCTION FOR INT 3 (0CCH)
                MOV     AL, 0CCH
                XCHG    AL, DS: [BX]
; SAVE THE TARGET INSTRUCTION
                MOV     ES: SAVE_INSTR, AL
; RESTORE AND RETURN
                POP     BX
                POP     AX
                POP     ES
                POP     DS
                POP     BP
                RET     4
SET_BREAK       ENDP

BREAKPOINT      PROC    FAR
; THE CPU WILL ACTIVATE THIS PROCEDURE WHEN IT
;   EXECUTES THE INT 3 INSTRUCTION SET BY THE
;   SET_BREAK PROCEDURE. THIS PROCEDURE
;   RESTORES THE SAVED INSTRUCTION BYTE TO ITS
;   ORIGINAL LOCATION AND BACKS UP THE
;   INSTRUCTION POINTER IMAGE ON THE STACK
;   SO THAT EXECUTION WILL RESUME WITH
;   THE RESTORED INSTRUCTION. IT THEN SETS
;   TF (THE TRAP FLAG) IN THE FLAG-IMAGE
;   ON THE STACK. THIS PUTS THE PROCESSOR
;   IN SINGLE-STEP MODE WHEN EXECUTION
;   RESUMES.
    FLAG_IMAGE  EQU     WORD PTR [BP+6]
    IP_IMAGE    EQU     WORD PTR [BP+2]
NEXT_INSTR      EQU     DWORD PTR [BP+2]
; SET UP BP TO ADDRESS STACK AND SAVE REGISTERS
                PUSH    BP
                MOV     BP, SP
                PUSH    DS
                PUSH    ES
                PUSH    AX
                PUSH    BX
; POINT ES AT THE SAVE AREA
                MOV     AX, SAVE_SEG
                MOV     ES, AX
; GET THE SAVED BYTE
                MOV     AL, ES: SAVE_INSTR
```

Figure 2-86. Breakpoint Example (Cont'd.)

```
; GET THE ADDRESS OF THE TARGET + 1
;    (INSTRUCTION FOLLOWING THE BREAKPOINT)
                LDS         BX, NEXT_INSTR
; BACK UP IP-IMAGE (IN BX) AND REPLACE ON STACK
                DEC         BX
                MOV         IP_IMAGE, BX

; RESTORE THE SAVED INSTRUCTION
                MOV         DS: [BX], AL
; SET TF ON STACK
                AND         FLAG_IMAGE, 0100H
; RESTORE EVERYTHING AND EXIT
                POP         BX
                POP         AX
                POP         ES
                POP         DS
                POP         BP
                IRET
BREAKPOINT      ENDP

SINGLE  STEP    PROC        FAR
; ONCE SINGLE-STEP MODE HAS BEEN ENTERED,
;    THE CPU "TRAPS" TO THIS PROCEDURE
;    AFTER EVERY INSTRUCTION THAT IS NOT IN
;    AN INTERRUPT PROCEDURE. IN THE CASE
;    OF THIS EXAMPLE, THIS PROCEDURE WILL
;    BE EXECUTED IMMEDIATELY FOLLOWING THE
;    "IN AL, 0FFFH" INSTRUCTION (WHERE THE
;    BREAKPOINT WAS SET) AND AFTER EVERY
;    SUBSEQUENT INSTRUCTION. THE PROCEDURE
;    COULD "TURN ITSELF OFF" BY CLEARING
;    TF ON THE STACK.
; SINGLE-STEP CODE GOES HERE.
; SINGLE_STEP   ENDP

BREAK           ENDS

                END         ;
```

Figure 2-86. Breakpoint Example (Cont'd.)

Interrupt Procedures

Figure 2-87 is a block diagram of a hypothetical system that is used to illustrate three different examples of interrupt handling: an external (maskable) interrupt, an external non-maskable interrupt and a software interrupt.

In this hypothetical system, an 8253 Programmable Interval Timer is used to generate a time base. One of the three timers on the 8253 is programmed to repeatedly generate interrupt requests at 50 millisecond intervals. The output from this timer is tied to one of the eight interrupt request lines of an 8259A Programmable Interrupt Controller. The 8259A, in turn, is connected to the INTR line of an 8086 or 8088.

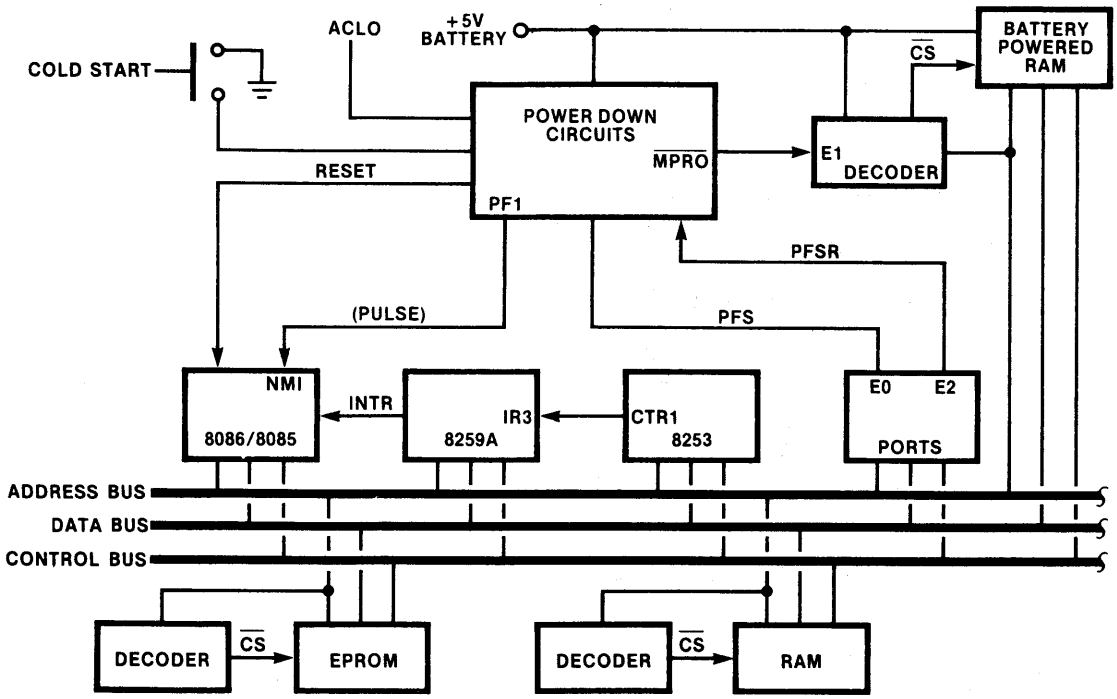


Figure 2-87. Interrupt Example Block Diagram

A power-down circuit is used in the system to illustrate one application of the 8086/8088 NMI (non-maskable interrupt) line. If the ac line voltage drops below a certain threshold, the power supply activates ACLO. The power-down circuit then sends a power-fail interrupt (PFI) pulse to the CPU's NMI input. After 5 milliseconds, the power-down circuit activates MPRO (memory protect) to disable reading from and writing to the system's battery-powered RAM. This protects the RAM from fluctuations that may occur when power is actually lost 7.5 milliseconds after the power failure is detected. The system software must save all vital information in the battery-powered RAM segment within 5 milliseconds of the activation of NMI.

When power returns, the power-down circuit activates the system RESET line. Pressing the "cold start" switch also produces a system RESET. The PFS (power fail status) line, which is connected to the low-order bit of port E0, identifies the source of the RESET. If the bit is set, the software executes a "warm start" to restore the information saved by the power-fail routine. If the PFS bit is cleared, the software executes a "cold start" from the beginning of the program. In either case, the software writes a "one" to the low-order bit of port E2. This line is connected to the power-down circuit's PFSR (power fail status reset) signal and is used to enable the battery-powered RAM segment.

A software interrupt is used to update a simple real-time clock. This procedure is written in PL/M-86, while the rest of the system is written in ASM-86 to demonstrate the interrupt handling capability of both languages. The system's main program simply initializes the system following receipt of a RESET and then waits for an interrupt. An example of this interrupt procedure is given in figure 2-88.

```
INT_POINTERS                    SEGMENT
; INTERRUPT POINTER TABLE, LOCATE AT 0H, ROM-BASED
TYPE_0          DD      ?               ; DIVIDE-ERROR NOT SUPPLIED IN EXAMPLE.
TYPE_1          DD      ?               ; SINGLE-STEP NOT SUPPLIED IN EXAMPLE.
TYPE_2          DD      POWER_FAIL      ; NON-MASKABLE INTERRUPT
TYPE_3          DD      ?               ; BREAKPOINT NOT SUPPLIED IN EXAMPLE.
TYPE_4          DD      ?               ; OVERFLOW NOT SUPPLIED IN EXAMPLE.
; SKIP RESERVED PART OF EXAMPLE
                ORG     32*4
TYPE_32         DD      ?               ; 8259A IR0 - AVAILABLE
TYPE_33         DD      ?               ; 8259A IR1 - AVAILABLE
TYPE_34         DD      ?               ; 8259A IR2 - AVAILABLE
TYPE_35         DD      TIMER_PULSE     ; 8259A IR3
TYPE_36         DD      ?               ; 8259A IR4 - AVAILABLE
TYPE_37         DD      ?               ; 8259A IR5 - AVAILABLE
TYPE_38         DD      ?               ; 8259A IR6 - AVAILABLE
TYPE_39         DD      ?               ; 8259A IR7 - AVAILABLE
;
; POINTER FOR TYPE 40 SUPPLIED BY PL/M-86 COMPILER
;
INT_POINTERS                    ENDS

BATTERY                         SEGMENT
; THIS RAM SEGMENT IS BATTERY-POWERED. IT CONTAINS VITAL DATA
;    THAT MUST BE MAINTAINED DURING POWER OUTAGES.
STACK_PTR       DW      ?               ; SP SAVE AREA
STACK_SEG       DW      ?               ; SS SAVE AREA
; SPACE FOR OTHER VARIABLES COULD BE DEFINED HERE.
BATTERY                         ENDS

DATA                            SEGMENT
; RAM SEGMENT THAT IS NOT BACKED UP BY BATTERY
N_PULSES        DB      1 DUP (0)       ; # TIMER PULSES

; ETC.
DATA                            ENDS

STACK                           SEGMENT
; LOCATED IN BATTERY-POWERED RAM
                DW      100 DUP (?)     ; THIS IS AN ARBITRARY STACKSIZE

STACK_TOP       LABEL   WORD            ; LABEL THE INITIAL TOS
STACK                           ENDS

INTERRUPT_HANDLERS      SEGMENT
; INTERRUPT PROCEDURES EXCEPT TYPE 40 (PL/M-86)

                ASSUME: CS:INTERRUPT_HANDLERS,DS:DATA,SS:STACK,ES:BATTERY

POWER_FAIL                      PROC            ; TYPE 2 INTERRUPT
; POWER FAIL DETECT CIRCUIT ACTIVATES NMI LINE ON CPU IF POWER IS
;    ABOUT TO BE LOST. THIS PROCEDURE SAVES THE PROCESSOR STATE IN
;    RAM (ASSUMED TO BE POWERED BY AN AUXILIARY SOURCE) SO THAT IT
;    CAN BE RESTORED BY A WARM START ROUTINE IF POWER RETURNS
```

Figure 2-88. Interrupt Procedures Example

```
; IP, CS, AND FLAGS ARE ALREADY ON THE STACK.
;   SAVE THE OTHER REGISTERS.
                    PUSH        AX
                    PUSH        BX
                    PUSH        CX
                    PUSH        DX
                    PUSH        SI
                    PUSH        DI
                    PUSH        BP
                    PUSH        DS
                    PUSH        ES

; CRITICAL MEMORY VARIABLES COULD ALSO BE SAVED ON THE STACK AT THIS
;     POINT. ALTERNATIVELY, THEY COULD BE DEFINED IN THE "BATTERY"
;     SEGMENT, WHERE THEY WILL AUTOMATICALLY BE PROTECTED IF MAIN POWER
;     IS LOST.

; SAVE SP AND SS IN FIXED LOCATIONS THAT ARE KNOWN BY WARM START ROUTINE.
                    MOV         AX,BATTERY
                    MOV         ES,AX
                    MOV         ES:STACK_PTR,SP
                    MOV         ES:STACK_SEG,SS
; STOP GRACEFULLY
                    HLT
POWER_FAIL                      ENDP

TIMER_PULSE                     PROC                ; TYPE 35 INTERRUPT
; THIS PROCEDURE HANDLES THE 50MS INTERRUPTS GENERATED BY THE 8253.
;     IT COUNTS THE INTERRUPTS AND ACTIVATES THE TYPE 40 INTERRUPT
;     PROCEDURE ONCE PER SECOND.
;
; DS IS ASSUMED TO BE POINTING TO THE DATA SEGMENT
;
; THE 8253 IS RUNNING FREE, AND AUTOMATICALLY LOWERS ITS INTERRUPT
;     REQUEST. IF A DEVICE REQUIRED ACKNOWLEDGEMENT, THE CODE MIGHT GO HERE.
;
; NOW PERFORM PROCESSING THAT MUST NOT BE INTERRUPTED (EXCEPT FOR NMI).
                    INC         N_PULSES
; ENABLE HIGHER-PRIORITY INTERRUPTS AND DO LESS CRITICAL PROCESSING
                    STI
                    CMP         N_PULSES,200    ; 1 SECOND PASSED?
                    JBE         DONE            ; NO, GO ON.
                    MOV         N_PULSES,0      ; YES, RESET COUNT.
                    INT         40              ; UPDATE CLOCK
; SEND NON-SPECIFIC END-OF-INTERRUPT COMMAND TO 8259A, ENABLING EQUAL
;     OR LOWER PRIORITY INTERRUPTS.
DONE:               MOV         AL,020H         ; EOI COMMAND
                    OUT         0C0H,AL         ; 8259A PORT
                    IRET
TIMER_PULSE                     ENDP

INTERRUPT_HANDLERS              ENDS


CODE                SEGMENT
; THIS SEGMENT WOULD NORMALLY RESIDE IN ROM.

                    ASSUME      CS:CODE,DS:DATA,SS:STACK,ES:NOTHING
```

Figure 2-88. Interrupt Procedures Example (Cont'd.)

```
INIT            PROC    NEAR
; THIS PROCEDURE IS CALLED FOR BOTH WARM AND COLD STARTS TO INITIALIZE
;    THE 8253 AND THE 8259A. THIS ROUTINE DOES NOT USE STACK, DATA, OR
;    EXTRA SEGMENTS, AS THEY ARE NOT SET PREDICTABLY DURING A WARM START.
;    INTERRUPTS ARE DISABLED BY VIRTUE OF THE SYSTEM RESET.

; INITIALIZE 8253 COUNTER 1 - OTHER COUNTERS NOT USED.
; CLK INPUT TO COUNTER IS ASSUMED TO BE 1.23 MHZ.

LO50MS          EQU     000H            ; COUNT VALUE IS
HI50MS          EQU     0F0H            ;    61440 DECIMAL.
CONTROL         EQU     0D6H            ; CONTROL PORT ADDRESS
COUNT_1         EQU     0D2H            ; COUNTER 1 ADDRESS
MODE2           EQU     01110100B       ; MODE 2, BINARY

                MOV     DX,CONTROL      ; LOAD CONTROL BYTE
                MOV     AL,MODE2
                OUT     DX,AL
                MOV     DX,COUNT_1      ; LOAD 50MS DOWNCOUNT
                MOV     AL,LO50MS
                OUT     DX,AL
                MOV     AL,HI50MS
                OUT     DX,AL
                ; COUNTER NOW RUNNING, INTERRUPTS STILL DISABLED.

; INITIALIZE 8259A TO: SINGLE INTERRUPT CONTROLLER, EDGE-TRIGGERED,
;    INTERRUPT TYPES 32-40 (DECIMAL) TO BE SENT TO CPU FOR INTERRUPT
;    REQUESTS 0-7 RESPECTIVELY, 8086 MODE, NON-AUTOMATIC END-OF-INTERRUPT.
;    MASK OFF UNUSED INTERRUPT REQUEST LINES.

ICW1            EQU     00010011B       ; EDGE-TRIGGERED, SINGLE 8259A. ICW4 REQUIRED.
ICW2            EQU     00100000B       ; TYPE 20H, 32 - 40D
ICW4            EQU     00000001B       ; 8086 MODE, NORMAL EOI
OCW1            EQU     11110111B       ; MASK ALL BUT IR3
PORT_A          EQU     0C0H            ; ICW1 WRITTEN HERE
PORT_B          EQU     0C2H            ; OTHER ICW'S WRITTEN HERE

                MOV     DX,PORT_A       ; WRITE 1ST ICW
                MOV     AL,ICW1
                OUT     DX,AL
                MOV     DX,PORT_B       ; WRITE 2ND ICW
                MOV     AL,ICW2
                OUT     DX,AL
                MOV     AL,ICW4         ; WRITE 4TH ICW
                OUT     DX,AL
                MOV     AL,0CW1         ; MASK UNUSED IR'S
                OUT     DX,AL
; INITIALIZATION COMPLETE, INTERRUPTS STILL DISABLED
                RET
INIT            ENDP


USER_PGM:
; "REAL" CODE WOULD GO HERE. THE EXAMPLE EXECUTES AN ENDLESS LOOP
;    UNTIL AN INTERRUPT OCCURS.
                JMP     USER_PGM


; EXECUTION STARTS HERE WHEN CPU IS RESET.
POWER_FAIL_STATUS       EQU     0E0H    ; PORT ADDRESS
ENABLE_RAM              EQU     0E2H    ; PORT ADDRESS
```

Figure 2-88. Interrupt Procedures Example (Cont'd.)

```
; ENABLE BATTERY-POWERED RAM SEGMENT
START:          MOV     AL,001H
                OUT     ENABLE_RAM,AL

; DETERMINE WARM OR COLD START
                IN      AL,POWER_FAIL_STATUS
                RCR     AL,1                    ; ISOLATE LOW BIT
                JC      WARM_START

COLD_START:
; INITIALIZE SEGMENT REGISTERS AND STACK POINTER.
                ASSUME  CS:CODE,DS:DATA,SS:STACK,ES:NOTHING
                ; RESET TAKES CARE OF CS AND IP.
                MOV     AX,DATA
                MOV     DS,AX
                MOV     AX,STACK
                MOV     SS,AX
                MOV     SP,OFFSET STACK_TOP

; INITIALIZE 8253 AND 8259A.
                CALL    INIT

; ENABLE INTERRUPTS
                STI

; START MAIN PROCESSING
                JMP     USER_PGM


WARM_START:
; INITIALIZE 8253 AND 8259A.
                CALL    INIT

; RESTORE SYSTEM TO STATE AT THE TIME POWER FAILED
                ; MAKE BATTERY SEGMENT ADDRESSABLE
                    MOV     AX,BATTERY
                    MOV     DX,AX
                ; VARIABLES SAVED IN THE "BATTERY" SEGMENT WOULD BE MOVED
                ;   BACK TO UNPROTECTED RAM NOW. SEGMENT REGISTERS AND
                ;   "ASSUME" DIRECTIVES WOULD HAVE TO BE WRITTEN TO GAIN
                ;   ADDRESSABILITY.

                ; RESTORE THE OLD STACK
                    MOV     SS,DS:STACK_SEG
                    MOV     SP,DS:STACK_PTR
                ; RESTORE THE OTHER REGISTERS
                    POP     ES
                    POP     DS
                    POP     BP
                    POP     DI
                    POP     SI
                    POP     DX
                    POP     CX
                    POP     BX
                    POP     AX
                ; RESUME THE ROUTINE THAT WAS EXECUTING WHEN NMI WAS ACTIVATED.
                ;   I.E., POP CS, IP, & FLAGS, EFFECTIVELY "RETURNING" FROM THE
                ;   NMI PROCEDURE.
                    IRET
CODE            ENDS

                ; TERMINATE ASSEMBLY AND MARK BEGINNING OF THE PROGRAM.
                                END     START
```

Figure 2-88. Interrupt Procedures Example (Cont'd.)

```
TYPE$40:  DO;
  DECLARE (HOUR, MIN, SEC) BYTE PUBLIC;
  UPDATE$TOD: PROCEDURE INTERRUPT 40;
    /*THE PROCESSOR ACTIVATES THIS PROCEDURE
     *TO HANDLE THE SOFTWARE INTERRUPT
     *GENERATED EVERY SECOND BY THE TYPE 35
     *EXTERNAL INTERRUPT PROCEDURE. THIS
     *PROCEDURE UPDATES A REAL-TIME CLOCK.
     *IT DOES NOT PRETEND TO BE "REALISTIC"
     *AS THERE IS NO WAY TO SET THE CLOCK.*/

  SEC = SEC + 1;
  IF SEC = 60 THEN DO;
    SEC = 0;
    MIN = MIN + 1;
    IF MIN = 60 THEN DO;
      MIN = 0;
      HOUR = HOUR + 1;
      IF HOUR = 24 THEN DO;
        HOUR = 0;
        END;
      END;
    END;
  END UPDATE$TOD;
END;
```

Figure 2-88. Interrupt Procedures Example (Cont'd.)

String Operations

Figure 2-89 illustrates typical use of string instructions and repeat prefixes. The XLAT instruction also is demonstrated. The first example simply moves 80 words of a string using MOVS. Then two byte strings are compared to find the alphabetically lower string, as might be done in a sort. Next a string is scanned from right to left (the index register is auto-decremented) to find the last period (".") in the string. Finally a byte string of EBCDIC characters is translated to ASCII. The translation is stopped at the end of the string or when a carriage return character is encountered, whichever occurs first. This is an example of using the string primitives in combination with other instructions to build up more complex string processing operations.

```
ALPHA           SEGMENT
; THIS IS THE DATA THE STRING INSTRUCTIONS WILL USE
OUTPUT          DW 100      DUP (?)
INPUT           DW 100      DUP (?)
NAME_1          DB 'JONES, JONA'
NAME_2          DB 'JONES, JOHN'
SENTENCE        DB 80       DUP (?)
EBCDIC_CHARS    DB 80       DUP (?)
ASCII_CHARS     DB 80       DUP (?)
CONV_TAB        DB 64       DUP(0H)          ; EBCDIC TO ASCII
```

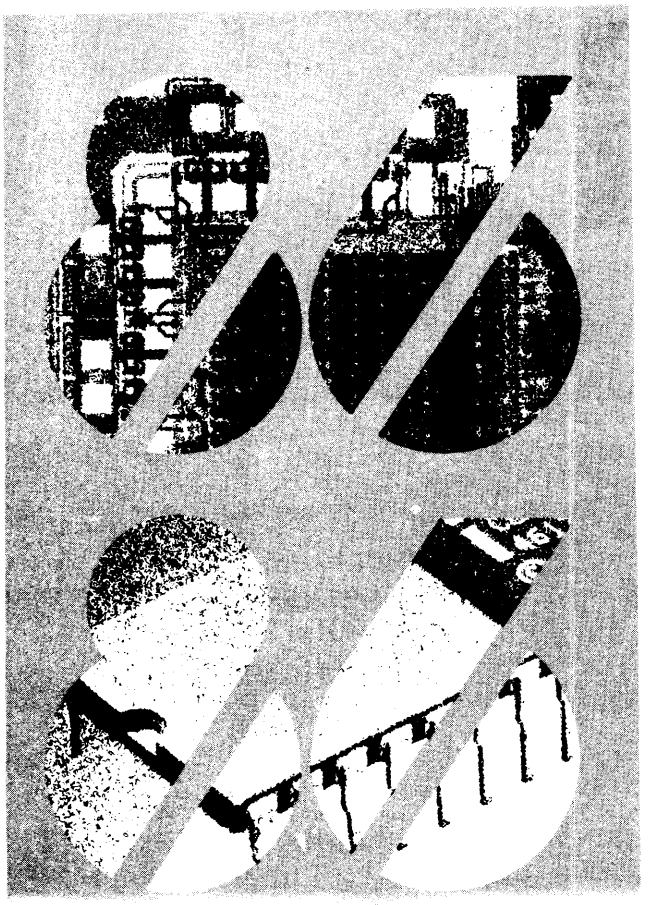
Figure 2-89. String Examples

```
        ; ASCII NULLS ARE SUBSTITUTED FOR "UNPRINTABLE" CHARS
                        DB 1        20H
                        DB 9        DUP (0H)
                        DB 7        '¢', '.', '<', '(', '+', 0H, '&'
                        DB 9        DUP (0H)
                        DB 8        '!', '$', '*', ')', ';', '¬', '-', '/'
                        DB 8        DUP (0H)
                        DB 6        '¦', ',', '%', '_', '>', '?'
                        DB 9        DUP (0H)
                        DB 17       '¨', ':', '#', '@', ''', '=', '"',
                                    0H, 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i'
                        DB 7        DUP (0H)
                        DB 9        'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r'
                        DB 7        DUP (0H)
                        DB 9        '≃', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z'
                        DB 22       DUP (0H)
                        DB 10       '¦', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I'
                        DB 6        DUP (0H)
                        DB 10       '¦', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R'
                        DB 6        DUP (0H)
                        DB 10       '¦', 0H, 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z'
                        DB 6        DUP (0H)
                        DB 10       '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'
                        DB 6        DUP (0H)

ALPHA                   ENDS

STACK                   SEGMENT
                        DW 100      DUP (?)         ; THIS IS AN ARBITRARY STACK SIZE
                                                    ; FOR ILLUSTRATION ONLY.
STACK_BASE              LABEL       WORD            ; INITIAL TOS
STACK                   ENDS

CODE                    SEGMENT
BEGIN:                  ; SET UP SEGMENT REGISTERS. NOTICE THAT
                        ; ES & DS POINT TO THE SAME SEGMENT, MEANING
                        ; THAT THE CURRENT EXTRA & DATA
                        ; SEGMENTS FULLY OVERLAP. THIS ALLOWS
                        ; ANY STRING IN "ALPHA" TO BE USED
                        ; AS A SOURCE OR A DESTINATION.
                        ASSUME CS: CODE, SS: STACK,
&                               DS: ALPHA, ES: ALPHA
                        MOV         AX, STACK
                        MOV         SS, AX
                        MOV         SP, OFFSET STACK_BASE ; INITIAL TOS
                        MOV         AX, ALPHA
                        MOV         DS, AX
                        MOV         ES, AX

; MOVE THE FIRST 80 WORDS OF "INPUT" TO
;     THE LAST 80 WORDS OF "OUTPUT".
                        LEA         SI, INPUT           ; INITIALIZE
                        LEA         DI, OUTPUT + 20     ; INDEX REGISTERS
```

Figure 2-89. String Examples (Cont'd.)

```
                MOV     CX, 80                  ; REPETITION COUNT
                CLD                             ; AUTO-INCREMENT
        REP     MOVS    OUTPUT, INPUT

; FIND THE ALPHABETICALLY LOWER OF 2 NAMES.
                MOV     SI, OFFSET NAME_1   ; ALTERNATIVE
                MOV     DI, OFFSET NAME_2   ; TO LEA
                MOV     CX, SIZE NAME_2     ; CHAR. COUNT
                CLD                         ; AUTO-INCREMENT
        REPE    CMPS    NAME_2, NAME_1      "WHILE EQUAL"
                JB      NAME_2_LOW
NAME_1_LOW:             ; NOT IN THIS EXAMPLE
NAME_2_LOW:             ; CONTROL COMES HERE IN THIS EXAMPLE.
                        ; DI POINTS TO BYTE ('H') THAT
                        ; COMPARED UNEQUAL.

; FIND THE LAST PERIOD ('.') IN A TEXT STRING.
                MOV     DI, OFFSET SENTENCE +
&                           LENGTH SENTENCE   ; START AT END
                MOV     CX, SIZE SENTENCE
                STD                         ; AUTO-DECREMENT
                MOV     AL, '.'             ; SEARCH ARGUMENT
        REPNE   SCAS    SENTENCE            ; "WHILE NOT ="
                JCXZ    NO_PERIOD           ; IF CX=0, NO PERIOD FOUND
PERIOD:         ; IF CONTROL COMES HERE THEN
                ;   DI POINTS TO LAST PERIOD IN SENTENCE.
NO_PERIOD:      ; ETC.

; TRANSLATE A STRING OF EBCDIC CHARACTERS
;       TO ASCII, STOPPING IF A CARRIAGE RETURN
;       (0DH ASCII) IS ENCOUNTERED.
                MOV     BX, OFFSET CONV_TAB   ; POINT TO TRANSLATE TABLE
                MOV     SI, OFFSET EBCDIC_CHARS   ; INITIALIZE
                MOV     DI, OFFSET ASCII_CHARS    ;    INDEX REGISTERS
                MOV     CX, SIZE ASCII_CHARS      ;    AND COUNTER
                CLD                         ; AUTO-INCREMENT
NEXT:           LODS    EBCDIC_CHARS        ; NEXT EBCDIC CHAR IN AL
                XLAT    CONV_TAB            ; TRANSLATE TO ASCII
                STOS    ASCII_CHARS         ; STORE FROM AL
                TEST    AL, 0DH             ; IS IT CARRIAGE RETURN?
                LOOPNE  NEXT                ; NO, CONTINUE WHILE CX NOT 0
                JE      CR_FOUND            ; YES, JUMP
                ; CONTROL COMES HERE IF ALL CHARACTERS
                ;       HAVE BEEN TRANSLATED BUT NO
                ;       CARRIAGE RETURN IS PRESENT.
                ; ETC.

CR_FOUND:
                ; DI-1 POINTS TO THE CARRIAGE RETURN
                ;   IN ASCII_CHARS.

CODE            ENDS
                END
```

Figure 2-89. String Examples (Cont'd.)



Chapter 3
The 8089
Input/Output Processor

CHAPTER 3
THE 8089 INPUT/OUTPUT PROCESSOR

This chapter describes the 8089 Input/Output Processor (IOP). Its organization parallels Chapter 2; that is, sections generally proceed from hardware to software topics as follows:

1. Processor Overview
2. Processor Architecture
3. Memory
4. Input/Output
5. Multiprocessing Features
6. Processor Control and Monitoring
7. Instruction Set
8. Addressing Modes
9. Programming Facilities
10. Programming Guidelines and Examples

As in Chapter 2, the discussion is confined to covering the hardware in functional terms; timing, electrical characteristics and other physical interfacing data are provided in Chapter 4.

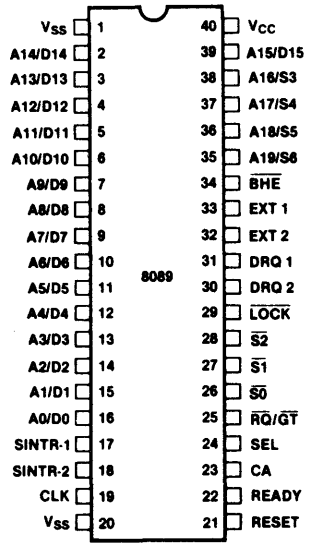


Figure 3-1. 8089 Input/Output Processor Pin Diagram

3.1 Processor Overview

The 8089 Input/Output Processor is a high-performance, general-purpose I/O system implemented on a single chip. Within the 8089 are two independent I/O channels, each of which combines attributes of a CPU with those of a very flexible DMA (direct memory access) controller. For example, channels can execute programs like CPUs; the IOP instruction set has about 50 different types of instructions specifically designed for efficient input/output processing. Each channel also can perform high-speed DMA transfers; a variety of optional operations allow the data to be manipulated (e.g., translated or searched) as it is transferred. The 8089 is contained in a 40-pin dual in-line package (figure 3-1) and operates from a single +5V power source. An integral member of the 8086 family, the IOP is directly compatible with both the 8086 and 8088 when these processors are configured in maximum mode. The IOP also may be used in any system that incorporates Intel's Multibus™ shared bus architecture, or a superset of the Multibus™ design.

Evolution

Figure 3-2 depicts the general trend in CPU and I/O device relationships in the first three generations of microprocessors. First generation CPUs were forced to deal directly with substantial numbers of TTL components, often performing transfers at the bit level. Only a very limited number of relatively slow devices could be supported.

Single-chip interface controllers were introduced in the second generation. These devices removed the lowest level of device control from the CPU and let the CPU transfer whole bytes at once. With the introduction of DMA controllers, high-speed devices could be added to a system, and whole blocks of data could be transferred without CPU intervention. Compared to the previous generation, I/O device and DMA controllers allowed microprocessors to be applied to problems that required moderate levels of I/O, both in terms of the numbers of devices that could be supported and the transfer speeds of those devices.

The controllers themselves, however, still required a considerable amount of attention from the CPU, and in many cases the CPU had to respond to an interrupt with every byte read or written. The CPU also had to stop while DMA transfers were performed.

The 8089 introduces the third generation of input/output processing. It continues the trend of simplifying the CPU's "view" of I/O devices by removing another level of control from the CPU. The CPU performs an I/O operation by building a message in memory that describes the function to be performed; the IOP reads the message, carries out the operation and notifies the CPU when it has finished. All I/O devices appear to the CPU as transmitting and receiving whole blocks of data; the IOP can make both byte- and word-level transfers invisible to the CPU. The IOP assumes all device controller overhead, performs both programmed and DMA transfers, and can recover from "soft" I/O errors without CPU intervention; all of these activities may be performed while the CPU is attending to other tasks.

Principles of Operation

Since the 8089 is a new concept in microprocessor components, this section surveys the basic operation of the IOP as background to the detailed descriptions provided in the rest of the chapter. This summary deliberately omits some operating details in order to provide an integrated overview of basic concepts.

CPU/IOP Communications

A CPU communicates with an IOP in two distinct modes: initialization and command. The initialization sequence is typically performed when the system is powered-up or reset. The CPU initializes the IOP by preparing a series of linked message blocks in memory. On a signal from the CPU, the IOP reads these blocks and determines from them how the data buses are configured and how access to the buses is to be controlled.

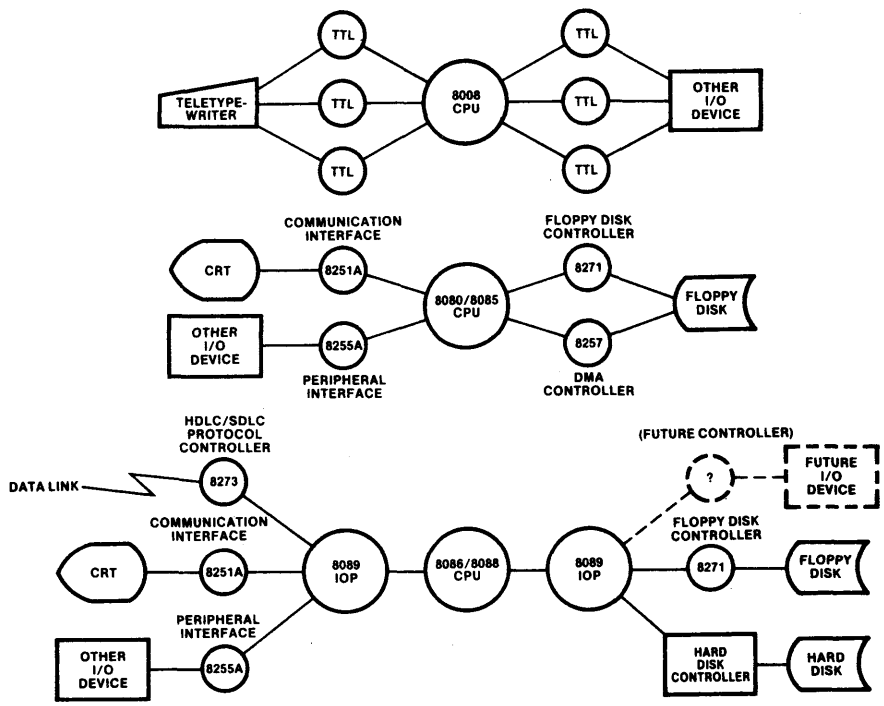


Figure 3-2. IOP Evolution

Following initialization, the CPU directs all communications to either of the IOP's two channels; indeed, during normal operation the IOP appears to be two separate devices—channel 1 and channel 2. All CPU-to-channel communications center on the channel control block (CB) illustrated in figure 3-3. The CB is located in the CPU's memory space, and its address is passed to the IOP during initialization. Half of the block is dedicated to each channel. The channel maintains the BUSY flag that indicates whether it is in the midst of an operation or is available for a new command. The CPU sets the CCW (channel command word) to indicate what kind of operation the IOP is to perform. Six different commands allow the CPU to start and stop programs, remove interrupt requests, etc.

If the CPU is dispatching a channel to run a program, it directs the channel to a parameter block (PB) and a task block (TB); these are also shown in figure 3-3. The parameter block is analogous to a parameter list passed by a program to a subroutine; it contains variable data that the channel program is to use in carrying out its assignment. The parameter block also may contain space for variables (results) that the channel is to return to the CPU. Except for the first two words, the format and size of a parameter block are completely open; the PB may be set up to exchange any kind of information between the CPU and the channel program.

A task block is a channel program—a sequence of 8089 instructions that will perform an operation. A typical channel program might use parameter block data to set up the IOP and a device controller for a transfer, perform the transfer, return the results, and then halt. However, there are no restrictions on what a channel program can do; its function may be simple or elaborate to suit the needs of the application.

Before the CPU starts a channel program, it links the program (TB) to the parameter block and the parameter block to the CB as shown in figure 3-3. The links are standard 8086/8088 doubleword pointer variables; the lower-addressed word contains an offset, and the higher-addressed word contains a segment base value. A system may have many different parameter and task blocks; however, only one of each is ever linked to a channel at any given time.

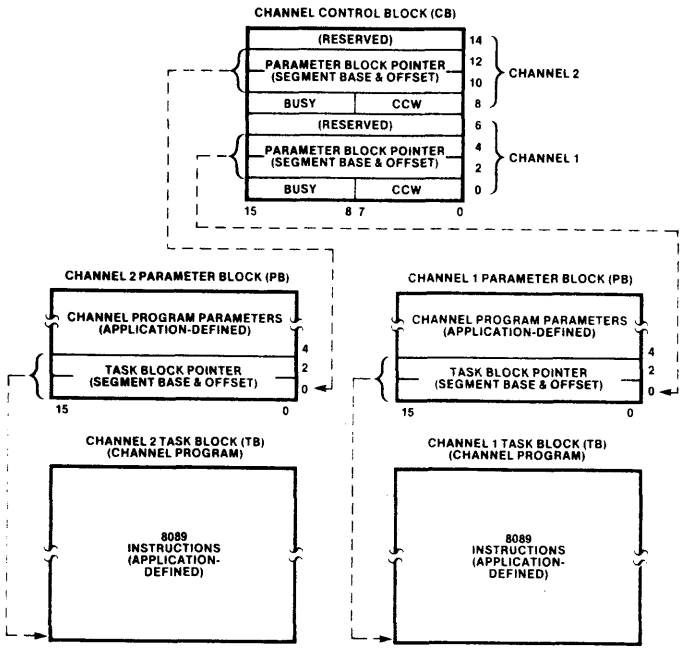


Figure 3-3. Command Communication Blocks

After the CPU has filled in the CCW and has linked the CB to a parameter block and a task block, if appropriate, it issues a channel attention (CA). This is done by activating the IOP's CA (channel attention) and SEL (channel select) pins. The state of SEL at the falling edge of CA directs the channel attention to channel 1 or channel 2. If the IOP is located in the CPU's I/O space, it appears to the CPU as two consecutive I/O ports (one for each channel), and an OUT instruction to the port functions as a CA. If the IOP is memory-mapped, the channels appear as two consecutive memory locations, and any memory reference instruction (e.g., MOV) to these locations causes a channel attention.

An IOP channel attention is functionally similar to a CPU interrupt. When the channel recognizes the CA, it stops what it is doing (it will typically be idle) and examines the command in the CCW. If it is to start a program, the channel loads the addresses of the parameter and task blocks into internal registers, sets its BUSY flag and starts executing the channel program. After it has issued the CA, the CPU is free to perform other processing; the channel can perform its function in parallel, subject to limitations imposed by bus configurations (discussed shortly).

When the channel has completed its program, it notifies the CPU by clearing its BUSY flag in the CB. Optionally, it may issue an interrupt request to the CPU.

The CPU/IOP communication structure is summarized in figure 3-4. Most communication takes place via "message areas" shared in common memory. The only direct hardware communications between the devices are channel attentions and interrupt requests.

Channels

Each of the two IOP channels operates independently, and each has its own register set, channel attention, interrupt request and DMA control signals. At a given point in time, a channel may be idle, executing a program, performing a DMA transfer, or responding to a channel attention. Although only one channel actually runs at a time, the channels can be active concurrently, alternating their operations (e.g., channel 1 may execute instructions in the periods between successive DMA transfer cycles run by channel 2). A built-in priority system allows high-priority activities on one channel to preempt less critical operations on the other channel. The CPU is able to further adjust priorities to handle special cases. The CPU starts the channel and can halt it, suspend it, or cause it to resume a suspended operation by placing different values in the CCW.

Channel Programs (Task Blocks)

Channel programs are written in ASM-89, the 8089 assembly language. About 50 basic instructions are available. These instructions operate on bit, byte, word and doubleword (pointer) variable types; a 20-bit physical address variable type (not used by the 8086/8088) can also be manipulated. Data may be taken from registers, immediate constants and memory. Four memory addressing modes allow flexible access to both memory variables and I/O devices located anywhere in either the CPU's megabyte memory space or in the 8089's 64k I/O space.

The IOP instruction set contains general purpose instructions similar to those found in CPUs as well as instructions specifically tailored for I/O

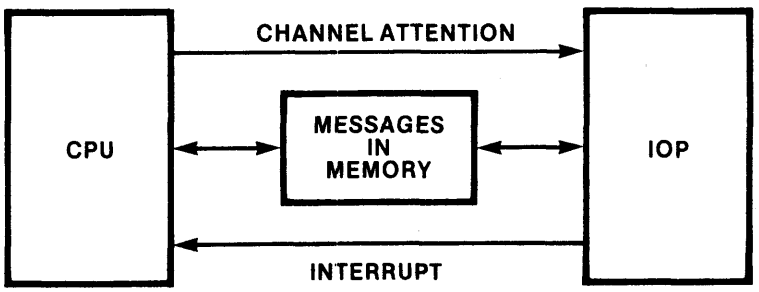


Figure 3-4. CPU/IOP Communication

operations. Data transfer, simple arithmetic, logical and address manipulation operations are available. Unconditional jump and call instructions also are provided so that channel programs can link to each other. An individual bit may be set or cleared with a single instruction. Conditional jumps can test a bit and jump if it is set (or cleared), or can test a value and jump if it is zero (or non-zero). Other instructions initiate DMA transfers, perform a locked test-and-set semaphore operation, and issue an interrupt request to the CPU.

DMA Transfers

The 8089 XFER (transfer) instruction prepares the channel for a DMA transfer. It executes one additional instruction, then suspends program execution and enters the DMA transfer mode. The transfer is governed by channel registers setup by the program prior to executing the XFER instruction.

Data is transferred from a source to a destination. The source and destination may be any locations in the CPU's memory space or in the IOP's I/O space; the IOP makes no distinction between memory components and I/O devices. Thus transfers may be made from I/O device to memory, memory to I/O device, memory to memory and I/O device to I/O device. The IOP automatically matches 8- and 16-bit components to each other.

Individual transfer cycles (i.e., the movement of a byte or a word) may be synchronized by a signal (DMA request) from the source or from the destination. In the synchronized mode, the channel waits for the synchronizing signal before starting the next transfer cycle. The transfer also may be unsynchronized, in which case the channel begins the next transfer cycle immediately upon completion of the previous cycle.

A transfer cycle is performed in two steps: fetching a byte or word from the source into the IOP and then storing it from the IOP into the destination. The IOP automatically optimizes the transfer to make best use of the available data bus widths. For example, if data is being transferred from an 8-bit device to memory that resides on a 16-bit bus (e.g., 8086 memory), the IOP will normally run two one-byte fetch cycles and then store the full word in a single cycle.

Between the fetch and store cycles, the IOP can operate on the data. A byte may be translated to another code (e.g., EBCDIC to ASCII), or compared to a search value, or both, if desired.

A transfer can be terminated by several programmer-specified conditions. The channel can stop the transfer when a specified number (up to 64k) of bytes has been transferred. An external device may stop a transfer by signaling on the channel's external terminate pin. The channel can stop the transfer when a byte (possibly translated) compares equal, or unequal, to a search value. Single-cycle termination, which stops unconditionally after one byte or word has been stored, is also available.

When the transfer terminates, the channel automatically resumes program execution. The channel program can determine the cause of the termination in situations where multiple terminations are possible (e.g., terminating when 80 bytes are transferred or a carriage return character is encountered, whichever occurs first). As an example of post-transfer processing, the channel program could read a result register from the I/O device controller to determine if the transfer was performed successfully. If not (e.g., a CRC error was detected by the controller), the channel program could retry the operation without CPU intervention.

A channel program typically ends by posting the result of the operation to a field supplied in the parameter block, optionally interrupting the CPU, and then halting. When the channel halts, its BUSY flag in the channel control block is cleared to indicate its availability for another operation. As an alternative to being interrupted by the channel, the CPU can poll this flag to determine when the operation has been completed.

Bus Configurations

As shown in figure 3-5, the IOP can access memory or ports (I/O devices) located in a 1-megabyte system space and memory or ports located in a 64-kilobyte I/O space. Although the IOP only has one physical data bus, it is useful to think of the IOP as accessing the system space via a system data bus and the I/O space over an I/O data bus. The distinction between the "two" buses is based on the type-of-cycle signals output

by the 8288 Bus Controller. Components in the system space respond to the memory read and memory write signals, whether they are memory or I/O devices. Components in the I/O space respond to the I/O read and I/O write signals. Thus I/O devices located in the system space are memory-mapped and memory in the I/O space is I/O-mapped. The two basic configuration options differ in the degree to which the IOP shares these buses with the CPU. Both configurations require an 8086/8088 CPU to be strapped in maximum mode.

In the local configuration, shown in figure 3-6, the IOP (or IOPs if two are used) shares both buses with the CPU. The system bus and the I/O bus are the same width (8 bits if the CPU is an 8088 or 16 bits if the CPU is an 8086). The IOP system space corresponds to the CPU memory space, and the IOP I/O space corresponds to the CPU I/O space. Channel programs are located in the system space; I/O devices may be located in either space. The IOP requests use of the bus for channel program instruction fetches as well as for DMA and programmed transfers. In the local configuration, either the IOP or the CPU may use the buses, but not both simultaneously. The advantage of the local configuration is that intelligent DMA may be added to a system with no additional components beyond the IOP. The disadvantage is that parallel operation of the processors is limited to cases in which the CPU has instruction in its queue that can be executed without using the bus.

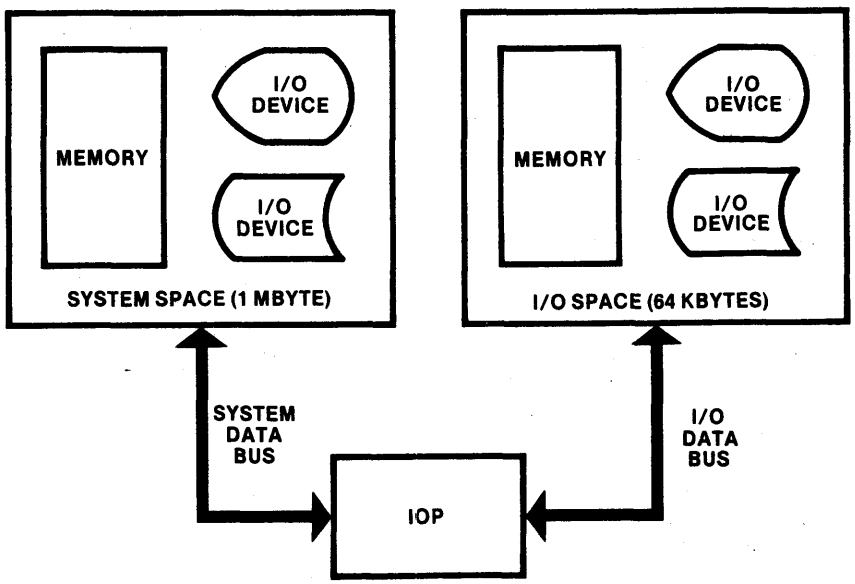


Figure 3-5. IOP Data Buses

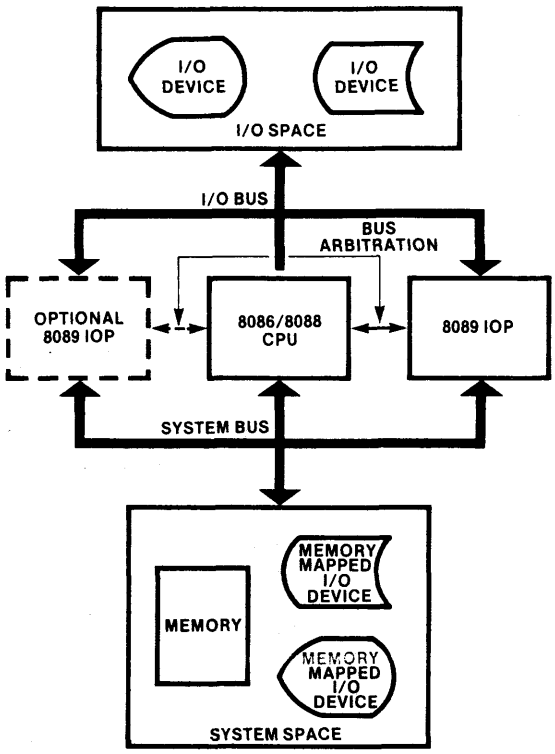


Figure 3-6. Local Configuration

In the remote configuration (figure 3-7), the IOP (or IOPs) shares a common system bus with the CPU. Access to this bus is controlled by 8289 Bus Arbiters. The IOP's I/O bus, however, is physically separated from the CPU in the remote configuration. Two IOPs can share the local I/O bus. Any number of remote IOPs may be contained in a system, configured in remote clusters of one or two. The local I/O bus need not be the same physical width as the shared system bus, allowing an IOP, for example, to interface 8-bit peripherals to an 8086. In the remote configuration, the IOP can access local I/O devices and memory without using the shared system bus, thereby reducing bus contention with the CPU. Contention can further be reduced by locating the IOP's channel programs in the local I/O space. The IOP can then also fetch instructions without accessing the system bus. Parameter, channel control and other CPU/IOP communication blocks must be located in system memory, however, so that both processors can access them. The remote configuration thus increases the degree to which an IOP and a CPU can operate in parallel and thereby increases a system's throughput potential. The price paid for this is that additional hardware must be added to arbitrate use of the shared bus, and to separate the shared and local buses (see Chapter 4 for details).

It is also possible to configure an IOP remote to one CPU, and local to another CPU (see figure 3-8). The local CPU could be used to perform heavy computational routines for the IOP.

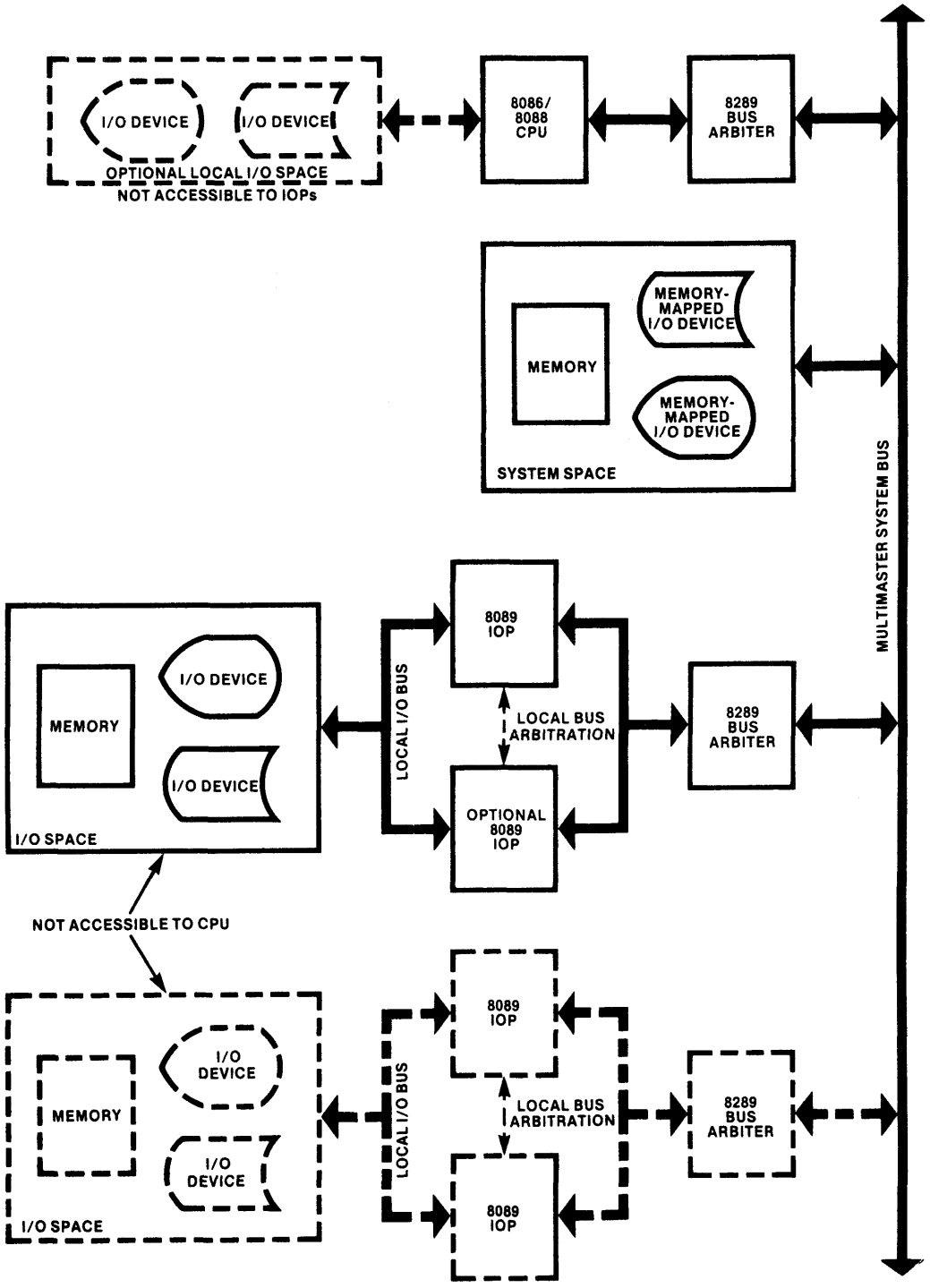


Figure 3-7. Remote Configuration

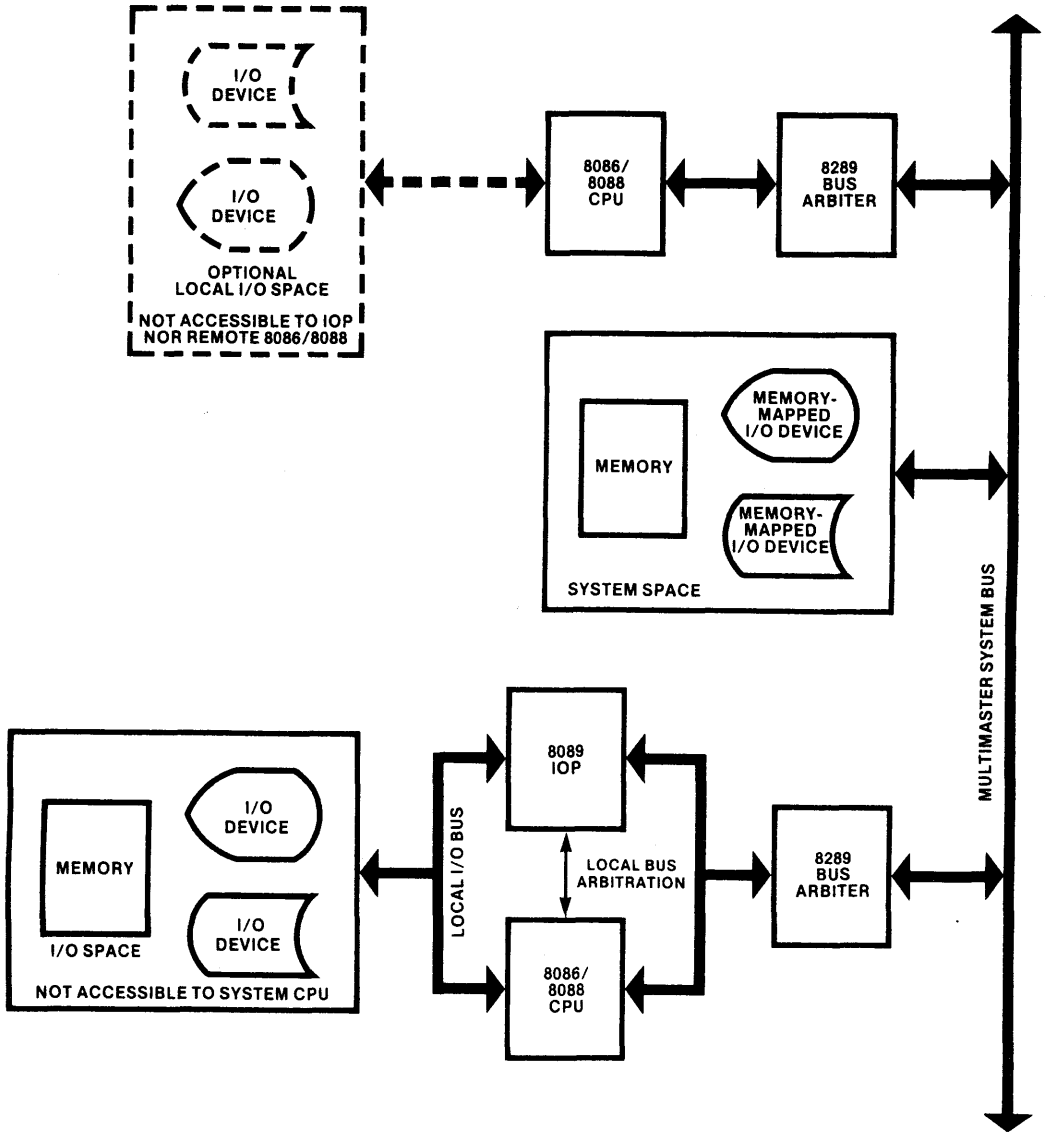


Figure 3-8. Remote IOP Configured With Local 8086/8088

A Sample Transaction

Figure 3-9 shows how a CPU and an IOP might work together to read a record (sector) from a floppy disk. This example is not illustrative of the IOP's full capabilities, but it does review its basic operation and its interaction with a CPU.

The CPU must first obtain exclusive use of a channel. This can be done by performing a "test and set lock" operation on the selected channel's BUSY flag. Assuming the CPU wants to use channel 1, this could be accomplished in PL/M-86 by coding similar to the following:

```
DO WHILE LOCKSET (@CH1.BUSY,0FFH);
    END;
```

In ASM-86 a loop containing the XCHG instruction prefixed by LOCK would accomplish the same thing, namely testing the BUSY flag until it is clear (0H), and immediately setting it to FFH (busy) to prevent another task or processor from obtaining use of the channel.

Having obtained the channel, the CPU fills in a parameter block (see figure 3-10). In this case, the CPU passes the following parameters to the channel: the address of the floppy disk controller, the address of the buffer where the data is to be placed, and the drive, track and sector to be read. It also supplies space for the IOP to return the result of the operation. Note that this is quite a "low-level" parameter block in that it implies that the CPU has detailed knowledge of the I/O system. For a "real" system, a higher-level parameter block would isolate the CPU from I/O device characteristics. Such a block might contain more general parameters such as file name and record key.

After setting up the parameter block, the CPU writes a "start channel program" command in channel 1's CCW. Then the CPU places the address of the desired channel program in the parameter block and writes the parameter block address in the CB. Notice that in this simple example, the CPU "knows" the address of the channel program for reading from the disk, and presumably also "knows" the address of another program for writing, etc. A more general solution would be to place a function code (read, write, delete, etc.) in the parameter block and let a single channel program execute different routines depending on which function is requested.

After the communication blocks have been setup, the CPU dispatches the channel by issuing a channel attention, typically by an OUT instruction for an I/O-mapped 8089, or a MOV or other memory reference instruction for a memory-mapped 8089.

The channel begins executing the channel program (task block) whose address has been placed in the parameter block by the CPU. In this case the program initializes the 8271 Floppy Disk Controller by sending it a "read data" command followed by a parameter indicating the track to be read. The program initializes the channel registers that define and control the DMA transfer.

Having prepared the 8271 and the channel itself, the channel program executes a XFER instruction and sends a final parameter (the sector to be read) to the 8271. (The 8271 enters DMA transfer mode immediately upon receiving the last of a series of parameters; sending the last parameter after the XFER instruction gives the channel time to setup for the transfer.) The DMA transfer begins when the 8271 issues a DMA request to the channel. The transfer continues until the 8271 issues an interrupt request, indicating that the data has been transferred or that an error has occurred. The 8271's interrupt request line is tied to the IOP's EXT1 (external terminate on channel 1) pin so that the channel interprets an interrupt request as an external terminate condition. Upon termination of the transfer, the channel resumes executing instructions and reads the 8271 result register to determine if the data was read successfully. If a soft (correctable) error is indicated, the IOP retries the transfer. If a hard (uncorrectable) error is detected, or if the transfer has been successful, the IOP posts the content of the result register to the parameter block result field, thus passing the result back to the CPU. The channel then interrupts the CPU (to inform the CPU that the request has been processed) and halts.

When the CPU recognizes the interrupt, it inspects the result field in the parameter block to see if the content of the buffer is valid. If so, it uses the data; otherwise it typically executes an error routine.

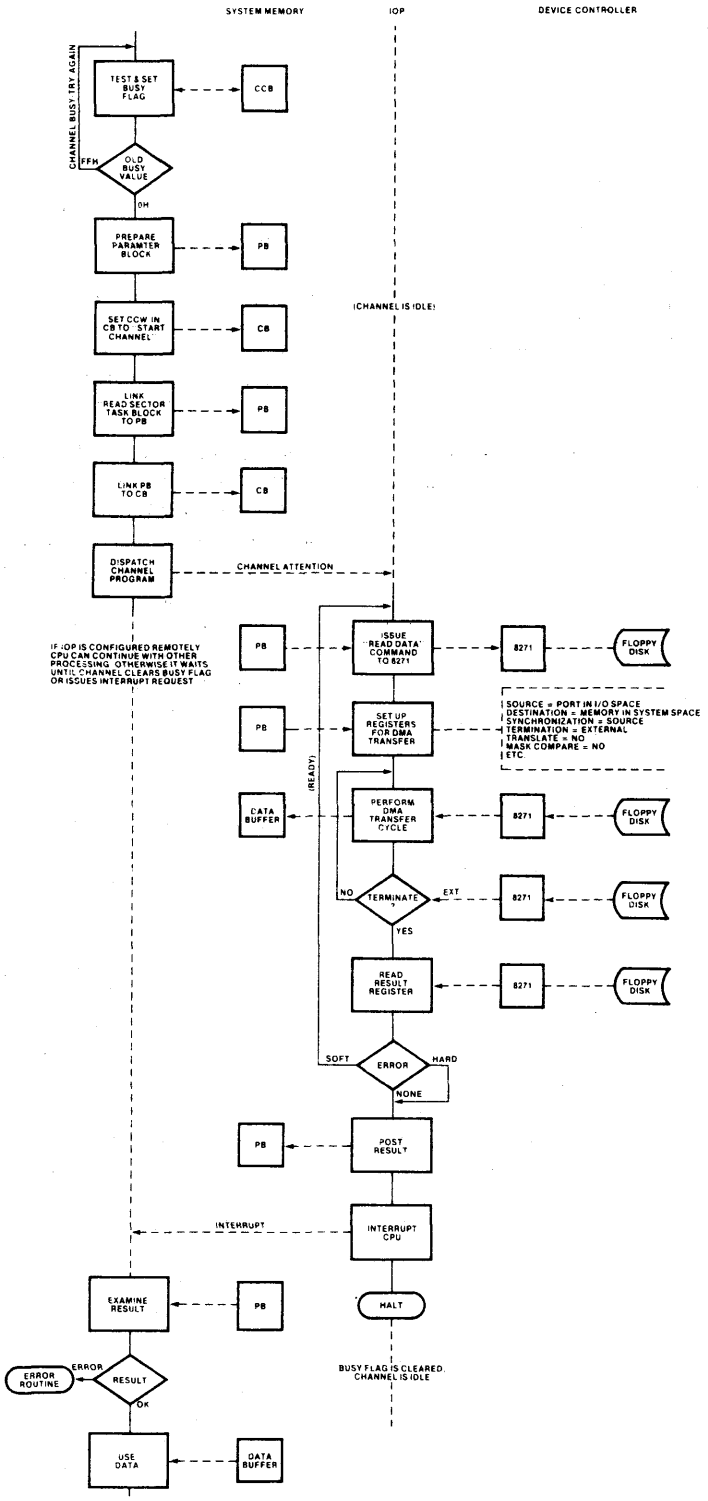


Figure 3-9. Sample CPU/IOP Transaction

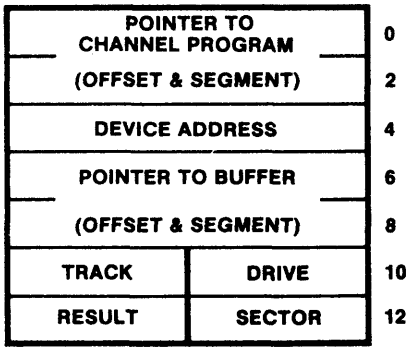


Figure 3-10. Sample Parameter Block

Applications

Combining the raw speed and responsiveness of a traditional DMA controller, an I/O-oriented instruction set, and a flexible bus organization, the 8089 IOP is a very versatile I/O system. Applications with demanding I/O requirements, previously beyond the abilities of microcomputer systems, can be undertaken with the IOP. These kinds of I/O-intensive applications include:

- systems that employ high-bandwidth, low-latency devices such as hard disks and graphics terminals;
- systems with many devices requiring asynchronous service; and
- systems with high-overhead peripherals such as intelligent CRTs and graphics terminals.

In addition, virtually every application that performs a moderate amount of I/O can benefit from the design philosophy embodied in the IOP: system functions should be distributed among special-purpose processors. An IOP channel program is likely to be both faster and smaller than an equivalent program implemented with a CPU. Programming also is more straightforward with the IOP's specialized instruction set.

Removing I/O from the CPU and assigning it to one or more IOPs simplifies and structures a system's design. The main interface to the I/O system can be limited to the parameter blocks. Once these are defined, the I/O system can be designed and implemented in parallel with the rest of the system. I/O specialists can work on the I/O system without detailed knowledge of the application; conversely, the operating system and application teams do not need to be expert in the operation of I/O devices. Standard high-level I/O systems can be used in multiple application systems. Because the application and I/O systems are almost independent, application system changes can be introduced without affecting the I/O system. New peripherals can similarly be incorporated into a system without impacting applications or operating system software. The IOP's simple CPU interface also is designed to be compatible with future Intel CPUs.

Keeping in mind the true general-purpose nature of the IOP, some of the situations where it can be used to advantage are:

- Bus matching - The IOP can transfer data between virtually any combination of 8- and 16-bit memory and I/O components. For example, it can interface a 16-bit peripheral to an 8-bit CPU bus, such as the 8088 bus. The IOP also provides a straightforward means of performing DMA between an 8-bit peripheral and 8086 memory that is split into odd- and even-addressed banks. The 8089 can access both 8- and 16-bit peripherals connected to a 16-bit bus.
- String processing - The 8089 can perform a memory move, translate, scan-for-match or scan-for-nonmatch operation much faster than the equivalent instructions in an 8086 or 8088. Translate and scan operations can be setup so that the source and destination refer to the same addresses to permit the string to be operated on in place.
- Spooling - Data from low-speed devices such as terminals and paper tape readers can be read by the 8089 and placed in memory or on disk until the transmission is complete. The IOP can then transfer the data at high speed when it is needed by an application program. Conversely, output data ultimately destined for a low-speed device such as a printer, can be temporarily spooled to disk and then printed later. This permits batches of data to be gathered or distributed by low-priority programs that run in the background, essentially using up "spare" CPU and IOP cycles. Application programs that use or produce the data can execute faster because they are not bound by the low-speed devices.

- Multitasking operating systems - A multitasking operating system can dispatch I/O tasks to channels with an absolute minimum of overhead. Because a remote channel can run in parallel with the CPU, the operating system's capacity for servicing application tasks can increase dramatically, as can its ability to handle more, and faster, I/O devices. If both channels of an IOP are active concurrently, the IOP automatically gives preference to the higher-priority activity (e.g., DMA normally preempts channel program execution). The operating system can adjust the priority mechanism and also can halt or suspend a channel to take care of a critical asynchronous event.

- Disk systems - The IOP can meet the speed and latency requirements of hard disks. It can be used to implement high-level, file-oriented systems that appear to application programs as simple commands: OPEN, READ, WRITE, etc. The IOP can search and update disk directories and maintain free space maps. "Hierarchical memory" systems that automatically transfer data among memory, high-speed disks and low-speed disks, based on frequency of use, can be built around IOPs. Complex database searches (reading data directly or following pointer chains) can appear to programs as simple commands and can execute in parallel with application programs if an IOP is configured remotely.

- Display terminals - The 8089 is well suited to handling the DMA requirements of CRT controllers. The IOP's transfer bandwidth is high enough to support both alphanumeric and graphic displays. The 8089 can assume responsibility for refreshing the display from memory data; in the remote configuration, the refresh overhead can be removed from the system bus entirely. Linked-list display algorithms may be programmed to perform sophisticated modes of display.

  Each time it performs a refresh operation, the IOP can scan a keyboard for input and translate the key's row-and-column format into an ASCII or EBCDIC character. The 8089 can buffer the characters, scanning the stream until an end-of-message character (e.g., carriage return) is detected, and then interrupt the CPU.

  A single IOP can concurrently support an alphanumeric CRT and keyboard on one channel and a floppy disk on the other channel. This configuration makes use of approximately 30 percent of the available bus bandwidth. Performance can be increased within the available bus bandwidth by adding an 8086 or 8088 CPU to a remote IOP configuration. This configuration can provide scaling, rotation or other sophisticated display transformations.

3.2 Processor Architecture

The 8089 is internally divided into the functional units depicted schematically in figure 3-11. The units are connected by a 20-bit data path to obtain maximum internal transfer rates.

Common Control Unit (CCU)

All IOP operations (instructions, DMA transfer cycles, channel attention responses, etc.) are composed of sequences of more basic processes called internal cycles. A bus cycle takes one internal cycle; the execution of an instruction may require several internal cycles. There are 23 different types of internal cycles each of which takes from two to eight clocks to execute, not including possible wait states and bus arbitration times.

The common control unit (CCU) coordinates the activities of the IOP primarily by allocating internal cycles to the various processor units; i.e., it determines which unit will execute the next internal cycle. For example, when both channels are active, the CCU determines which channel has priority and lets that channel run; if the channels have equal priority, the CCU "interleaves" their execution (this is discussed more fully later in this section). The CCU also initializes the processor.

Arithmetic/Logic Unit (ALU)

The ALU can perform unsigned binary arithmetic on 8- and 16-bit binary numbers. Arithmetic results may be up to 20 bits in length. Available arithmetic instructions include addition, increment and decrement. Logical operations ("and," "or" and "not") may be performed on either 8- or 16-bit quantities.

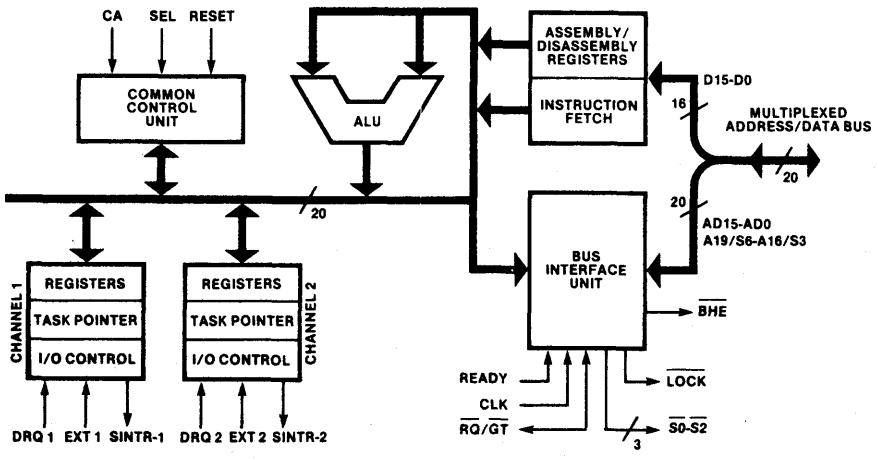


Figure 3-11. 8089 Block Diagram

Assembly/Disassembly Registers

All data entering the chip flows through these registers. When data is being transferred between different width buses, the 8089 uses the assembly/disassembly registers to effect the transfer in the fewest possible bus cycles. In a DMA transfer from an 8-bit peripheral to 16-bit memory, for example, the IOP runs two bus cycles, picking up eight bits in each cycle, assembles a 16-bit word, and then transfers the word to memory in a single bus cycle. (The first and last cycles of a transfer may be performed differently to accommodate odd-addressed words; the IOP automatically adjusts for this condition.)

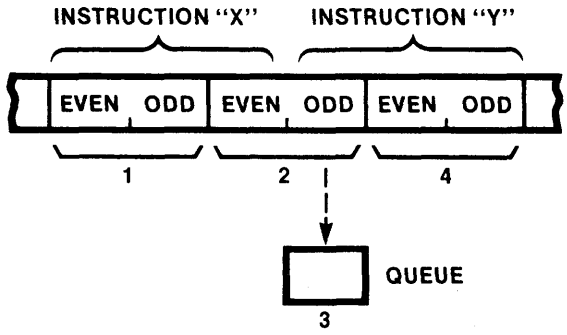
Instruction Fetch Unit

This unit controls instruction fetching for the executing channel (one channel actually runs at a time). If the bus over which the instructions are being fetched is eight bits wide, then the instructions are obtained one byte at a time, and each fetch requires one bus cycle. If the instructions are being fetched over a 16-bit bus, then the instruction fetch unit automatically employs a 1-byte queue to reduce the number of bus cycles. Each channel has its own queue, and the activity of one channel does not affect the other's queue.

During sequential execution, instructions are fetched one word at a time from even addresses; each fetch requires one bus cycle. This process is shown graphically in figure 3-12. When the last byte of an instruction falls on an even address, the odd-addressed byte (the first byte of the following instruction) of the fetched word is saved in the queue. When the channel begins execution of the next instruction, it fetches the first byte from the queue rather than from memory. The queue, then, keeps the processor fetching words, rather than bytes, thereby reducing its use of the bus and increasing throughput.

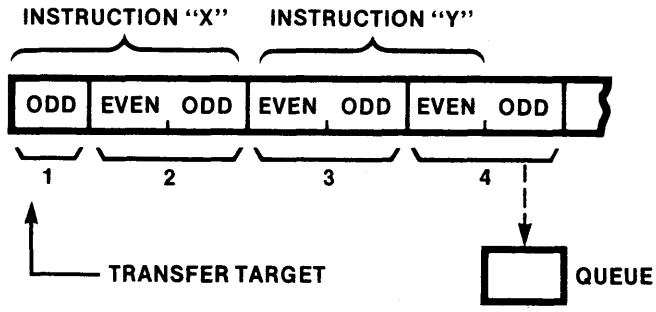
The processor fetches bytes rather than words in two cases. If a program transfer instruction (e.g., JMP or CALL) directs the processor to an instruction located at an odd address, the first byte of the instruction is fetched by itself as shown in figure 3-13. This is because the program transfer invalidates the content of the queue by changing the serial flow of execution.

The second case arises when an LPDI instruction is located at an odd address. In this situation, the six-byte LPDI instruction is fetched: byte, word, byte, byte, byte, and the queue is not used. The first byte of the following instruction is fetched in one bus cycle as if it had been the target of a program transfer. Word fetching resumes with this instruction's second byte.



FETCH	INSTRUCTION BYTES
1	FIRST TWO BYTES OF "X"
2	THIRD BYTE OF "X" PLUS FIRST BYTE OF "Y", WHICH IS SAVED IN QUEUE
3	FIRST BYTE OF "Y" FROM QUEUE—NO BUS CYCLE
4	LAST TWO BYTES OF "Y"

Figure 3-12. Sequential Instruction Fetching (16-Bit Bus)



FETCH	INSTRUCTION BYTES
1	FIRST (ODD-ADDRESSED) BYTE OF "X" (8-BIT BUS CYCLE)
2	SECOND AND THIRD BYTES OF "X"
3	FIRST AND SECOND BYTES OF "Y".
4	THIRD BYTE OF "Y" PLUS FIRST BYTE OF NEXT INSTRUCTION, WHICH IS SAVED IN QUEUE

Figure 3-13. Instruction Fetching Following a Program Transfer to an Odd Address (16-Bit Bus)

Bus Interface Unit (BIU)

The BIU runs all bus cycles, transferring instructions and data between the IOP and external memory or peripherals. Every bus access is associated with a register tag bit that indicates to the BIU whether the system or I/O space is to be addressed. The BIU outputs the type of bus cycle (instruction fetch from I/O space, data store into system space, etc.) on status lines $\overline{S0}$, $\overline{S1}$, and $\overline{S2}$. An 8288 Bus Controller decodes these lines and provides signals that selectively enable one bus or the other (see Chapter 4 for details).

The BIU further distinguishes between the physical and logical widths of the system and I/O buses. The physical widths of the buses are fixed and are communicated to the BIU during initialization. In the local configuration, both buses must be the same width, either 8 or 16 bits (matching the width of the host CPU bus). In the remote configuration, the IOP system bus must be the same physical width as the bus it shares with the CPU. The width of the IOP's I/O bus, which is local to the 8089, may be selected independently. If any 16-bit peripherals are located in the I/O space, then a 16-bit I/O bus must be used. If only 8-bit devices reside on the I/O bus, then either an 8- or a 16-bit I/O bus may be selected. A 16-bit I/O bus has the advantage of easy accommodation of future 16-bit devices and fewer instruction fetches if channel programs are placed in the I/O space.

For a given DMA transfer, a channel program specifies the logical width of the system and the I/O buses; each channel specifies logical bus widths independently. The logical width of an 8-bit physical bus can only be eight bits. A 16-bit physical bus, however, can be used as either an 8- or 16-bit logical bus. This allows both 8- and 16-bit devices to be accessed over a single 16-bit physical bus. Table 3-1 lists the permissible physical and logical bus widths for both locally and remotely configured IOPs. Logical bus width pertains to DMA transfers only. Instructions are fetched and operands are read and written in bytes or words depending on physical bus width.

In addition to performing transfers, the BIU is responsible for local bus arbitration. In the local configuration, the BIU uses the $\overline{RQ}/\overline{GT}$ (request/grant) line to obtain the bus from the CPU and to return it after a transfer has been performed. In the remote configuration, the BIU uses $\overline{RQ}/\overline{GT}$ to coordinate use of the local I/O bus with another IOP or a local CPU, if present. System bus arbitration in the remote configuration is performed by an 8289 Bus Arbiter that operates invisibly to the IOP. The BIU automatically asserts the $\overline{LOCK}$ (bus lock) signal during execution of a TSL (test and set lock) instruction and, if specified by the channel program, can assert the $\overline{LOCK}$ signal for the duration of a DMA transfer. Section 3.5 contains a complete discussion of bus arbitration.

Table 3-1. Physical/Logical Bus Combinations

Configuration	System Bus Physical:Logical	I/O Bus Physical:Logical
Local	8:8	8:8
	16:8/16	16:8/16
Remote	8:8	8:8
	16:8/16	16:8/16
	16:8/16	8:8
	8:8	16:8/16

Channels

Although the 8089 is a single processor, under most circumstances it is useful to think of it as two independent channels. A channel may perform DMA transfers and may execute channel programs; it also may be idle. This section describes the hardware features that support these operations.

I/O Control

Each channel contains its own I/O control section that governs the operation of the channel during DMA transfers. If the transfer is synchronized, the channel waits for a signal on its DRQ (DMA request) line before performing the next fetch-store sequence in the transfer. If the transfer is to be terminated by an external signal, the channel monitors its EXT (external terminate) line and stops the transfer when this line goes active. Between the fetch and store cycles (when the data is in the IOP) the channel optionally counts,

translates, and scans the data, and may terminate the transfer based on the results of these operations. Each channel also has a SINTR (system interrupt) line that can be activated by software to issue an interrupt request to the CPU.

Registers

Figure 3-14 illustrates the channel register set, and table 3-2 summarizes the uses of each register. Each channel has an independent set of registers; they are not accessible to the other channel. Most of the registers play different roles during channel program execution than in DMA transfers. Channel programs must be careful to save these registers in memory prior to a DMA transfer if their values are needed following the transfer.

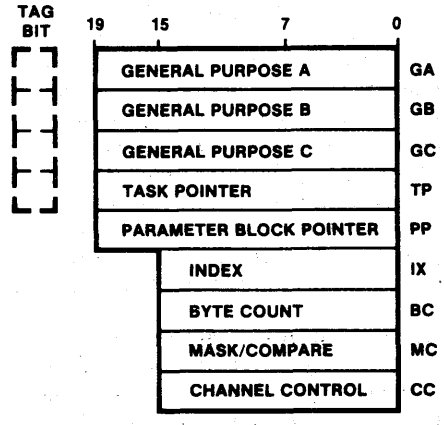


Figure 3-14. Channel Register Set

General Purpose A (GA). A channel program may use GA for a general register or a base register. A general register can be an operand of most IOP instructions; a base register is used to address memory operands (see section 3.8). Before initiating a DMA transfer, the channel program points GA to either the source or destination address of the transfer.

General Purpose B (GB). GB is functionally interchangeable with GA. If GA points to the source of a DMA transfer, then GB points to the destination, and vice versa.

General Purpose C (GC). GC may be used as a general register or a base register during channel program execution. If data is to be translated during a DMA transfer, then the channel program loads GC with the address of the first byte of a translation table before initiating the transfer. GC is not altered by a transfer operation.

Task Pointer (TP). The CCU loads TP from the parameter block when it starts or resumes a channel program. During program execution, the channel automatically updates TP to point to the

Table 3-2. Channel Register Summary

Register	Size	Program Access	System or I/O Pointer	Use by Channel Programs	Use in DMA Transfers
GA	20	Update	Either	General, base	Source/destination pointer
GB	20	Update	Either	General, base	Source/destination pointer
GC	20	Update	Either	General, base	Translate table pointer
TP	20	Update	Either	Procedure return, instruction pointer	Adjusted to reflect cause of termination
PP	20	Reference	System	Base	N/A
IX	16	Update	N/A	General, auto-increment	N/A
BC	16	Update	N/A	General	Byte counter
MC	16	Update	N/A	General, masked compare	Masked compare
CC	16	Update	N/A	Restricted use recommended	Defines transfer options

next instruction to be executed; i.e., TP is used as an instruction pointer or program counter. Program transfer instructions (JMP, CALL, etc.) update TP to cause nonsequential execution. A procedure (subroutine) returns to the calling program by loading TP with an address previously saved by the CALL instruction. The task pointer is fully accessible to channel programs; it can be used as a general register or as a base register. Such use is not recommended, however, as it can make programs very difficult to understand.

Parameter Block Pointer (PP). The CCU loads this register with the address of the parameter block before it starts a channel program. The register cannot be altered by a channel program, but is very useful as a base register for accessing data in the parameter block. PP is not used during DMA transfers.

Index (IX). IX may be used as a general register during channel program execution. It also may be used as an index register to address memory operands (the address of the operand is computed by adding the content of IX to the content of a base register). When specified as an index register, IX may be optionally auto-incremented as the last step in the instruction to provide a convenient means of "stepping" through arrays or strings. IX is not used in DMA transfers.

Byte Count (BC). BC may be used as a general register during channel program execution. If DMA is to be terminated when a specific number of bytes has been transferred, BC should be loaded with the desired byte count before initiating the transfer. During DMA, BC is decremented for each byte transferred, whether byte count termination has been selected or not. If BC reaches zero, the transfer is stopped only if byte count termination has been specified. If byte count termination has not been selected, BC "wraps around" from 0H to FFFFH and continues to be decremented.

Mask/Compare (MC). A channel program may use MC for a general register. This register also may be used in either a channel program or in a DMA transfer to perform a masked compare of a byte value. To use MC in this way, the program loads a compare value in the low-order eight bits of the register and a mask value in the upper eight bits (see figure 3-15). A "1" in a mask bit *selects* the bit in the corresponding position in the compare value; a "0" in a mask bit *masks* the corresponding bit in the compare value. In figure 3-15, a value compared with MC will be considered equal if its low-order five bits contain the value 00100; the upper three bits may contain any value since they are masked out of the comparison.

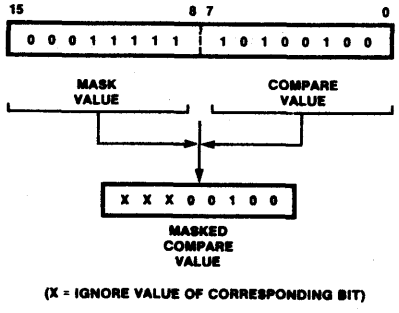


Figure 3-15. Mask/Compare Register

Channel Control (CC). The content of the channel control register governs a DMA transfer (see figure 3-16). A channel program loads this register with appropriate values before beginning the transfer operation; section 3.4 covers the encoding of each field in detail. Bit 8 (the chain bit) of CC pertains to channel program execution rather than to a DMA transfer. When this bit is zero, the channel program runs at normal priority; when it is one, the priority of the program is raised to the same level as DMA (priorities are covered later in this section). Although a channel program may use CC as a general register, such use is not recommended because of the side effects on the chain bit and thus on the priority of the channel program. Channel programs should restrict their use of CC to loading control values in preparation for a DMA transfer, setting and clearing the chain bit, and storing the register.

Program Status Word (PSW)

Each channel maintains its own program status word (PSW) as shown in figure 3-17. Channel programs do not have access to the PSW. The PSW records the state of the the channel so that channel operation may be suspended and then resumed later. When the CPU issues a "suspend" command, the channel saves the PSW, task pointer, and task pointer tag bit in the first four bytes of the channel's parameter block as shown in figure 3-18. Upon receipt of a subsequent

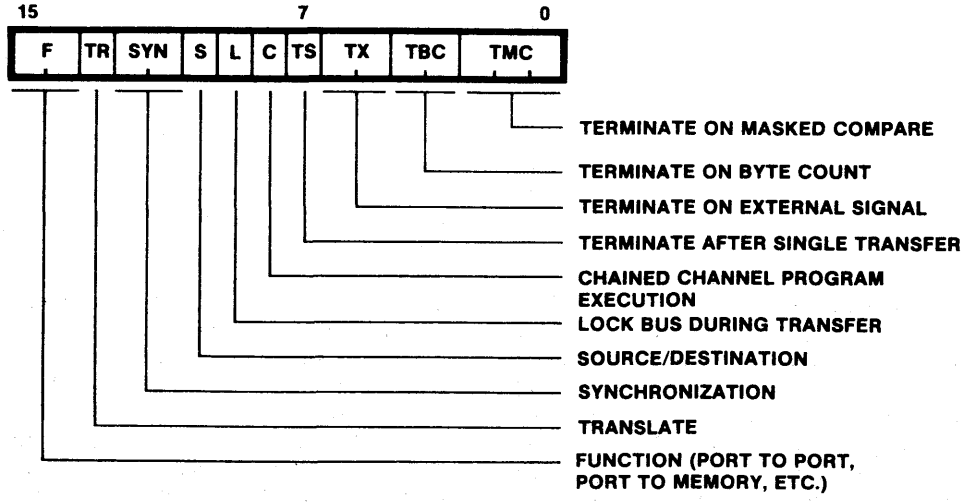


Figure 3-16. Channel Control Register

"resume" command, the PSW, TP, and TP tag bit are restored from the parameter block save area and execution resumes.

Two conditions override the normal channel priority mechanism. If one channel is performing DMA (priority 1) and the channel receives a channel attention (priority 2), the channel attention is serviced at the end of the current DMA transfer cycle. This override prevents a synchronized DMA transfers from "shutting out" a channel attention. DMA terminations and chained channel programs postpone recognition of a CA on the *other* channel; the CA is latched, however, and is serviced as soon as priorities permit.

The IOP's $\overline{\text{LOCK}}$ (bus lock) signal also supersedes channel switching. A running channel will not relinquish control of the processor while $\overline{\text{LOCK}}$ is active, regardless of the priorities of the activities on the two channels. This is consistent with the purpose of the $\overline{\text{LOCK}}$ signal: to guarantee exclusive access to a shared resource in a multiprocessing system. Refer to sections 3.5 and 3.7 for futher information on the $\overline{\text{LOCK}}$ signal and the TSL instruction.

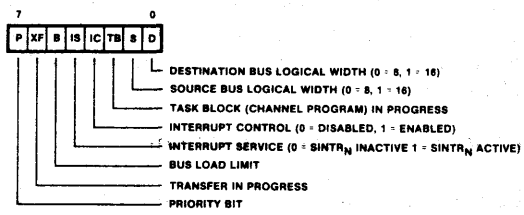


Figure 3-17. Program Status Word

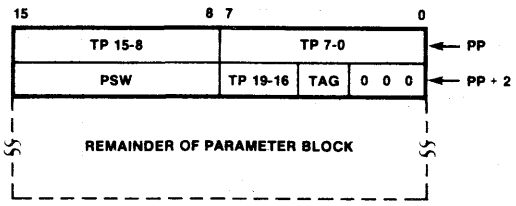


Figure 3-18. Channel State Save Area

Tag Bits

Registers GA, GB, GC, and TP are called pointer registers because they may be used to access, or

point to, addresses in either the system space or the I/O space. The pointer registers may address either memory or I/O devices (IOP instructions do not distinguish between memory and I/O devices since the latter are memory-mapped). The tag bit associated with each register (figure 3-14) determines whether the register points to an address in the system space (tag=0) or the I/O space (tag=1).

The CCU sets or clears TP's tag bit depending on whether the command it receives from the CPU is "start channel program in system space," or "start channel program in I/O space." Channel programs alter the tag bits of GA, GB, GC, and TP by using different instructions for loading the registers. Briefly, a "load pointer" instruction clears a tag bit, a "move" instruction sets a tag bit, and a "move pointer" instruction moves a memory value (either 0 or 1) to a tag bit. Section 3.9 covers these instructions in detail.

If a register points to the system space, all 20 bits are placed on the address lines to allow the full megabyte to be directly addressed. If a register points to the I/O space, the upper four bits of the address lines are undefined; the lower 16 bits are sufficient to access any location in the 64k byte I/O space.

Concurrent Channel Operation

Both channels may be active concurrently, but only one can actually run at a time. At the end of each internal cycle, the CCU lets one channel or the other execute the next internal cycle. No extra overhead is incurred by this channel switching. The basis for making the determination is a priority mechanism built into the IOP. This mechanism recognizes that some kinds of activities (e.g., DMA) are more important than others. Each activity that a channel can perform has a priority that reflects its relative importance (see table 3-3).

Two new activities are introduced in table 3-3. When a DMA transfer terminates, the channel executes a short internal channel program. This DMA termination program adjusts TP so that the user's program resumes at the instruction specified when the transfer was setup (this is discussed in detail in section 3.4). Similarly, when a channel attention is recognized, the channel executes an internal program that examines the CCW and carries out its command. Both of these programs consist of standard 8089 instructions that are fetched from internal ROM. Intel Application Note AP-50, *Debugging Strategies and Considerations for 8089 Systems*, lists the instructions in these programs. Users monitoring the bus during debugging may see operands read or written by the termination or channel attention programs. The instructions themselves, however, will not appear on the bus as they are resident in the chip.

Notice also that, according to table 3-3, a channel program may run at priority 3 or at priority 1.

Table 3-3. Channel Priorities and Interleave Boundaries

Channel Activity	Priority (1 = highest)	Interleave Boundary	
		By DMA	By Instruction
DMA transfer	1	Bus cycle[1]	Bus cycle[1]
DMA termination sequence	1	Internal cycle	None
Channel program (chained)	1	Internal cycle[2]	Instruction
Channel attention sequence	2	Internal cycle	None
Channel program (not chained)	3	Internal cycle[2]	Instruction
Idle	4	Two clocks	Two clocks

[1]DMA is not interleaved while $\overline{\text{LOCK}}$ is active.
[2]Except TSL instruction; see section 3.7.

Channel program priority is determined by the chain bit in the channel control register. If this bit is cleared, the program runs at normal priority (3); if it is set, the program is said to be chained, and it runs at the same priority as DMA. Thus, the chain bit provides a way to raise the priority of a critical channel program.

The CCU lets the channel with the highest priority run. If both channels are running activities with the same priority, the CCU examines the priority bits in the PSWs. If the priority bits are unequal, the channel with the higher value (1) runs. Thus, the priority bit serves as a "tie breaker" when the channels are otherwise at the same priority level. The value of the priority bit in the PSW is loaded from a corresponding bit in the CCW; therefore, the CPU can control which channel will run when the channels are at the same priority level. The priority bit has no effect when the channel priorities are different. If both channels are at the same priority level and if both priority bits are equal, the channels run alternately without any additional overhead.

The CCU switches channels only at certain points called interleave boundaries; these vary according to the type of activity running in each channel and are shown in table 3-3. In table 3-3 and in the following discussion, the terms "channel A" and "channel B" are used to identify two active channels that are bidding for control of an IOP. "Channel A" is the channel that last ran and will run again unless the CCU switches to "channel B." Where the CCU switches from one channel (channel A) to another (channel B) depends on whether channel B is performing DMA or is executing instructions. For this determination, instructions in the internal ROM are considered the same as instructions executed in user-written channel programs (chained or not chained). Table 3-3 shows that a switch from channel A to channel B will occur sooner if channel B is running DMA. DMA, then, interleaves instruction execution at internal cycle boundaries. Since instructions are often composed of several internal cycles, instruction execution on channel A can be suspended by DMA on channel B (when channel A next runs, the instruction is resumed from the point of suspension). DMA on channel A is interleaved by DMA on channel B after any bus cycle (when channel A runs again, the DMA transfer sequence is resumed from the point of suspension). If both channels are executing programs, the interleave boundaries are extended to instruction boundaries: a program on channel B will not run until channel A reaches the end of an instruction. Note that a DMA termination sequence or channel attention sequence on channel A cannot be interleaved by instructions on channel B, regardless of channel B's priority. These internal programs are short, however, and will not delay channel B for long (see Chapter 4 for timing information).

Table 3-4 summarizes the channel switching mechanism with several examples. It is important to remember that channel switching occurs only when both channels are ready to run. In typical applications, one of the channels will be idle much of the time, either because it is waiting to be dispatched by the CPU or because it is waiting for a DMA request in a synchronized transfer. (During a synchronized transfer, the channel is idle between DMA requests; for many peripherals, the channel will spend much more time idling than executing DMA cycles.) The real potential for one channel "shutting out" a priority 1 activity on the other channel is largely limited to unsynchronized DMA transfers and locked transfers (synchronized or unsynchronized). Long, chained channel programs and high-speed synchronized DMA will slow a priority 1 activity on the other channel, but will not shut it out because the channels will alternate (assuming their priority bits are equal). A chained channel program will shut out any lower priority activity on the other channel, including a channel attention. (The channel attention is latched by the IOP, however, so it will execute when the other channel drops to a lower priority.) Chained channel programs should therefore be used with discretion and should be made as short as possible.

3.3 Memory

The 8089 can access memory components located in two different address spaces. The system space, which coincides with the CPU's memory space, may contain up to 1,048,576 bytes. The I/O space, which may either coincide with the CPU's I/O space or be local (private) to the IOP, may contain up to 65,536 bytes. Memory components in the system space should respond to the memory read and write commands issued by the 8288 Bus Controller. Memory components in the I/O space must respond to 8288 I/O read and write commands. Memory in either space may be

Table 3-4. Channel Switching Examples

Channel A (Ran Last)				Channel B			Result
Activity	Chain Bit	Priority Bit	LOCK	Activity	Chain Bit	Priority Bit	
DMA transfer	X	X	Inactive	Idle	X	X	A runs.
DMA transfer	X	X	Inactive	Channel attention	X	X	A runs until end of current transfer cycle; then B runs.
Channel program	X	0	Inactive	Channel program	X	1	B runs.
Channel program	X	0	Inactive	Channel program	X	0	A and B alternate by instruction.
Channel program	1	X	Inactive	Channel program	0	X	A runs.
DMA transfer	X	1	Inactive	Channel program	1	1	B runs one bus or internal cycle following each bus cycle run by A.*
Channel attention	X	X	Inactive	Channel program	1	X	A runs if it has started the sequence; otherwise B runs.
DMA transfer	X	X	Active	Channel attention	X	X	A runs until DMA terminates.
Channel program (TSL instruction)	0	X	Active	DMA transfer	X	X	A completes TSL instruction, LOCK goes inactive and B runs.

*If transfer is synchronized, B also runs when A goes idle between transfer cycles.

implemented like 8086 memory (16-bit words split into even- and odd-addressed 8-bit banks) or 8088 memory (a single 8-bit bank). See Chapter 4 for physical implementation considerations.

Storage Organization

From a software point of view, both 8089 memory spaces are organized as unsegmented arrays of individually addressable 8-bit bytes (figure 3-19). Instructions and data may be stored at any address without regard for alignment (figure 3-20).

The IOP views the system space differently from the 8086 or 8088 with which it typically shares the space. The 8086 and 8088 differentiate between a location's logical (segment and offset) address and its physical (20-bit) address.

The 8089 does not "see" the logically segmented structure of the memory space; it uses its 20-bit pointer registers to access all locations in the system space by their physical addresses. Memory in the 8089 I/O space is treated similarly except that only 16 bits are needed to address any location.

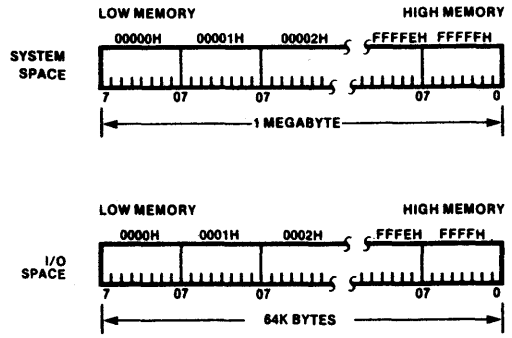


Figure 3-19. Storage Organization

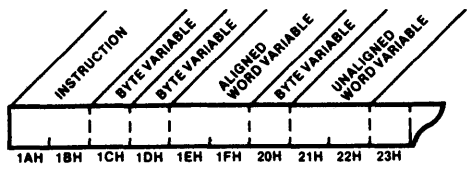


Figure 3-20. Instruction and Variable Storage

Following Intel convention, word data is stored with the most-significant byte in the higher address (see figure 3-21). The 8089 recognizes the doubleword pointer variable used by the 8086 and 8088 (figure 3-22). The lower-addressed word of the pointer contains an offset value, and the higher-addressed word contains a segment base address. Each word is stored conventionally, with the higher-addressed byte containing the most-significant eight bits of the word. The 8089 can convert a doubleword pointer into a 20-bit physical address when it is loaded into a pointer register to address system memory. A special 3-byte variable, called a physical address pointer (figure 3-23), is used to save and restore pointer registers and their associated tag bits.

Dedicated and Reserved Memory Locations

The extreme low and high addresses of the system space are dedicated to specific processor functions or are reserved for use by other Intel hardware and software products; the locations are 0H through 7FH (128 bytes) and FFFF0H through FFFFFH (16 bytes), as shown in figure 3-24. The low addresses are used for part of the 8086/8088 interrupt pointer table. Locations FFFF0H-FFFFBH are used for 8086, 8088 and 8089 startup sequences; the remaining locations are reserved by Intel.

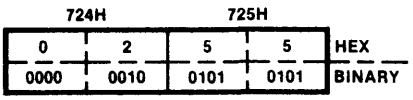
If an IOP is configured locally, its I/O space coincides with the CPU's I/O space, and it must respect the reserved addresses F8H-FFH. The entire I/O space of a remotely-configured IOP may be used without restriction.

Using any dedicated or reserved addresses may inhibit the compatibility of a system with current or future Intel hardware and software products.

Dynamic Relocation

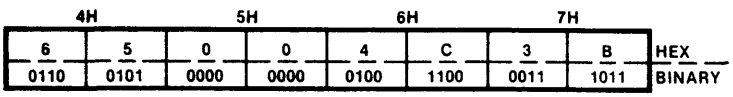
The 8089 is very well-suited to environments in which programs do not occupy static memory locations, but are moved about during execution. Dynamic code relocation allows systems to make efficient use of limited memory resources by transferring programs between external storage and memory, and by combining scattered free areas of memory into larger, more useful, continuous spaces.

IOP channel programs are inherently position-independent, the only restriction being that channel programs that transfer to each other or share data must be moved as a unit. Since the IOP



VALUE OF WORD STORED AT 724H: 5502H

Figure 3-21. Storage of Word Variables



VALUE OF DOUBLEWORD POINTER STORED AT 4H:
SEGMENT BASE ADDRESS: 3B4CH
OFFSET: 65H

Figure 3-22. Storage of Doubleword Pointer Variables

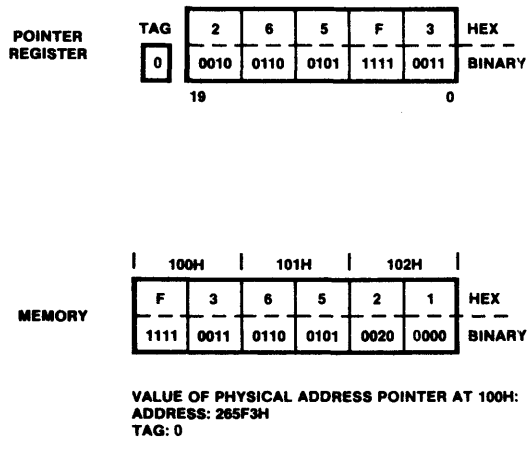


Figure 3-23. Storage of Physical Address Pointer Variables

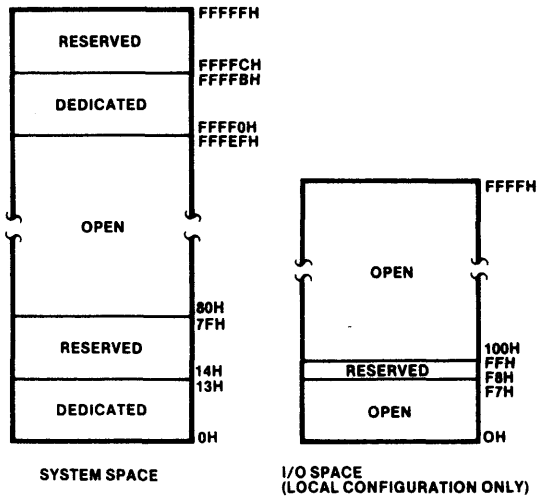


Figure 3-24. Reserved Memory Locations

receives the address of a channel program and its associated parameter block when it is dispatched by the CPU, the location of these blocks is immaterial and can change from one dispatch to the next. (Note, however, that the channel control block cannot be moved without reinitializing the IOP.) Typically, then, the CPU would direct the movement of IOP channel programs and parameter blocks. These blocks, of course, cannot be moved while they are in use.

While the CPU may be in charge of relocation, the IOP is an excellent vehicle for performing the actual transfer of channel programs, parameter blocks, and CPU programs as well. A very simple channel program can transfer code between memory locations by DMA much faster than the equivalent CPU instructions, and transfers between disk and memory also can be performed more efficiently.

Memory Access

Memory accesses are always performed using a pointer register and its associated tag bit. The tag bit indicates whether the access is to the system space (tag=0) or the I/O space (tag=1). The pointer register contains the base address of the location; i.e., the pointer register is used as a base register. Only the low-order 16 bits of the pointer register are used for I/O space locations; all 20 bits are used for system space addresses. Different types of memory accesses use base registers as shown in table 3-5. The 8089 addressing modes allow the base address of a memory operand to be modified by other registers and constant values to yield the effective address of the operand (see section 3.8).

Notice that table 3-5 indicates that memory operands may be addressed using register PP in addition to GA, GB, and GC. PP is maintained by the IOP and can neither be read nor written by a channel program; it can be used, however, to access data in the parameter block. PP has no associated tag bit; a reference to it implies the system space, where a parameter block always resides.

Table 3-5. Base Register Use in Memory Access

Memory Access	Base Register
Instruction Fetch	TP
DMA Source	GA or GB[1]
DMA Destination	GA or GB[1]
DMA Translate Table	GC
Memory Operand	GA or GB or GC or PP[2]

[1]As specified in CC register
[2]As specified in instruction

The IOP is told the physical widths of the system and I/O buses when it is initialized. If a bus is eight bits wide, the IOP accesses memory on this bus like an 8088. Instruction fetches and operand reads and writes are performed one byte at a time; one bus cycle is run for each memory access. Word operands are accessed in two cycles, completely transparent to software. Instruction fetches are made as needed, and the instruction stream is not queued.

The IOP accesses memory on a 16-bit bus like an 8086. As mentioned in the previous section, the instruction stream is generally fetched in words from even addresses with the second byte held in the one-byte queue. If a word operand is aligned (i.e., located at an even address), the 8089 will access it in a single 16-bit bus cycle. If a word operand is unaligned (i.e., located at an odd address), the word will be accessed in two consecutive 8-bit bus cycles. Byte operands are always accessed in 8-bit bus cycles.

For memory on 16-bit buses, performance is improved and bus contention is reduced if word operands are stored at even addresses. The instruction queue tends to reduce the effect of alignment on instructions fetched on a 16-bit bus. In tight loops, performance can be increased by word-aligning transfer targets.

Notice that the correct operation of a program is completely independent of memory bus width. A channel program written for one system that uses an 8-bit memory bus will execute without modification if the bus is increased to 16 bits. It is good practice, though, to write all programs as though they are to run on 16-bit systems; i.e., to align word operands. Such programs will then make optimal use of the bus in whatever system they are run.

3.4 Input/Output

The 8089 combines the programmed I/O capabilities of a CPU with the high-speed block transfer facility of a DMA controller. It also provides additional features (e.g., compare and translate during DMA) and is more flexible than a typical CPU or DMA controller. The 8089 transfers data from a source address to a destination address. Whether the component mapped into a given address is actually memory or I/O is immaterial. All addresses in both the system and I/O spaces are equally accessible, and transfers may be made between the two spaces as well as within either address space.

Programmed I/O

A channel program performs I/O similar to the way a CPU communicates with memory-mapped I/O devices. Memory reference instructions perform the transfer rather than "dedicated" I/O instructions, such as the 8086/8088 IN and OUT instructions. Programmed I/O is typically used to prepare a device controller for a DMA transfer and to obtain status/result information from the controller following termination of the transfer. It may be used, however, with any device whose transfer rate does not require DMA.

I/O Instructions

Since the 8089 does not distinguish between memory components and I/O devices, any instruction that accepts a byte or word memory operand can be used to access an I/O device. Most memory reference instructions take a source operand or a destination operand, or both. The instructions generally obtain data from the source operand, operate on the data, and then place the result of the operation in the destination operand. Therefore, when a source operand refers to an address where an I/O device is located, data is input from the device. Similarly, when a destination operand refers to an I/O device address, data is output to the device.

Most I/O device controllers have one or more internal registers that accept commands and supply status or result information. Working with these registers typically involves:

- reading or writing the entire register;
- setting or clearing some bits in a register while leaving others alone; or
- testing a single bit in a register.

Table 3-6 shows some of the 8089 instructions that are useful for performing these kinds of operations. Section 3.7 covers the 8089 instruction set in detail.

Table 3-6. Memory Reference Instructions Used for I/O

Instruction	Effect on I/O Device
MOV/MOVB	Read or write word/byte
AND/ANDB	Clear multiple bits in word/byte
OR/ORB	Set multiple bits in word/byte
CLR	Clear single bit (in byte)
SET	Set single bit (in byte)
JBT	Read (byte) and jump if single bit =1
JNBT	Read (byte) and jump if single bit =0

Device Addressing

Since memory reference instructions are used to perform programmed I/O, device addressing is very similar to memory addressing. An operand that refers to an I/O device always specifies one of the pointer registers GA, GB, or GC (PP is legal, but an I/O device would not normally be mapped into a parameter block). The base address of the device is taken from the specified pointer register. Any of the memory addressing modes (see section 3.8) may be used to modify the base address to produce the effective (actual) address of the device. The pointer register's tag bit locates the device in the system space (tag=0) or in the I/O space (tag=1). If the device is in the I/O space, only the low-order 16 bits of the pointer register are used for the base address; all 20 bits are used for a system space address. The IOP's system and I/O spaces are fully compatible with the corresponding address spaces of the other 8086 family processors.

I/O Bus Transfers

Table 3-7 shows the number of bus cycles the IOP runs for all combinations of bus size, transfer size (byte or word), and transfer address (even or odd). Bus width refers to the physical bus implementation; the instruction mnemonic determines whether a byte or a word is transferred.

Both 8- and 16-bit devices may reside on a 16-bit bus. All 16-bit devices should be located at even addresses so that transfers will be performed in one bus cycle. The 8-bit devices on a 16-bit bus may be located at odd or even addresses. The internal registers in an 8-bit device on a 16-bit bus must be assigned all-odd or all-even addresses that are two bytes apart (e.g., 1H, 3H, 5H, or 2H, 4H, 6H). All 8-bit peripherals should be referenced with byte instructions, and 16-bit devices should be referenced with word instructions. Odd-addressed 8-bit devices must be able to transfer data on the upper eight bits of the 16-bit physical data bus.

Only 8-bit devices should be connected to an 8-bit bus, and these should only be referenced with byte instructions. An 8-bit device on an 8-bit bus may be located at an odd or even address, and its internal registers may be assigned consecutive addresses (e.g., 1H, 2H, 3H). Assigning all-odd or all-even addresses, however, will simplify conversion to a 16-bit bus at a later date.

Table 3-7. Programmed I/O Bus Transfers

Bus Width:	8				16			
Instruction:	byte		word*		byte		word	
Device Address:	even	odd	even	odd	even	odd	even	odd*
Bus Cycles:	1	1	2	2	1	1	1	2

* not normally used

DMA Transfers

In addition to byte- and word-oriented programmed I/O, the 8089 can transfer blocks of data by direct memory access. A block may be transferred between any two addresses; memory-to-memory transfers are performed as easily as memory-to-port, port-to-memory or port-to-port exchanges. There is no limitation on the size of the block that can be transferred except that the block cannot exceed 64k bytes if byte count termination is used. A channel program typically prepares for a DMA transfer by writing commands to a device controller and initializing channel registers that are used during the transfer. No instructions are executed during the transfer, however, and very high throughput speeds can be achieved.

Preparing the Device Controller

Most controllers that can peform DMA transfers are quite flexible in that they can perform several different types of operations. For example, an 8271 Floppy Disk Controller can read a sector, write a sector, seek to track 0, etc. The controller typically has one or more internal registers that are "programmed" to perform a given operation. Often, certain registers will contain status information that can be read to determine if the controller is busy, if it has detected an error, etc.

An 8089 channel program views these device registers as a series of memory locations. The channel program typically places the device's base address in a pointer register and uses programmed I/O to communicate with the registers.

Some controllers start a DMA transfer immediately upon receiving the last of a series of parameters. If this type of controller is being used, the channel program instruction that sends the last parameter should *follow* the 8089 XFER instruction. (The XFER instruction places the channel in DMA mode after the next instruction; this is explained in more detail later in this section.)

Preparing the Channel

For a channel to perform a DMA transfer, it must be provided with information that describes the operation. The channel program provides this information by loading values into channel registers and, in one case, by executing a special instruction (see table 3-8).

Source and Destination Pointers. One register is loaded to point to the transfer source; the other points to the destination. A bit in the channel control register is set to indicate which register is the source pointer. If a register is pointed at a memory location, it should contain the address where the transfer is to begin — i.e., the lowest address in the buffer. The channel automatically increments a memory pointer as the transfer proceeds. If the tag bit selects the I/O space, the upper four bits of the register are ignored; if the tag selects the system space, all 20 bits are used. The source and destination may be located in the same or in different address spaces.

Translate Table Pointer. If the data is to be translated as it is transferred, GC should be pointed at the first (lowest-addressed) byte in a 256-byte translation table. The table may be located in either the system or I/O space, and GC

Table 3-8. DMA Transfer Control Information

Information	Register or Instruction	Required or Optional
Source Pointer	GA or GB	Required
Destination Pointer	GA or GB	Required
Translate Table Pointer	GC	Optional
Byte Count	BC	Optional
Mask/Compare Values	MC	Optional
Logical Bus Width	WID	Optional*
Channel Control	CC	Required

*Must be executed once following processor RESET.

should be loaded by an instruction that sets or clears its tag bit as appropriate. The translate operation is only defined for byte data; source and destination logical bus widths must both be set to eight bits.

The channel translates a byte by treating it as an unsigned 8-bit binary number. This number is added to the content of register GC to form a memory address; GC is not altered by the operation. If GC points to the I/O space, its upper four bits are ignored in the operation. The byte at this address (which is in the translate table) is then fetched from memory, replacing the source byte. Figure 3-25 illustrates the translate process.

Byte Count. If the transfer is to be terminated on byte count— i.e., after a specific number of bytes have been transferred—the desired count should be loaded into register BC as an unsigned 16-bit number. The channel decrements BC as the transfer proceeds, whether or not byte count termination has been specified. There are cases (discussed later in this section) where the difference between BC's value before and after the transfer does not accurately reflect the number of bytes transferred to the destination.

Mask/Compare Values. If the transfer is to be terminated when a byte (possibly translated) is found equal or unequal to a search value, MC should be loaded as described in section 3.2. MC is not altered during the transfer. Normally, the logical destination bus width is set to eight bits when transferred data is being compared. If the logical destination width is 16 bits, only the low-order byte of each word is compared.

Logical Bus Width. The 8089 WID (logical bus width) instruction is used to set the logical width of the source and destination buses for a DMA transfer. Any bus whose physical width is eight bits can only have a logical width of eight bits. A 16-bit physical bus, however, can have a logical width of 8 or 16 bits; i.e., it can be used as either an 8-bit or 16-bit bus in any given transfer. Logical bus widths are set independently for each channel.

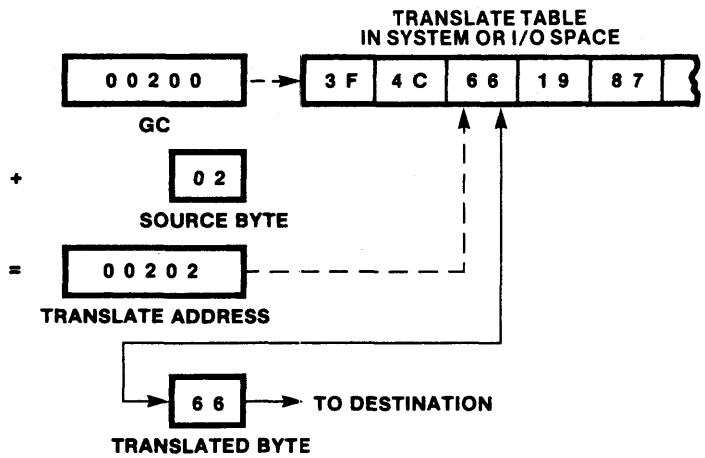


Figure 3-25. Translate Operation

For a transfer to or from an I/O device on a 16-bit physical bus, the logical bus width should be set equal to the peripheral's width; i.e., 8 or 16 bits. Transfers to or from 16-bit memory will run at maximum speed if the logical bus width is set to 16 since the channel will fetch/store words. In the following cases, however, the logical width should be set to 8:

- the data is being translated,
- the data is being compared under mask, and the 16-bit memory is the destination of the transfer.

The WID instruction sets both logical widths and remains in effect until another WID instruction is executed. Following processor reset, the settings of the logical bus widths are unpredictable. Therefore, the WID instruction must be executed before the first DMA transfer.

Channel Control. The 16 bits of the CC register are divided into 10 fields that specify how the DMA transfer is to be executed (see figure 3-26). A channel program typically sets these fields by loading a word into the register.

The *function field* (bits 15-14) identifies the source and destination as memory or ports (I/O devices). During the transfer, the channel increments source/destination pointer registers that refer to memory so that the data will be placed in successive locations. Pointers that refer to I/O devices remain constant throughout the transfer.

The *translate field* (bit 13) controls data translation. If it is set, each incoming byte is translated using the table pointed to by register GC. Translate is defined only for byte transfers; the destination bus must have a logical width of eight.

The *synchronization field* (bits 12-11) specifies how the transfer is to be synchronized. Unsynchronized ("free running") transfers are typically used in memory-to-memory moves. The channel begins the next transfer cycle immediately upon completion of the current cycle (assuming it has the bus). Slow memories, which cannot run as fast as the channel, can extend bus cycles by signaling "not ready" to the 8284 Clock Generator, which will insert wait states into the bus cycle. A similar technique may be used with peripherals whose speed exceeds the channel's ability to execute a synchronized transfer: in effect, the peripheral synchronizes the transfer through the use of wait states. Chapter 4 discusses synchronization in more detail.

Source synchronization is typically selected when the source is an I/O device and the destination is memory. The I/O device starts the next transfer cycle by activating the channel's DRQ (DMA request) line. The channel then runs one transfer cycle and waits for the next DRQ.

Destination synchronization is most often used when the source is memory and the destination is an I/O device. Again, the I/O device controls the transfer frequency by signaling on DRQ when it is ready to receive the next byte or word.

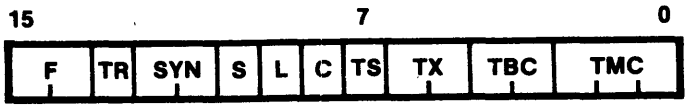
The *source field* (bit 10) identifies register GA or GB as the source pointer (and the other as the destination pointer).

The *lock field* (bit 9) may be used to instruct the channel to assert the processor's bus lock ($\overline{\text{LOCK}}$) signal during the transfer. In a source-synchronized transfer, $\overline{\text{LOCK}}$ is active from the time the first DMA request is received until the channel enters the termination sequence. In a destination-synchronized transfer $\overline{\text{LOCK}}$ is active from the first fetch (which precedes the first DMA request) until the channel enters the termination sequence.

The *chain field* (bit 8) is not used during the transfer. As discussed previously, setting this bit raises channel program execution to priority level 1.

The *terminate on single transfer field* (bit 7) can be used to cause the channel to run one complete transfer cycle only—i.e., to transfer one byte or word and immediately resume channel program execution. When single transfer is specified, any other termination conditions are ignored. Single transfer termination can be used with low-speed devices, such as keyboards and communication lines, to translate and/or compare one byte as it transferred.

The *three low-order fields* in register CC instruct the channel when to terminate the transfer, assuming that single transfer has not been selected. Three termination conditions may be specified singly or in combination.



F	FUNCTION
00	PORT TO PORT
01	MEMORY TO PORT
10	PORT TO MEMORY
11	MEMORY TO MEMORY

TR	TRANSLATE
0	NO TRANSLATE
1	TRANSLATE

SYN	SYNCHRONIZATION
00	NO SYNCHRONIZATION
01	SYNCHRONIZE ON SOURCE
10	SYNCHRONIZE ON DESTINATION
11	RESERVED BY INTEL

S	SOURCE
0	GA POINTS TO SOURCE
1	GB POINTS TO SOURCE

L	LOCK
0	NO LOCK
1	ACTUATE LOCK DURING TRANSFER

C	CHAIN
0	NO CHAINING
1	CHAINED: RAISE TB TO PRIORITY 1

TS	TERMINATE ON SINGLE TRANSFER
0	NO SINGLE TRANSFER TERMINATION
1	TERMINATE AFTER SINGLE TRANSFER

TX	TERMINATE ON EXTERNAL SIGNAL
00	NO EXTERNAL TERMINATION
01	TERMINATE ON EXT ACTIVE; OFFSET = 0
10	TERMINATE ON EXT ACTIVE; OFFSET = 4
11	TERMINATE ON EXT ACTIVE; OFFSET = 8

TBC	TERMINATE ON BYTE COUNT
00	NO BYTE COUNT TERMINATION
01	TERMINATE ON BC = 0; OFFSET = 0
10	TERMINATE ON BC = 0; OFFSET = 4
11	TERMINATE ON BC = 0; OFFSET = 8

TMC	TERMINATE ON MASKED COMPARE
000	NO MASK/COMPARE TERMINATION
001	TERMINATE ON MATCH; OFFSET = 0
010	TERMINATE ON MATCH; OFFSET = 4
011	TERMINATE ON MATCH; OFFSET = 8
100	(NO EFFECT)
101	TERMINATE ON NON-MATCH; OFFSET = 0
110	TERMINATE ON NON-MATCH; OFFSET = 4
111	TERMINATE ON NON-MATCH; OFFSET = 8

Figure 3-26. Channel Control Register Fields

External termination allows an I/O device (typically, the one that is synchronizing the transfer) to stop the transfer by activating the channel's EXT (external terminate) line. If byte count termination is selected, the channel will stop when BC=0. If masked compare termination is specified, the channel will stop the transfer when a byte is found that is equal or unequal (two options are available) to the low-order byte in MC as masked by MC's high-order byte. The byte that stops the termination is transferred. If translate has been specified, the translated byte is compared.

When a DMA transfer ends, the channel adds a value called the termination offset to the task pointer and resumes channel program execution at that point in the program. The termination offset may assume a value of 0, 4, or 8. Single transfer termination always results in a termination offset of 0. Figure 3-27 shows how the termination offsets can be used as indices into a three-element "jump table" that identifies the condition that caused the termination.

As an example of using the jump table, consider a case in which a transfer is to terminate when 80 bytes have been transferred or a linefeed character is detected, whichever occurs first. The program would load 80H into BC and 000AH into MC (ASCII line feed, no bits masked). The channel program could assign byte count termination an offset of 0 and masked compare termination an offset of 4. If the transfer is terminated by byte count (no linefeed is found), the instruction at location TP+0 will be executed first after the termination. If the linefeed is found before the byte count expires, the instruction at TP+4 will be executed first. The LJMP (long unconditional jump, see section 3.7) instruction is four bytes long and can be placed at TP+0 and TP+4 to cause the channel program to jump to a different routine, depending on how the transfer terminates.

If the transfer can only terminate in one way and that condition is assigned an offset of 0, there is no need for the jump table. Code which is to be unconditionally executed when the transfer ends can immediately follow the instruction after XFER. This is also the case when single transfer is specified (execution always resumes at TP+0).

It is possible, however, for two, or even three, termination conditions to arise at the same time. In

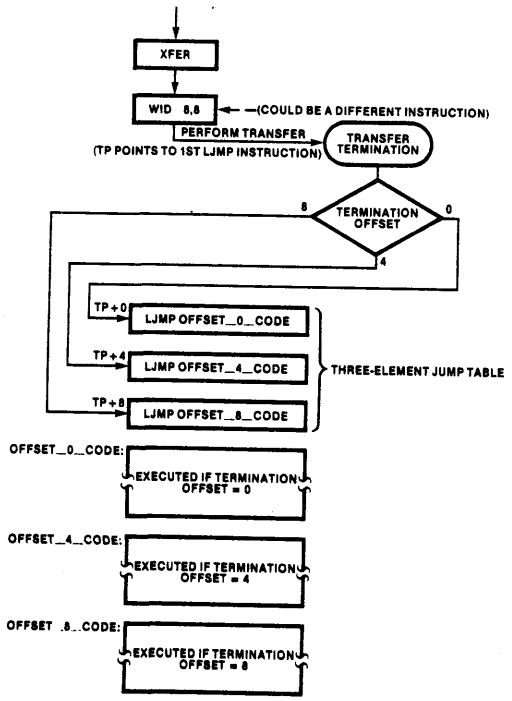


Figure 3-27. Termination Jump Table

the preceding example, this would occur if the 80th character were a linefeed. When multiple terminations occur simultaneously, the channel indicates that termination resulted from the condition with the largest offset value. In the preceding example, if byte count and search termination occur at the same time, the channel program resumes at TP+4.

Beginning the Transfer

The 8089 XFER (transfer) instruction puts the channel into DMA transfer mode after the *following instruction* has been executed. This technique gives the channel time to set itself up when it is used with device controllers, such as the 8271 Floppy Disk Controller, that begin transferring immediately upon receipt of the last in a series of parameters or commands. If the transfer is to or from such a device, the last parameter should be sent to the device after the XFER instruction. If this type of device is not being used, the instruction following XFER would

typically send a "start" command to the controller. If a memory-to-memory transfer is being made, any instruction may follow XFER except one that alters GA, GB, or CC. The HLT instruction should normally not be coded after the XFER; doing so clears the channel's BUSY flag, but allows the DMA transfer to proceed.

DMA Transfer Cycle

A DMA transfer cycle is illustrated in figure 3-28; a complete transfer is a series of these cycles run until a termination condition is encountered. The figure is deliberately simplified to explain the general operation of a DMA transfer; in particular, the updating of the source and destination pointers (GA and GB) can be more complex than the figure indicates. Notice that it is possible to start an unending transfer by not specifying a termination condition in CC or by specifying a condition that never occurs; it is the programmer's responsibility to ensure that the transfer eventually stops.

If the transfer is source-synchronized, the channel waits until the synchronizing device activates the channel's DRQ line. The other channel is free to run during this idle period. The channel fetches a byte or a word, depending on the source address (contained in GA or GB) and the logical bus width. Table 3-9 shows how a channel performs the fetch/store sequence for all combinations of addresses and bus widths. If the destination is on a 16-bit logical bus and the source is on an 8-bit logical bus, and the transfer is to an even address, the channel fetches a second byte and assembles a word internally. During each fetch, the channel decrements BC according to whether a byte or word is obtained. Thus BC always indicates the number of bytes fetched.

The channel samples its EXT line after every bus cycle in the transfer. If EXT is recognized after the first of two scheduled fetches, the second fetch is not run. After the fetch sequence has been completed, the channel translates the data if this option is specified in CC.

If a word has been fetched or assembled, and bytes are to be stored (destination bus is eight bits or transfer is to an odd address), the channel disassembles the word into two bytes. If the transfer is destination-synchronized (only one type of synchronization may be specified for a given transfer), the channel waits for DRQ before running a store cycle. It stores a word or the lower-addressed byte (which may be the only byte or the first of two bytes). Table 3-9 shows the possible combinations of even/odd addresses and logical bus widths that define the store cycle. Whenever stores are to memory on a 16-bit logical bus, the channel stores words, except that bytes may be stored on the first and last cycles.

Table 3-9. DMA Transfer Assembly/Disassembly

Address (Source→ Destination)	Logical Bus Width (Source→Destination)			
	8→8	8→16	16→8	16→16
EVEN→EVEN	B→B	B/B→W	W→B/B	W→W
EVEN→ODD	B→B	B→B	W→B/B	W→B/B
ODD→EVEN	B→B	B/B→W	B→B	B/B→W
ODD→ODD	B→B	B→B	B→B	B→B

B= Byte Fetched or Stored in 1 Bus Cycle
W= Word Fetched or Stored in 1 Bus Cycle
B/B= 2 Bytes Fetched or Stored in 2 Bus Cycles

The channel samples EXT again after the first store cycle and, if it is active, the channel prevents the second store cycle from running. If specified in the CC register, the low-order byte is compared to the value in MC. A "hit" on the comparison (equal or unequal, as indicated in CC) also prevents the second of two scheduled store cycles from running. In both of these cases, one byte has been "overfetched," and this is reflected in BC's value. It would be unusual, however, for a synchronizing device to issue EXT in the midst of a DMA cycle. Note also that EXT is valid only when DRQ is inactive. Chapter 4 covers the timing requirements for these two signals in detail.

GA and GB are updated next. Only memory pointers are incremented; pointers to I/O devices remain constant throughout the transfer.

If any termination condition has occurred during this cycle, the channel stops the transfer. It uses the content of the CC register to assign a value to the termination offset, to reflect the cause of the termination. The channel adds this offset to TP and resumes channel program execution at the location now addressed by TP. This offset will

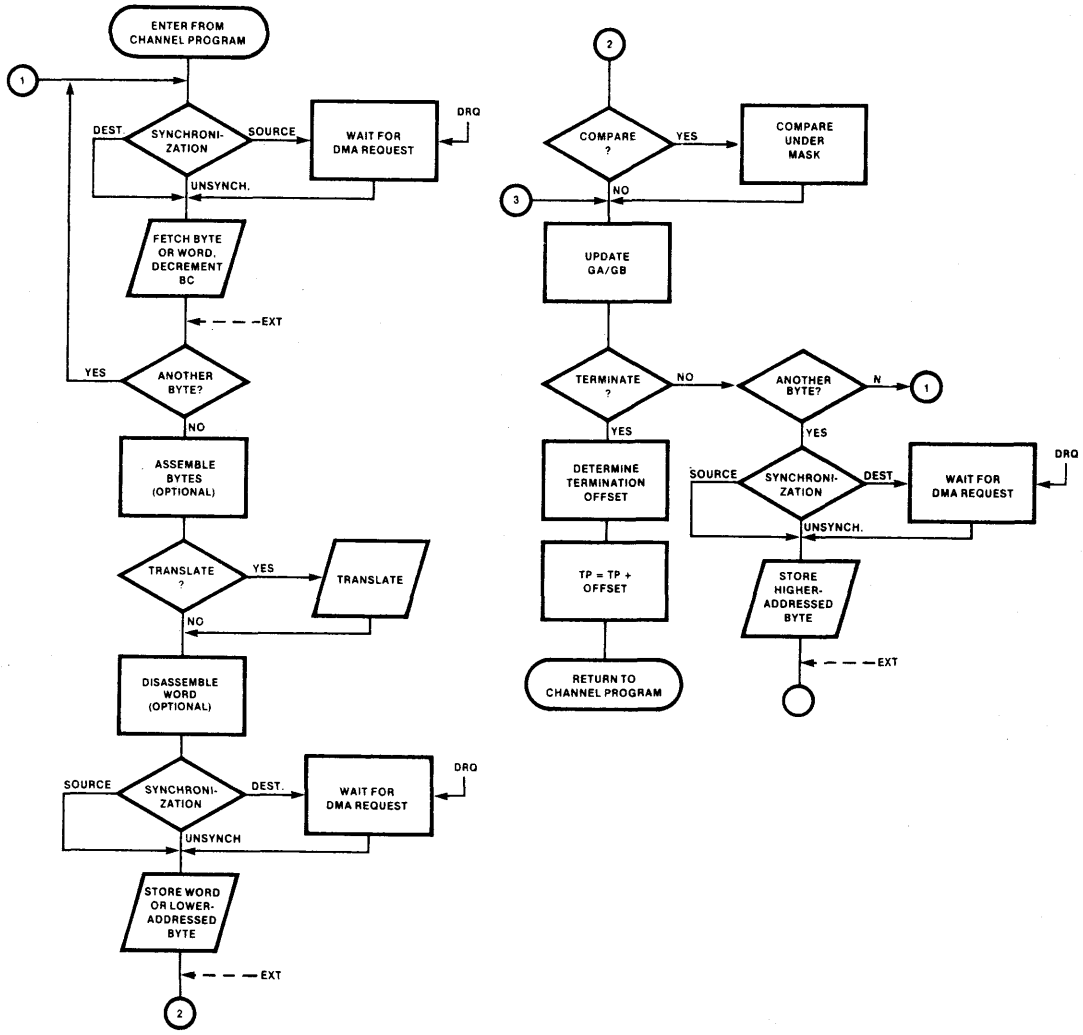


Figure 3-28. Simplified DMA Transfer Flowchart

always be zero, four, or eight bytes past the end of the instruction following the XFER instruction.

If no termination condition is detected and another byte remains to be stored, the channel stores this byte, waiting for DRQ if necessary, and updates the source and destination pointers. After the store, it again checks for termination.

Following the Transfer

A DMA transfer updates register BC, register GA (if it points to memory), and register GB (if it points to memory). If the original contents of these registers are needed following the transfer, the contents should be saved in memory prior to executing the XFER instruction.

A program may determine the address of the last byte stored by a DMA transfer by inspecting the pointer registers as shown in table 3-10. The number of bytes stored is equal to:

last_byte_address — first_byte_address + 1.

For port-to-port transfers, the number of bytes transferred can be determined by subtracting the final value of BC from its original value provided that:

- the original BC > final BC,
- a transfer cycle is not "chopped off" before it completes by a masked compare or external termination.

In general, programs should not use the contents of GA, GB and BC following a transfer except as noted above and in table 3-10. This is because the contents of the registers are affected by numerous conditions, particularly when the transfer is terminated by EXT. In particular, when a program is performing a sequence of transfers, it should reload these registers before each transfer.

3.5 Multiprocessing Features

The 8089 shares the multiprocessing facilities common to the 8086 family of processors. It has on-chip logic for arbitrating the use of the local bus with a CPU or another IOP; system bus arbitration is delegated to an 8289 Bus Arbiter. The 8089's TSL (test and set while locked) instruction enables it to share a resource, such as a buffer, with other processors by means of semaphore (see section 2.5 for a discussion of the use of semaphores to control access to shared resources). Finally, the 8089 can lock the system bus for the duration of a DMA transfer to ensure that the transfer completes without interference from other processors on the bus.

In the remote configuration, the 8089 is electrically compatible with Intel's Multibus™ multimaster bus design. This means that the power and convenience of 8089 I/O processing can be used in 8080- or 8085-based systems that implement the Multibus protocol or a superset of it. This includes single-board computers such as Intel's iSBC 80/20™ and iSBC 80/30™ boards. In addition, the IOP can access other iSBC board products such as memory and communications controllers.

Bus Arbitration

The 8089 shares its system bus with a CPU, and may also share its I/O bus with an IOP or another CPU. Only one processor at a time may drive a bus. When two (or more) processors want to use a shared bus, the system must provide an arbitration mechanism that will grant the bus to one of the processors. This section describes the bus arbitration facilities that may be used with the 8089 and covers their applicability to different IOP configurations.

Table 3-10. Address of Last Byte Stored

Termination	Source	Destination	Synchronization	Last Byte Stored
byte count	memory memory port	memory port memory	any any any	destination pointer[1] source pointer destination pointer
masked compare	memory memory port	memory port memory	any any any	destination pointer source pointer destination pointer
external	memory memory port	memory port memory	unsynchronized destination source	destination pointer source pointer[2] destination pointer

[1]Source pointer may also be used.
[2]If transfer is B/B→W, source pointer must be decremented by 1 to point to last byte transferred.

Request/Grant Line

When an 8089 is directly connected to another 8089, an 8086 or an 8088, the $\overline{RQ}/\overline{GT}$ (request/grant) lines built into all of these processors are used to arbitrate use of a local bus. In the local mode, $\overline{RQ}/\overline{GT}$ is used to control access to both the system and the I/O bus.

As discussed in section 2.6, the CPU's request/grant lines ($\overline{RQ}/\overline{GT0}$ and $\overline{RQ}/\overline{GT1}$) operate as follows:

- an external processor sends a pulse to the CPU to request use of the bus;
- the CPU finishes its current bus cycle, if one is in progress, and sends a pulse to the processor to indicate that it has been granted the bus; and
- when the external processor is finished with the bus, it sends a final pulse to the CPU, to indicate that it is releasing the bus.

The 8089's request/grant circuit can operate in two modes; the mode is selected when the IOP is initialized (see section 3.6). Mode 0 is compatible with the 8086/8088 request/grant circuit and must be specified when the 8089's $\overline{RQ}/\overline{GT}$ line is connected to $\overline{RQ}/\overline{GT0}$ or $\overline{RQ}/\overline{GT1}$ of one of these CPUs. Mode 0 *may* be specified when $\overline{RQ}/\overline{GT}$ of one 8089 is tied to $\overline{RQ}/\overline{GT}$ of another 8089. When mode 0 is used with a CPU, the CPU is designated the master, and the IOP is designated a slave. When mode 0 is used with another IOP, one IOP is the master, and the other is the slave. Master/slave designation also is made at initialization time as discussed in section 3.6. The master has the bus when the system is initialized and keeps the bus until it is requested by the slave. When the slave requests the bus, the master grants it if the master is idle. In this sense, the CPU becomes idle at the end of the current bus cycle. An IOP master, on the other hand, does not become idle until both channels have halted program execution or are waiting for DMA requests. Once granted the bus, the slave (always an IOP) uses it until both channels are idle, and then releases it to the master. In mode 0, the master has no way of requesting the slave to return the bus.

Mode 1 operation of the request/grant lines may only be used to arbitrate use of a private I/O bus between two IOPs. In this case, one IOP is designated the master, and the other is designated the slave. However, the only difference between a master and a slave running in mode 1 is that the master has the bus at initialization time. Both processors may request the bus from each other at any time. The processor that has the bus will grant it to the requester as soon as one of the following occurs on either channel:

- an unchained channel program instruction is completed, or
- a channel goes idle due to a program halt or the completion of a synchronized transfer cycle (the channel waits for a DMA request).

Execution of a chained channel program, a DMA termination sequence, a channel attention sequence, or a synchronized DMA transfer (i.e., a high-priority operation) on either channel prevents the IOP from granting the bus to the requesting IOP.

The handshaking sequence in mode 1 is:

- the requesting processor pulses once on $\overline{RQ}/\overline{GT}$;
- the processor with the bus grants it by pulsing once; and
- if the processor granting the bus wants it back immediately (for example, to fetch the next instruction), it will pulse $\overline{RQ}/\overline{GT}$ again, two clocks after the grant pulse.

The fundamental difference between the two modes is the frequency with which the bus can be switched between the two processors when both are active. In mode 0, the processor that has the bus will tend to keep it for relatively long periods if it is executing a channel program. Mode 1 in effect places unchained channel programs at a lower priority since the processor will give up the bus at the end of the next instruction. Therefore, when both processors are running channel programs or synchronized DMA, they will share the bus more or less equally. When a processor changes to what would typically be considered a higher-priority activity such as chained program execution or DMA termination, it will generally be able to obtain the bus quickly and keep the bus for the duration of the more critical activity.

8289 Bus Arbiter

When an IOP is configured remotely, an 8289 Bus Arbiter is used to control its access to the shared system bus (the CPU also has its own 8289). In a remote cluster of two IOPs or an IOP and a CPU, one 8289 controls access to the system bus for both processors in the cluster. The 8289 has several operating modes; when used with an 8089, the 8289 is usually strapped in its IOB (I/O Peripheral Bus) mode.

The 8289 monitors the IOP's status lines. When these indicate that the IOP needs a cycle on the system bus, and the IOP does not presently have the bus, the 8289 activates a bus request signal. This signal, along with the bus request lines of other 8289s on the same bus, can be routed to an external priority-resolving circuit. At the end of the current bus cycle, this circuit grants the bus to the requesting 8289 with the highest priority. Several different prioritizing techniques may be used; in a typical system, an IOP would have higher bus priority than a CPU. If the 8289 does not obtain the bus for its processor, it makes the bus appear "not ready" as if a slow memory were being accessed. The processor's clock generator responds to the "not ready" condition by inserting wait states into the IOP's bus cycle, thereby extending the cycle until the bus is acquired.

Bus Arbitration for IOP Configurations

When the CPU initializes an IOP, it must inform the IOP whether it is a master or a slave, and which request/grant mode is to be used. This section covers the requirements and options available for each IOP configuration; section 3.6 describes how the information is communicated at initialization time.

Table 3-11 summarizes the bus arbitration requirements and options by IOP configuration. In the local configuration, all bus arbitration is performed by the request/grant lines without additional hardware. One IOP may be connected to each of the CPU's $\overline{RQ}/\overline{GT}$ lines. The IOP connected to $\overline{RQ}/\overline{GT0}$ will obtain the bus if both processors make simultaneous requests.

Since a single IOP in a remote configuration does not use $\overline{RQ}/\overline{GT}$, its mode may be set to 0 or 1 without affect. The single remote IOP, however, must be initialized as a master. If two remote IOPs share an I/O bus, one must be a master and the other a slave; both must be initialized to use the same request/grant mode. Normally, mode 1 will be selected for its improved responsiveness, and the designation of master will be arbitrary. If one IOP must have the I/O bus when the system comes up, it should be initialized as the master.

When a remote IOP shares its I/O bus with a local CPU, it must be a slave and must use request/grant mode 0.

Bus Load Limit

A locally configured IOP effectively has higher bus priority than the CPU since the CPU will grant the bus upon request from the IOP. One or two local IOPs can potentially monopolize the bus at the expense of the CPU. Of course, if the IOP activities are time-critical, this is exactly what should happen. On the other hand, there may be low-priority channel programs that have less demanding performance requirements.

In such cases, the CPU may set a CCW bit called bus load limit to constrain the channel's use of the bus during normal (unchained) channel program

Table 3-11. Bus Arbitration Requirements and Options

IOP	Local		Remote		Remote With Local CPU	
	Master/ Slave	$\overline{RQ}/\overline{GT}$ Mode	Master/ Slave	$\overline{RQ}/\overline{GT}$ Mode	Master/ Slave	$\overline{RQ}/\overline{GT}$ Mode
IOP1	Slave	0	Master	0 or 1	Slave	0
IOP2	Slave	0	Slave	Same as Master	N/A	N/A

execution. When this bit is set, the channel decrements a 7-bit counter from 7F (127) to 0H with each instruction executed. Since the counter is decremented once per clock period, the channel waits a minimum of 128 clock cycles before it executes the next instruction. By forcing the execution time of all instructions to 128 clocks, the use of the bus is reduced to between 3 and 25 percent of the available bus cycles.

Setting the bus load limit effectively enables a CPU to slow the execution of a normal channel program, thus freeing up bus cycles. This is of most use in local configurations, but also may be effective in remote configurations, particularly when channel programs are executed from system memory. Bus load limit has no effect on chained channel programs, DMA transfers, DMA termination, or channel attention sequences.

Bus Lock

Like the 8086 and 8088, the 8089 has a $\overline{\text{LOCK}}$ (bus lock) signal which can be activated by software. The $\overline{\text{LOCK}}$ output is normally connected to the $\overline{\text{LOCK}}$ input of an 8289 Bus Arbiter. When $\overline{\text{LOCK}}$ is active, the bus arbiter will not release the bus to another processor regardless of its priority. A channel automatically locks the bus during execution of the TSL (test and set while locked) instruction and may lock the bus for the duration of a DMA transfer.

If bit 9 of register CC is set, the 8089 activates its $\overline{\text{LOCK}}$ output during a DMA transfer on that channel. If the transfer is synchronized, $\overline{\text{LOCK}}$ is active from the time that the first DRQ is recognized. If the transfer is unsynchronized, $\overline{\text{LOCK}}$ is active throughout the entire transfer (there are no idle periods in an unsynchronized transfer). $\overline{\text{LOCK}}$ goes inactive when the channel begins the DMA termination sequence.

A locked transfer ensures that the transfer will be completed in the shortest possible time and that the transferring channel has exclusive use of the bus. Once the channel obtains the bus and starts a locked transfer, the channel, in effect, becomes the highest-priority processor on that bus.

The 8089 TSL (test and set while locked) instruction can be used to implement a semaphore. (See section 2.5 for a discussion of how a semaphore may be used to control the access of multiple processors to a shared resource.) The instruction activates $\overline{\text{LOCK}}$ and inspects the value of a byte in memory. If the value of the byte is 0H, it is changed (set) to a value specified in the instruction and the following instruction is executed. If the byte does not contain 0H, control is transferred to another location specified in the instruction. The bus is locked from the time the byte is read until it is either written or control is transferred to ensure that another processor does not access the variable after TSL has read it, but before it has updated it (i.e., between bus cycles). The following line of code will repeatedly test a semaphore pointed to by GA until it is found to contain zero:

```
TEST_FLAG: TSL [GA], 0FFH, TEST_FLAG
```

When the semaphore is found to be zero, it is set to FFH and the program continues with the next instruction.

3.6 Processor Control and Monitoring

This section focuses on IOP/CPU interaction, i.e., how the CPU initializes the IOP and subsequently sends commands to channels, and how the channels may interrupt the CPU. It also covers the channels' DMA control signals and the status signals that external devices can use to monitor IOP activities.

Initialization

Before the 8089 channels can be dispatched to perform I/O tasks, the IOP must be initialized. The initialization sequence (figure 3-29) provides the IOP with a definition of the system environment: physical bus widths, request/grant mode, and the location of the channel control block.

The sequence begins when the IOP's RESET line is activated. This halts any operation in progress, but does not affect any registers. Upon the first

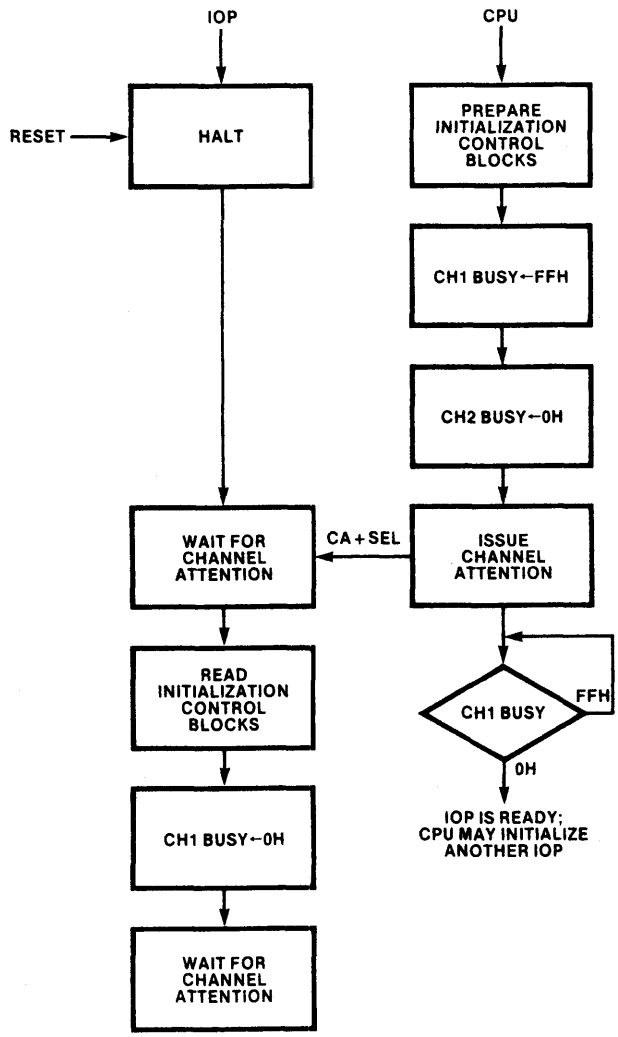


Figure 3-29. Initialization Sequence

RESET after power-up, the content of all IOP registers is undefined. Register contents are preserved if the IOP is subsequently RESET, except that RESET always clears the chain bit in register CC.

The IOP initializes itself by reading information from initialization control blocks located in the system space (see figure 3-30). The three blocks are the SCP (system configuration pointer), SCB (system configuration block) and the CB (channel control block). The CB is normally RAM-based; the SCP and the SCB may be in RAM or ROM. It is the CPU's responsibility to properly setup the control blocks.

The CPU starts the initialization sequence by issuing a channel attention to channel 1 (SEL low) or to channel 2 (SEL high). The CPU typically accesses the channels as two consecutive addresses in its I/O or memory space. An OUT instruction (for an I/O-mapped IOP) or a memory reference instruction (such as MOV) then issues the channel attention.

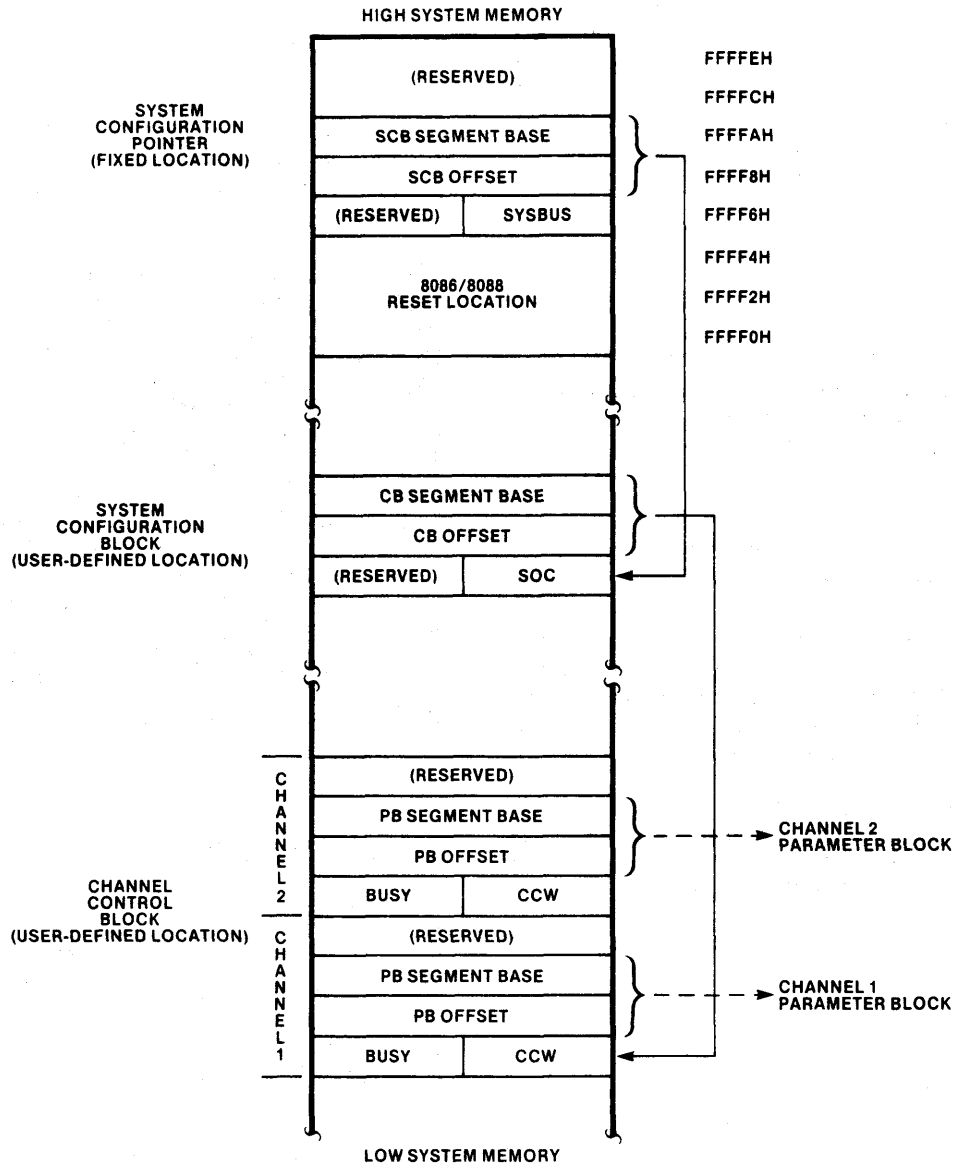


Figure 3-30. Initialization Control Blocks

If channel 1 is selected (SEL=low), the IOP considers itself a master (as discussed in section 3.5). If channel 2 is selected (SEL=high), the IOP operates as a slave. The IOP ignores, and does not latch, any subsequent channel attentions that occur during initialization.

If the IOP is a master, it assumes that it has the bus immediately. If it is a slave, it pulses $\overline{RQ}/\overline{GT}$ to request the bus from the CPU (local configuration) or the other IOP (remote configuration). When the IOP has obtained the bus, it assumes that the system bus is eight bits wide and reads the

SYSBUS field (figure 3-31) from location FFFF6H in system memory. This byte tells the IOP the actual physical width of the system bus; all subsequent accesses take advantage of a 16-bit bus if it is available; i.e., even-addressed words are fetched in single bus cycles. It is therefore advantageous to word-align the control blocks.

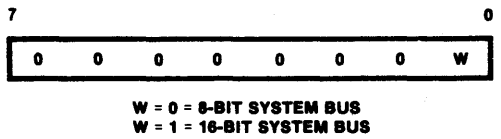


Figure 3-31. SYSBUS Encoding

Next, the IOP reads the SCB address located at FFFF8H. This is a standard doubleword pointer, and the IOP constructs a 20-bit physical address from it by shifting the segment base left four bits and adding the offset word of the pointer.

Having obtained the SCB address, the IOP reads the SOC (system operation command). This byte (see figure 3-32) tells the IOP the request/grant mode and the width of the I/O bus.

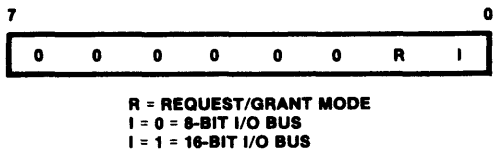


Figure 3-32. SOC Encoding

Then the IOP reads the doubleword pointer to the channel control block, converts the pointer into a 20-bit physical address, and stores it in an internal register. This register is not accessible to channel programs and is only loaded during initialization. The CB, therefore, cannot be moved during execution except by reinitializing the IOP.

After loading the address of the CB, the IOP clears the channel 1 BUSY flag to 0H. The other fields in the CB are used when a channel is dispatched and are not read or altered in the initialization sequence.

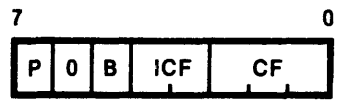
After the CPU has started the initialization sequence, it should monitor channel 1's BUSY flag in the CB to determine when the sequence has been completed. When the BUSY flag has been cleared, the CPU can dispatch either channel. It also can begin the initialization of another IOP. Since each IOP normally has a separate CB, the CPU must allocate the CB and update the pointer in the SCB before initializing the next IOP. Alternatively, multiple SCBs could be employed, each pointing to a different CB area. In this case the CPU would update the pointer in the SCP before initializing the next IOP. It follows from this that in multi-IOP systems, either the SCB or SCP, or both, must be RAM-based. When all IOPs have been initialized, the CPU may use RAM occupied by the SCB for another purpose.

Channel Commands

After initialization, any channel attention is interpreted as a command to channel 1 (SEL=low) or to channel 2 (SEL=high). As discussed in section 3.2, the channel attention, depending on the activities of both channels, may not be recognized immediately. The channel attention is latched, however, so that it will be serviced as soon as priorities allow.

When the channel recognizes the CA, it sets its BUSY flag in the CB to FFH. This does not prevent the CPU from issuing another CA, but provides status information only. In its response to a CA, the channel reads various control fields from system memory. It is the responsibility of the CPU to ensure that the appropriate fields are properly initialized before issuing the CA.

After setting its BUSY flag, the channel reads its CCW from the CB. It examines the command field (see figure 3-33) and executes the command encoded there by the CPU.



CF COMMAND FIELD
000 UPDATE PSW
001 START CHANNEL PROGRAM LOCATED IN I/O SPACE.
010 (RESERVED)
011 START CHANNEL PROGRAM LOCATED IN SYSTEM SPACE.
100 (RESERVED)
101 RESUME SUSPENDED CHANNEL OPERATION
110 SUSPEND CHANNEL OPERATION
111 HALT CHANNEL OPERATION

ICF INTERRUPT CONTROL FIELD
00 IGNORE, NO EFFECT ON INTERRUPTS.
01 REMOVE INTERRUPT REQUEST; INTERRUPT IS ACKNOWLEDGED.
10 ENABLE INTERRUPTS.
11 DISABLE INTERRUPTS.

B BUS LOAD LIMIT
0 NO BUS LOAD LIMIT
1 BUS LOAD LIMIT

P PRIORITY BIT

Figure 3-33. Channel Command Word Encoding

Figure 3-34 illustrates the channel's response to each type of command. Note that if CF contains a reserved value (010 or 100), the channel's response is unpredictable.

The CPU can use the "update PSW" command to alter the bus load limit and priority bits in the PSW (see figure 3-17) without otherwise affecting the channel. This command also allows the CPU to control interrupts originating in the channel; this topic is discussed in more detail later in this section.

The two "start program" commands differ only in their affect on the TP tag bit. If CF=001, the channel sets the tag to 1 to indicate that the program resides in the I/O space. If CF=011, the tag is cleared to 0, and the program is assumed to be in the system space. The channel converts the doubleword parameter block pointer to a 20-bit physical address and loads this into PP. It loads the doubleword task block (channel program) pointer into TP, updates the PSW as specified by the ICF, B and P fields of the CCW and starts the program with the instruction pointed to by TP.

The CPU may suspend a channel operation (either program execution or DMA transfer) by setting CF to 110. The channel saves its state (TP, its tag bit, and PSW) in the first two words of the parameter block (see figure 3-18 for format) and clears its BUSY flag to 0H. Note the following in regard to a suspended operation:

- The content of the doubleword pointer to the beginning of the channel program is replaced by the channel state save data. Therefore, a suspended operation may be resumed, but cannot be started from the beginning without recreating the doubleword pointer.

- TP is the only register saved by this operation. If another channel program is started on this channel, the other registers, including PP, are subject to being overwritten. In general, suspend is used to temporarily halt a channel, not to "interrupt" it with another program. Section 3.10 provides an example of a program that can be used to save another program's registers.

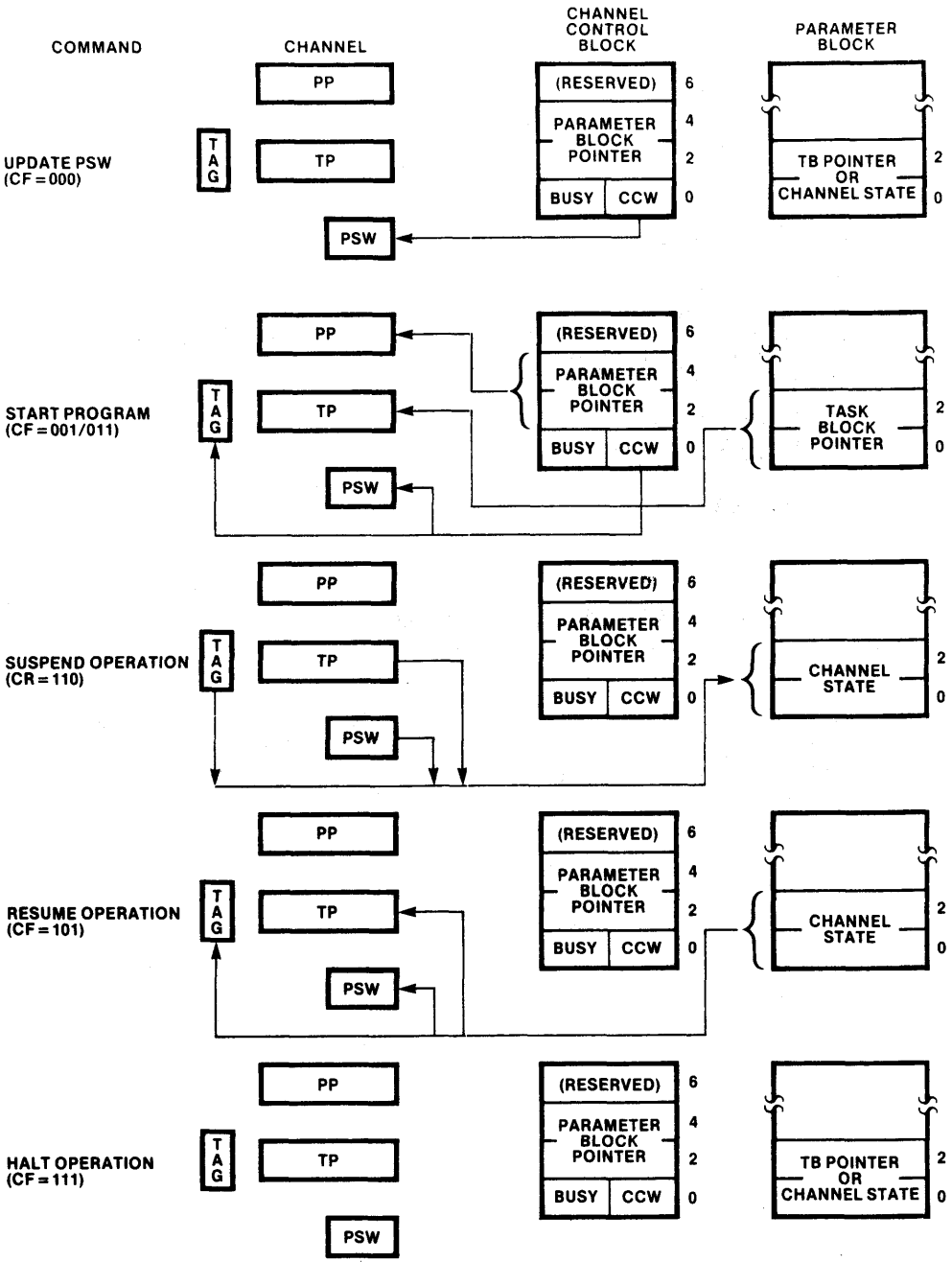


Figure 3-34. Channel Commands

- Suspending a DMA transfer does not affect any I/O devices (an I/O device will act as though the transfer is proceeding). The CPU must provide for conditions that may arise if, for example, a device requests a DMA transfer, but the channel does not acknowledge the request because it has been suspended. Similarly, an I/O device may be in a different condition when the operation is resumed.

A suspended operation may be resumed by setting CF to 101. This command causes the channel to reload TP, its tag bit, and the PSW from the first two words of PB. Resuming an operation that has not been suspended will give unpredictable results since the first two words of PB will not contain the required channel state data. A resume command does not affect any channel registers other than TP.

The CPU may abort a channel operation by issuing a "halt" command (CF=111). The channel clears its BUSY flag to 0H and then idles. Again, the CPU must be prepared for the effect aborting a DMA transfer may have on an I/O device.

DRQ (DMA Request)

The synchronizing device in a DMA transfer uses the DRQ line to indicate when it is ready to send or receive the next byte or word. The channel recognizes a signal on this line only during a DMA transfers, i.e., after the instruction following XFER has been executed and before a termination condition has occurred. The channels have separate DMA request lines (DRQ1 and DRQ2).

EXT (External Terminate)

An external device (typically the synchronizing device) can terminate a DMA transfer by signaling on this line. Each channel has its own external terminate line (EXT1 and EXT2). The channel stops the transfer as soon as the current fetch or store cycle is completed. An external terminate in an unsynchronized transfer could result in a loss of data, although this would not be a typical use of EXT. In a synchronized transfer, the synchronizing device will normally issue EXT instead of DRQ following the last transfer cycle. If EXT is activated during a transfer cycle, a fetched byte may not be stored as explained in section 3.4.

A channel does not recognize EXT if it is not performing a DMA transfer. If EXT1 and EXT2 are activated simultaneously, EXT1 is recognized first.

Interrupts

Each channel has a separate system interrupt line (SINTR1 and SINTR2). A channel program may generate a CPU interrupt request by executing a SINTR instruction. Whether this instruction actually activates the SINTR line, however, depends upon the state of the interrupt control bit (bit 3 of the PSW; see figure 3-17). If this bit is set, interrupts from the channel are enabled, and execution of the SINTR instruction activates SINTR. If the interrupt control bit is cleared, the SINTR instruction has no effect; interrupts from the channel are disabled.

The CPU can alter a channel's interrupt control bit by sending any command to the channel with the value of ICF (interrupt control field) in the CCW set to 10 (enable) or 11 (disable). Thus, the CPU can prevent interrupts from either channel.

Once activated, SINTR remains active until the CPU sends a channel command with ICF set to 01 (interrupt acknowledge). When the channel receives this command, it clears the interrupt service bit in the PSW (figure 3-17) and removes the interrupt request. Disabling interrupts also clears the interrupt service bit and lowers SINTR.

Status Lines

The IOP emits signals on the $\overline{S0}$-$\overline{S2}$ status lines to indicate to external devices the type of bus cycle the processor is starting. Table 3-12 shows the signals that are output for each type of cycle. These status lines are connected to an 8288 Bus Controller. The bus controller decodes these lines and outputs the signals that control components attached to the bus. The IOP indicates "instruction fetch" on these lines when it is reading and writing memory operands as well as when it is fet-

ched instructions. In the remote configuration, an 8289 Bus Arbiter monitors the $\overline{S0}$-$\overline{S2}$ status lines to determine when a system bus access is required.

Table 3-12. Status Signals S0-S2

$\overline{S2}$	$\overline{S1}$	$\overline{S0}$	Type of Bus Cycle
0	0	0	Instruction fetch from I/O space
0	0	1	Data fetch from I/O space
0	1	0	Data store to I/O space
0	1	1	(not used)
1	0	0	Instruction fetch from system space
1	0	1	Data fetch from system space
1	1	0	Data store to system space
1	1	1	Passive; no bus cycle run

Status lines S3-S6 indicate whether the bus cycle is DMA or non-DMA, and which channel is running the cycle (see table 3-13). Note that when the IOP is not running a bus cycle (e.g., when it is idle or when it is executing an internal cycle that does not use the bus), the status lines reflect the last bus cycle run.

Table 3-13. Status Signals S3-S6

S6	S5	S4	S3	Bus Cycle
1	1	0	0	DMA cycle on channel 1
1	1	0	1	DMA cycle on channel 2
1	1	1	0	Non-DMA cycle on channel 1
1	1	1	1	Non-DMA cycle on channel 2

3.7 Instruction Set

This section divides the IOP's 53 instructions into five functional categories:

1. data transfer,
2. arithmetic,
3. logic and bit manipulation,
4. program transfer,
5. processor control.

The description of each instruction in these categories explains how the instruction operates and how it may be used in channel programs. Instructions that perform essentially the same operation (e.g., ADD and ADDB, which add words and bytes respectively), are described together. A reference table at the end of the section lists every instruction alphabetically and provides execution time, encoded length, and sample ASM-89 coding for each permissable operand combination. For information on how the 8089 machine instructions are encoded in memory, see section 4.3.

In reading this section, it is important to recall that the instruction set does not differentiate between memory addresses and I/O device addresses. Instructions that are described as accepting byte and word memory operands may also be used to read and write I/O devices.

Data Transfer Instructions

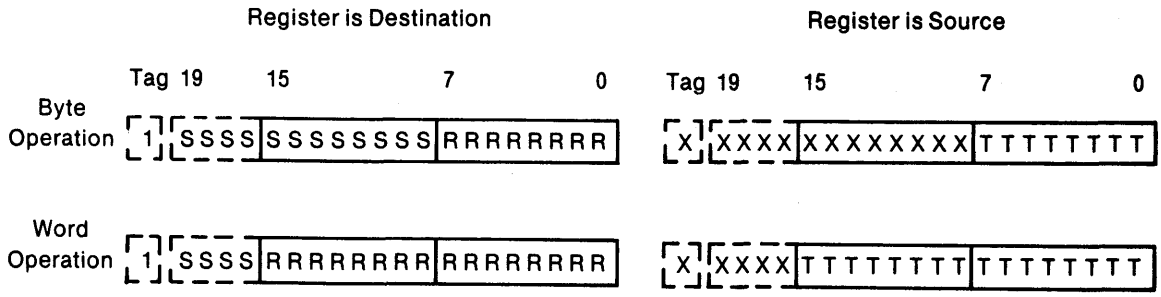
These instructions move data between memory and channel registers. Traditional byte and word moves (including memory-to-memory) are available, as are special instructions that load addresses into pointer registers and update tag bits in the process.

MOV *destination, source*

MOV transfers a byte or word from the source to the destination. Four instructions are provided:

MOV	Move Word Variable,
MOVB	Move Byte Variable,
MOVI	Move Word Immediate,
MOVBI	Move Byte Immediate.

Figure 3-35 shows how these instructions affect register operands. Notice that when a pointer register is specified as the destination of a MOV, its tag bit is unconditionally set to 1. MOV instructions are therefore used to load I/O space addresses into pointer registers.



T = bit is transferred to destination operand
R = bit is replaced by source operand
S = bit is sign extension of high-order bit transferred
X = bit is ignored
1 = bit is unconditionally set

Figure 3-35. Register Operands in MOV Instructions

MOVP *destination, source*

MOVP (move pointer) transfers a physical address variable between a pointer register and memory. If the source is a pointer register, its content and tag bit are converted to a physical address pointer (see figure 3-23). If the source is a memory location, the three bytes are converted to a 20-bit physical address and a tag value, and are loaded into the pointer register and its tag bit. MOVP is typically used to save and restore pointer registers.

LPD *destination, source*

LPD (load pointer with doubleword) converts a doubleword pointer (see figure 3-22) to a 20-bit physical address and loads it into the destination, which must be a pointer register. The pointer register's tag bit is unconditionally cleared to 0, indicating a system address. Two instructions are provided:

LPD	Load Pointer With Doubleword Variable
LPDI	Load Pointer With Doubleword Immediate

An 8086 or 8088 can pass any address in its megabyte memory space to a channel program in the form of a doubleword pointer. The channel program can access the location by using LPD to load the location address into a pointer register.

Arithmetic Instructions

The arithmetic instructions interpret all operands as unsigned binary numbers of 8, 16 or 20 bits. Signed values may be represented in standard two's complement notation with the high-order bit representing the sign (0=positive, 1=negative). The processor, however, has no way of detecting an overflow into a sign bit so this possibility must be provided for in the user's software.

The 8089 performs arithmetic operations to 20 significant bits as follows. Byte and word operands are sign-extended to 20 bits (e.g., bit 7 of a byte operand is propagated through bits 8-19 of an internal register). Sign extension does not affect the magnitude of the operand. The operation is then performed, and the 20-bit result is

returned to the destination operand. High-order bits are truncated as necessary to fit the result in the available space. A carry out of, or borrow into, the high-order bit of the result is not detected. However, if the destination is a register that is larger than the source operand, carries will be reflected in the upper register bits, up to the size of the register.

Figure 3-36 shows how the arithmetic instructions treat registers when they are specified as source and destination operands.

ADD *destination, source*

The sum of the two operands replaces the destination operand. Four addition instructions are provided:

ADD	Add Word Variable
ADDB	Add Byte Variable
ADDI	Add Word Immediate
ADDBI	Add Byte Immediate

INC *destination*

The destination is incremented by 1. Two instructions are available:

INC	Increment Word
INCB	Increment Byte

DEC *destination*

The destination is decremented by 1. Word and byte instructions are provided:

DEC	Decrement Word
DECB	Decrement Byte

Logical and Bit Manipulation Instructions

The logical instructions include the boolean operators AND, OR and NOT. Two bit manipulation instructions are provided for setting or

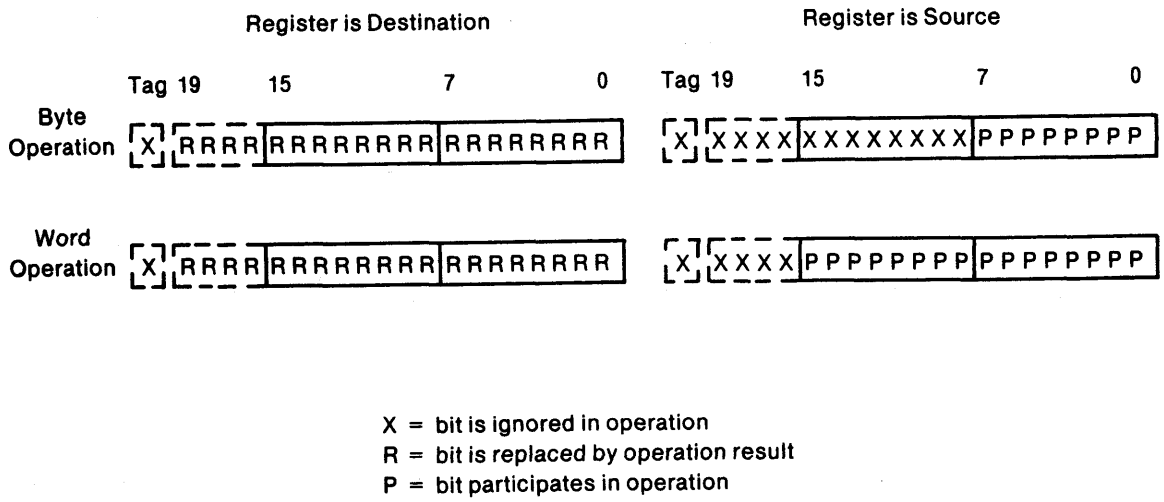


Figure 3-36. Register Operands in Arithmetic Instructions

clearing a single bit in memory or in an I/O device register. As shown in figure 3-37, the logical operations always leave the upper four bits of 20-bit destination registers undefined. These bits should not be assumed to contain reliable values or the same values from one operation to the next. Notice also that when a register is specified as the destination of a byte operation, bits 8-15 are overwritten by bit 7 of the result. Bits 8-15 can be preserved in AND and OR instructions by using word operations in which the upper byte of the source operand is FFH or 00H, respectively.

AND *destination, source*

The two operands are logically ANDed and the result replaces the destination operand. A bit in the result is set if the bits in the corresponding positions of the operands are both set, otherwise the result bit is cleared. The following AND instructions are available:

AND	Logical AND Word Variable
ANDB	Logical AND Byte Variable
ANDI	Logical AND Word Immediate
ANDBI	Logical AND Byte Immediate

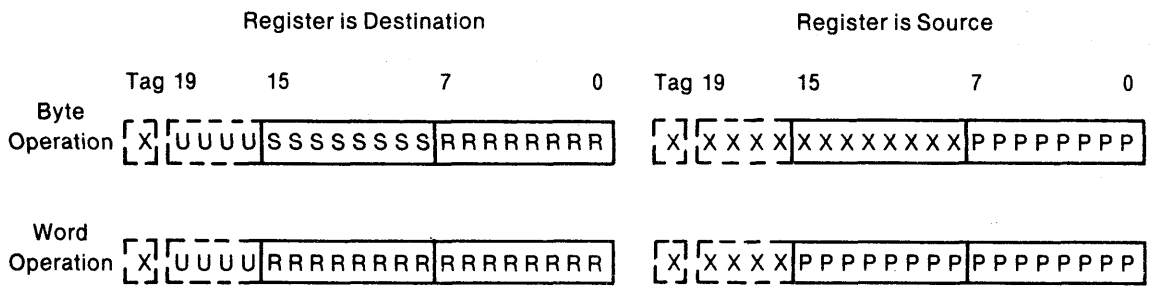
AND is useful when more than one bit of a device register must be cleared while leaving the remaining bits intact. For example, ANDing an 8-bit register with EEH only clears bits 0 and 4.

OR *destination, source*

The two operands are logically ORed, and the result replaces the destination operand. A bit in the result is set if either or both of the corresponding bits of the operands are set; if both operand bits are cleared, the result bit is cleared. Four types of OR instructions are provided:

OR	Logical OR Word Variable
ORB	Logical OR Byte Variable
ORI	Logical OR Word Immediate
ORBI	Logical OR Byte Immediate

OR can be used to selectively set multiple bits in a device register. For example, ORing an 8-bit register with 30H sets bits 4 and 5, but does not affect the other bits.



X = bit is ignored in operation
U = bit is undefined following operation
R = bit participates in operation and is replaced by result
S = bit is sign-extension of high-order result bit
P = bit participates in operation, but is unchanged

Figure 3-37. Register Operands in Logical Instructions

NOT *destination/destination, source*

NOT inverts the bits of an operand. If a single operand is coded, the inverted result replaces the original value. If two operands are coded, the inverted bits of the source replace the destination value (which must be a register), but the source retains its original value. In addition to these two operand forms, separate mnemonics are provided for word and byte values:

NOT	Logical NOT Word
NOTB	Logical NOT Byte

NOT followed by INC will negate (create the two's complement of) a positive number.

SETB *destination, bit-select*

The bit-select operand specifies one bit in the destination, which must be a memory byte, that is unconditionally set to 1. A bit-select value of 0 specifies the low-order bit of the destination while the high-order bit is set if bit-select is 7. SETB is handy for setting a single bit in an 8-bit device register.

CLR *destination, bit-select*

CLR operates exactly like SETB except that the selected bit is unconditionally cleared to 0.

Program Transfer Instructions

Register TP controls the sequence in which channel program instructions are executed. As each instruction is executed, the length of the instruction is added to TP so that it points to the next sequential instruction. The program transfer instructions can alter this sequential execution by adding a signed displacement value to TP. The displacement is contained in the program transfer instruction and may be either 8 or 16 bits long. The displacement is encoded in two's complement notation, and the high-order bit indicates the sign (0=positive displacement, 1=negative displacement). An 8-bit displacement may cause a transfer to a location in the range −128 through +127 bytes from the end of the transfer instruction, while a 16-bit displacement can transfer to any location within −32,768 through +32,767 bytes. An instruction containing an 8-bit displacement is called a short transfer and an instruction containing a 16-bit displacement is called a long transfer.

The program transfer instructions have alternate mnemonics. If the mnemonic begins with the letter "L," the transfer is long, and the distance to the transfer target is expressed as a 16-bit displacement regardless of how far away the target is located. If the mnemonic does not begin with "L," the ASM-89 assembler may build a short or long displacement according to rules discussed in section 3.9.

The "self-relative" addressing technique used by program transfer instructions has two important consequences. First, it promotes position-independent code, i.e., code that can be moved in memory and still execute correctly. The only restriction here is that the entire program must be moved as a unit so that the distance between the transfer instruction and its target does not change. Second, the limited addressing range of these instructions must be kept in mind when designing large (over 32k bytes of code) channel programs.

CALL/LCALL *TPsave, target*

CALL invokes an out-of-line routine, saving the value of TP so that the subroutine can transfer back to the instruction following the CALL. The instruction stores TP and its tag bit in the TPsave operand, which must be a physical address variable, and then transfers to the target address formed by adding the target operand's displacement to TP. The subroutine can return to the instruction following the CALL by using a MOVP instruction to load TPsave back into TP.

Notice that the 8089's facilities for implementing subroutines, or procedures, is less sophisticated than its counterparts in the 8086/8088. The principal difference is that the 8089 does not have a built in stack mechanism. 8089 programs can implement a stack using a base register as a stack pointer. On the other hand, since channel programs are not subject to interrupts, a stack will not be required for most channel programs.

JMP/LJMP *target*

JMP causes an unconditional transfer (jump) to the target location. Since the task pointer is not saved, no return to the instruction following the JMP is implied.

JZ/LJZ *source, target*

JZ (jump if zero) effects a transfer to the target location if the source operand is zero; otherwise the instruction following JZ is executed. Word and byte values may be tested by alternate instructions:

JZ/LJZ	Jump/Long Jump if Word Zero
JZB/LJZB	Jump/Long Jump if Byte Zero

If the source operand is a register, only the low-order 16 bits are tested; any additional high-order bits in the register are ignored. To test the low-order byte of a register, clear bits 8-15 and then use the word form of the instruction.

JNZ/LJNZ *source, target*

JNZ operates exactly like JZ except that control is transferred to the target if the source operand does not contain all 0-bits. Word and byte sources may be tested using these mnemonics:

JNZ/LJNZ	Jump/Long Jump if Word Not Zero
JNZB/LJNZB	Jump/Long Jump if Byte Not Zero.

JMCE/LJMCE *source, target*

This instruction (jump if masked compare equal) effects a transfer to the target location if the source (a memory byte) is equal to the lower byte in register MC as masked by the upper byte in MC. Figure 3-15 illustrates how 0-bits in the upper half of MC cause the corresponding bits in the lower half of MC and the source operand to compare equal, regardless of their actual values. For example, if bits 8-15 of MC contain the value 01H, then the transfer will occur if bit 0 of the source and register MC are equal. This instruction is useful for testing multiple bits in 8-bit device registers.

JMCNE/LJMCNE *source, target*

This instruction causes a jump to the target location if the source is not equal to the mask/compare value in MC. It otherwise operates identically to JMCE.

JBT/LJBT *source, bit-select, target*

JBT (jump if bit true) tests a single bit in the source operand and jumps to the target if the bit is a 1. The source must be a byte in memory or in an I/O device register. The bit-select value may range from 0 through 7, with 0 specifying the low-order bit. This instruction may be used to test a bit in an 8-bit device register. If the target is the JBT instruction itself, the operation effectively becomes "wait until bit is 0."

JNBT/LJNBT *source, bit-select, target*

This instruction operates exactly like JBT, except that the transfer is made if the bit is not true, i.e., if the bit is 0.

Processor Control Instructions

These instructions enable channel programs to control IOP hardware facilities such as the $\overline{\text{LOCK}}$ and SINTR1-2 pins, logical bus width selection, and the initiation of a DMA transfer.

TSL *destination, set-value, target*

Figure 3-38 illustrates the operation of the TSL (test and set while locked) instruction. TSL can be used to implement a semaphore variable that controls access to a shared resource in a multiprocessor system (see section 2.5). If the target operand specifies the address of the TSL instruction, the instruction is repetively executed until the semaphore (destination) is found to contain zero. Thus the channel program does not proceed until the resource is free.

WID *source-width, dest-width*

WID (set logical bus widths) alters bits 0 and 1 of the PSW, thus specifying logical bus widths for a DMA transfer. The operands may be specified as

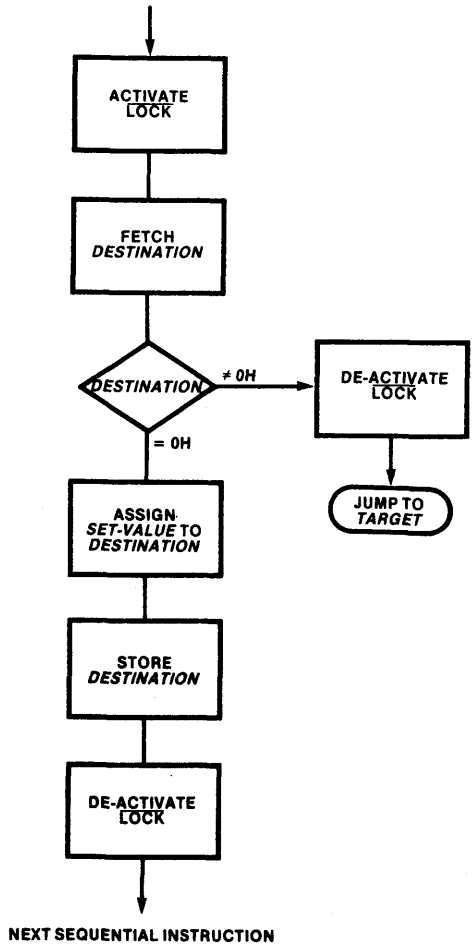


Figure 3-38. Operation of TSL Instruction

8 or 16 (bits), with the restriction that the logical width of a bus cannot exceed its physical width. The logical bus widths are undefined following a processor RESET; therefore the WID instruction must be executed before the first transfer. Thereafter the logical widths retain their values until the next WID instruction or processor RESET.

XFER *(no operands)*

XFER (enter DMA transfer mode after following instruction) prepares the channel for a DMA transfer operation. In a synchronized transfer, the instruction following XFER may ready the synchronizing device (e.g., send a "start" command or the last of a series of parameters). Any instruction, including NOP and WID, may follow XFER, except an instruction that alters GA, GB or GC.

SINTR *(no operands)*

This instruction sets the interrupt service bit in the PSW and activates the channel's SINTR line if the interrupt control bit in the PSW is set. If the

interrupt control bit is cleared (interrupts from this channel are disabled), the interrupt service bit is set, but SINTR1-2 is not activated. A channel program may use this instruction to interrupt a CPU.

NOP *(no operands)*

This instruction consumes clock cycles but performs no operation. As such, it is useful in timing loops.

HLT *(no operands)*

This instruction concludes a channel program. The channel clears its BUSY flag and then idles.

Instruction Set Reference Information

Table 3-16 lists every 8089 instruction alphabetically by its ASM-89 mnemonic. The ASM-89 coding format is shown (see table 3-14 for an explanation of operand identifiers) along with the instruction name. For every combination of operand types (see table 3-15 for key), the instruction's execution time and its length in bytes, and a coding example are provided.

The instruction timing figures are the number of clock periods required to execute the instruction with the given combination of operands. At 5 MHz, one clock period is 200 ns; at 8 MHz a clock period is 125 ns. Two timings are provided when an instruction operates on a memory word. The first (lower) figure indicates execution time when the word is aligned on an even address and is accessed over a 16-bit bus. The second figure is for odd-addressed words on 16-bit buses and any word accessed via an 8-bit bus.

Instruction fetch time is shown in table 3-17 and should be added to the execution times shown in table 3-16 to determine how long a sequence of instructions will take to run. (Section 3.2 explains the effect of the instruction queue on 16-bit instruction fetches.) External delays such as bus arbitration, wait states and activity on the other channel will increase the elapsed time over the figures shown in tables 3-16 and 3-17. These delays are application dependent.

Table 3-14. Key to ASM-89 Operand Identifiers

IDENTIFIER	USED IN	EXPLANATION
destination	data transfer, arithmetic, bit manipulation	A register or memory location that may contain data operated on by the instruction, and which receives (is replaced by) the result of the operation.
source	data transfer, arithmetic, bit manipulation	A register, memory location, or immediate value that is used in the operation, but is not altered by the instruction.
target	program transfer	Location to which control is to be transferred.
TPsave	program transfer	A 24-bit memory location where the address of the next sequential instruction is to be saved.
bit-select	bit manipulation	Specification of a bit location within a byte; 0=least-significant (rightmost) bit, 7=most-significant (leftmost) bit.
set-value	TSL	Value to which destination is set if it is found 0.
source-width	WID	Logical width of source bus.
dest-width	WID	Logical width of destination bus.

Table 3-15. Key to Operand Types

IDENTIFIER	EXPLANATION
(no operands)	No operands are written
register	Any general register
ptr-reg	A pointer register
immed8	A constant in the range 0-FFH
immed16	A constant in the range 0-FFFFH
mem8	An 8-bit memory location (byte)
mem16	A 16-bit memory location (word)
mem24	A 24-bit memory location (physical address pointer)
mem32	A 32-bit memory location (doubleword pointer)
label	A label within −32,768 to +32,767 bytes of the end of the instruction
short-label	A label within −128 to +127 bytes of the end of the instruction
0-7	A constant in the range: 0-7
8/16	The constant 8 or the constant 16

Table 3-16. Instruction Set Reference Data

ADD destination, source		Add Word Variable	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, mem16	11/15	2-3	ADD BC, [GA].LENGTH
mem16, register	16/26	2-3	ADD [GB], GC

ADDB destination, source		Add Byte Variable	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, mem8	11	2-3	ADDB GC, [GA].N_CHARS
mem8, register	16	2-3	ADDB [PP].ERRORS, MC

ADDBI destination, source		Add Byte Immediate	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, immed8	3	3	ADDBI MC,10
mem8, immed8	16	3-4	ADDBI [PP+IX+].RECORDS, 2CH

ADDI destination, source		Add Word Immediate	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, immed16	3	4	ADDI GB, 0C25BH
mem16, immed16	16/26	4-5	ADDI [GB].POINTER, 5899

Table 3-16. Instruction Set Reference Data (Cont'd.)

AND destination, source		Logical AND Word Variable	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, mem16	11/15	2-3	AND MC, [GA].FLAG_WORD
mem16, register	16/26	2-3	AND [GC].STATUS, BC

ANDB destination, source		Logical AND Byte Variable	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, mem8	11	2-3	AND BC, [GC]
mem8, register	16	2-3	AND [GA+IX].RESULT, GA

ANDBI destination, source		Logical AND Byte Immediate	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, immed8	3	3	GA, 01100000B
mem8, immed8	16	3-4	[GC+IX], 2CH

ANDI destination, source		Logical AND Word Immediate	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, immed16	3	4	IX, 0H
mem16, immed16	16/26	4-5	[GB+IX].TAB, 40H

CALL TPsave, target		Call	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem24, label	17/23	3-5	CALL [GC+IX].SAVE, GET_NEXT

CLR destination, bit select		Clear Bit To Zero	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, 0-7	16	2-3	CLR [GA], 3

DEC destination		Decrement Word By 1	
Operands	**Clocks**	**Bytes**	**Coding Example**
register	3	2	
mem16	16/26	2-3	DEC [PP].RETRY

Table 3-16. Instruction Set Reference Data (Cont'd.)

DECB destination		Decrement Byte By 1	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8	16	2-3	DECB [GA+IX+].TAB

HLT (no operands)		Halt Channel Program	
Operands	**Clocks**	**Bytes**	**Coding Example**
(no operands)	11	2	HLT

INC destination		Increment Word by 1	
Operands	**Clocks**	**Bytes**	**Coding Example**
register	3	2	INC GA
mem16	16/26	2-3	INC [GA].COUNT

INCB destination		Increment Byte by 1	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8	16	2-3	INCB [GB].POINTER

JBT source, bit-select, target		Jump if Bit True (1)	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, 0-7, label	14	3-5	JBT [GA].RESULT_REG, 3, DATA_VALID

JMCE source, target		Jump if Masked Compare Equal	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, label	14	3-5	JMCE [GB].FLAG, STOP_SEARCH

JMCNE source, target		Jump if Masked Compare Not Equal	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, label	14	3-5	JMCNE [GB+IX], NEXT_ITEM

JMP target		Jump Unconditionally	
Operands	**Clocks**	**Bytes**	**Coding Example**
label	3	3-4	JMP READ_SECTOR

Table 3-16. Instruction Set Reference Data (Cont'd.)

JNBT source, bit-select, target		Jump if Bit Not True (0)	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, 0-7, label	14	3-5	JNBT [GC], 3, RE_READ

JNZ source, target		Jump if Word Not Zero	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, label	5	3-4	JNZ BC, WRITE_LINE
mem16, label	12/16	3-5	JNZ [PP].NUM_CHARS, PUT_BYTE

JNZB source, target		Jump if Byte Not Zero	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, label	12	3-5	JNZB [GA], MORE_DATA

JZ source, target		Jump if Word is Zero	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, label	5	3-4	JZ BC, NEXT_LINE
mem16, label	12/16	3-5	JZ [GC+IX].INDEX, BUF_EMPTY

JZB source, target		Jump if Byte Zero	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, label	12	3-5	JZB [PP].LINES_LEFT, RETURN

LCALL TPsave, target		Long Call	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem24, label	17/23	4-5	LCALL [GC].RETURN_SAVE, INIT_8279

LJBT source, bit-select, target		Long Jump if Bit True (1)	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, 0-7, label	14	4-5	LJBT [GA].RESULT, 1, DATA_OK

LJMCE source, target		Long jump if Masked Compare Equal	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, label	14	4-5	LJMCE [GB], BYTE_FOUND

Table 3-16. Instruction Set Reference Data (Cont'd.)

LJMCNE source, target		Long jump if Masked Compare Not Equal	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, label	14	4-5	LJMCNE [GC+IX+], SCAN_NEXT

LJMP target		Long Jump Unconditional	
Operands	**Clocks**	**Bytes**	**Coding Example**
label	3	4	LJMP GET_CURSOR

LJNBT source, bit-select, target		Long Jump if Bit Not True (0)	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, 0-7, label	14	4-5	LJNBT [GC], 6, CRCC_ERROR

LJNZ source, target		Long Jump if Word Not Zero	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, label	5	4	LJNZ BC, PARTIAL_XMIT
mem16, label	12/16	4-5	LJNZ [GA+IX].N_LEFT, PUT_DATA

LJNZB source, target		Long Jump if Byte Not Zero	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, label	12	4-5	LJNZB [GB+IX+].ITEM, BUMP_COUNT

LJZ source, target		Long Jump if Word Zero	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, label	5	4	LJZ IX, FIRST_ELEMENT
mem16, label	12/16	4-5	LJZ [GB].XMIT_COUNT, NO_DATA

LJZB source, target		Long Jump if Byte Zero	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, label	12	4-5	LJZB [GA], RETURN_LINE

LPD destination, source		Load Pointer With Doubleword Variable	
Operands	**Clocks**	**Bytes**	**Coding Example**
ptr-reg, mem32	20/28*	2-3	LPD GA, [PP].BUF_START

*20 clocks if operand is on even address; 28 if on odd address

Table 3-16. Instruction Set Reference Data (Cont'd.)

JNBT source, bit-select, target		Jump if Bit Not True (0)	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, 0-7, label	14	3-5	JNBT [GC], 3, RE_READ

JNZ source, target		Jump if Word Not Zero	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, label	5	3-4	JNZ BC, WRITE_LINE
mem16, label	12/16	3-5	JNZ [PP].NUM_CHARS, PUT_BYTE

JNZB source, target		Jump if Byte Not Zero	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, label	12	3-5	JNZB [GA], MORE_DATA

JZ source, target		Jump if Word is Zero	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, label	5	3-4	JZ BC, NEXT_LINE
mem16, label	12/16	3-5	JZ [GC+IX].INDEX, BUF_EMPTY

JZB source, target		Jump if Byte Zero	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, label	12	3-5	JZB [PP].LINES_LEFT, RETURN

LCALL TPsave, target		Long Call	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem24, label	17/23	4-5	LCALL [GC].RETURN_SAVE, INIT_8279

LJBT source, bit-select, target		Long Jump if Bit True (1)	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, 0-7, label	14	4-5	LJBT [GA].RESULT, 1, DATA_OK

LJMCE source, target		Long jump if Masked Compare Equal	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, label	14	4-5	LJMCE [GB], BYTE_FOUND

Table 3-16. Instruction Set Reference Data (Cont'd.)

LJMCNE source, target		Long jump if Masked Compare Not Equal	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, label	14	4-5	LJMCNE [GC+IX+], SCAN_NEXT

LJMP target		Long Jump Unconditional	
Operands	**Clocks**	**Bytes**	**Coding Example**
label	3	4	LJMP GET_CURSOR

LJNBT source, bit-select, target		Long Jump if Bit Not True (0)	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, 0-7, label	14	4-5	LJNBT [GC], 6, CRCC_ERROR

LJNZ source, target		Long Jump if Word Not Zero	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, label	5	4	LJNZ BC, PARTIAL_XMIT
mem16, label	12/16	4-5	LJNZ [GA+IX].N_LEFT, PUT_DATA

LJNZB source, target		Long Jump if Byte Not Zero	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, label	12	4-5	LJNZB [GB+IX+].ITEM, BUMP_COUNT

LJZ source, target		Long Jump if Word Zero	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, label	5	4	LJZ IX, FIRST_ELEMENT
mem16, label	12/16	4-5	LJZ [GB].XMIT_COUNT, NO_DATA

LJZB source, target		Long Jump if Byte Zero	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, label	12	4-5	LJZB [GA], RETURN_LINE

LPD destination, source		Load Pointer With Doubleword Variable	
Operands	**Clocks**	**Bytes**	**Coding Example**
ptr-reg, mem32	20/28*	2-3	LPD GA, [PP].BUF_START

*20 clocks if operand is on even address; 28 if on odd address

Table 3-16. Instruction Set Reference Data (Cont'd.)

LPDI destination, source		Load Pointer With Doubleword Immediate	
Operands	**Clocks**	**Bytes**	**Coding Example**
ptr-reg, immed32	12/16*	6	LPDI GB, DISK_ADDRESS

*12 clocks if instruction is on even address; 16 if on odd address

MOV destination, source		Move Word	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, mem16	8/12	2-3	MOV IX, [GC]
mem16, register	10/16	2-3	MOV [GA].COUNT, BC
mem16, mem16	18/28	4-6	MOV [GA].READING, [GB]

MOVB destination, source		Move Byte	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, mem8	8	2-3	MOVB BC, [PP].TRAN_COUNT
mem8, register	10	2-3	MOVB [PP].RETURN_CODE, GC
mem8, mem8	18	4-6	MOVB [GB+IX+], [GA+IX+]

MOVBI destination, source		Move Byte Immediate	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, immed8	3	3	MOVBI MC, 'A'
mem8, immed8	12	3-4	MOVBI [PP].RESULT, 0

MOVI destination, source		Move Word Immediate	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, immed16	3	4	MOVI BC, 0
mem16, immed16	12/18	4-5	MOVI [GB], 0FFFFH

MOVP destination, source		Move Pointer	
Operands	**Clocks**	**Bytes**	**Coding Example**
ptr-reg, mem24	19/27*	2-3	MOVP TP, [GC+IX]
mem24, ptr-reg	16/22*	2-3	MOVP [GB].SAVE_ADDR, GC

*First figure is for operand on even address; second is for odd-addressed operand.

NOP (no operands)		No Operation	
Operands	**Clocks**	**Bytes**	**Coding Example**
(no operands)	4	2	NOP

Table 3-16. Instruction Set Reference Data (Cont'd.)

NOT destination/destination, source — Logical NOT Word

Operands	Clocks	Bytes	Coding Example
register	3	2	NOT MC
mem16	16/26	2-3	NOT [GA].PARM
register, mem16	11/15	2-3	NOT BC, [GA+IX].LINES_LEFT

NOTB destination/destination, source — Logical NOT Byte

Operands	Clocks	Bytes	Coding Example
mem8	16	2-3	NOTB [GA].PARM_REG
register, mem8	11	2-3	NOTB IX, [GB].STATUS

OR destination, source — Logical OR Word

Operands	Clocks	Bytes	Coding Example
register, mem16	11/15	2-3	OR MC, [GC].MASK
mem16, register	16/26	2-3	OR [GC], BC

ORB destination, source — Logical OR Byte

Operands	Clocks	Bytes	Coding Example
register, mem8	11	2-3	ORB IX, [PP].POINTER
mem8, register	16	2-3	ORB [GA+IX+], GB

ORBI destination, source — Logical OR Byte Immediate

Operands	Clocks	Bytes	Coding Example
register, immed8	3	3	ORBI IX, 00010001B
mem8, immed8	16	3-4	ORBI [GB].COMMAND, 0CH

ORI destination, source — Logical OR Word Immediate

Operands	Clocks	Bytes	Coding Example
register, immed16	3	4	ORI MC, 0FF0DH
mem16,immed16	16/26	4-5	ORI [GA], 1000H

SETB destination, bit-select — Set Bit to 1

Operands	Clocks	Bytes	Coding Example
mem8, 0-7	16	2-3	SETB [GA].PARM_REG, 2

SINTR (no operands) — Set Interrupt Service Bit

Operands	Clocks	Bytes	Coding Example
(no operands)	4	2	SINTR

Table 3-16. Instruction Set Reference Data (Cont'd.)

TSL destination, set-value, target		Test and Set While Locked	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, immed8, short-label	14/16*	4-5	TSL [GA].FLAG, 0FFH, NOT_READY

*14 clocks if destination ≠ 0; 16 clocks if destination = 0

WID source-width, dest-width		Set Logical Bus Widths	
Operands	**Clocks**	**Bytes**	**Coding Example**
8/16, 8/16	4	2	WID 8, 8

XFER (no operands)		Enter DMA Transfer Mode After Next Instruction	
Operands	**Clocks**	**Bytes**	**Coding Example**
(no operands)	4	2	XFER

Table 3-17. Instruction Fetch Timings (Clock Periods)

INSTRUCTION LENGTH (BYTES)	BUS WIDTH		
	8	16	
		(1)	(2)
2	14	7	11
3	18	14	11
4	22	14	15
5	26	18	15

(1) First byte of instruction is on an even address.

(2) First byte of instruction is on an odd address. Add 3 clocks if first byte is not in queue (e.g., first instruction following program transfer).

3.8 Addressing Modes

8089 instruction operands may reside in registers, in the instruction itself or in the system or I/O address spaces. Operands in the system and I/O spaces may be either memory locations or I/O device registers and may be addressed in four different ways. This section describes how the channel processes different types of operands and how it calculates addresses using its addressing modes. Section 3.9 describes the ASM-89 conventions that programmers use to specify these operands and addressing modes.

Register and Immediate Operands

Registers may be specified as source or destination operands in many instructions. Instructions that operate on registers are generally both shorter and faster than instructions that specify immediate or memory operands.

Immediate operands are data contained in instructions rather than in registers or in memory. The data may be either 8 or 16 bits in length. The limitations of immediate operands are that they may only serve as source operands and that they are constant values.

Memory Addressing Modes

Whereas the channel has direct access to register and immediate operands, operands in the system and I/O space must be transferred to or from the IOP over the bus. To do this, the IOP must calculate the address of the operand, called its

effective address (EA). The programmer may specify that an operand's address be calculated in any of four different ways; these are the 8089's memory addressing modes.

The Effective Address

An operand in the system space has a 20-bit effective address, and an operand in the I/O space has a 16-bit effective address. These addresses are unsigned numbers that represent the distance (in bytes) of the low-order byte of the operand from the beginning of the address space. Since the 8089 does not "see" the segmented structure of the system space that it may share with an 8086 or 8088, 8089 effective addresses are equivalent to 8086/8088 physical addresses.

All memory addressing modes use the content of one of the pointer registers, and the state of that register's tag bit determines whether the operand lies in the system or the I/O space. If the operand is in the I/O space (tag = 1), bits 16-19 of the pointer register are ignored in the effective address calculation. Section 4.3 describes the two fields (AA and MM) in the encoded machine instruction that specify addressing mode and base (pointer) register.

Based Addressing

In based addressing (figure 3-39), the effective address is taken directly from the content of GA, GB, GC or PP. Using this addressing mode, one instruction may access different locations if the register is updated before the instruction executes. LPD, MOV, MOVP or arithmetic instructions might be used to change the value of the base register.

Offset Addressing

In this mode (figure 3-40) an 8-bit unsigned value contained in the instruction is added to the content of a base register to form the effective address. The offset mode provides a convenient way to address elements in structures (a parameter block is a typical example of a structure). As shown in figure 3-41, a base register can be pointed at the base (first element) in the structure, and then different offsets can be used to access the elements within the structure. By changing the base address, the same structure can be relocated elsewhere in memory.

Indexed Addressing

An indexed address is formed by adding the content of register IX (interpreted as an unsigned quantity) to a base register as shown in figure 3-42. Indexed addressing is often used to access

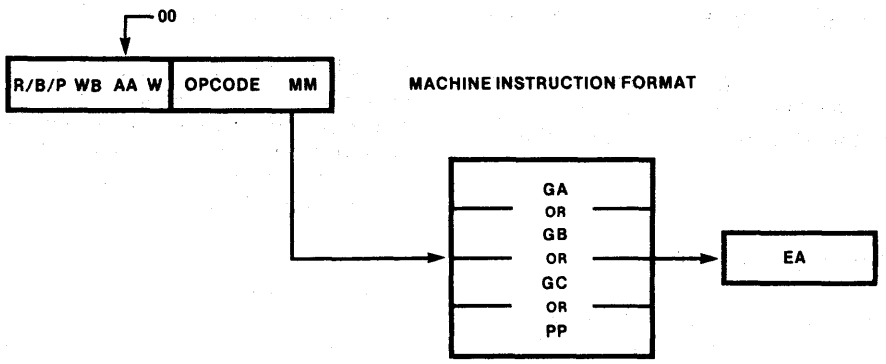


Figure 3-39. Based Addressing

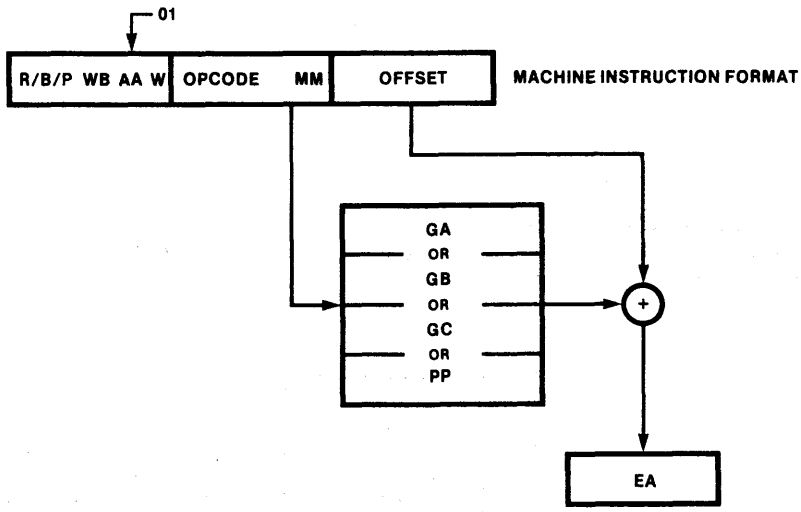


Figure 3-40. Offset Addressing

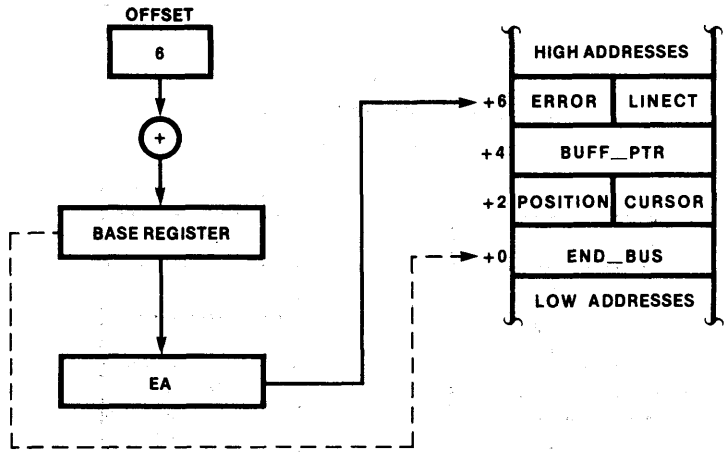


Figure 3-41. Accessing a Structure with Offset Addressing

array elements (see figure 3-43). A base register locates the beginning of the array and the value in IX selects one element, i.e., it acts as the array subscript. The *i*th element of a byte array is selected when IX contains $(i - 1)$. To access the *i*th element of a word array, IX should contain $((i - 1) * 2)$.

Indexed Auto-Increment Addressing

In this variation of indexed addressing, the effective address is formed by summing IX and a base register, and then IX is incremented automatically. (See figure 3-44.) The addition takes place

after the EA is calculated. IX is incremented by 1 for a byte operation, by 2 for a word operation and by 3 for a MOVP instruction. This addressing mode is very useful for "stepping through" successive elements of an array (e.g., a program loop that sums an array).

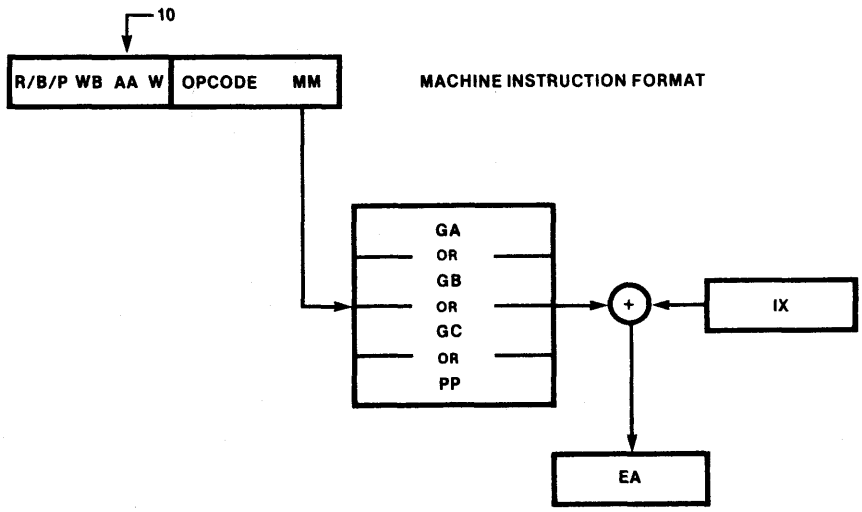


Figure 3-42. Indexed Addressing

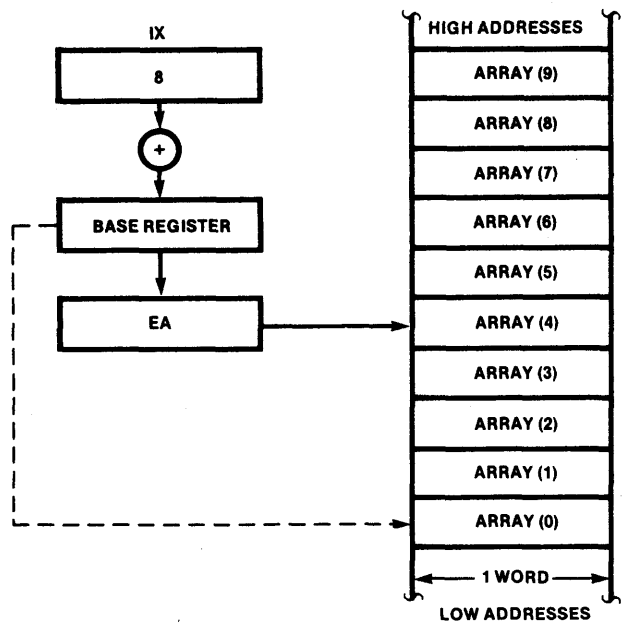


Figure 3-43. Accessing a Word Array with Indexed Addressing

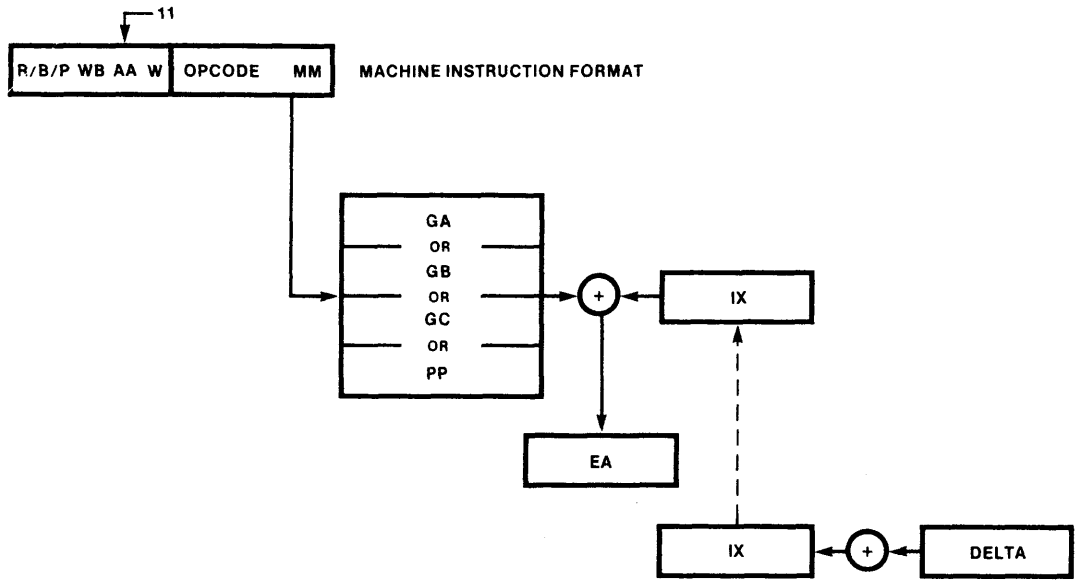


Figure 3-44. Indexed Auto-Increment Addressing

3.9 Programming Facilities

The compatibility of the 8089 with the 8086 and 8088 extends beyond the hardware interface. Comparing figure 3-45, with figure 2-45, one can see that, except for the translate step, the software development process is identical for both 8086/8088 and 8089 programs. The ASM-89 assembler produces a relocatable object module that is compatible with the 8086 family software development utilities LIB-86, LINK-86, LOC-86 and OH-86, described in section 2.9. All of these development tools run on an Intellec® 800 or Series II microcomputer development system.

This section surveys the facilities of the ASM-89 assembler and discusses how LINK-86 and LOC-86 can be used in 8089 software development. For a complete description of the 8089 assembly language, consult *8089 Assembly Language User's Guide*, Order No. 9800938, available from Intel's Literature Department.

ASM-89

The ASM-89 assembler reads a disk file containing 8089 assembly language statements, translates these statements into 8089 machine instructions, and writes the result into a second disk file. The assembly input is called a source module, and the principal output is a relocatable object module. The assembler also produces a file that lists the module and flags any errors detected during the assembly.

Statements

Statements are the building blocks of ASM-89 programs. Figure 3-46 shows several examples of ASM-89 statements. The ASM-89 assembler gives programmers considerable flexibility in formatting program statements. Variable names and labels (identifiers) may be up to 31 characters long, the underscore (_) character may be used to improve the readability of longer names (e.g.,

WAIT_UNTIL_READY). The component parts of statements (fields) need not be located at particular "columns" of the statement. Any number of blank characters may separate fields and multiple identifiers within the operand field. Long statements may be continued onto the next link by coding an ampersand (&) as the first character of the continued line.

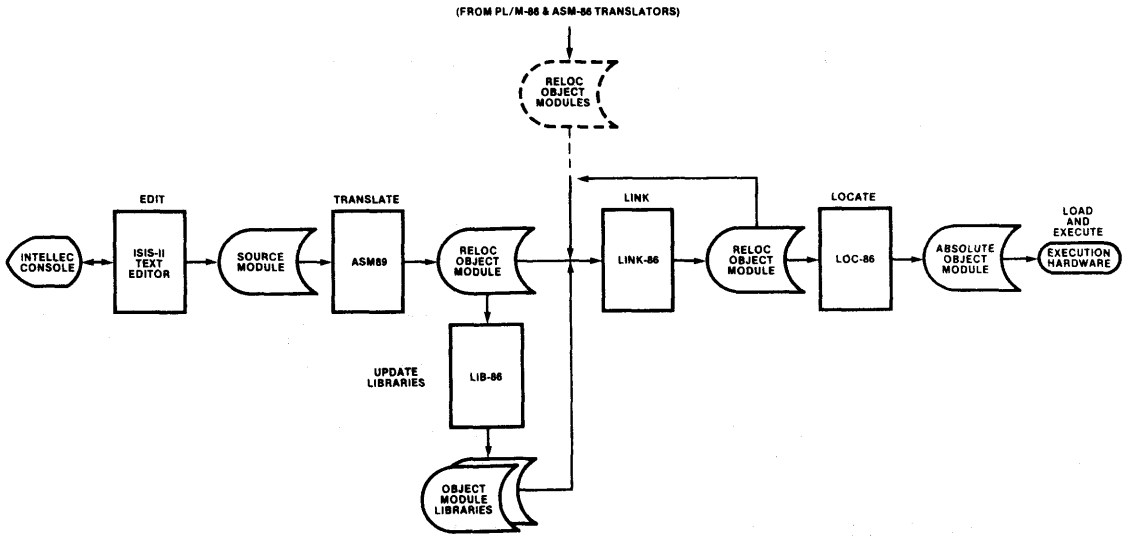


Figure 3-45. 8089 Software Development Process

```
; THIS STATEMENT CONTAINS A COMMENT FIELD ONLY
ADDI   BC,5                    ; TYPICAL ASM89 INSTRUCTION
  ADDI     BC,       5         ; NO "COLUMN" REQUIREMENTS
MOV      [GA].STATUS,
&        6                     ; A CONTINUED STATEMENT
SOURCE     EQU GA              ; A SIMPLE ASM89 DIRECTIVE
LINE_BUFFER_ADDRESS DD         ; A LONG IDENTIFIER
```

Figure 3-46. ASM-89 Statements

A statement whose first non-blank character is a semicolon is a comment statement. Comments have no affect on program execution and, in fact, are ignored by the ASM-89 assembler. Nevertheless, carefully selected comments are included in all well written ASM-89 programs. They summarize, annotate and clarify the logic of the program where the instructions are too "microscopic" to make the operation of the program self-evident.

An ASM-89 instruction statement (figure 3-47) directs the assembler to build an 8089 machine instruction. The optional label field assigns a symbolic identifier to the address where the instruction will be stored in memory. A labelled instruction can be the target of a program transfer; the transferring instruction specifies the label for its target operand. In figure 3-47 the labelled instruction conditionally transfers to itself; the program will loop on this one instruction as long as bit 3 of the byte addressed by [GA].STATUS is not true. The mnemonic field of an instruction statement specifies the type of 8089 machine instruction that the assembler is to build.

The operand field may contain no operands or one or more operands as required by the instruction. Multiple operands are separated by commas and, optionally, by blanks. Any instruction statement may contain a comment field (comment fields are initiated by a semicolon).

An ASM-89 directive statement (figure 3-48) does not produce an 8089 machine instruction. Rather, a directive gives the assembler information to use during the assembly. For example, the DS (define storage) directive in figure 3-48 tells the assembler to reserve 80 bytes of storage and to assign a symbolic identifier (INPUT_BUFFER) to the first (lowest-addressed) byte of this area. The ASM-89 assembler accepts 14 directives; the more commonly used directives are discussed in this section.

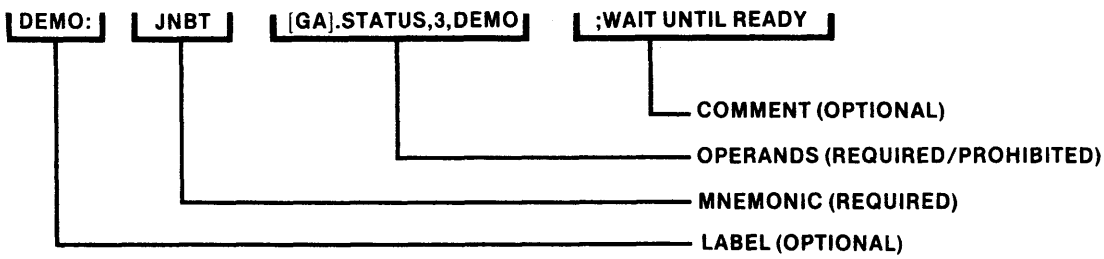


Figure 3-47. ASM-89 Instruction Format

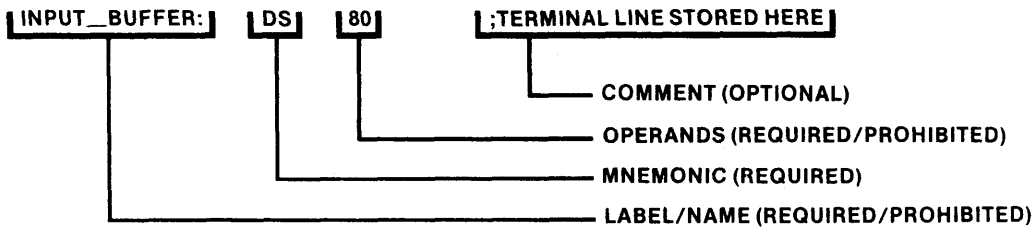


Figure 3-48. ASM-89 Directive Format

The first field in a directive may be a label or a name; individual directives may require or prohibit names, while labels are optional for directives that accept them. A label ends in a colon like an instruction statement label. However, a directive label cannot be specified as the target of a program transfer. A name does not have a colon. The second field is the directive mnemonic, and the assembler distinguishes between instructions and directives by this field. Any operands required by the directive are written next; multiple operands are separated by commas and, optionally, by blanks. A comment may be included in any directive by beginning the text with a semicolon.

Constants

Binary, decimal, octal and hexadecimal numeric constants (figure 3-49) may be written in ASM-89 instructions and directives. The assembler can add and subtract constants at assembly time. Numeric constants, including the results of arithmetic operations, must be representable in 16 bits. Positive numbers cannot exceed 65,535 (decimal); negative numbers, which the assembler represents in two's complement notation, cannot be "more negative" than −32,768 (decimal).

Character constants are enclosed in single quote marks as shown in figure 3-49. Strings of characters up to 255 bytes long may be written when initializing storage. Instruction operands, however, can only be one or two characters long (for byte and word instructions respectively).

As an aid to program clarity, The EQU (equate) directive may be used to give names to constants (e.g., DISK_STATUS EQU 0FF20H).

Defining Data

Four ASM-89 directives reserve space for memory variables in the ASM-89 program (see figure 3-50). The DB, DW and DD directives allocate units of bytes, words and doublewords, respectively, initialize the locations, and optionally label them so that they may be referred to by name in instruction statements. The label of a storage directive always refers to the first (lowest-addressed) byte of the area reserved by the directive.

The DB and DW directives may be used to define byte- and word-constant scalars (individual data items) and arrays (sequences of the same type of item). For example, a character string constant could be defined as a byte array:

```
SIGN_ON_MSG: DB 'PLEASE ENTER PASSWORD'
```

The DD directive is typically used to define the address of a location in the system space, i.e., a doubleword pointer variable. The address may be loaded into a pointer register with the LPD instruction.

The DS directive reserves, and optionally names, storage in units of bytes, but does not initialize any of the reserved bytes. DS is typically used for RAM-based variables such as buffers. As there is no special directive for defining a physical address pointer, DS is typically used to reserve the three bytes used by the MOVP instruction.

```
MOVBI   GA, 'A'          ; CHARACTER
MOVBI   GA, 41H          ; HEXADECIMAL
MOVBI   GA, 65           ; DECIMAL
MOVBI   GA, 65D          ; DECIMAL ALTERNATIVE
MOVBI   GA, 101Q         ; OCTAL
MOVBI   GA, 101O         ; OCTAL ALTERNATIVE
MOVBI   GA, 01000001B    ; BINARY
; NEXT TWO STATEMENTS ARE EQUIVALENT AND
;    ILLUSTRATE TWO'S COMPLEMENT REPRESENTATION
;    OF NEGATIVE NUMBERS
MOVBI   GA, -5
MOVBI   GA, 11111011B
```

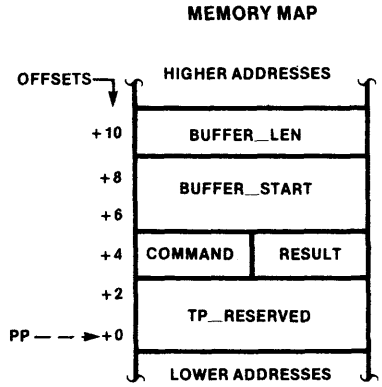
Figure 3-49. ASM89 Constants

```
; ASM89 DIRECTIVE                ; MEMORY CONTENT (HEX)
ALPHA:  DB      1               ; 01
        DB      -2              ; FE (TWO'S COMPLEMENT)
        DB      'A', 'B'        ; 4142
BETA:   DW      1               ; 0100
        DW      -5              ; FAFF
        DW      'AB'            ; 4241
        DW      400, 500        ; 2410F401
        DW      400H, 500H      ; 0004 0005
gamma:  DW      BETA            ; OFFSET OF BETA ABOVE,
                                ; FROM BEGINNING OF PROGRAM
DELTA   DD      GAMMA           ; ADDRESS (SEGMENT & OFFSET)
                                ; OF GAMMA
ZETA:   DS      80              ; 80 BYTES, UNINITIALIZED
```

Figure 3-50. ASM-89 Storage Directives

Structures

An ASM-89 structure is a map or template that gives names and relative locations to a collection of related variables that are called structure elements or members. Defining a structure, however, does not allocate storage. The structure is, in effect, overlaid on a particular area of memory when one of its elements is used as an instruction operand. Figure 3-51 shows how a structure representing a parameter block could be defined and then used in a channel program. The assembler uses the structure element name to produce an offset value (structures are used with the offset addressing mode). Compared to "hard-coded" offsets, structures improve program clarity and simplify maintenance. If the layout of a memory block changes, only the structure definition must be modified. When the program is reassembled, all symbolic references to the structure are automatically adjusted. When multiple areas of memory are laid out identically, a single structure can be used to address any area by changing the content of the pointer (base) register that specifies the structure's "starting address."




```
STRUCTURE DEFINITION

PARM_BLOCK      STRUC
  TP_RESERVED:  DS   4
  COMMAND:      DS   1
  RESULT:       DS   1
  BUFFER_START: DS   4
  BUFFER_LEN:   DS   2
PARM_BLOCK      ENDS
```

USING "HARD-CODED" OFFSETS

```
LPD GA, [PP].6
MOVBI   [PP].5,0
```

USING STRUCTURE ELEMENT NAMES

```
LPD GA, [PP].BUFFER_START
MOVBI   [PP].RESULT,0
```

Figure 3-51. ASM-89 Structure Definition and Use

Addressing Modes

Table 3-18 summarizes the notation a programmer uses to specify how the effective address of a memory operand is to be computed. Examples of typical ASM-89 coding for each addressing mode, as well as register and immediate operands, are provided in figure 3-52. Notice that a bracketed reference to a register indicates that the content of the register is to be used to form the effective address of a memory operand, while an unbracketed register reference specifies that the register itself is the operand.

The following examples summarize how the memory addressing modes can be used to access simple variables, structures and arrays.

- If GA contains the address of a memory operand, then [GA] refers to that operand.
- If GA contains the base address of a structure, then [GA].DATA refers to the DATA element (field) in that structure. If DATA is six bytes from the beginning of the structure, then [GA].6 refers to the same location.
- If GA contains the starting address of an array, then [GA+IX] addresses the array element indexed by IX. For example, if IX contains the value 4H, the effective address refers to the fifth element of a byte array, or the third element of a word array. [GA+IX+] selects the same element and additionally auto-increments IX by 1 (byte operation), 2 (word operation) or 3 (MOVP instruction) in anticipation of accessing the next array element.

Note that any pointer register could have been substituted for GA in the previous examples.

Table 3-18. ASM-89 Memory Addressing Mode Notation

Notation	Addressing Mode
[*ptr-reg*]	Based
[*ptr-reg*].*offset*	Offset
[*ptr-reg* + IX]	Indexed
[*ptr-reg* + IX +]	Indexed Post Auto-increment

ptr-reg = GA, GB, GC or PP
offset = 8-bit signed value; may be structure element

Program Transfer Targets

As discussed in section 3.7, program transfer instructions operate by adding a signed byte or word displacement to the task pointer. Table 3-19 shows how the ASM-89 assembler determines the sign and size of the displacement value it places in a program transfer machine instruction. In the table, the terms "backward" and "forward" refer to the location of a label specified as a transfer target relative to the transfer instruction. "Backward" means the label physically precedes the instruction in the source module, and "forward" means the label follows the instruction in the source text. The distances are from the end of the transfer instruction; the distance to the instruction immediately following the transfer is 0 bytes.

```
ADDI    GA, 5            ; REGISTER, IMMEDIATE
ADD     GC, [GB]         ; REGISTER, MEMORY (BASED)
ADDBI   [PP],10          ; MEMORY (BASED), IMMEDIATE
ADDB    IX, [GB].5       ; REGISTER, MEMORY (OFFSET)
ADDB    BC, [GC].COUNT   ; REGISTER, MEMORY (OFFSET)
ADD     [GC+IX], BC      ; MEMORY (INDEXED), REGISTER
ADDI    [GA+IX+],5       ; MEMORY (INDEXED AUTO-INCREMENT), IMMED
ADDB    [PP].ERROR, [GA] ; MEMORY (OFFSET), MEMORY (BASED)
```

Figure 3-52. ASM-89 Operand Coding Examples

Two important points can be drawn from table 3-19. First, a target must lie within 32k bytes of a transfer instruction; this should not prove restrictive except in very large programs. Second, one byte can be saved in the assembled instruction by writing the short mnemonic when the target is known to be within −128 through +127 assembled bytes of the transfer.

It is also important to note that a program transfer target must reside in the same module as the transferring instruction, i.e., the target address must be known at assembly time.

Procedures

An ASM-89 program may invoke an out-of-line procedure (subroutine) with the CALL/LCALL instruction. The first instruction operand specifies a memory location where the content of TP will be stored as a physical address pointer before control is transferred to the procedure. The procedure may return to the instruction following the CALL/LCALL by using the MOVP instruction to restore TP from the save area. Figure 3-53 illustrates one approach to procedure linkage.

A channel program may use the first two words of its parameter block (pointed to by PP) as a task pointer save area. However, this is not recommended if there is any chance that the CPU will issue a "suspend" command to the channel; this command stores the current value of TP in the same location, possibly overwriting a return address.

As in any program transfer, the target of a CALL/LCALL instruction must be contained in the same module and within 32k bytes of the instruction.

Segment Control

The relocatable object module produced by the ASM-89 assembler consists of a single logical segment. (A segment is a storage unit up to 64k bytes long; for a more complete description, refer to sections 2.3 and 2.7.) The ASM-89 SEGMENT and ENDS directives name the segment as shown in figure 3-54. Typically, all instructions and most directives are coded in between these directives. The END directive, which terminates the assembly, is an exception.

The LOC-86 utility can assign this logical segment to any memory address that is a physical segment boundary (i.e., whose low-order four bits are 0000). In a ROM-based system, variable data (which must be in RAM) can be "clustered" together at one "end" of the program as shown in figure 3-55. The ORG directive can then be used to force assembly of the variables to start at a given offset from the beginning of the segment (2,000 hexadecimal bytes in figure 3-55). As the

Table 3-19. Program Transfer Displacement

Target Location			
Mnemonic Form	**Direction**	**Distance**	**Displacement Sign Bytes**
Short (e.g., JMP)	Backward	≤128	− 1
	Forward	≤127	+ 1
	Backward	≤32,768	− 2
	Forward	≤32,767	Error
	Backward	>32,768	Error
	Forward	>32,767	Error
Long (e.g., LJMP)	Backward	≤128	− 2
	Forward	≤127	+ 2
	Backward	≤32,768	− 2
	Forward	≤32,767	+ 2
	Backward	>32,768	Error
	Forward	>32,767	Error

```
.
.
.
CALL SAVE:   DS   3   ; TP SAVE AREA
.
.
.
; SET UP TP SAVE AREA
;    NOTE: EXAMPLE ASSUMES PROGRAM
;          IS IN I/O SPACE. USE LPDI
;          IF IN SYSTEM SPACE.
;          MOVI   GC, CALLSAVE      ; LOAD ADDRESS TO GC
; CALL IT.
          LCALL   [GC],DEMO
.
.
.
          HLT   ; LOGICAL END OF PROGRAM

; DEFINE THE PROCEDURE.
DEMO:
; PROCEDURE INSTRUCTIONS GO HERE.
; NOTE: PROCEDURE MUST NOT UPDATE GC
;       AS IT POINTS TO THE RETURN ADDRESS.
.
.
.
; RETURN TO CALLER.
          MOVP   TP, [GC]
```

Figure 3-53. ASM-89 Procedure Example

```
CHANNEL1     SEGMENT     ; START OF SEGMENT
    .
    .
    .
    ASM89 SOURCE STATEMENTS
    .
    .
    .
CHANNEL1     ENDS        ; END OF SEGMENT
             END         ; END OF ASSEMBLY
```

Figure 3-54. ASM-89 SEGMENT and ENDS Directives

figure shows, the segment can then be located so that instructions and constants fall into the ROM portion of memory, while the variable part of the segment is located in RAM. The entire segment, including any "unused" portions, of course, cannot exceed 64k bytes.

Intermodule Communication

An ASM-89 module can make some of its addresses available to other modules by defining symbols with the PUBLIC directive. At a minimum, a channel program must make the address of its first instruction available to the CPU module that starts the channel program. Figure 3-56 shows an ASM-89 module that contains three channel programs labelled READ, WRITE and DELETE. The example shows how a PL/M-86 program and an ASM-86 program could define these "entry points" as EXTERNAL and EXTRN symbols respectively. When the modules are linked together, LINK-86 will match the externals with the publics, thus providing the CPU programs with the addresses they need.

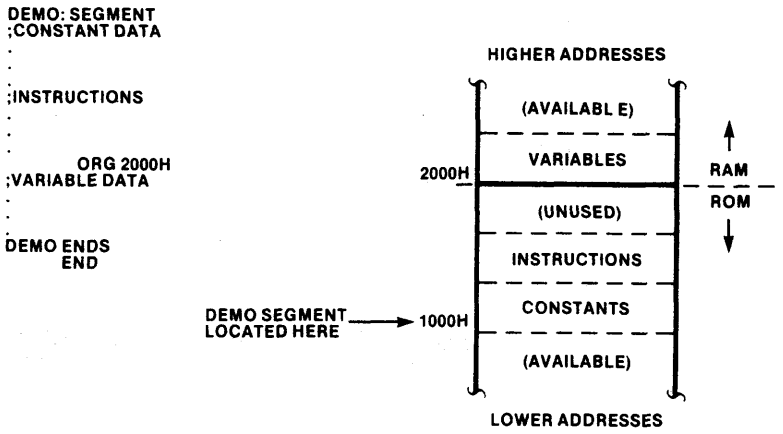


Figure 3-55. Using the ASM-89 ORG Directive

```
                ASM-89 MODULE DEFINES THREE PUBLIC SYMBOLS
.
.
.
PUBLIC      READ, WRITE, DELETE
.
.
.
READ:       ; ASM89 INSTRUCTIONS FOR "READ" OPERATION
.
.
            HLT
WRITE:      ; ASM89 INSTRUCTIONS FOR "WRITE" OPERATION
.
.
            HLT
DELETE:     ; ASM89 INSTRUCTIONS FOR "DELETE" OPERATION
.
.
            HLT
```

Figure 3-56. ASM-89 PUBLIC Directive

PL/M-86 MODULE USES "WRITE" SYMBOL

```
DECLARE   (READ,WRITE,DELETE) POINTER EXTERNAL;
DECLARE   PARM$BLOCK  STRUCTURE
          (TP$START        POINTER,
          BUFFER$ADDR       POINTER,
          BUFFER$LEN        WORD);
.
.
.
/*SET UP "WRITE" CHANNEL OPERATION*/
PARM$BLOCK. TP$START = WRITE;
.
.
.
```

ASM-86 MODULE USES "READ" SYMBOL

```
                .
                .
                .
EXTRN           READ,WRITE,DELETE
.
.
.
.
READ_PTR    DD  READ
WRITE_PTR   DD  WRITE
DELETE_PTR  DD  DELETE
.
.
.
; PARM_BLOCK
                EVEN        ; FORCE TO EVEN ADDRESS
TP_START    DD ?
BUFFER_ADDRDD ?
BUFFER_LEN  DW ?
.
.
.
; SET UP "READ" CHANNEL OPERATION
      MOV   AX, WORD PTR READ_PTR          ; 1ST WORD
      MOV   WORD PTR TP_START, AX
      MOV   AX, WORD PTR READ_PTR          ; 2ND WORD
      MOV   WORD PTR TP_START + 2, AX
      .
      .
      .
```

Figure 3-56. ASM-89 PUBLIC Directive (Cont'd.)

Conversely, an ASM-89 module can obtain the address of a public symbol in another module by defining it with the EXTRN directive. An external symbol, however, can only appear as the initial value operand of a DD directive (see figure 3-57). This effectively means that an ASM-89 program's use of external symbols is limited to obtaining the addresses of data located in the system space. Another way of doing this, which may be preferable in many cases, is to have the CPU program place system space addresses in the parameter block.

PL/M-86 PROGRAM DECLARES PUBLIC SYMBOL "BUFFER"

```
.
.
.
DECLARE BUFFER (80) BYTE PUBLIC;
.
.
.
```

ASM-89 PROGRAM OBTAINS ADDRESS OF PUBLIC SYMBOL "BUFFER"

```
.
.
.
EXTRN BUFFER
.
.
.
BUF_ADDRESS   DD   BUFFER
.
.
.
LPD   GA, BUF_ADDRESS   ; POINT TO SYSTEM BUFFER
.
.
.
```

Figure 3-57. ASM-89 EXTRN Directive

Sample Program

Figure 3-58 diagrams the logic of a simple ASM-89 program; the code is shown in figure 3-59. The program reads one physical record (sector) from a diskette drive controlled by an 8271 Floppy Disk Controller. No particular system configuration is implied by the program, except that the 8271 resides in the IOP's I/O space.

Hardware address decoding logic is assumed to be set up as follows:

- reading location FF00H selects the 8271 status register,
- writing location FF00H selects the 8271 command register,
- reading location FF01H selects the 8271 result register
- writing location FF01H selects the 8271 parameter register
- decoding the address FF04H provides the 8271 DACK (DMA acknowledge) signal.

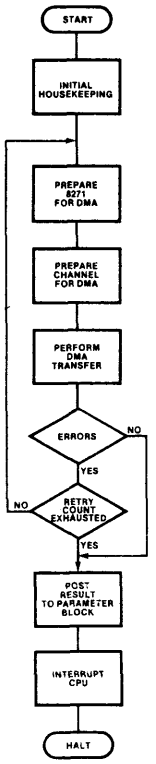


Figure 3-58. ASM-89 Sample Program Flow

The program uses structures to address the parameter block and the 8271 registers. Register PP contains the address of the parameter block, and the program loads GC with FF00H to point to the 8271 registers. The program's entry point (the label START) is defined as a PUBLIC symbol so that the CPU program can place its address in the parameter block when it starts the program.

Register IX is used as a retry counter. If the transfer is not completed successfully (bit 3 of the 8271 result register ≠ 0), the program retries the transfer up to 10 times.

Since the 8271 automatically requests a DMA transfer upon receipt of the last parameter, this parameter is sent immediately following the XFER command.

```
8089 ASSEMBLER


ISIS-II 8089 ASSEMBLER V1.0 ASSEMBLY OF MODULE FLOPPY
OBJECT MODULE PLACED IN :F0:FLOPPY.OBJ
ASSEMBLER INVOKED BY ASM89 FLOPPY.A89


                                          1
0000                                      2 FLOPPY            SEGMENT
                                          3 ;***
                                          4 ;*** 8089 PROGRAM TO READ SECTOR FROM FLOPPY DISK
                                          5 ;***
                                          6
                                          7 ;*** LAY OUT PARAMETER BLOCK.
                                          8 PARM BLOCK        STRUC
0000                                      9    RESERVED TP:      DS   4
0004                                     10    BUFF PTR:         DS   4
0008                                     11    TRACK:            DS   1
0009                                     12    SECTOR:           DS   1
000A                                     13    RETURN CODE:      DS   1
000B                                     14    PARM BLOCK    ENDS
                                         15
                                         16 ;***LAY OUT 8271 DEVICE REGISTERS.
                                         17 FLOPPY REGS       STRUC
0000                                     18    COMMAND STAT:     DS   1
0001                                     19    PARM RESULT:      DS   1
0002                                     20    FLOPPY REGS   ENDS
                                         21
                                         22 ;***8271 ADDRESSES.
  FF00                                   23 FLOPPY REG ADDR  EQU   OFF00H        ;LOW-ADDRESSED REGISTER
  FF04                                   24 DACK 8271        EQU   OFF04H        ;DMA ACKNOWLEDGE
                                         25
                                         26 ;***MAKE PROGRAM ENTRY POINT ADDRESS
                                         27 ;       AVAILABLE TO OTHER MODULES.
                                         28 PUBLIC           START
                                         29
                                         30 ;***CLEAR RETURN CODE IN PARAMETER BLOCK.
0000   0A4F 0A 00                        31 START:           MOVBI   [PP].RETURN CODE,0
                                         32
                                         33 ;***INITIALIZE RETRY COUNT.
0004   B130 0A00                        34                  MOVI    IX,10
                                         35
                                         36 ;***POINT GC AT LOW-ORDER 8271 REGISTER.
0008   5130 00FF                        37                  MOVI    GC,FLOPPY REG ADDR
                                         38
                                         39 ;***SEND COMMAND SEQUENCE TO 8271, HOLDING FINAL PARM.
                                         40 ;***WAIT UNTIL 8271 IS NOT BUSY.
000C   EABA 00 FC                       41 RETRY:           JNBT    [GC].COMMAND STAT,7,RETRY
                                         42 ;***SEND "READ SECTOR, DRIVE 0" COMMAND.
0010   0A4E 00 12                       43                  MOVBI   [GC].COMMAND STAT,012H
                                         44 ;***SEND TRACK ADDRESS PARAMETER.
0014   0293 08 02CE 01                  45                  MOVB    [GC].PARM RESULT,[PP].TRACK
                                         46
                                         47 ;***LOAD CHANNEL CONTROL REGISTER SPECIFYING:
                                         48 ;   FROM PORT TO MEMORY,
                                         49 ;   SYNCHRONIZE ON SOURCE,
                                         50 ;   GA POINTS TO SOURCE,
                                         51 ;   TERMINATE ON EXT,
                                         52 ;   TERMINATION OFFSET = 0.
001A   D130 2088                        53                  MOVI    CC,08820H
                                         54
```

Figure 3-59. ASM-89 Sample Program

```
001E   A000                                ;***SET SOURCE BUS = 8, DEST BUS = 16.
                                           WID     8,16

                                           ;***POINT GB AT DESTINATION, GA AT SOURCE.
0020   238B 04                             LPD     GB,[PP].BUFF_PTR
0023   1130 04FF                           MOVI    GA,DACK_8271

                                           ;***INSURE THAT 8271 IS READY FOR LAST PARAMETER.
0027   AABA 00 FC                  WAIT1:  JNBT    [GC].COMMAND_STAT,5,WAIT1

                                           ;***PREPARE FOR DMA.
002B   6000                                XFER

                                           ;***START DMA BY SENDING FINAL PARAMETER TO 8271.
002D   0293 09 02CE 01                     MOVB    [GC].PARM_RESULT,[PP].SECTOR

                                           ;***PROGRAM RESUMES HERE FOLLOWING EXT.

                                           ;***IF TRANSFER IS OK THEN EXIT, ELSE TRY AGAIN.
0033   6ABE 01 05                          JBT     [GC].PARM_RESULT,3,EXIT

                                           ;***DECREMENT RETRY COUNT.
0037   A03C                                DEC     IX

                                           ;***TRY AGAIN IF COUNT NOT EXHAUSTED.
0039   A840 D0                             JNZ     IX,RETRY

                                           ;***WAIT UNTIL 8271 IS NOT BUSY.
003C   EABA 00 FC                  EXIT:   JNBT    [GC].COMMAND_STAT,7,EXIT

                                           ;***SEND "READ RESULT" COMMAND TO 8271.
0040   0A4E 00 2C                          MOVBI   [GC].COMMAND_STAT,02CH

                                           ;***WAIT FOR RESULT.
0044   8ABA 00 FC                  WAIT2:  JNBT    [GC].COMMAND_STAT,4,WAIT2

                                           ;***POST RESULT IN PARAMETER BLOCK FOR CPU.
0048   0292 01 02CF 0A                     MOVB    [PP].RETURN_CODE,[GC].PARM_RESULT

                                           ;***INTERRUPT CPU.
004E   4000                                SINTR

                                           ;***STOP EXECUTION.
0050   2048                                HLT

0052                               FLOPPY  ENDS
                                           END
```

SYMBOL TABLE

DEFN	VALUE	TYPE	NAME
10	0004	SYM	BUFF_PTR
18	0000	SYM	COMMAND_STAT
24	FF04	SYM	DACK_8271
83	003C	SYM	EXIT
2	0000	SYM	FLOPPY
17	0000	STR	FLOPPY_REGS
23	FF00	SYM	FLOPPY_REG_ADDR
8	0000	STR	PARM_BLOCK
19	0001	SYM	PARM_RESULT
9	0000	SYM	RESERVED_TP
41	000C	SYM	RETRY
13	000A	SYM	RETURN_CODE
12	0009	SYM	SECTOR
31	0000	PUB	START
11	0008	SYM	TRACK
63	0027	SYM	WAIT1
89	0044	SYM	WAIT2

```
ASSEMBLY COMPLETE; NO ERRORS FOUND
```

Figure 3-59. ASM-89 Sample Program (Cont'd.)

Linking and Locating ASM-89 Modules

The LINK-86 utility program combines multiple relocatable object modules into a single relocatable module. The input modules may consist of modules produced by any of the 8086 family language translators: ASM-89, ASM-86, or PL/M-86. LINK-86's principal function is to satisfy external references made in the modules. Any symbol that is defined with the EXTRN directive in ASM-89 or ASM-86 or is declared EXTERNAL in PL/M-86 is an external reference, i.e., a reference to an address contained in another module. Whenever LINK-86 encounters an external reference, it searches the other modules for a PUBLIC symbol of the same name. If it finds the matching symbol, it replaces the external reference with the address of the object.

The most common occurrence of an external reference in a system that employs one or more 8089s is the channel program address. In order for a CPU program to start a channel program, it must ensure that the address of the first channel program instruction is contained in the first two words of the parameter block. Since the channel program is assembled separately, the translator that processes the CPU program will not typically know its address. If this address is defined as an external symbol (see figure 3-56), LINK-86 will obtain the address from the ASM-89 channel program when the two are linked together. (The ASM-89 program must, of course, define the symbol in a PUBLIC directive.)

Other external references may arise when one module uses data (e.g., a buffer) that is contained in another module, and (in PL/M-86 and ASM-86 modules) when one module executes another module, typically by a CALL statement or instruction.

When an 8089 module (or modules) is to be located in the system space, it may be linked together with PL/M-86 or ASM-86 modules as described above and shown in figure 3-60. LINK-86 resolves external references and combines the input modules into a single relocatable object module. This module can be input to LOC-86 (LOC-86 assigns final absolute memory addresses to all of the instructions and data). This absolute object module may, in turn, be processed by the OH-86 utility to translate the module into the hexadecimal format. This format makes the module readable (the records are written in ASCII characters) and is required by some PROM programmers and RAM loaders. Intel's Universal PROM Programmer (UPP) and iSBC 957™ Execution Package (loader) use the hexadecimal format.

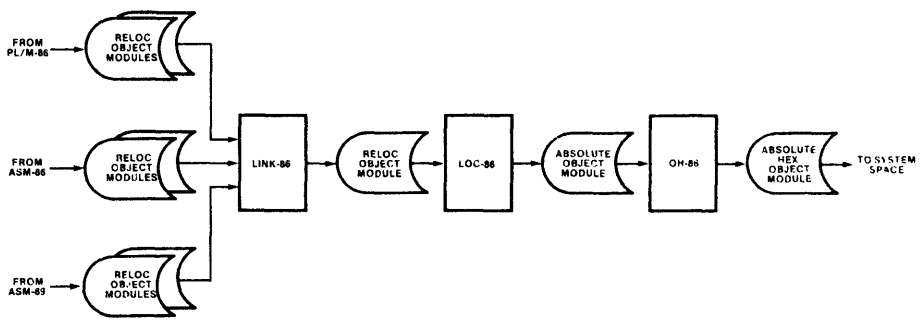


Figure 3-60. Creating a Single Absolute Object Module

If the 8089 code is to reside in its I/O space, a different technique is required since separate absolute object modules must be produced for the system and I/O spaces. Figure 3-61 shows how to link and locate when there are external references between I/O space modules and system space modules.

The normal link and locate sequence is followed and culminates in the production of an absolute module in hexadecimal format. Since the records in this file are human-readable, the file can be edited using the ISIS-II text editor. The editing task involves finding the 8089 I/O space records in the file, writing them to one file, and then writing the 8086/8088 records (destined for the system space) to another file. *MCS-86 Absolute Object File Formats*, Order No. 9800921, available from Intel's Literature Department, describes the records in absolute (including hexadecimal) object modules.

When using the previous method, it is likely that LOC-86 will issue messages warning that segments overlap. For example, the 8089 code would typically be located starting at absolute location 0H of the I/O space. However, the 8086/8088 interrupt pointer table occupies these low memory addresses in the system space. Since LOC-86 has no way to know that the segment will ultimately be located in different address spaces, it will warn of the conflict; the warning may be ignored.

An alternative to linking the modules together and then separating them is to link system space modules separately from I/O space modules as shown in figure 3-62. This approach avoids the manual edit of the absolute object module and the segment conflict messages from LOC-86. It requires, however, that modules in the two spaces not use the EXTRN/PUBLIC mechanism to refer to each other. Modules in the same space can define external and public symbols, however.

External references from I/O space modules to system space modules can be eliminated if the CPU programs pass all system space addresses in parameter blocks. In other words, a channel program can obtain any address in the system space if the address is in the parameter block. Using this approach allows the system space addresses to be changed during execution. If the addresses are constant values, they may also be altered as system development proceeds without relinking the channel programs.

External references from system space modules to addresses in the I/O space may be eliminated by assigning these addresses values that are known at assembly or compilation time. Figure 3-63 illustrates how the ASM-89 ORG directive can be used to force the first instruction (entry point) of a channel program to an absolute address. In the case of the example, one module contains two entry points labelled "READ" and "WRITE." Assuming the module is located at absolute address 0H in the I/O space, the channel programs will begin at 200H and 600H respectively. In the example, these values have been chosen arbitrarily; in a typical application they would be based on the length of the programs and the location of RAM and ROM areas. By starting the programs at fixed addresses that are known to the CPU programs that activate them, the channel programs can be reassembled without needing to relink the CPU programs.

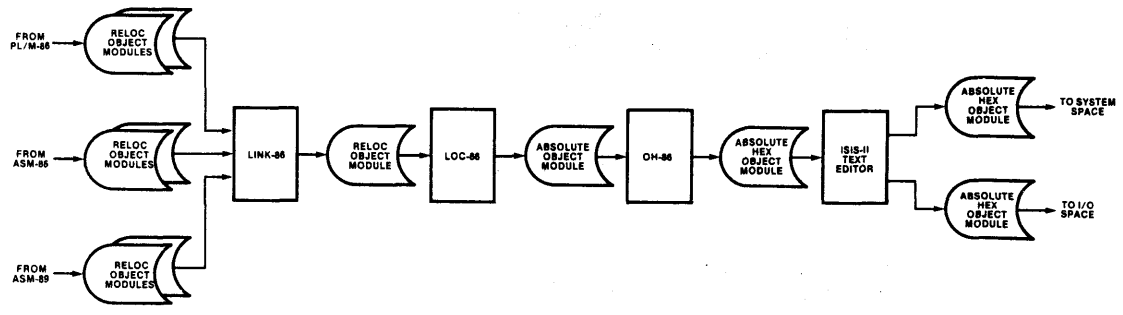


Figure 3-61. Creating Separate Absolute Object Modules—External References in Relocatable Modules

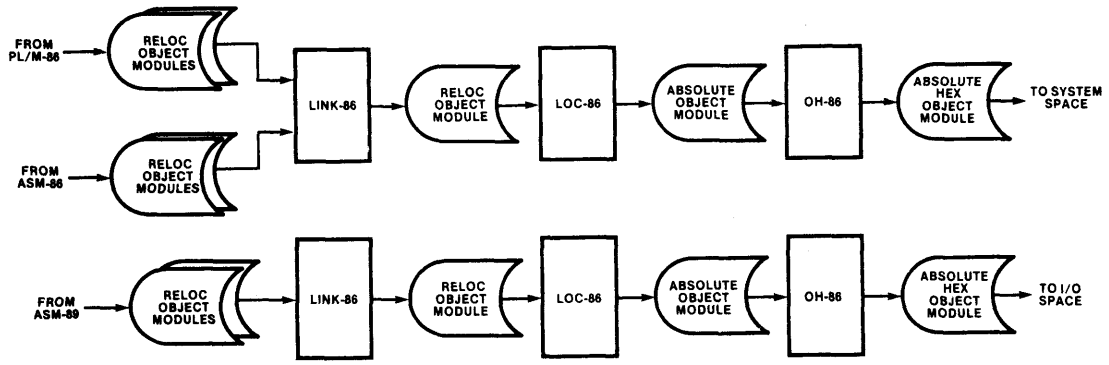


Figure 3-62. Creating Separate Absolute Object Modules—No External References in Relocatable Modules

```
ASM-89 ENTRY POINT DEFINITIONS
.
.
.
        ORG 200H
READ:
.
.
.
; INSTRUCTIONS FOR "READ" CHANNEL PROGRAM
.
.
.
        ORG 600H
WRITE:
.
.
.
; INSTRUCTIONS FOR "WRITE" CHANNEL PROGRAM
.
.
.

ASM-86 DEFINITION OF ENTRY POINT ADDRESSES
.
.
.
READ_ADDR       DD  200H
WRITE_ADDR      DD  600H
.
.
.

PL/M-86 DECLARATION OF ENTRY POINT ADDRESSES
.
.
.
DECLARE READ$ADDR POINTER;
DECLARE WRITE$ADDR POINTER;
READ$ADDR = 200H;
WRITE$ADDR = 600H;
```

Figure 3-63. Using Absolute Entry Point Addresses

3.10 Programming Guidelines and Examples

This section provides two types of 8089 programming information. A series of general guidelines, which apply to system and program design, is presented first. These guidelines are followed by specific coding examples that illustrate programming techniques that may be applied to many different types of applications.

Programming Guidelines

The practices in this section are recommended to simplify system development and, particularly, for system maintenance and enhancement. Software that is designed in accordance with these **guidelines will be adaptable to the changing environment in which most systems operate, and will be in the best position to take advantage of new Intel hardware and software products.**

Segments

Although the IOP does not "see" the segmented organization of system memory, it should respect this logical structure. The IOP should only address the system space through pointers passed by the CPU in the parameter block. It should not perform arithmetic on these addresses or otherwise manipulate them except for the automatic incrementing that occurs during DMA transfers. It is the responsibility of the CPU to pass addresses such that transfer operations do not cross segment boundaries.

Self-Modifying Code

Programs that alter their own instructions are difficult to understand and modify, and preclude placing the code in ROM. They may also inhibit compatibility with future Intel hardware and software products.

Note also that when the 8089 is on a 16-bit bus, its instruction fetch queue can interfere with the attempt of one instruction to modify the next sequential instruction. Although the instruction may be changed in memory, its unmodified first byte will be fetched from the queue rather than memory if it is on an odd address. The processor will thus execute a partially-modified instruction with unpredictable results.

I/O System Design

Section 2.10 notes that I/O systems should be designed hierarchically. Application programs "see" only the topmost level of the structure; all details pertaining to the physical characteristics and operation of I/O devices are relegated to lower levels. Figure 3-64 shows how this design approach might be employed in a system that uses an 8089 to perform I/O. The same concept can be expanded to larger systems with multiple IOPs.

The application system is clearly separated from the I/O system. No application programs perform I/O; instead they send an I/O request to the I/O supervisor. (In systems with file-oriented I/O, the request might be sent to a file system that would then invoke the I/O supervisor.) The I/O request should be expressed in terms of a logical block of data—a record, a line, a message, etc. It should also be devoid of any device-dependent information such as device address, sector size, etc.

The I/O supervisor transforms the application program's request for service into a parameter block and dispatches a channel program to carry out the operation. The I/O supervisor controls the channels; therefore, it knows the correspondence between channels and I/O devices, the locations of CBs and channel programs, and the format of all of the parameter blocks. The I/O supervisor also coordinates channel "events," monitoring BUSY flags and responding to channel-generated interrupt requests. The I/O supervisor does not, however, communicate with I/O devices that are controlled by the channels. If the CPU performs some I/O itself (this should be restricted to devices other than those run by the channels), the I/O supervisor invokes the equivalent of a channel program in the CPU to do the physical I/O. Note that although the I/O supervisor is drawn as a single box in figure 3-64, it is likely to be structured as a hierarchy itself, with separate modules performing its many functions.

The software interface between the CPU's I/O supervisor and an IOP channel program should be completely and explicitly defined in the

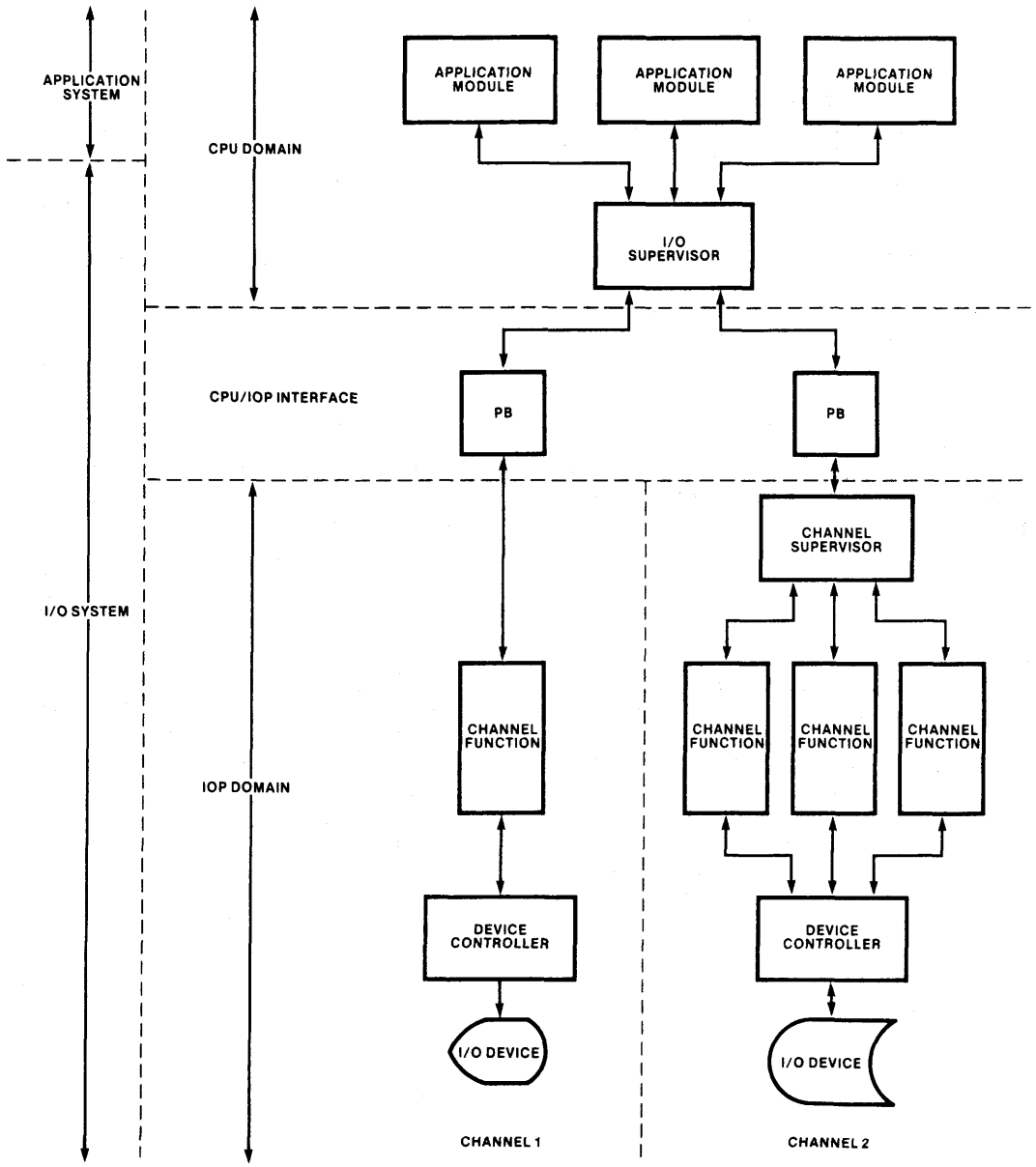


Figure 3-64. 8089-Based I/O System Design

parameter block. For example, the I/O supervisor should pass the addresses of all system memory areas that the channel program will use. The channel program should not be written so that it "knows" any of these addresses, even if they are constants. Concentrating the interface into one place like this makes the system easier to understand and reduces the likelihood of an undesirable side effect if it is modified. It also generalizes the design so that it may be used in other application systems.

Figure 3-64 shows a simple channel program running on channel 1 and a more complex program running on channel 2. Channel 1's program performs a single function and is therefore designed as a simple program. The program on channel 2 performs three functions (e.g., "read," "write," "delete") and is structured to separate its functions. The functions might be implemented as procedures called by the "channel supervisor" depending on the content of the parameter block. Notice that to the I/O supervisor, both programs appear alike; in particular, both have a single entry point.

In some channel programs, different functions will need different information passed to them in the parameter block. Figure 3-65 shows one technique that accommodates different formats while still allowing the channel supervisor to determine which procedure to call from the PB. The parameter block is divided into fixed and variable portions, and a function code in the fixed area indicates the type of operation that is to be performed. Part of the fixed area has been set aside so that additional parameters can be added in the future.

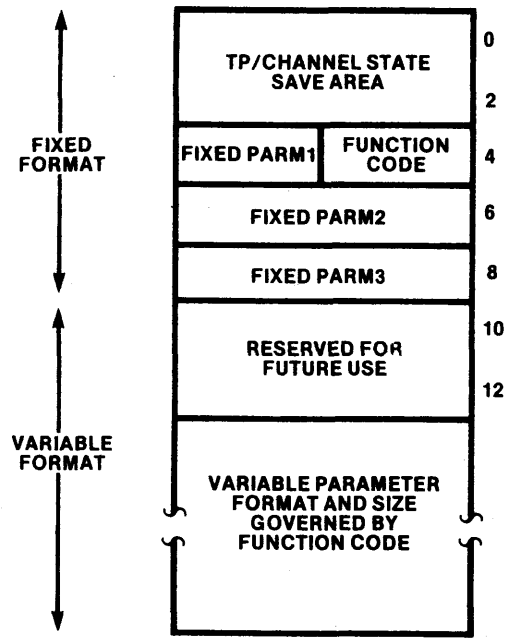


Figure 3-65. Variable Format Parameter Block

Programming Examples

The first example in this section illustrates how a CPU can initialize a group of IOPs and then dispatch channel programs. This code is written in PL/M-86.

The remaining examples, written in ASM-89, demonstrate the 8089 instruction set and addressing modes in various commonly-encountered programming situations. These include:

- memory-to-memory transfers
- saving and restoring registers

Initialization and Dispatch

The PL/M-86 code in figure 3-66 initializes two IOPs and dispatches two channel programs on one of the IOPs. The same general technique can be used to initialize any number of IOPs. The hypothetical system that this code runs on is configured as follows:

- 8086 CPU (16-bit system bus);
- two remote IOPs share an 8-bit local I/O bus via the request/grant lines operating in mode 1;
- 8089 channel attentions are mapped into four port addresses in the CPU's I/O space;
- channel programs reside in the 8089 I/O space;
- one 8089 controls a CRT terminal, one channel running the display, the other scanning the keyboard and building input messages;
- the function of the second 8089 is not defined in the example.

The code declares one CB (channel control block) for each 8089. The CBs are declared as two-element arrays, each element defining the structure of one channel's portion of the CB. The SCB (system configuration block) and SCP (system configuration pointer) are also declared as structures. The SCP is located at its dedicated system space address of FFFF6H. The other structures are not located at specific addresses since they are all linked together by a chain of pointers "anchored" at the SCP.

Two simple parameter blocks define messages to be transmitted between the PL/M-86 program and the CRT. Each PB contains a pointer to the beginning of the message area and the length of the message. In the case of the keyboard (input) message, the channel program builds the message in the buffer pointed to by the pointer in the PB and returns the length of the message in the PB.

The code initializes one IOP at a time since the chain of control blocks read by the IOP during initialization must remain static until the process is complete. To initialize the first IOP, the code fills in the SYSBUS and SOC fields and links the blocks to each other using the PL/M-86 @ (address) operator. It sets channel 1's BUSY flag to FFH so that it can monitor the flag to determine when the initialization has been completed (the IOP clears the flag to 0H when it has finished). Channel 2's BUSY flag is cleared, although this could just as well have been done after the initialization (the IOP does not alter channel 2's BUSY flag during initialization). The code starts the IOP by issuing a channel attention to channel 1 to indicate that the IOP is a bus master. PL/M-86's OUT function is used to select the port address to which the IOP's CA and SEL lines have been mapped. The data placed on the bus (0H) is ignored by the IOP. It then waits until the IOP clears the channel 1 BUSY flag.

The second IOP is initialized in the same manner, first changing the pointer in the SCB to point to the second IOP's channel control block. If this IOP were on a different I/O bus, the SOC field would have been altered if a different request/grant mode were being used or if the IOP had a 16-bit I/O bus. The second IOP is a slave so its initialization is started by issuing a CA to channel 2 rather than channel 1.

After both IOPs are ready, the code dispatches two channel programs (not coded in the example); one program is dispatched to each channel of one of the IOPs. To avoid external references, the system has been set up so that the PL/M-86 code "knows" the starting addresses of these channel programs (200H and 600H). The code uses the PL/M-86 LOCKSET function to:

- lock the system bus;
- read the BUSY flag;
- set the BUSY flag to FFH if it is clear;
- unlock the system bus.

This operation continues until the BUSY flag is found to be clear (indicating that the channel is available). Setting the flag immediately to FFH prevents another processor (or another task in this program activated as a result of an interrupt) from using the channel. The code fills in the parameter block with the address and length of the message to be displayed, sets the CCW and then links the channel program (task block) start address to the parameter block and links the parameter block to the CB. The channel is dispatched with the OUT function that effects a channel attention for channel 1.

A similar procedure is followed to start channel 2 scanning the terminal keyboard. In this case, the code allows channel 2 to generate an interrupt request (which it might do to signal that a message has been assembled). An interrupt procedure would then handle the interrupt request.

```
/*ASSIGN NAMES TO CONSTANTS*/
DECLARE     CHANNEL$BUSY        LITERALLY '0FFH';
DECLARE     CHANNEL$CLEAR       LITERALLY '0H';
DECLARE     CR /*CARR. RET.*/   LITERALLY '0DH';
DECLARE     LF /*LINE FEED*/    LITERALLY '0AH';
DECLARE     DISPLAY$TB          LITERALLY '200H';
DECLARE     KEYBD$TB            LITERALLY '600H';
```

Figure 3-66. Initialization and Dispatch Example

```
DECLARE  /*IOP CHANNEL ATTENTION ADDRESSES*/
IOP$A$CH1      LITERALLY          '0FFE0H',
IOP$A$CH2      LITERALLY          '0FFE1H',
IOP$B$CH1      LITERALLY          '0FFE2H',
IOP$B$CH2      LITERALLY          '0FFE3H';

DECLARE  /*CHANNEL CONTROL BLOCK FOR IOP$A)
           CB$A(2)          STRUCTURE
           (BUSY            BYTE,
           CCW              BYTE,
           PB$PTR           POINTER,
           RESERVED         WORD);

DECLARE  /*CHANNEL CONTROL BLOCK FOR IOP$B*/
           CB$B(2)          STRUCTURE
           (BUSY            BYTE,
           CCW              BYTE,
           PB$PTR           POINTER,
           RESERVED         WORD);

DECLARE  /*SYSTEM CONFIGURATION BLOCK*/
           SCB              STRUCTURE
           (SOC             BYTE,
           RESERVED         BYTE,
           CB$PTR           POINTER);

DECLARE  /*SYSTEM CONFIGURATION POINTER*/
           SCP              STRUCTURE
           (SYSBUS          BYTE,
           SCB$PTR          POINTER) AT (0FFFF6H);

DECLARE  MESSAGE$PB STRUCTURE
           (TB$PTR          POINTER,
           MSG$PTR          POINTER,
           MSG$LENGTH       WORD);

DECLARE  KEYBD$PB STRUCTUE
           (TP$PTR          POINTER,
           BUFF_PTR         POINTER,
           MSG$SIZE         WORD);

DECLARE  SIGN$ON BYTE (*) DATA
         (CR, LF, 'PLEASE ENTER USER ID');

DECLARE  KEYBD$BUFF BYTE (256);

/*
 *INITIALIZE IOP$A, THEN IOP$B
*/

/*PREPARE CONTROL BLOCKS FOR IOP$A*/
SCP.SCB$PTR = @ SCB;
SCP.SYSBUS = 01H; /*16-BIT SYSTEM BUS*/
SCB.SOC = 02H; /*RQ/GT MODE1, 8-BIT I/O BUS*/
SCB.CB$PTR = @ CB$A(0);
CB$A(0).BUSY = CHANNEL$BUSY
CB$A(1).BUSY = CHANNEL$CLEAR;
```

Figure 3-66. Initialization and Dispatch Example (Cont'd.)

```
/*ISSUE CA FOR CHANNEL1, INDICATING IOP IS MASTER*/
OUT (IOP$A$CH1) = 0H;

/*WAIT UNTIL FINISHED*/
DO WHILE CB$A(0).BUSY = CHANNEL$BUSY;
    END;

/*PREPARE CONTROL BLOCKS FOR IOP$B*/
SCB.CB$PTR = @CB$B(0);
CB$B(0).BUSY = CHANNEL$BUSY;
CB$B(1).BUSY = CHANNEL$CLEAR;

/*ISSUE CA FOR CHANNEL2, INDICATING SLAVE STATUS*/
OUT (IOP$B$CH2) = 0H;

/*WAIT UNTIL IOP IS READY*/
DO WHILE CB$B(0).BUSY = CHANNEL$BUSY;
    END;



/*
 *SEND SIGN ON MESSAGE TO CRT CONTROLLED
 *BY CHANNEL 1 OF IOP$A
 */
/*WAIT UNTIL CHANNEL IS CLEAR, THEN SET TO BUSY*/
DO WHILE LOCKSET (@CB$A(0).BUSY, CHANNEL$BUSY);
    END;

/*SET CCW AS FOLLOWS:
 *      PRIORITY = 1,
 *      NO BUS LOAD LIMIT,
 *      DISABLE INTERRUPTS,
 *      START CHANNEL PROGRAM IN I/O SPACE*/
CB$A(0).CCW = 10011001B;

/*LINK MESSAGE PARAMETER BLOCK TO CB*/
CB$A(0).PB$PTR = @ MESSAGE$PB;

/*FILL IN PARAMETER BLOCK*/
MESSAGE$PB.TB$PTR = DISPLAY$TB;
MESSAGE$PB.MSG$PTR = @SIGN$ON;
MESSAGE$PB. MSB$LENGTH = LENGTH (SIGN$ON);

/*DISPATCH THE CHANNEL*/
OUT (IOP$A$CH1) = 0H;

/*
 *DISPATCH CHANNEL 2 OF IOP$A TO
 *CONTINUOUSLY SCAN KEYBOARD, INTERRUPTING
 *WHEN A COMPLETE MESSAGE IS READY
 */
/*WAIT UNTIL CHANNEL IS CLEAR, THEN SET TO BUSY*/
DO WHILE LOCKSET (@ CB$A(1).BUSY, CHANNEL$BUSY);
    END;
```

Figure 3-66. Initialization and Dispatch Example (Cont'd.)

```
/*SET CCW AS FOLLOWS:
 *      PRIORITY = 0
 *      BUS LOAD LIMIT,
 *      ENABLE INTERRUPTS,
 *      START CHANNEL PROGRAM IN I/O SPACE*/
CB$A(1).CCW = 00110001B;
/*LINK KEYBOARD PARAMETER BLOCK TO CB*/
CB$A(1).PB$PTR = @ KEYBD$PB;
/*FILL IN PARAMETER BLOCK*/
KEYBD$PB.TB$PTR = KEYBD$TB;
KEYBD$PB.BUFF$PTR = @ KEYBD$BUFF;
KEYBD$PB.MSG$SIZE = 0H;
/*DISPATCH THE CHANNEL*/
OUT (IOP$A$CH2) = 0H;
```

Figure 3-66. Initialization and Dispatch Example (Cont'd.)

Memory-to-Memory Transfer

Figure 3-67 shows a channel program that performs a memory-to-memory block transfer in seven instructions. The program moves up to 64k bytes between any two locations in the system space. A 16-bit system bus is assumed, and the CPU is assumed to be monitoring the channel's BUSY flag to determine when the program has finished.

To attain maximum transfer speed, the program locks the bus during each transfer cycle. This ensures that another processor does not acquire the bus in the interval between the DMA fetch and store operations. By setting this channel's priority bit in the CCW to 1 and the other channel's to 0, the CPU could effectively prevent the other channel from running during the transfer. Byte count termination is selected so that the transfer will stop when the number of bytes specified by the CPU has been moved. Since there is only a single termination condition, a termination offset of 0 is specified. The transfer begins after the WID instruction, and the HLT instruction is executed immediately upon termination.

Saving and Restoring Registers

A CPU program can "interrupt" a channel program by issuing a "suspend" channel command. The channel responds to this command by saving the task pointer and PSW in the first two words of the parameter block. The suspended program can be restarted by issuing a "resume" command that loads TP and the PSW from the save area.

If the CPU wants to execute another channel program between the suspend and resume operations, the suspended program's registers will usually have to be saved first. If the "interrupting" program "knows" that the registers must be saved, it can perform the operation and also restore the registers before it halts.

A more general solution is shown in figure 3-68. This is a program that does nothing but save the contents of the channel registers. The registers are saved in the parameter block because PP is the only register that is known to point to an available area of memory. A similar program could be written to restore registers from the same parameter block.

Using this approach, the CPU would "interrupt" a running program as follows:

- suspend the running program,
- run the register save program,
- run the "interrupting" program,
- run the register restore program,
- resume the suspended program.

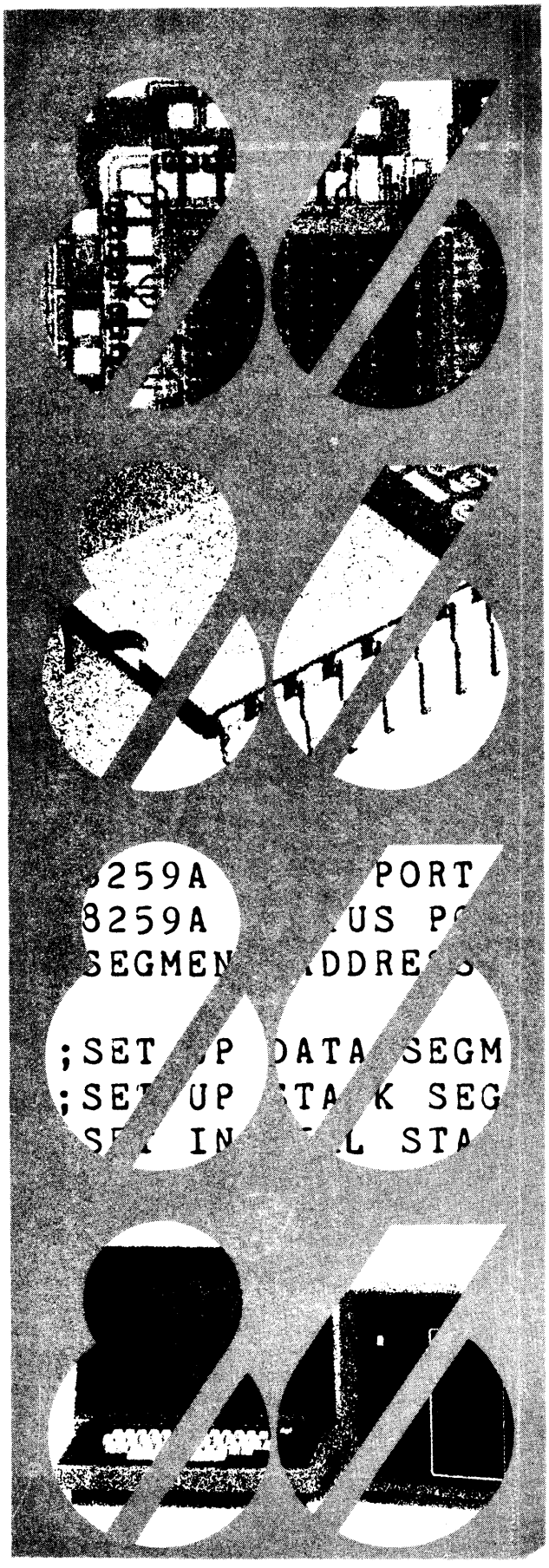
```
MEMEXAMP        SEGMENT
;**MEMORY-TO-MEMORY TRANSFER PROGRAM**
PB              STRUC
TP_RESERVED:    DS      4
FROM_ADDR:      DS      4
TO_ADDR:        DS      4
SIZE:           DS      2
PB              ENDS

;POINT GA AT SOURCE, GB AT DESTINATION.
                LPD             GA, [PP].FROM_ADDR
                LPD             GB, [PP].TO_ADDR
;LOAD BYTE COUNT INTO BC.
                MOV             BC, [PP].SIZE
;LOAD CC SPECIFYING:
;       MEMORY TO MEMORY,
;       NO TRANSLATE,
;       UNSYNCHRONIZED,
;       GA POINTS TO SOURCE,
;       LOCK BUS DURING TRANSFER,
;       NO CHAINING,
;       TERMINATING ON BYTE COUNT,OFFSET = 0.
                MOV             CC, 0C208H
;PREPARE CHANNEL FOR TRANSFER.
                XFER

;SET LOGICAL BUS WIDTH.
                WID             16,16

;STOP EXECUTION AFTER DMA.
                HLT
MEMEXAMP        ENDS
                END
```

Figure 3-67. Memory-to-Memory Transfer Example

```
SAVEREGS        SEGMENT
;SAVE ANOTHER CHANNEL'S REGISTERS IN PB
PB              STRUC
TP_RESERVED:    DS      4
GA_SAVE:        DS      3
GB_SAVE:        DS      3
GC_SAVE:        DS      3
IX_SAVE:        DS      2
BC_SAVE:        DS      2
MC_SAVE:        DS      2
CC_SAVE:        DS      2
PB              ENDS

                MOVP    [PP].GA_SAVE, GA
                MOVP    [PP].GB_SAVE, GB
                MOVP    [PP].GC_SAVE, GC
                MOV     [PP].IX_SAVE, IX
                MOV     [PP].BC_SAVE, BC
                MOV     [PP].MC_SAVE, MC
                MOV     [PP].CC_SAVE, CC
                HLT
SAVEREGS        ENDS
                END
```

Figure 3-68. Register Save Example

8087 Numeric Data Processor



THE 8087 NUMERIC DATA PROCESSOR

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

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

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

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

S.1 Processor Overview

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

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

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

Evolution

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

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

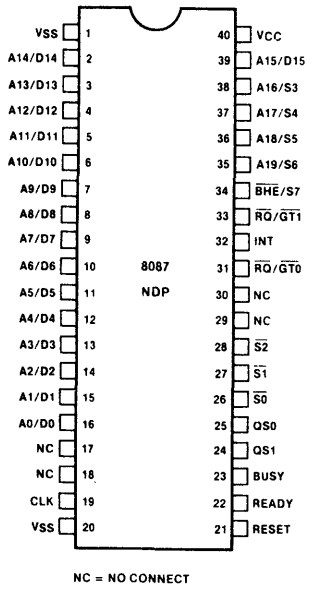


Figure S-1. 8087 Numeric Data Processor Pin Diagram

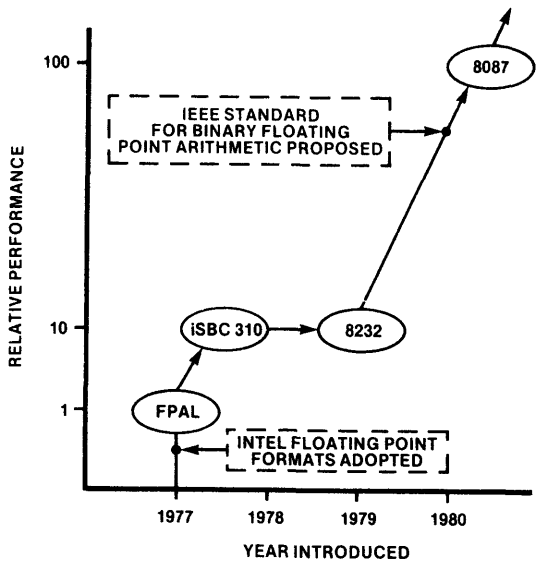


Figure S-2. 8087 Evolution and Relative Performance

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

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

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

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

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

Performance

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

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

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

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

Usability

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

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

Table S-1. 8087 Emulator Speed Comparison

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

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

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

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

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

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

Applications

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

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

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

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

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

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

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

Programming Interface

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

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

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

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

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

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

Table S-2. Data Types

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

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

Table S-3. Principal Instructions

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

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

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

Hardware Interface

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

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

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

S.2 Processor Architecture

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

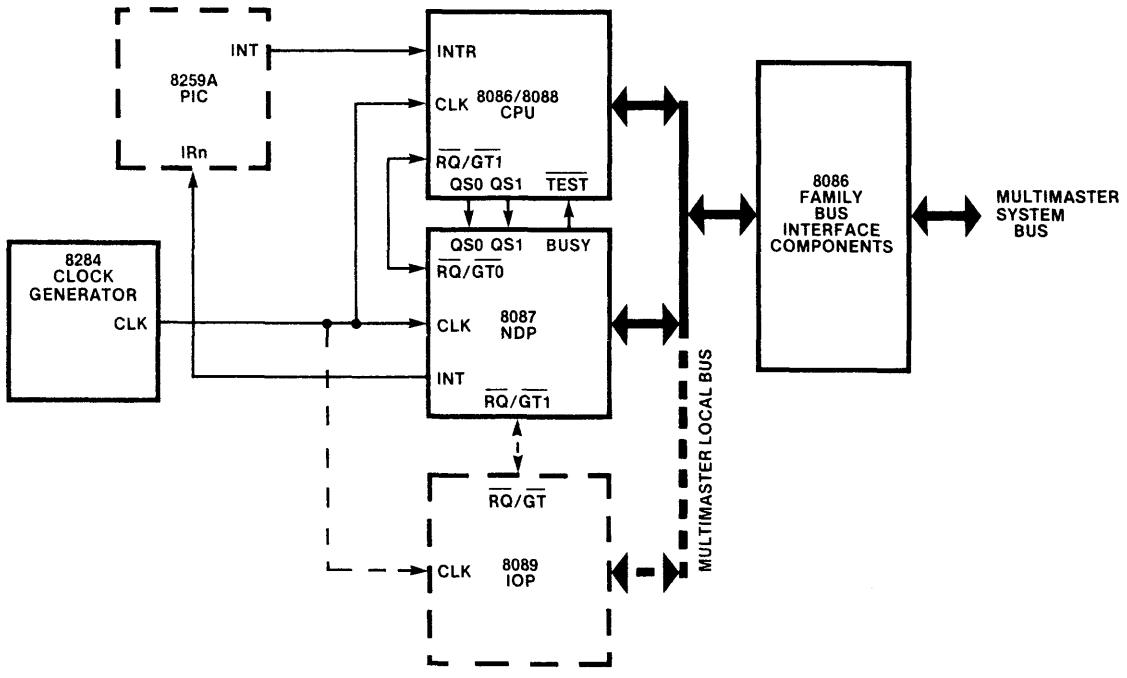


Figure S-3. NDP Interconnect

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

Control Unit

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

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

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

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

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

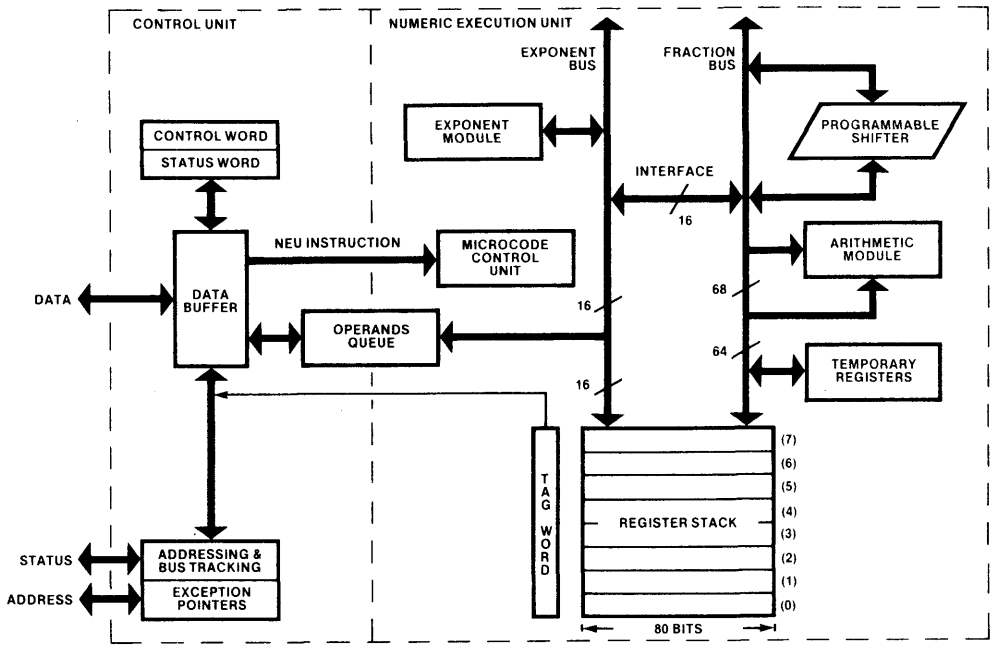


Figure S-4. 8087 Block Diagram

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

Numeric Execution Unit

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

Register Stack

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

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

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

Figure S-5. Register Structure

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

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

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

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

Status Word

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

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

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

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

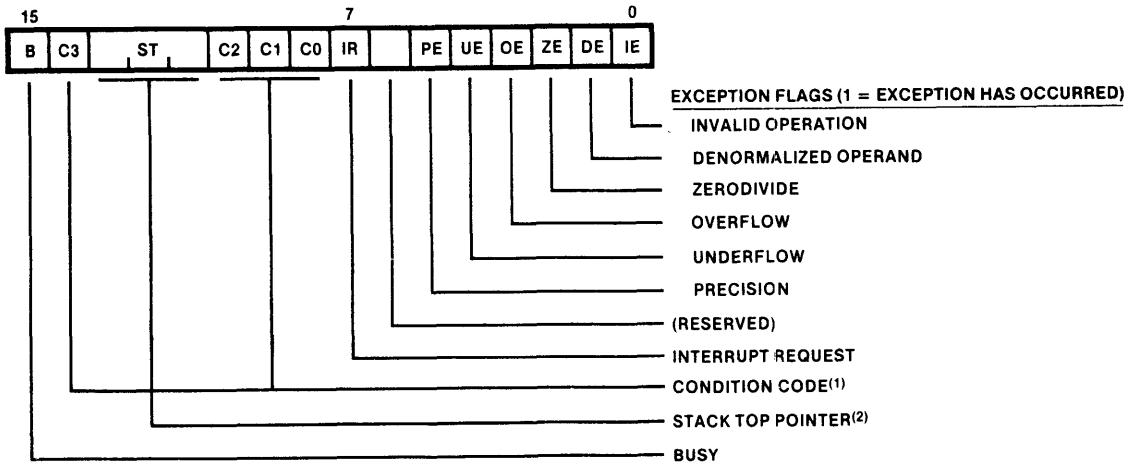
Bits 5-0 are set to indicate that the NEU has detected an exception while executing an instruction. Section S.3 explains these exceptions.

Control Word

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

Tag Word

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



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

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

Figure S-6. Status Word Format

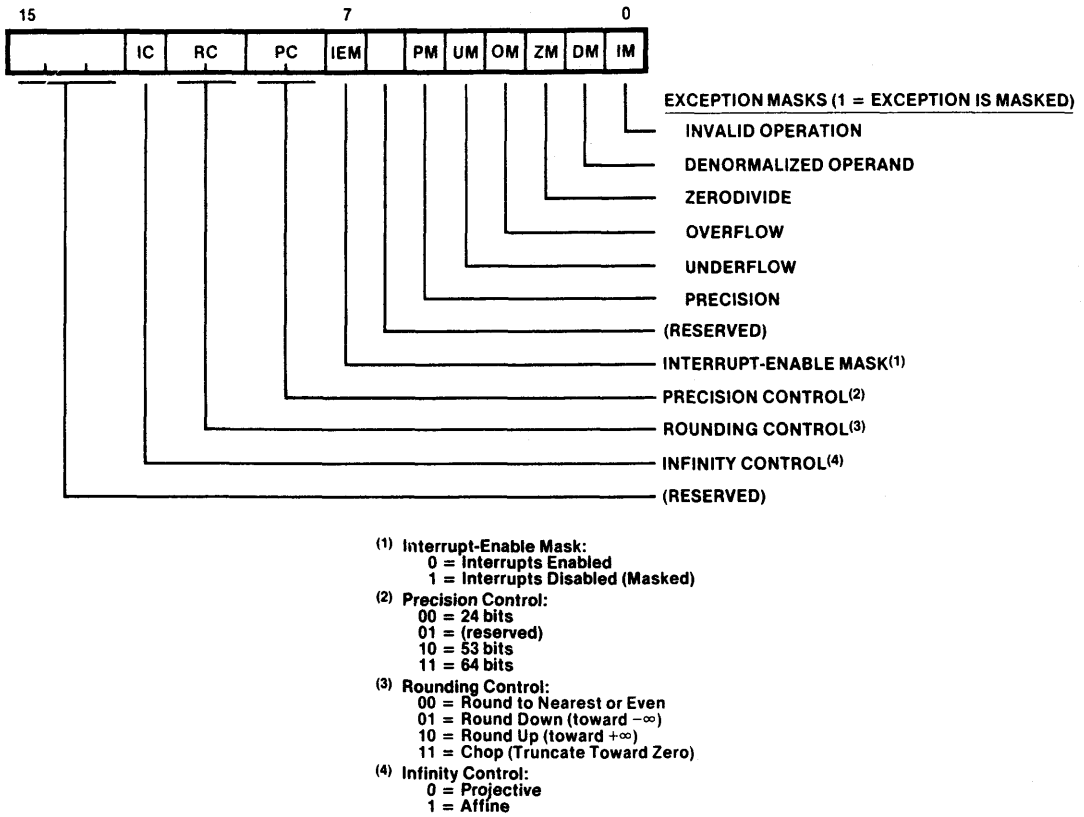


Figure S-7. Control Word Format

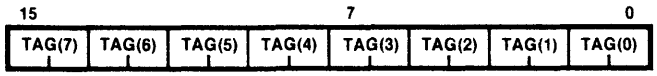
of the tag word is to optimize the NDP's performance under certain circumstances and programmers ordinarily need not be concerned with it.

Exception Pointers

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

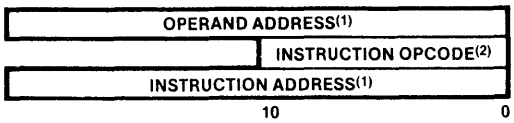
S.3 Computation Fundamentals

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



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

Figure S-8. Tag Word Format



(1) 20-bit physical address

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

Figure S-9. Exception Pointers Format

Number System

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

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

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

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

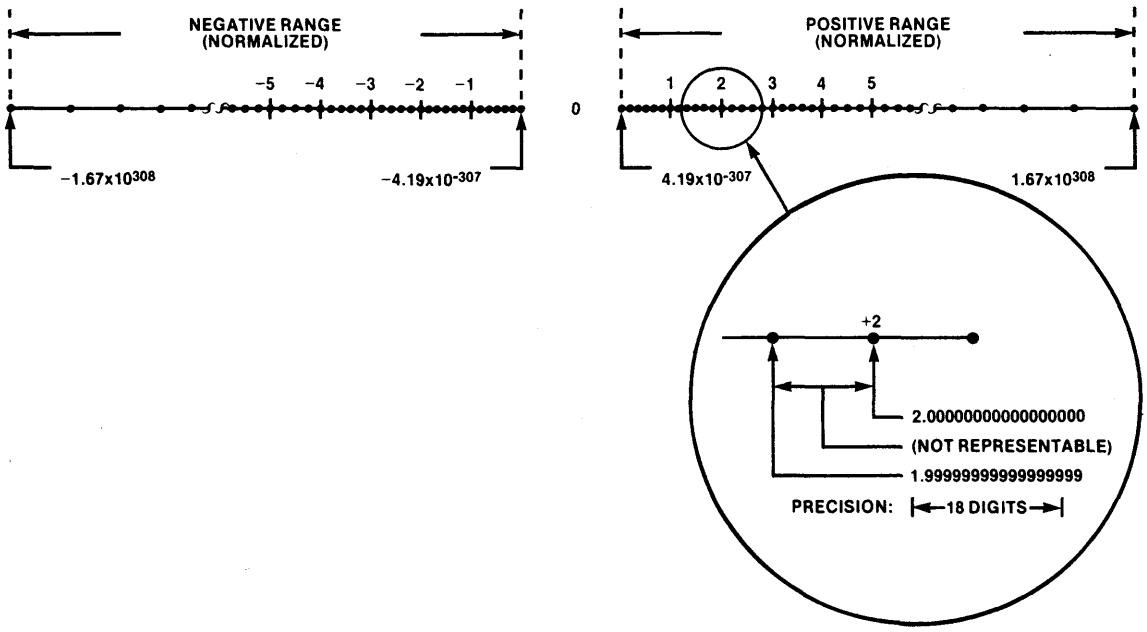


Figure S-10. 8087 Number System

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

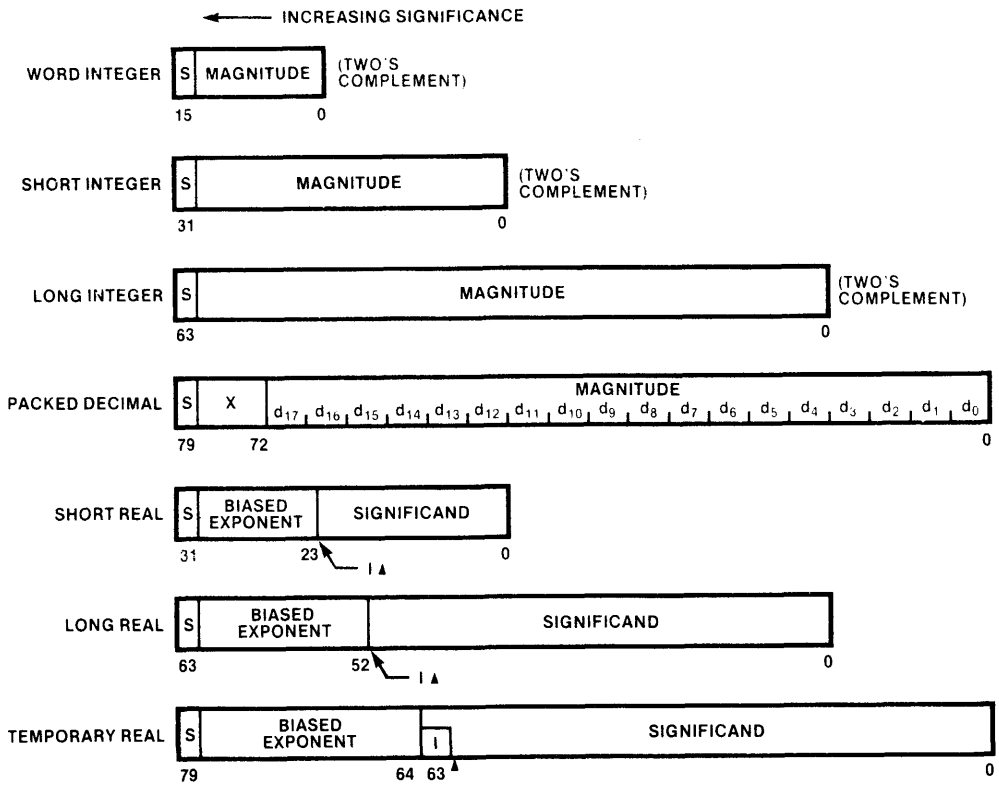
In its internal operations, the 8087 actually employs a number system that is a substantial superset of that shown in figure S-10. The internal format (called temporary real) extends the 8087's range to about $\pm 3.4 \times 10^{-4932}$ to $\pm 1.2 \times 10^{4932}$, and its precision to about 19 (equivalent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

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

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

Data Types and Formats

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



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

Figure S-11. Data Formats

Binary Integers

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

Decimal Integers

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

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

Real Numbers

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

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

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

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

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

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

Table S-4. Real Number Notation

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

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

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

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

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

Special Values

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

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

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

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

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

Table S-5. Rounding Modes

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

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

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

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

Rounding Control

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

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

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

Precision Control

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

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

Infinity Control

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

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

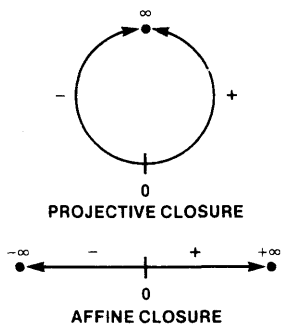


Figure S-12. Projective Versus Affine Closure

Exceptions

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

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

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

An invalid operation generally indicates a program error.

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

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

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

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

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

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

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

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

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

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

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

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

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

Table S-6. Exception and Response Summary

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

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

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

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

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

Possible "application-dependent actions" include:

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

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

S.4 Memory

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

Data Storage

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

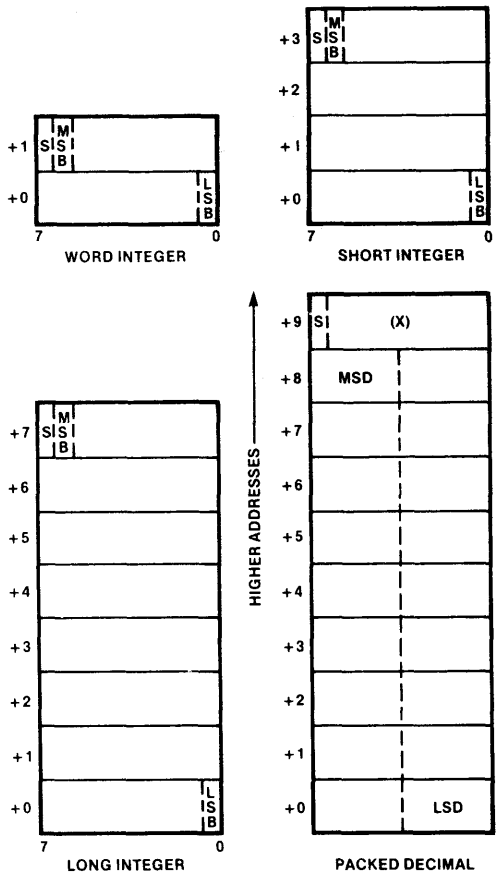


Figure S-13. Storage of Integer Data Types

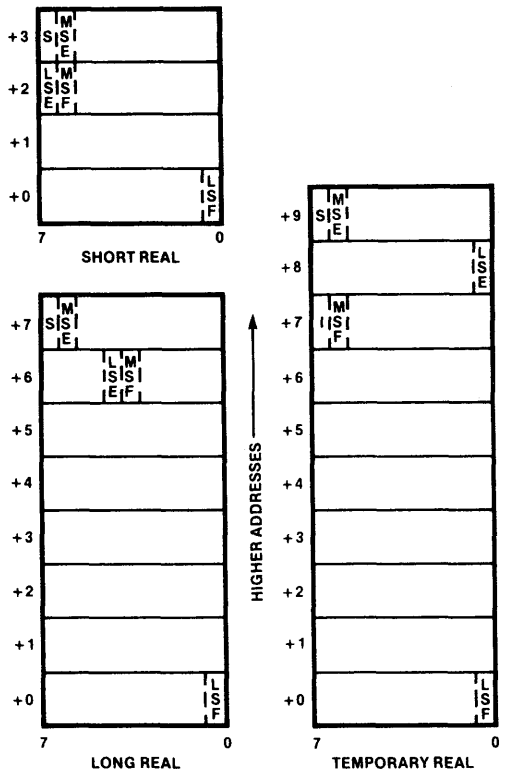


Figure S-14. Storage of Real Data Types

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

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

Storage Access

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

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

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

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

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

Dynamic Relocation

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

Dedicated and Reserved Memory Locations

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

S.5 Multiprocessing Features

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

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

Instruction Synchronization

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

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

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

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

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

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

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

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

```
WAIT
FMUL
WAIT
FDIV
```

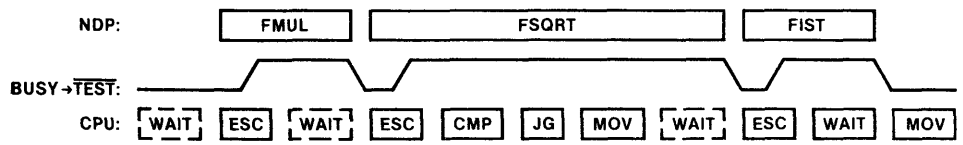
This ensures that the multiply runs to completion before the CPU and the 8087 CU decode the divide.

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

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

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

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



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

Figure S-15. Synchronizing Execution With WAIT

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

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

Local Bus Arbitration

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

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

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

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

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

System Bus Arbitration

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

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

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

Controlled Variable Access

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

S.6 Processor Control and Monitoring

Initialization

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

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

CPU Identification

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

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

Table S-7. Processor State Following Initialization

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

Interrupt Requests

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

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

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

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

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

Interrupt Priority

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

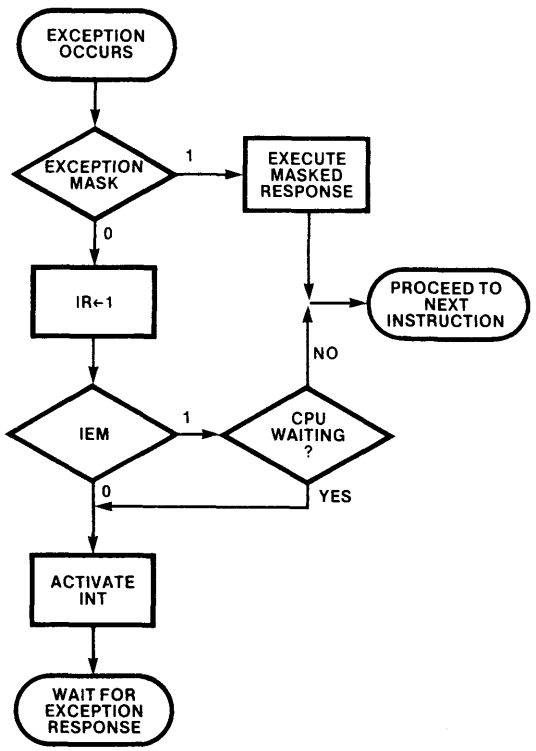


Figure S-16. Interrupt Request Logic

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

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

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

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

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

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

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

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

Endless Wait

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

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

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

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

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

Status Lines

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

Table S-8. Bus Cycle Status Signals

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

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

S.7 Instruction Set

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

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

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

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

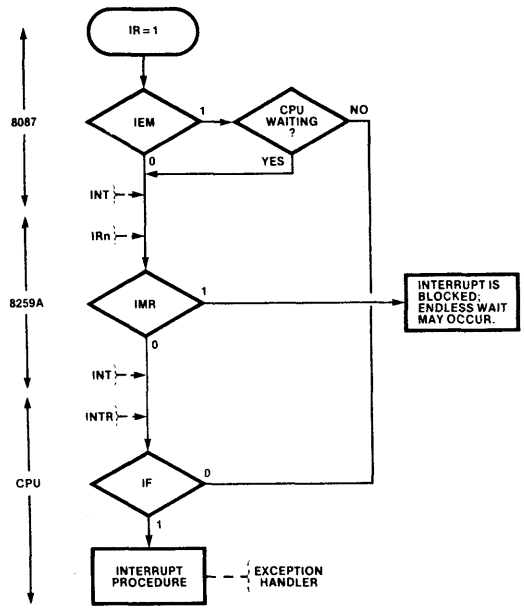


Figure S-17. Interrupt Request Path

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

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

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

//source/destination, source

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

FADD
FADD *source*
FADD *destination, source*

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

Data Transfer Instructions

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

FLD *source*

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

Table S-9. Data Transfer Instructions

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

FST *destination*

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

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

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

FSTP *destination*

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

FXCH *//destination*

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

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

FILD *source*

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

FIST *destination*

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

FISTP *destination*

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

FBLD *source*

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

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

FBSTP *destination*

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

Arithmetic Instructions

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

Table S-10. Arithmetic Instructions

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

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

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

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

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

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

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

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

Table S-11. Basic Arithmetic Instructions and Operands

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Normal Division

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

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

Reversed Division

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

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

FSQRT

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

FSCALE

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

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

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

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

FPREM

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

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

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

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

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

FRNDINT

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

FXTRACT

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

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

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

FABS

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

FCHS

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

Comparison Instructions

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

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

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

FCOM *//source*

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

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

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

Table S-12. Comparison Instructions

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

FCOMP *//source*

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

FCOMPP

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

FICOM *source*

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

FICOMP *source*

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

FTST

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

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

FXAM

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

Transcendental Instructions

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

Table S-13. FXAM Condition Code Settings

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

Table S-14. Transcendental Instructions

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

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

FPTAN

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

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

FPATAN

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

F2XM1

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

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

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

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

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

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

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

FYL2X

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

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

$$LOG_n 2 \bullet LOG_2 X$$

FYL2XP1

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

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

Constant Instructions

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

Table S-15. Constant Instructions

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

FLDZ

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

FLD1

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

FLDPI

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

FLDL2T

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

FLDL2E

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

FLDLG2

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

FLDLN2

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

Processor Control Instructions

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

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

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

FINIT/FNINIT

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

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

FDISI/FNDISI

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

Table S-16. Processor Control Instructions

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

FENI/FNENI

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

FLDCW *source*

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

FSTCW/FNSTCW *destination*

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

FSTSW/FNSTSW *destination*

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

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

FCLEX/FNCLEX

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

FSAVE/FNSAVE *destination*

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

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

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

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

FRSTOR *source*

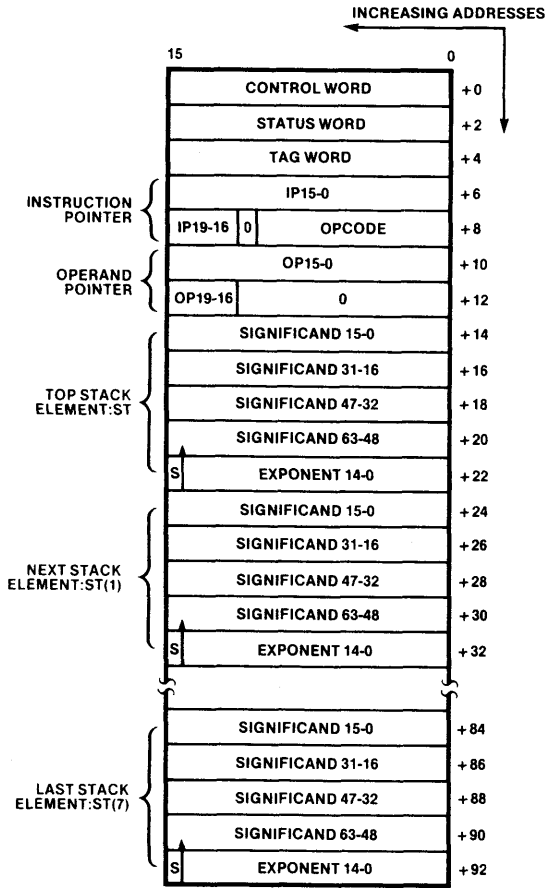
FRSTOR (restore state) reloads the 8087 from the 94-byte memory area defined by the source operand. This information should have been written by a previous FSAVE/FNSAVE instruction and not altered by any other instruction. CPU instructions (that do not reference the save image) may immediately follow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

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

FSTENV/FNSTENV *destination*

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

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



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

Figure S-18. FSAVE/FRSTOR Memory Layout

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

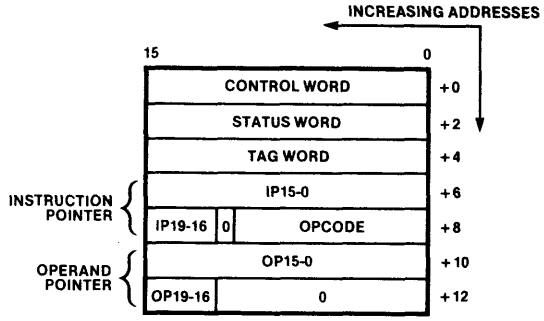


Figure S-19. FSTENV/FLDENV Memory Layout

FLDENV *source*

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

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

FINCSTP

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

FDECSTP

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

FFREE *destination*

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

FNOP

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

FWAIT (CPU instruction)

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

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

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

Instruction Set Reference Information

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

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

Table S-17. Key to Operand Types

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

Execution Time

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

- Instruction fetch
- Instruction execution
- Operand transfer

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

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

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

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

Table S-18. Execution Penalties

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

Bus Transfers

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

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

Instruction Length

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

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

Table S-19. Instruction Set Reference Data

FABS

FABS (no operands)
Absolute value

Exceptions: I

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

FADD

FADD //source/destination,source
Add real

Exceptions: I, D, O, U, P

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

FADDP

FADDP destination,source
Add real and pop

Exceptions: I, D, O, U, P

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

FBLD

FBLD source
Packed decimal (BCD) load

Exceptions: I

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

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

FBSTP

FBSTP destination
Packed decimal (BCD) store and pop

Exceptions: I

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

FCHS

FCHS (no operands)
Change sign

Exceptions: I

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

FCLEX/FNCLEX

FCLEX (no operands)
Clear exceptions

Exceptions: None

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

FCOM

FCOM //source
Compare real

Exceptions: I, D

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

FCOMP

FCOMP //source
Compare real and pop

Exceptions: I, D

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

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

FCOMPP

FCOMPP (no operands)
Compare real and pop twice

Exceptions: I, D

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

FDECSTP

FDECSTP (no operands)
Decrement stack pointer

Exceptions: None

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

FDISI/FNDISI

FDISI (no operands)
Disable interrupts

Exceptions: None

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

FDIV

FDIV //source/destination,source
Divide real

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

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

FDIVP

FDIVP destination,source
Divide real and pop

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

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

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

FDIVR

FDIVR //source/destination,source
Divide real reversed

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

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

FDIVRP

FDIVRP destination,source
Divide real reversed and pop

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

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

FENI/FNENI

FENI (no operands)
Enable interrupts

Exceptions: None

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

FFREE

FFREE destination
Free register

Exceptions: None

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

FIADD

FIADD source
Integer add

Exceptions: I, D, O, P

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

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

FICOM

FICOM source
Integer compare

Exceptions: I, D

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

FICOMP

FICOMP source
Integer compare and pop

Exceptions: I, D

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

FIDIV

FIDIV source
Integer divide

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

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

FIDIVR

FIDIVR source
Integer divide reversed

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

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

FILD

FILD source
Integer load

Exception: I

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

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

FIMUL

FIMUL source
Integer multiply

Exceptions: I, D, O, P

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

FINCSTP

FINCSTP (no operands)
Increment stack pointer

Exceptions: None

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

FINIT/FNINIT

FINIT (no operands)
Initialize processor

Exceptions: None

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

FIST

FIST destination
Integer store

Exceptions: I, P

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

FISTP

FISTP destination
Integer store and pop

Exceptions: I, P

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

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

FISUB

FISUB source
Integer subtract

Exceptions: I, D, O, P

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

FISUBR

FISUBR source
Integer subtract reversed

Exceptions: I, D, O, P

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

FLD

FLD source
Load real

Exceptions: I, D

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

FLDCW

FLDCW source
Load control word

Exceptions: None

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

FLDENV

FLDENV source
Load environment

Exceptions: None

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

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

FLDLG2

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

Exceptions: I

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

FLDLN2

FLDLN2 (no operands)
Load $\log_e 2$

Exceptions: I

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

FLDL2E

FLDL2E (no operands)
Load $\log_2 e$

Exceptions: I

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

FLDL2T

FLDL2T (no operands)
Load $\log_2 10$

Exceptions: I

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

FLDPI

FLDPI (no operands)
Load π

Exceptions: I

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

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

FLDZ

FLDZ (no operands)
Load +0.0

Exceptions: I

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

FLD1

FLD1 (no operands)
Load +1.0

Exceptions: I

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

FMUL

FMUL //source/destination,source
Multiply real

Exceptions: I, D, O, U, P

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

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

FMULP

FMULP destination,source
Multiply real and pop

Exceptions: I, D, O, U, P

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

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

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

FNOP

FNOP (no operands)
No operation

Exceptions: None

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

FPATAN

FPATAN (no operands)
Partial arctangent

Exceptions: U, P (operands not checked)

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

FPREM

FPREM (no operands)
Partial remainder

Exceptions: I, D, U

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

FPTAN

FPTAN (no operands)
Partial tangent

Exceptions: I, P (operands not checked)

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

FRNDINT

FRNDINT (no operands)
Round to integer

Exceptions: I, P

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

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

FRSTOR

FRSTOR source
Restcre saved state

Exceptions: None

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

FSAVE/FNSAVE

FSAVE destination
Save state

Exceptions: None

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

FSCALE

FSCALE (no operands)
Scale

Exceptions: I, O, U

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

FSQRT

FSQRT (no operands)
Square root

Exceptions: I, D, P

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

FST

FST destination
Store real

Exceptions: I, O, U, P

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

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

FSTCW/FNSTCW

FSTCW destination
Store control word

Exceptions: None

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

FSTENV/FNSTENV

FSTENV destination
Store environment

Exceptions: None

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

FSTP

FSTP destination
Store real and pop

Exceptions: I, O, U, P

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

FSTSW/FNSTSW

FSTSW destination
Store status word

Exceptions: None

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

FSUB

FSUB //source/destination,source
Subtract real

Exceptions: I,D,O,U,P

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

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

FSUBP

FSUBP destination,source
Subtract real and pop

Exceptions: I,D,O,U,P

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

FSUBR

FSUBR //source/destination,source
Subtract real reversed

Exceptions: I,D,O,U,P

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

FSUBRP

FSUBRP destination,source
Subtract real reversed and pop

Exceptions: I,D,O,U,P

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

FTST

FTST (no operands)
Test stack top against +0.0

Exceptions: I, D

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

FWAIT

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

Exceptions: None (CPU instruction)

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

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

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

FXAM

FXAM (no operands)
Examine stack top

Exceptions: None

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

FXCH

FXCH //destination
Exchange registers

Exceptions: I

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

FXTRACT

FXTRACT (no operands)
Extract exponent and significand

Exceptions: I

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

FYL2X

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

Exceptions: P (operands not checked)

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

FYL2XP1

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

Exceptions: P (operands not checked)

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

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

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

S.8 Programming Facilities

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

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

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

PL/M-86

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

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

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

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

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

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

[1]Also initializes interrupt pointers for emulation.

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

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

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

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

ASM-86

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

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

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

Defining Data

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

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

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

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

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

Records and Structures

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

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

Table S-21. 8087 Storage Allocation Directives

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

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

Figure S-20. Sample 8087 Constants

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

Figure S-21. Status Word RECORD Definition

```
SAMPLE      STRUC

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

Figure S-22. Structure Definition

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

Addressing Modes

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

8087 Emulators

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

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

Table S-22. Addressing Mode Examples

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

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

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

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

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

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

PL/M-86 automatically observes corresponding conventions.

Programming Example

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

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

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

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

```
PL/M-86 COMPILER     ARRAYSUM

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


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

    1             ARRAY$SUM:  DO;

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

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

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

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

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

                  /* ETC...*/

   14   1         END ARRAY$SUM;
```

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

```
PL/M-86 COMPILER    ARRAYSUM

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

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

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


MODULE INFORMATION:

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

END OF PL/M-86 COMPILATION
```

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

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

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

```
8086/8087/8088 MACRO ASSEMBLER    ARRSUM

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

LOC  OBJ                  LINE     SOURCE

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

Figure S-24. Sample ASM-86 Program

```
8086/8087/8088 MACRO ASSEMBLER    ARRSUM

LOC  OBJ                  LINE     SOURCE

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

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

```
8086/8087/8088 MACRO ASSEMBLER    ARRSUM

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


NAME          TYPE     VALUE   ATTRIBUTES, XREFS

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

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

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

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

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

S.9 Special Topics

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

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

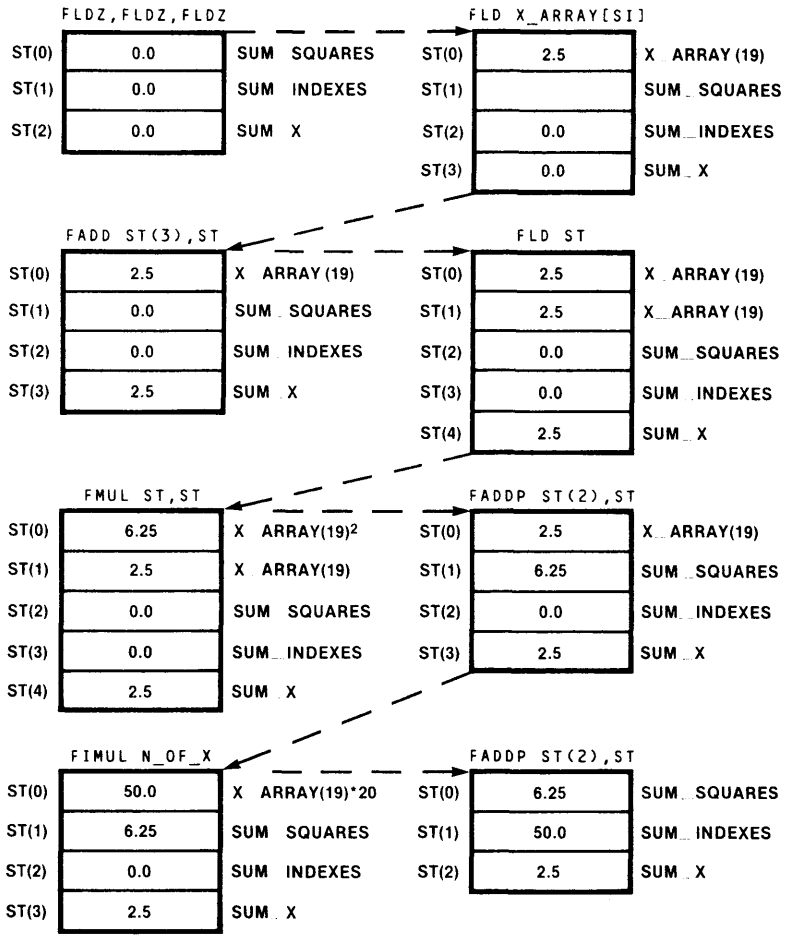


Figure S-25. Instructions and Register Stack

Nonnormal Real Numbers

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

Denormals

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

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

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

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

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

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

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

Table S-23. Denormalization Process

Operation	Sign	Exponent(1)	Significand
True Result	0	−129	1_{Δ}01011100...00
Denormalize	0	−128	0_{Δ}101011100...00
Denormalize	0	−127	0_{Δ}0101011100...00
Denormalize	0	−126	0_{Δ}00101011100...00
Denormal Result(2)	0	−126	0_{Δ}00101011100...00

Notes:
(1)expressed as unbiased, decimal number

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

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

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

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

Unnormals

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

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

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

Table S-24. Exceptions Due to Denormal Operands

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

Table S-25. Unnormal Operands and Results

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

Zeros and Pseudo-Zeros

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

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

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

Table S-26. Zero Operands and Results

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

Notes:

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

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

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

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

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

[6] † = sign of X.

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

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

(9) When Y divides into X exactly.

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

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

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

Infinities

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

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

Table S-27. Infinity Operands and Results

Operation	Projective Result	Affine Result
Addition		
+∞ plus +∞	Invalid operation	+∞
−∞ plus −∞	Invalid operation	−∞
+∞ plus −∞	Invalid operation	Invalid operation
−∞ plus +∞	Invalid operation	Invalid operation
±∞ plus ±X	*∞	*∞
±X plus ±∞	*∞	*∞
Subtraction		
+∞ minus −∞	Invalid operation	+∞
−∞ minus +∞	Invalid operation	−∞
+∞ minus +∞	Invalid operation	Invalid operation
−∞ minus −∞	Invalid operation	Invalid operation
±∞ minus ±X	*∞	*∞
±X minus ±∞	†∞	†∞
Multiplication		
±∞ • ±∞	⊕∞	⊕∞
±∞ • ±Y	⊕∞	⊕∞
±0 • ±∞, ±∞ * ±0	Invalid operation	Invalid operation

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

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

Notes: X = zero or nonzero operand

Y = nonzero operand

* = sign of original operand

† = sign is complement of original operand's sign

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

NANs

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

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

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

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

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

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

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

Data Type Encodings

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

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

Table S-28. Binary Integer Encodings

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

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

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

1) if the result is the largest negative number,

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

Exception Handling Details

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

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

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

Table S-29. Packed Decimal Encodings

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

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

Table S-30. Real and Long Real Encodings

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

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

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

* Integer bit is implied and not stored.

Table S-31. Temporary Real Encodings

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

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

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

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

Class			Sign	Biased Exponent	Significand $I_{\Delta}ff...ff$
Negatives	NANs	*Indefinite*	1	11...11	110...00
			1	11...11	111...11

←15 bits→ ←64 bits→

Table S-32. Exception Conditions and Masked Responses

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

Exception Conditions and Masked Responses (Cont'd.)

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

Exception Conditions and Masked Responses (Cont'd.)

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

Table S-33. Masked Overflow Response for Directed Rounding

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

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

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

S.10 Programming Examples

Conditional Branching

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

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

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

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

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

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

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

Figure S-26. Conditional Branching for Compares

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

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

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

Figure S-27. Conditional Branching for FXAM

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

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

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

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

Exception Handlers

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

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

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

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

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

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

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

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

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

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

Figure S-28. Full State Exception Handler

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

Figure S-29. Reduced Latency Exception Handler

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

Figure S-30. Reentrant Exception Handler

intel

AR-164

iAPX 286 Microprocessor Architecture Overview

Revised February 1981

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

CONTENTS

This booklet contains reprints of material on Intel's iAPX 286 microprocessor. All material is from the Intel International Invitational Technical Symposium (October 1980). The book includes four articles. They are as follows:

1. iAPX 286 Microprocessor: Architectural Overview Page 1
   This paper discusses the iAPX 286 processor architecture. It details the memory management and memory protection features and their operation. It also describes some of the instructions new to the 286 not included on the 8086.

2. iAPX 286 Architecture: Memory Management and Protection Model Page 34
   This paper, referenced in the iAPX 286 Architecture Overview, goes into greater depth on the operation of the 286 processors memory management and protection mechanism.

3. iAPX 286 Microarchitecture to Maximize Performance Page 40
   Discusses the four main sub-units of the 286 processor: the bus unit, instruction decode unit, execution unit, and address unit. The paper describes how they work together to greatly improve throughput over the 8086.

4. The Implementation of Operating Systems on the iAPX 286 Page 44
   This paper pulls all the pieces of the 286 architecture together and tells why they are well suited to the needs of multi-user/multi-tasking operating systems.

iAPX 286 Architecture Overview

R. Childs
J. Klebanoff

October 14, 1980

iAPX ∠86 ARCHITECTURE

Introduction

The iAPX ∠86 is a single vLSI processor chip which provides very high performance and an integrated memory management and protection mechanism. It is part of the iAPX 86 series of 16 bit microprocessors. The ∠86 is an evolutionary extension of the 86 architecture and is upward compatible with the 86 at the object code level. The extension is also revolutionary in that the iAPX ∠86 provides a memory management and protection scheme of significantly greater sophistication and performance than previously available in micro-processors or even on contemporary mini and maxi-computers.

The iAPX ∠86 can operate in either an iAPX 86 compatible mode or in native protected mode. The compatible mode is also totally software compatible with the iAPX 186 highly integrated processor.

The memory management and protection scheme of the ∠86 is called PM∠86. It is compatible with well structured user level code written for the iAPX 86 and 186. This paper will discuss PM∠86 in detail. Familarity with the 8086 is assumed. The concurrently published paper entitled "The iAPX ∠86 Architecture: Memory Management and Protection Model" gives an overview of the motivations and theory behind the architecture.

PM∠86 provides a number of advantages to its users. It assists in debugging software by detecting such errors as stack overflows, pointers out of range, and invalid instructions. It limits the effects of bugs by protecting the operating system from user level software, and by isolating one application from another in multi-application systems. It allows the implementation of virtual memory. It increases the memory space available on the iAPX 86; the virtual memory is increased to one gigabyte, and the physical address space is increased to 16 megabytes. Therefore PM∠86 is well suited to the implementation of large applications.

PM∠86 extends and strengthens the segmentation of the iAPX 86. While the iAPX 86 interprets a segment number as a segment base address, PM∠86 interprets it as an index into a memory based descriptor table. The actual base address is found in the descriptor table along with the segment length and other protection information. The 8086's segment registers are widened in the iAPX ∠86 to contain the extra information.

PM∠86 was designed for high performance. For most instructions there is no performance penalty paid for protection; the protection checking is done in parallel with instruction execution. A performance penalty is paid only when a segment register is changed. This is not done frequently in a program, and the iAPX ∠86 is designed to minimize the penalty.

iAPX ∠86 SYSTEM MODEL

The iAPX ∠86 Protection Model (PM∠86) was developed to optomize the execution of an assumed "system model". The ∠86 design was driven by the "high-end" application requirements. A typical high-end application is assumed to have the following characteristics:

- Protection Required for Application and System Reliability
- Heavily Multi-tasked
- Performance Critical
- Large Real Memory
- Interrupt Driven

The overall system is assumed to be organized with a clear distinction between application level and operating system. The operating system provides all resource scheduling, input/output functions and a variety of general services for usage by the applications. The applications are controlled, from a scheduling and resource allocation standpoint, by the operating system.

The range of system services includes a large body of functions and library procedures, which are utilized at both the application and system levels. A set of low level system functions also exist for the exclusive use of the operating system in performing the system control, I/O, resource allocation and application service functions.

The iRMX-86 Operating System may be considered as a specific example of a set of basic operating system functions which would satisify the system model. However, many other implementations exist which also satisfy the system model.

Typical system applications might include large multi-terminal interactive business systems, main-frame front end processors and complex communications systems.

PM∠86 OVERVIEW

The iAPX ∠86 provides a powerful address translation, segment access control and state transition control mechanism to support the protection model specified below as PM∠86.

The implementation of protection is based on protected memory resident descriptor tables. The descriptor tables are automatically accessed to obtain both physical location and protection information for segments and for control of all major system state transitions. The PM∠86 protection rules are completely enforced by hardware within the iAPX ∠86. Violations result in hardware invoked traps which transfer control to violation handlers within the operating system. All protection control information is directly traceable to either program source declarations or explicit activity of a protected operating system kernel function.

The protection kernel of the operating system functions is an extension of the hardware protection mechanism. PM∠86 does not specify the operation of the kernel. However, the kernel is expected to be the only software with the capability to modify the content of the PM∠86 specified descriptor tables.

THE MEMORY MANAGEMENT AND PROTECTION MODEL (PM⊿86)

FEATURE OVERVIEW

he PM⊿86 technical requirements and objectives have been rigorously efined and are a natural extension of the 8086 architecture. The esulting features are listed below for ease of reference:

PM⊿86 provides a simple application migration path for 8086 application programs.

M⊿86 PROTECTION provides:

- Hierarchical, four level, protection mechanism
- Protection of a task from other tasks.
- Protection of a segment at one level from any task at a less privileged level.
- Immediate detection of attempted protection violations.
- Means for two or more tasks at the same level to cooperate (e.g., share data)
- Enforcement of access restrictions of memory segments (e. g.: read only, execute only, accessable only from certain privilege levels, etc.)
- Support of high performance synchronous requests of functions to be performed by more privileged levels on the behalf of less privileged levels.
- Support of conforming (library) segments; the requested procedure assumes the protection level of the caller and may call procedures at any more privileged level.
- Support of logical segments having restricted access rights to the corresponding physical segment.
  The following segment classifications are provided:

  1. Read only
  2. Execute only
  3. Execute and Read only
  4. Read/Write only (i. e. not executable)
  * 5. Privilege level only visibility
  * 6. Segment length less than or equal to physical maximium.
  * 7. Not Present

  * Orthogonal restrictions to the others; a segment may be accessible from only a certain privilege level AND be read only. etc.

M⊿86 MEMORY MANAGEMENT provides:

- virtual address mapping by supporting:

  Hardware mapping of logical segments into the physical address space of the iAPX ⊿86 based system.

Dynamically alterable logical to physical address mapping permitting dynamic relocation of part or all of the task's real address space.

- Support of 16 MByte real address space.

  - The support necessary for operating systems to provide dynamic segment swapping.

PM∠86 I/O PROCESSING supports:

- Interrupt handlers that may be invoked at any of the 4 protection levels. It is therefore possible to protect the operating system kernel from interrupt handlers.

- I/O instructions that may be executed at all 4 protection levels.

- I/O operations fully protected via memory-mapped I/O devices. Thereby permitting I/O usage of the complete protection mechanism.

- Restricting the tasks that may execute the I/O instructions.

- An interrupt system that is capable of performing a complete, high performance, task switch when an interrupt handler is invoked. It is therefore possible to isolate an interrupt handler from all other tasks.

iAPX ∠86 PROTECTION MODEL (PM∠86)

The protection model consists of three components: protected entities, active subjects and rules governing the access of the subjects to the protected entities.

Access to protected entities is controlled, they are represented by descriptors. The entities represented in PM∠86 are: main memory segments, descriptor tables, gates and task state segments. Each protected entity is accessed via a selector, which comprises an index integer assigned to the descriptor at the time of its creation and PM∠86 control information.

Subjects are active entities which may perform accesses and are therefore subject to control. In PM∠86, only tasks are active entities and thus subjects. A task is a single sequential thread of execution characterized by an processor execution state and a set of logical named entities, which have as attributed access rights, and which are a dynamic function of the execution state and execution history.

Protection rules govern the access of subjects to the protected entities and represent the relations between them. In PM∠86, each and every access by an active task to a protected entity (an active task is one executing on an iAPX ∠86 processor under PM∠86) is validated against the protection rules. The access is permitted IF AND ONLY IF the request is valid according to the access rights of the task to that protected entity.

PM286 CONCEPTS AND ENTITIES

TASKS

All operations within PM286 are performed by tasks. A task is an executing program. A task is characterized by a single sequential thread of execution, a current task state, and an addressable logical address space including a set of four stack segments. The static attributes of a task are tables which contain information concerning all of the accessible entities. The dynamic attributes of a task are the register contents, the current privilege level, the currently selected procedure and the values of all accessible data elements.

PRIVILEGE LEVEL

PM286 supports a 4 level protection hierarchy. The most privileged level is number 0 and the least privileged level is number 3.

Each protected entity within the system is assigned a specific privilege level. Each task within the system operates at one and only one privilege level at any instant in time. Protected entities which reside at a privilege level which is equal or less privileged than the current privilege level (CPL) of the task are generally visible and accessible, as long as they reside within the logical address space of the task. (Protected entities which do not reside within the logical address space of the active task are neither addressable nor accessible.) Protected entities which reside at a more privileged level than the current privilege level of a task are not accessible.

POINTERS

A full pointer in PM286 is a 32 bit logical name. A pointer consists of a 16 bit selector and a 16 bit offset. A pointer is always evaluated within the logical address space of the task. The selector is encoded as a 13 bit index into a descriptor table, a 1 bit table selector and a 2 bit quantity which indicates the requested privilege level for protected entity access. The offset is a 16 bit integer which indicates the desired byte within the protected entity. Segments may contain any number of bytes up to a maximium of 64K. The offset field is not defined when the indicated protected entity is not a segment.

A short pointer, consisting of an 8 bit selector, is supported for accessing non-segment protected entities in the interrupt vector table.

DESCRIPTOR TABLES

PM286 utilizes three classes of descriptor tables: Interrupt Descriptor Table (IDT), Global Descriptor Table (GDT) and Local Descriptor Table (LDT).

The IDT and GDT are system tables which are always present in the system's address space. The LDT is a task dependent table which is a portion of each tasks' state. The logical address space of a task consists of the descriptors in the GDT, IDT, and currently selected LDT.

Each of the descriptor tables is organized as a linear array of descriptors. The index portion of a selector indicates which descriptor in the tables is to be selected.

The IDT, GDT and LDT may be located at any real memory address. The GDT and LDT may contain a maximium of 819⁄ descriptors. In the IDT, however, only ⁄56 descriptors are addressable.

DESCRIPTORS

PM⁄86 utilizes descriptors to contain all information about the attributes and addressing of all protected entities in the system. Descriptors are organized into two categories; segment descriptors and control descriptors.

Segment descriptors are the primary mechanism for enforcing protection rules upon all accesses to segments and performing the logical address to real address translation function. Segment descriptors contain a type indicator, privilege level, base address and length of real memory segments.

Control descriptors are the primary mechanism for enforcing protection rules upon all major state transitions. Control descriptors contain a type indicator, privilege level and pointer to the target of the state transition.

FLAGS

PM⁄86 introduces 3 special flag bits in the flag register of the iAPX ⁄86. Nested Context (NC) is a single bit flag which indicates the existence of a non-empty chain of task state segments. It is created and maintained by the hardware task switch operation. I/O privilege level (IO_PL) is a ⁄ bit value in the flags which indicates the least privileged level permitted to perform I/O instructions.

MACHINE STATUS WORD

A status register, the Machine Status Word (MSW), is included in the iAPX ⁄86. This register is used to indicate the system configuration and processor status.

Four MSW bits are defined. Protection enable (PE) is a single bit flag that indicates invocation of the protection model. Three other bits are used to manage usage of a coprocessor and/or for emulation of a coprocessor. The math present (MP) bit indicates whether an floating point math coprocessor is present. The emulation mode (EM) switch indicates whether the coprocessor function is to be (software) emulated. The task switched (TS) flag indicates whether the iAPX ⁄86 has performed a task switch. The TS flag is used to manage the state of the math co-processor. The TS bit is set by the iAPX ⁄86 on each task switch but is reset by software.

PROTECTED ENTITY ACCESSING

Segments are accessed by the implicit or explicit loading of a segment register and a subsequent request for a real memory access to the contents of the segment. PM⁄86 obtains, under hardware control, the descriptor indicated by the selector being loaded into a segment register. Basic validity checking is immediately performed to assure that that the selected descriptor is a segment which may lawfully be accessed via the segment register type and is accessible from the current privilege level. Failure of these basic tests will result in an

immediate protection violation trap. All subsequent references to main memory are checked for full conformance to the access restrictions and for a byte offset value within the defined limit of the segment size. Failure of these tests will also result in a protection violation trap. The supported segment types are as follows:

- Read Only Data
- Read/Write Data

- Execute Only
- Execute and Read Only
- Conforming Execute Only
- Conforming Execute and Read Only

Execute only segments and conforming execute only segments may only be selected via CS (see 8086 User's Manual). Readable or Read/Write segments, including those with the execute or conforming attributes, may be selected via DS or ES. Data segments must have write permit in order to be accessed via SS.

PM286 enforces protection rules upon all major state transitions by the usage of control descriptors. All state transitions which involve the increase of the privilege level or a change of processor task must reference a control descriptor. The control descriptor contains a type field, a privilege level and a pointer to the new processor state. The type field is checked to confirm that it is the correct type of control descriptor for the desired state transition. The privilege level is checked to assure the accessability of the control descriptor from the current privilege level. Successful passing of these checks will result in transition of the processor state to the value indicated by the control descriptor.

The supported types of control descriptors are:

- Call Gate
- Trap Gate
- Interrupt Gate
- Task Gate
- Task State Segment
- Local Descriptor Table

The call gate is utilized for CALL transitions within a task which result in an increase of privilege level. The call gate may also be utilized for transitions that stay at the same privilege level, if desired, in order to support delayed binding. These operations include CALL, JUMP, INT n, processor trap and external interrupt operations. The trap gate is used for interrupt and trap operations that stay within the same task but do not disable interrupts. The interrupt gate is utilized for interrupt and trap operations which stay within the same task but result in disabling interrupts. The task gate and task state segment are utilized for all operations which result in the invocation of an entirely different task.

A special control descriptor type is used to indicate the space allocated to a Local Descriptor Table (LDT). An LDT control descriptor may only be selected for initializing an on-chip register which points to the LDT.

IMPLEMENTATION OF PM⊿86

OVERVIEW

This section contains a technical specification of the operation of PM⊿86 on the iAPX ⊿86. The first portion characterizes the address space, protected entity access control, segment descriptors and segment access control. Then, the various means of transfering control and using control descriptors are examined. Next are specifics of the CPU registers and the new or modified iAPX ⊿86 instructions which support the PM⊿86 model. The next section specifies the interrupts which have been defined for protection violations and use INTEL reserved interrupt vectors and error codes returned for each interrupt. Finally, we examine special circumstances that arise from coprocessor and multiprocessor configurations.

ADDRESS SPACE AND PROTECTED ENTITY ACCESS CONTROL

The PM⊿86 mechanism uses descriptors to regulate and manage the contents of the address space of a task through control of the entities that are allowed to be accessed. These entities are represented in the address space by descriptors.

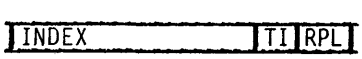
Descriptors in memory are maintained in descriptor table segments, which are entities known to the hardware. There are three types of tables: global descriptor table (GDT), local descriptor table (LDT) and interrupt descriptor table (IDT). The location and limit information of the active IDT, GDT and LDT are kept in hardware registers. These three tables completely represent the address space of a task.

A task when active under PM⊿86 must have a GDT and an IDT (which handles external interrupts and protection violations). In order to achieve isolation between tasks, it is necessary to create for each task a LDT in PM⊿86 to represent its private portion of the logical address space. Tasks created in such a format may now acquire a complete address space through the inclusion of a task state segment descriptor in the GDT.

Privilege levels are numbered 0,1,⊿,3 in order of decreasing privilege. Privilege levels are available as a system resource; it is not necessary to use all privilege levels in every application. Use of levels 0,3 alone will support a ⊿ level system. Sophisticated applications, as typified by the iRMX-86 Operating System can specify the application of all four levels. The privilege level of a task is used by the hardware in a manner specified below.

SELECTORS

A selector is a 16-bit quantity comprising 3 fields as follows:



Selector

FIELD NAME	SYMBOL
Requested Privilege level	RPL
Table Indicator(0 = GDT, 1 = LDT)	TI
Index	INDEX

A selector that contains all zeros in the table indicator and index fields is a special case used primarily to allow the hardware the means of representing an invalid segment selector in a segment register. Thus descriptor number 0 is not allowed in the GDT.

MEMORY ADDRESSING

The address of a byte (or a word) in memory consists of a selector and an offset. The selector indicates the segment in which the byte is located. Most iAPX 286 instructions will use a short form of address specifying only the offset with the implied usage of a segment register which has been previously loaded with a segment selector (and descriptor).

INTERRUPT ADDRESSING

Interrrupt addressing is within the interrupt descriptor table. This table contains control descriptors. An interrrupt address is an 8-bit address.

I/O ADDRESSING

I/O port addressing is via a 16-bit quantity. There is no address mapping for an I/O port. (All I/O ports use real addresses.)

ACCESS CONTROL

This section describes access control. Every time a descriptor is loaded the iAPX 286 validates the protection access status. A violation of the protection rules will cause an interrupt.

The access control rules for an entity are encoded in its descriptor. The descriptor is a 4 word quantity. The access byte is the sixth byte in all descriptors. A portion of the access byte is common for all descriptors. The information in the common portion is:

NAME	SYMBOL
PRESENT	P
DESCRIPTOR PRIVILEGE LEVEL	DPL
SEGMENT (segment/control)	S

The complete format of the access byte differs depending on whether the descriptor is a segment descriptor or a control descriptor.

The present indicator is used to indicate that the entity is present in real memory. If not-present, then the descriptor does not point to an accessible entity in real memory. Any hardware access of the descriptor will result in a protection trap. This is useful both for deletion of previously valid descriptors and for support of demand swapping functions.

The descriptor privilege level (DPL) is the privilege level assigned to the protected entity. Semantics for usage of the privilege level are discussed below.

The segment indicator (S) is the gross classification of descriptors into segment or control types. Descriptor formats and semantics of usage are discussed below.

PRIVILEGE LEVEL

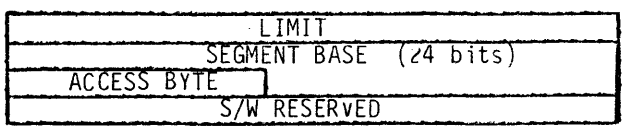
Privilege level is a major part of access control. At every moment, the executing task is at some privilege level, called the current privilege level (CPL).
The effective privilege level (EPL) of an access to an protected entity changes dynamically and is defined as the numeric maximum of the CPL and the requested privilege level (RPL) present in the selector. An access is permitted IF AND ONLY IF the EPL is numerically less than or equal to the OPL.

The Requested Privilege level has been introduced to solve the "Trojan Horse" problem. Consider a file system procedure, fread(file_id, nbytes, buffer_ptr). It reads data from a file into a buffer, overwriting whatever is in the buffer. Normally fread would be available at the user level but the file system procedures and data would be privileged so that user level procedures cannot directly change the file tables. However a user level procedure could use a pointer into the file tables as his buffer pointer, causing the fread procedure to unwittingly corrupt the file table.

Use of the requested privilege level feature can prevent this. The called procedure need only ensure that all selectors passed to it have an RPL at least as high (numerically) as the caller's CPL. Then a protection fault will result if the selector is used to access a segment that the caller would not be able to access directly. The caller's CPL is found in the CS selector which was pushed on the stack, and a special instruction can be used to appropriately adjust the RPL field of a selector parameter.

SEGMENT DESCRIPTORS AND SEGMENT ACCESS CONTROL

Segments are classified into two categories: executable segments and data segments. The basic format of both types of descriptors are identical as shown below:



BASIC SEGMENT DESCRIPTOR FORMAT

The segment base is the 24 bit real memory address of the beginning (lowest) byte address of the segment. The limit is the maximum (minimium for expand down segments) value of offset which may be validly utilized in accessing the contents of the segment.

EXECUTABLE SEGMENTS

The detail portion of the access byte is utilized to distinguish between executable and data segments and provide specific protection and activity information for the segment.

The detail portion of the access byte for executable segments contains the following:

NAME	SYMBOL
Executable	E
Conforming	C
Readable	R
Accessed	A

Executable segments may never be written and may not be loaded into SS. A segment must be executable in order to be selected by CS.

When a normal (i. e. non-conforming) executable segment is correctly invoked, the CPL is changed to be equal to DPL of the segment. If a level change occurs, this operation will result in the invocation of a new stack segment. RPL is also forced equal to DPL in CS.

When a conforming executable segment is invoked CPL does not change. The RPL will be adjusted in the CS to be equal to the CPL. A conforming executable segment whose DPL is numerically greater than CPL may not be accessed.

Executable segments may be readable; as indicated by R = 1. Readable executable segments, both normal and conforming, may be accessed via DS and ES. Readable conforming segments are readable from any privilege level.

The accessed bit (A) is provided in both executable and data segment descriptors. Whenever the descriptor is accessed by the iAPX 286 hardware, the A bit will be set in memory, if not already set, when the descriptor is loaded. The reading of the access byte and the restoring of the access byte with the accessed bit (A) set is an indivisible operation (i. e. performed as a read-modify-write with bus lock).

The purpose of this feature is to provide a mechanism for developing a segment usage profile in demand swapping systems.

DATA SEGMENTS

The detail portion of the access byte is utilized to distinguish between executable and data segments and provide specific protection and activity information for the segment.

The detail portion of the access byte for data segments contains information as indicated below:

NAME	SYMBOL
Executable	E
Expand down	ED
Writeable	W
Accessed	A

Data segments are indicated by E = 0. Data segments may always be read and may not be selected by CS. A data segment is always accessible via DS and ES.

A data segment may expand up or down. Normal data segments, indicated by ED = 0, expand up. In any access to the segment the offset used must satisfy the condition:

0 LESS THAN OR EQUAL TO offset LESS THAN OR EQUAL TO limit.

If such a segment is expanded (by the operating system) it is expanded by increasing the limit. Data segments may instead expand down, indicated by ED = 1. In this case offsets used to access the segments must satisfy the condition:

limit LESS THAN offset LESS THAN 64K-1.

When this type of segment is expanded it is expanded by decreasing the limit. This is preferable for stack segments.

A data segment may be writeable; indicated by W = 1. Writable data segments are accessible via DS, ES, or SS.

A data segment must be writeable in order to be accessible via SS.

The accessed bit (A) of the access byte is as defined for executable segments.

CONTROL DESCRIPTORS AND STATE TRANSITIONS

The careful and precise control of all state transitions, that is changes in the control flow of the processor, are of crucial importance to successful protection. All such transitions are checked for conformance to the protection rules.

There exist two major classifications of system state transition: sequential within a task and transition from one task to another. An orthogonal classification is into call, return, and branch transitions. On a call transition the hardware saves a pointer back to the previous state. In a return transition the hardware follows such a pointer back to a previous state. In a branch transition no pointer is saved. A return is not normally desired subsequent to a branch.

Task sequential transitions have two major varieties: intra-level and inter-level. Intra-level transitions remain within the same privilege level. They present little difficulty since there is no change to the addressable entities. Inter-level transitions, those which cross privilege level boundaries, are substantially more complex. This complexity is due to the necesssity for assuring the integrity of the "fire wall" between levels and control of the change in the addressable entities.

The inter-task transition is the most complex form of change to the control flow. An inter-task transition represents a complete change of processor state and the selection of a new set of addressable protected entities.

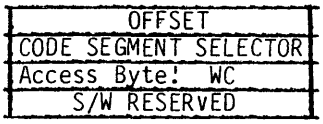
CONTROL DESCRIPTORS

There exist six types of control descriptors: call gates, trap gates, interrupt gates, task gates, task state segments, and descriptor tables. In general, the call gate is used for synchronous transitions. The gate is used for asynchronous intra-task transitions. The interrupt

is utilized for asynchronous intra-task transitions which result in interrupt disable. The task gate is used for transitions between tasks. The task state segment contains the entire task state and certain task control information. The descriptor table holds descriptors for the protected entities.

CALL GATES

A call gate consists of a present bit, privilege level, type identifier, argument word count, code segment selector, and offset. A call gate may exist in either of the basic descriptor tables: GDT or LDT. A call gate specifies a procedure that returns with a return instruction, as opposed to an interrupt return instruction. The complete layout of a call gate is illustrated below.



NAME	SYMBOL
WORD COUNT	WC

INTERRUPT GATE

An interrupt gate consists of a present bit, privilege level, type identifier, code segment selector and offset. An interrupt gate may only exist in the IDT. An interrupt gate specifies a procedure that enters with interrupts disabled and returns via an interrupt return instruction. The layout of an interrupt gate is identical to a call gate except that the type is different and the word count value is ignored.

TRAP GATE

A trap gate consists of a present bit, privilege level, type identifier, code segment selector, and offset. A trap gate descriptor may only exist in the IDT. A trap gate specifies a procedure that enters with interrupt enable status unchanged and returns via an interrupt return instruction. The layout of a trap gate is identical to an interrupt gate except that the type is different.

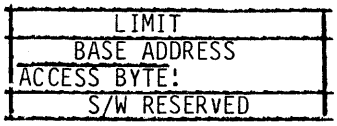
TASK GATE

A task gate consists of a present bit, protection level, type identifier and a task state segment selector. A task gate may exist in any of the three descriptor tables: interrupt, global, or local. The layout of a task gate is illustrated below:

TASK STATE SEGMENT

A task state segment is a special, fixed format, segment. The purpose of this segment is to contain all of the state information for a task and a linkage field. Task state segments come in two forms: busy, and

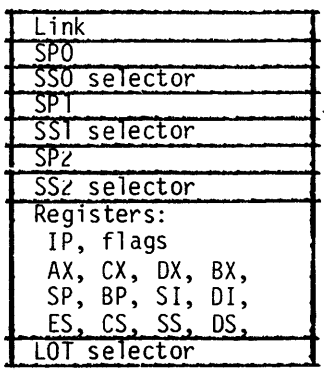
available. A busy task state segment is one that is active or is on a chain of task state segments. A task state segment descriptor consists of a present bit, privilege level, type identifier, segment base address and segment limit. The value of the type identifier of a task state segment descriptor is different for available or busy. A task state segment descriptor is expected to reside only in the global descriptor table (GDT). The detailed layout of a task state segment descriptor is illustrated below.



The task state segment contains sections which are accessed but not modified (static) and sections which are modified under hardware control by PM286. The static sections consist of the LDT selector and the pointers for the initial (empty) stacks of the three most privileged levels. Each stack pointer consists of a segment selector (SS) and a stack offset (SP) value.

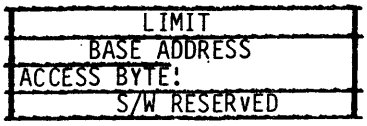
A stack pointer is not needed for the least privileged level since that stack is either the current (active) stack or can be located via the back link chain from the current stack.

The modified (dynamic) portion of the task state segment consists of all of the dynamically variable and programmer visible processor registers. This portion also contains a linkage word that is used to chain nested invocations of different tasks. The actual content of the linkage word is either null or a task state segment selector. The format of a task state segment is illustrated below.



DESCRIPTOR TABLE

A descriptor table is a special type of segment that contains descriptors. A descriptor table descriptor consists of a present bit, privilege level, type identifier, segment base address, and segment limit. The layout of a descriptor table descriptor follows.



Only the LDT is described by a descriptor table descriptor. The GDT and IDT are defined normally only at system initialization, using real addresses.

CONTROL TRANSFERS

INTRA-LEVEL TRANSITIONS

An intra-level transition is a transition which begins and ends at the same privilege level within the same task. Intra-level transitions in the iAPX 286 are directly comparable to transitions in the 8086. The basic function of this type of transition is the selection of a new point of execution and the establishment, on the stack, of a return link as necessary.

The new point of execution may exist in the same code segment. In this case, the short form of branch, jump or call may be utilized. Protection is provided by limit checking on the code and stack segments.

Alternatively the new point of execution may exist in a different code segment. In this case a full pointer, consisting of a code segment selector and an offset, must be provided. The full pointer may be contained in the instruction in the inter-segment form of the call or jump instructions. In this case the protection mechanism ensures that the indicated segment is indeed directly addressable and an executable segment. It will also ensure that there is no privilege level transition; that is, it checks that the indicated segment is either at the current level or that it is a conforming segment with an OPL less than or equal to the CPL.

An inter-segment call, or an inter-segment jump may also utilize a call gate. The existing 8086 instruction formats are retained for all cases. The offset specified in inter-segment call and jump instructions will be ignored if the instruction's selector points to a call gate. The actual code segment selector and offset used in the transition will then be taken from the call gate. The word count in the gate is ignored for intra-level transitions. The protection restrictions which apply in this case are as follows.

1. The instruction's selector is selecting a call gate.

2. The current level and the requested privilege level in the call gate selector (i.e. the EPL) must both be of equal or greater privilege (numerically less than or equal to) the level of the call gate.

3. The selector in the call gate must point to a code segment. For an intra-level transition the code segment must be at the current level. The RPL in the selector in the call gate is ignored.

Violation of any of the above rules will result in a protection violation trap. Conformance with the above rules will result in a successful control transfer and, in the case of call or interrupt instructions, establishment of an appropriate return linkage on the stack. The return linkage will be identical to the return linkage established by the comparable 8086 instructions.

An interrupt or trap operation may utilize either a trap or interrupt gate in the IDT. Utilization of a trap gate will result in no change to the interrupt enable flag. Utilization of an interrupt gate will result in resetting the interrupt enable flag. For either a trap or interrupt gate, the flags will be pushed on the stack, prior to any change in the interrupt enable flag, and the resulting stack image will be identical to that established by the comparable 8086 operation.

The return instructions (RET and IRET) will directly utilize the return linkage established on the stack.

INTER-LEVEL TRANSITIONS

The inter-level transitions are, by definition, a transition in the current privilege level (CPL) and, hence, a major change in the addressable entities. The implications of this are that the legality of the transition must be carefully monitored and the integrity of the "fire wall" between the levels must be assured.

The basic philosophy for the support of this type of transition is as follows.

1. Call, trap and interrupt operations may only be performed from a less privileged level to a more privileged level. (Calls, jumps, traps, interrupts, and returns to the same level are allowed, but fall under the class of intra-level, not inter-level, transitions.)

2. Returns to a more privileged level are not allowed. (Returns to the same level come under intra-level transitions).

3. Transitions to a more privileged level may only be performed through call, trap, or interrupt gates.

4. A new stack, whose privilege level is identical to the target privilege level, is selected.

5. Inter-level jumps are not allowed.

The utilization of a gate ensures that all transitions to a more privileged level will go to a valid entry point, not into the middle of a procedure, or worse still into the middle of an instruction. The utilization of a separate stack segment whose privilege level is identical to the target level assures that other tasks which may be operating at a lesser privilege level within the same address space will not corrupt the stack of a task operating at a more privileged level. Isolation of the stacks on a per task basis must be provided in order to assure that different tasks do not try to use the same stack space.

An inter-level transition may be performed by the inter-segment call, interrupt, RET, or IRET instructions, or it may be invoked by the occurance of an internally or externally generated interrupt. The following protection rules apply to calls and interrupts.

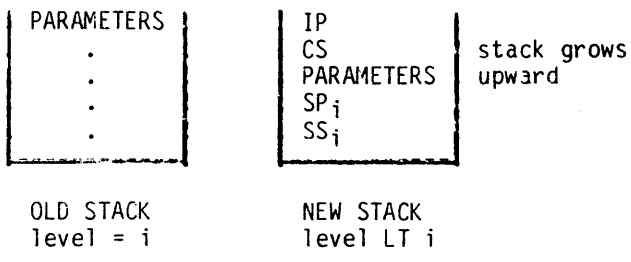
1. The transition must be via a gate.

2. Both the requested privilege level (RPL) in the gate selector and the initial level (CPL) must be of equal or greater privilege (numerically less than or equal to) than the gate (DPL). This restriction is not applicable for external interrupts or protection traps.

3. The code segment selected in the gate must be of greater privilege than the initial level. (If the selected code segment is at the same level then the transition is intra-level and the rules discussed in the previous section apply.) If the selected code segment is less privileged than the current level, then a protection violation trap will be generated.

4. The offset indicated in the call gate must be within the limit of the code segment.

The actual operation of the transition will appear, from the programmer's view, to be essentially identical to that provided on the 8086, with the exception of the stack image. The new, more privileged, stack will be selected from the appropriate element in the task state segment. Initially, this stack is empty. Upon completion of the state transition, the new stack will contain a pointer to the top of the less privileged stack, the procedure parameters, and the normal return linkage. The number of words to be moved over as parameters from the original stack is specified in the word count byte in the call gate. (The word count byte is ignored in intra-level transitions). If this count is specified as zero then no parameters are moved. The illustration below shows the general form of the stack image upon the successful completion of an inter-level call.



In the case of a transition through an interrupt or trap gate the flags are pushed on the new stack between PARAMETERS and CS, and the nested context flag is then reset. In a few interrupt cases, to be defined later, an error code is pushed on the new stack after IP.

The SS_j and SP_j values pushed on the stack constitute a pointer into the previous stack. Procedures that have a variable number of parameters or that have more than 32 parameter words cannot use the parameter copying feature of call gates. These types of procedures can use the SS_j and SP_j values to get the parameters.

An important part of assuring the integrity of the "fire wall" between levels is parameter verification. Parameter verification consists of checking the RPL of any selectors passed as parameters to assure that the RPL is numerically greater than or equal to the CPL of the caller. The CPL of the caller may be determined from the RPL of the CS selector on the stack. The procedure called may in fact only be an interface procedure that after moving and verifying the parameters only calls another procedure at the same level to do the actual work.

INTER-TASK TRANSITIONS

Inter-task transitions are the transistion of the processor from active execution of one task to the active execution of another task. This is clearly a common occurance in a multi-task system. A transition from executing one task to executing another task occurs for two basic reasons: completion of the current computational requirements of a task or the occurance of an external event. The necessary operation consists of the termination of active execution of the first (outgoing) task, saving its complete processor state in memory, loading a complete processor state for the next (incoming) task and beginning execution of the incoming task. The major requirement upon a protection system is the careful control of the circumstances upon which a different task may be invoked and the complete isolation of the logical address spaces and task state segments.

PM286 supports, in hardware as a single instruction, inter-task transitions. These operations include the saving of the complete processor state and the loading of a different processor state. These transitions are controlled with task gates which are analogous to call gates. Task state segment descriptors point to memory-resident task state segments. These segments are special protected control segments which are used for saving the processor state and for storing the linkage for nested invocations. Task gate descriptors may exist in any of the descriptor tables. Task segment descriptors must reside in the GDT.

An inter-task transition is invoked by external interrupts, internal traps, or by interrupt n, inter-segment call, or inter-segment jump instructions that select a task gate or task state segment descriptor. If a task gate descriptor is selected, then a task state segment descriptor is selected by the gate. The new (incoming) task state is obtained from the selected task state segment. The new task state segment must be marked as available (not busy) to ensure that it has only one current usage. If the task state segment is busy, i.e. already in use, then a protection fault results. If the new task is successfully entered into execution then the task state segment descriptor is marked busy. In the case of a transition initiated by a jump the outgoing task segment is set not busy. In the case of transitions initiated through traps, interrupts or calls the task state segment selector of the

outgoing task (previously located in the TS Register) is stored in the linkword of the new task state segment and the nested context flag is set.

An inter-task interrupt or call transition may be reversed with an IRET instruction. The IRET instruction causes the nested context flag to be examined. If the flag is off then execution will remain within the same task; Flags and the return link are taken from the stack. If the nested context flag is on then an inter-task transition will occur: the current task is suspended, its state saved in its task state segment and the linkword of its task segment fetched and verified, if valid the linkage word points to the (incoming) task which then resumes execution. The outgoing task's segment becomes available by being marked not busy.

To ensure correct operation in a multi-processor environment a bus lock is applied during the testing and setting of the task busy bit. This is sufficient to ensure that two processors do not invoke the same task at the same time, however, it will not avoid a protection fault. Some other mechanism for resolving conflicts must be used to avoid protection traps. This will be discussed further in the section on multi processor considerations.

CPU REGISTERS AND INSTRUCTIONS

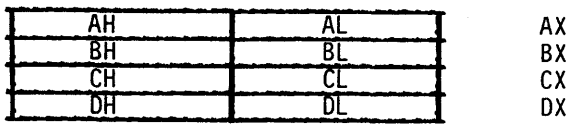
This section describes all of the CPU functionally visible registers and modified instruction semantics required to support PM286 within the iAPX 286.

REGISTERS

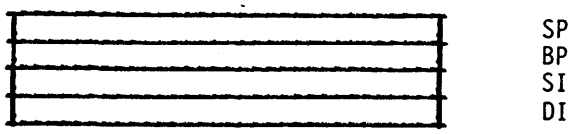
GENERAL REGISTERS

The iAPX 286 provides the same eight 16 bit registers as the 8086 for general purpose arithmetic and effective offset address computation. The general accumulator registers are AX, BX, CX and DX. These registers may also be utilized as eight 8 bit registers. The index/pointer registers are SP, BP, SI and DI. SP is used as a stack pointer. The other registers are available as general index registers.

The usages of the eight general registers, as implemented in the 8086 instruction set, are identical in the iAPX 286.



Accumulators

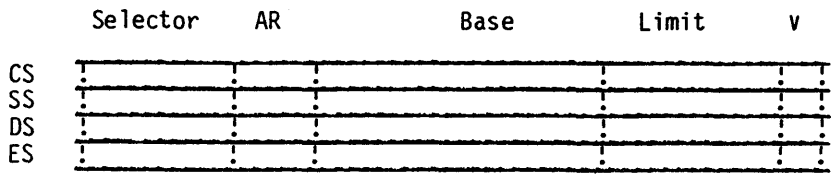


Pointer and Index Registers

SEGMENT REGISTERS

PM286 modifies the way in which the general program visible segment registers are utilized and extends each of the registers by the addition of a corresponding descriptor register. The descriptor registers are invisible at the general instruction set level. However, the on-chip descriptor registers are essential within the iAPX 286 hardware in order to obtain the performance objectives and the functionality of PM286.

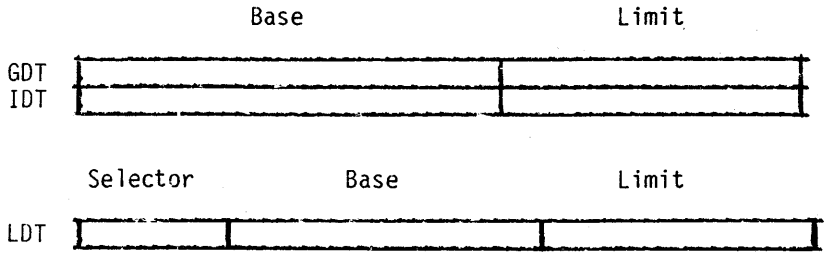
The normal program visible segment registers in PM286 are CS (Code Segment), SS (Stack Segment), DS (Data Segment) and ES (Extra Segment). PM286 uses the segment registers to contain segment selectors instead of segment base addresses, as in the 8086. Each segment register has an associated descriptor register as shown.



Each of the segment/descriptor registers is composed of 5 fields. The selector field is a 16 bit field. It is the only program visible field. The AR field is a 8 bit field which contains the access rights of the segment. The base field is the 24 bit real address of the lowest byte address in the segment. The limit field is a 16 bit quantity which indicates the segment limit. The V field is a 1 bit field which, when set, indicates the valid status of the descriptor.

DESCRIPTOR TABLE REGISTERS

PM286 has three active descriptor tables, IDT, GDT and LDT. Three new registers are used to locate the currently active descriptor tables.



Each of these registers contains a 24 bit base field and a 16 bit limit field. The base field gives the real memory address of the beginning of the table. The limit field gives the maximum offset that may be used in accessing table entries. The selector field of the LDT gives the selector for the LDT descriptor. LDT descriptors must reside in the GDT.

TASK REGISTER

The Task Register (TR) is a new register required for PM286. This register points to the task state segment for the currently active task. This register is similar to a segment register, with selector, base, and limit fields. Only the selector field is readable under normal circumstances. Special semantics of the normal state transition instructions are utilized to select a new task state segment and associated descriptor.

FLAG REGISTER

PM286 adds three new flag bits to the 8086 flag register definition. The new flag bits are a single bit for nested context (NC) and two bits to indicate I/O privilege level (IO_PL)

The Nested Context flag indicates the valid presence of a back link to a previous task state segment in the current task state segment. See the discussion of the inter-task transition mechanism for the intepretation of this flag.

The I/O Instruction Privilege Level flags indicate the maximum privilege level permitted to perform I/O instructions. Level 0 is always permitted to perform I/O.

Alteration of the I/O Instruction privilege level flags (IO_PL) is restricted to level 0 and task switches.

INSTRUCTION POINTER

The Instruction Pointer register (IP) contains the 16 bit offset of the current instruction to be executed within the current code segment. Operation is identical to the 8086.

MACHINE STATUS WORD

The machine status word indicates the iAPX 286 configuration and status. It is not part of a task's state. The usage of the bits is as follows:

task switched (TS)
emulation mode (EM)
math unit present (MP)
protection enable (PE)

The task switched flag indicates that a task switch has been done. It is set under hardware control and reset under software control.

The emulation mode flag (EM) indicates that a coprocessor function is to be software emulated. If EM = 1 but MP = 0, then all escape instructions will be trapped.

The math present (MP) flag indicates whether a math coprocessor is present. If MP = 1 then escapes and waits will be trapped when TS = 1.

The protection enable (PE) flag indicates that PM286 mapping and protection rule enforcement will be performed. When reset, the iAPX 286 will emulate the 8086.

iAPX 286 INSTRUCTIONS

The iAPX 286 supports all 8086 instructions. Only new instructions or instructions whose semantics are changed to support PM286 are discussed below. The remainder of the instruction set is described in the 8086 Family User's Manual.

LOAD/STORE IDT, GDT

Four instructions are provided to load and store the contents of IDT and GDT (LIDT, LGDT, SIDT, and SGDT). These instructions move 3 words starting at the effective real memory address to or from the indicated descriptor table base/limit register. These instructions are executable only at level 0. The format of the 3 words is a one word limit, a 3 byte real base address, followed by an unused byte.

These instructions will not normally be utilized except for initialization.

LOAD/STORE LDT

Two instructions are provided to load and store the contents of the LDT register. LLDT (load LDT) takes as its operand a selector for a descriptor table. It loads the LDT register from the descriptor for the descriptor table.

SLDT (store LDT) stores the local descriptor table selector from the LDT register. LLDT is only executable at privilege level 0. SLDT is unprivileged. Note that LDT descriptors must reside in the GDT.

LOAD/STORE TASK STATE SEGMENT SELECTOR

Load and store of the task state segment selector register is performed by LTR and STR respectively. LTR instructions will not normally be utilized except for initialization. LTR is executable only at level 0. STR is unprivileged.

READ ACCESS RIGHTS

The Read Access Rights instruction obtains the access rights field of the descriptor selected by a selector . If the indicated descriptor is not legally accessible from the current task state, or if it does not exist, then only the flags are affected.

This instruction is useful for parameter verification operations which are more extensive than the verification of the RPL.

VERIFY ACCESS

The verify for Read and verify for Write instructions determine whether the action (read or write) is permitted on a specified selector. These instructions can be used to verify parameters without getting a protection fault.

READ SEGMENT LENGTH

The Read Segment Length instruction loads the length of a segment into a register. If the indicated descriptor is not legally accessible from the current task state then only the flags are affected. It can be used in verifying a parameter without getting a protection fault.

ADJUST RPL

The Adjust RPL instruction changes the RPL field of a selector, allowing a procedure to set it to the maximum of the original value and some specified level.

LOAD/STORE MACHINE STATUS WORD

These two instructions respectively load the MSW from memory and store it to memory. Load MSW may only be executed at level 0. Store MSW is not privileged.

CLEAR TS FLAG

This instruction resets the TS flag. Setting of the flag will be done implicitly when the hardware performs a task switch. The clear TS flag instruction may only be performed at privilege level 0.

GENERAL REDEFINITION OF 8086 INSTRUCTIONS

The major change in the semantics of the 8086 instructions is the enforcement of protection rules and the formation of real memory addresses. The majority of the 8086 instructions will perform the same functions under PM286. Move from a segment register will store the currently defined selector if the register is valid, else zero.

Exceptions to the above are explicitly discussed below.

MOVE TO SEGMENT REGISTER

PM286 changes the semantics of Move to Segment Register. The move of a selector to a segment register causes an immediate descriptor table reference. The selected descriptor is obtained from the table and protection rule checking is performed. The move to CS operation is not permitted and will result in a protection violation trap.

LONG CALL/RETURN

The long call operations have been augmented in PM286. The same machine language codes are used.

A long call containing a executable segment selector will result in the logically identical operation to that provided in the 8086. However, the selector may also be a call gate, task gate or task descriptor.

Usage of a call gate may result in either an intra-level or an inter-level transition.

Usage of a task gate or task state segment descriptor will result in a task switch. A task switch is accomplished by the storing of the entire processor state in the outgoing task state segment, the loading of a new processor state from the indicated (incoming) task state segment and the saving of the outgoing task selector in the incoming task segment linkword. Execution then continues with the new task state.

The inter segment return reverses the action of an intra-level or inter-level call operation if a task switch was not made. An inter-level return will result in popping the old SS and SP values from the stack, thereby restoring the less privileged stack. RET n instructions will result in the addition of n to SP at the new level.

An IRET via the task selector stored in the task linkword is required in order to reverse the operation of a call or interrupt via task gate or call or interrupt to task descriptor. An inter-segment return (RET) may not be used to perform this operation.

INTERRUPT/IRET

PM286 supports a broad range of interrupt functions. In all cases, control is transfered via the selected descriptor in the IDT. The IDT may contain a trap, interrupt, or task gate.

An interrupt through a trap or interrupt gate operates much like an inter-segment call. The differences are that an interrupt causes the flag register to be pushed into the stack before the CS and IP, and that an interrupt changes the NC, TF and, perhaps IF flags. The nested context flag is reset. The procedure called by a trap or interrupt gate must return via an interrupt return instruction to pop the flags. An interrupt via an interrupt gate will additionally reset the interrupt enable flag. Protection fault interrupt operations will also push the error code onto the final stack. The error code is the last value to be pushed.

An interrupt via a task gate will operate in exactly the same way as a call via a task gate. Protection fault interrupts will additionally push the error code onto the stack as the last operation.

The IRET is the reverse of an interrupt. IRET uses the nested context flag to determine if the return is inter-segment or inter-task.

If the nested context flag is off, then IP, CS and flags are popped. If the requested privilege level (RPL) in the CS selector is different from the current privilege level (CPL), then SS and SP are also popped, thus switching stacks.

If the nested context flag is set, then IRET performs a task switch to the task state segment indicated by the selector in the linkword of the outgoing task state segment. The nested context flag is reset before storing the outgoing (initial) task state.

INTER-SEGMENT JUMP

An inter-segment jump may select a code segment, call gate, task gate or task segment. A jump to a different privilege level within the same task is not supported and will result in a protection violation. A jump to a context gate or task state segment will cause a task switch. The current task state is saved and the new task state becomes the processor state. The nested context flag is not set.

PRIVILEGED INSTRUCTIONS

PM286 restricts the execution of some instructions in order to protect the data structures and registers required by the protection mechanism and the state of I/O devices.

The following instructions may only be executed at level 0:

LGDT	Load GDT
LLDT	Load LDT
LIDT	Load IDT
LTR	Load Context Block Selector
LMSW	Load MSW
CTS	Clear TS flag
HALT	Halt

Additionally, I/O Privilege level will remain unchanged by a POPF or IRET operation except at level 0. POPF does not change the nested context flag except at level 0. A task switch will also change the I/O privilege level and nested context flags.

I/O instructions, and interrupt enable/disable are restricted to execution at a floating level. The least privileged level which may execute the following instructions is indicated by the I/O privilege level value in the flag register:

IN
INW
OUT
OUTW
STI
CLI
INSB
INSW
OUTSB
OUTSW

Additionally, modification of the IO privilege level, and interrupt enable, is restricted. The IO privilege level flags are only modified at privilege level 0 (the most privleged level). The interrupt enable flag is only modified at privilege levels that may perform IO. Execution of set and clear interrupt instructions at privilege levels not allowed to modify the interrupt enable flag will cause a protection trap. A POPF or intra-task IRET instruction that illegally attempts to modify the IO privilege level, nested context or interrupt enable flags will not change these flags.

PROTECTION FAULTS

A protection fault will cause a trap, an interrupt that is unmaskable. There are a large number of possible protection violations. Since it is quite difficult for software to interpret them without hardware support, the iAPX 286 uses a 16 bit error code and several new interrupt vectors to identify protection violations.

When a protection fault is detected a trap is generated. The trap may invoke a trap, interrupt or task gate depending upon the gate type in the IDT; the fault may be handled by the task that caused it or it may be handled by a different task.

Protection faults can be classified into two types: either an implicit request for service or a program error. Stack overflow and not present faults are examples of implicit service requests. An attempt to write into a read only segment or a regular limit violation are examples of program errors. The iAPX 286 is designed to ease the handling of service request faults. The different types of service requests use different interrupt vectors. By contrast, many different types of protection faults use the same general protection fault vector.

Furthermore, instructions that cause service request faults are generally re-startable without any back-out on the part of the fault handling software. Re-startable means that it appears that the fault was invoked immediately before the violating instruction was started. The saved instruction pointer points to the first byte of the violating instruction and all other saved machine state is exactly as it was before starting execution of the violating instruction. In this case, if the fault handler clears up the fault condition and performs an IRET the (previously) violating instruction will execute properly. Except for execution time, it will appear that there was no fault. The iAPX 286 does not provide full restartability on all fault types. The description of each interrupt will indicate whether it allows instruction re-start.

Some of the protection traps cause a one word error code to be pushed on a stack. The error code is always the last thing to be pushed and is pushed onto the stack that will be active when the trap handler begins execution. This ensures that the trap handler will not have to access another stack segment to find the error code.

PROTECTION INTERRUPTS

INVALID OP-CODE

When an invalid op-code is detected interrupt 6 is invoked. The saved IP will point at the invalid op-code. No error code is pushed. It may be handled within the faulting task. This interrupt allows full restartability.

MATH ADDRESS ERROR

This interrupt is used to signal that the math coprocessor has overrun its segment limit. It is generated by the coprocessor Data Channel during the limit test which is performed on each transfer of data between memory and math coprocessor.

INVALID TASK STATE SEGMENT

This interrupt is invoked when it is discovered that a task state segment is invalid. It is essential that this interrupt be handled through a task gate.

This fault is invoked if the task state segment is too small, if any of the LDT, SS, CS, DS or ES selectors are invalid or point to inappropriate descriptors or if the LDT is marked not present. The fault is generally found in an inter-task transition, it is also detected in an inter-level transition if the new stack selector points to an invalid descriptor.

If the fault was detected on an inter-level transition then the fault handling task will be linked to the original task and the IP stored in the original task's state segment will point at the offending call, or interrupt instruction. The previous task's IP will point at the next instruction if the transition was being made in response to an external interrupt. The error code will be of the form INDEX, TI, 0, EX where index and TI form the selector that the processor attempted to use as the new stack selector. EX = 1 indicates, that the transition was on behalf of an external interrupt. EX = 0 indicates that the transition was not on behalf of an external interrupt.

If the fault is detected in an inter-task transition to a task state segment that is too small then the fault handler's task state segment will be linked to the original task state segment. If the fault is detected in an inter task transition to a task state segment that has some other problem (LDT, SS, CS, DS, or ES selector or descriptor inappropriate, or LDT not present) then the fault handler's task state segment will be linked to the new one. In either of these cases the error code will be of the form index, TI, 0, EX, where index and TI are taken from the selector that points to the offending descriptor. EX indicates whether the transition was in response to an external interrupt or not.

NOT PRESENT

This interrupt is invoked when an attempt is made to load a segment or use a control descriptor that is marked not present.

There are two exceptions. An attempt to load a not present LDT segment in a task switch results in an invalid task state segment fault. An attempt to load a not present stack segment as part of an inter-task or inter-level transition results in a stack fault.

The error code is of the form INDEX, TI, I, EX. The index field is the table index of the descriptor. The TI field is the table selector, undefined if the descriptor is in the IDT. The EX field is 1 if the fault was detected while receiving an external interrupt, 0 otherwise. The I field is 1 if the descriptor is in the IDT, 0 otherwise.

If a not present fault is detected in the loading of CS, DS, or ES in a task switch then the selectors for the other segment registers will have been loaded but their descriptors may not be loaded; hence these other segment registers will not be valid for memory accesses. If control is transferred out of the faulting task and then returned back after the segment is marked present then the other segment registers will get loaded in the return. The stack segment will be checked before CS, DS or

ES so that it will be usable and the not present fault handler can operate, within the faulting task.

The not present interrupt fully supports instruction restart.

STACK FAULT

This interrupt is invoked when a stack underflow or overflow is detected, and when a not present stack segment is encountered in an intertask or inter-level transition. An error code is pushed. It has the form: index, TI,0,EX. The index and TI (table indicator) fields form a selector for the stack segment involved. The EX field indicates whether the fault was detected while trying to process an external interrupt or not. EX = 1 indicates that an external interrupt was in process.

It is expected that this interrupt will be fielded through a task gate, since the interrupt handler may not otherwise have a valid stack segment. However, if the software can assure that the stack used by the stack fault handler is valid, then the fault can be handled within the same task.

This interrupt fully supports instruction restart.

This interrupt is inactive in Compatibility Mode except for the very unusual case in which a stack is aligned to odd addresses and there is an attempt to push or pop a word at the maximum effective address, i.e., the high order byte is outside the segment.

GENERAL PROTECTION

The general protection fault is invoked on all protection faults not covered by the previous cases. In the case of limit read or write violations the error code is zero. If the violation is detected on attempting to load a segment register or use a control descriptor the error code will have the form index, TI, I, EX. The index and TI fields came from the selector being processed when the fault was detected. The EX field is 1 if the fault was detected while an external interrupt was being processed; the EX field is 0 otherwise. The I field is 1 if the index pertains to the IDT; it is 0 otherwise.

On general protection faults the original machine state is not always saved. Therefore full instruction restart is not provided in all cases.

This interrupt is activate in Compatibility Mode except for two cases. One is for the case in which an attempt is made to execute an instruction which is not entirely contained on the current code segment. The other case is for attempts to read or write a data word at the maximum effective address, i.e., the high order byte is outside the segment.

MULTI-PROCESSOR CONSIDERATIONS

A great variety of multiprocessor and coprocessor configurations are supported by the iAPX 286 product line. For the purposes of PM286 it is sufficient to examine the impact of all attempted concurrent usages of descriptors in the system. This prevents the necessity of enumerating and examining all of the possible system configurations while assuring the adequacy of provisions for these cases.

MATH UNIT CO-PROCESSOR SUPPORT

The iAPX 286 family includes a math coprocessor unit that is software compatible with the 8087. Much of the its state is stored internally. Any system that uses an math coprocessor must ensure that a task's math unit state is protected. It must also ensure that the math unit itself does not violate other protection rules. Support for this is built into the hardware architecture of the iAPX 286. Furthermore, the hardware supports sharing of the math coprocessor between several tasks.

The math coprocessor state is quite large. It may consist of up to 47 words. Thus saving and restoring the math coprocessor state is a relatively expensive operation, and should not be performed frequently. Typically only a few of the active tasks in a system will use the math unit. Therefore it would be wasteful to do a math coprocessor state save and restore on each task switch. The preferable alternative is to have the system keep track of which task the math coprocessor is assigned and to only swap the math coprocessor state when some other task attempts to use it.

The iAPX 286 supports math coprocessor sharing by providing Math Present (MP) and Task Switched (TS) flags. If a math unit is present then the MP flag should be set at initialization time and left set thereafter. When the TS flag is on the use of an escape or wait instruction causes a trap. The OS may thereby keep track of the task to which the math coprocessor is assigned at any time. The first time a task tries to use the math coprocessor after a task switch, a trap is generated. The operating system will check to see if the math coprocessor is allocated to the current task. If so the TS flag is turned off, and a return from the trap handler is made. If it is found that the math coprocessor is not allocated to the currently active task then the math coprocessor state will be swapped, the TS flag turned off, and a return from the trap handler performed. The active task can then use the math coprocessor.

CONCURRENT ACCESS TO PROTECTED ENTITIES

This section will examine the permitted or prevented multiple usage of all protected entities in the system by multiple processors.

TASK STATE SEGMENTS

A Task State Segment represents a task which is a single sequential thread of execution. Any attempt to concurrently execute a task by two or more processors must therefore be prevented. The status of the task, as indicated in the descriptor, effectively prevents any attempted concurrent execution.

DESCRIPTOR TABLES

Descriptor tables are owned by processes. A process may consist of more than one task. Each task in a process may be in a different and independent state of execution including active execution. Therefore, descriptor tables may be actively utilized by concurrently executing tasks. The problems occur only when the contents of the tables are modified.

Generally, multiple usage of gates and segments are permissible. However, the contents of segments must be modified in accordance with a synchronization protocol which is outside of the scope of PM286. The base instruction set does provide the necessary indivisible semaphore operation.

The multiple usage of gates is generally permissible. The problem occurs when the entity pointed to by the gate is in a dynamically changing state. Code segments may not be modified and therefore may be utilized by multiple concurrent tasks. Coordination, via semaphores, will be required for dispatch operations in order to prevent excessive frequency of software induced protection traps on task state segment collisions. Hardware induced collisions on task state segments may occur unless there is substantial coordination of the interrupt and trap vectoring between multiple concurrent processors.

Ordinary interrupts may be structured as reentrant procedures. Alternately, a separate IDT may exist for each processor, thereby providing separate tasks for handling each interrupt for each processor. Separate IDT's may be conveniently provided by placing each processor's IDT at a different address, or by placing each IDT at the same address in physical memory that is local to each processor. Alternately, separate GDT's may be provided for each processor, thereby achieving the same result of permitting separate tasks to handle the same interrupt vector on a per processor basis.

DESCRIPTORS

Multiple concurrent usage of descriptors is not a problem as long as the descriptors do not change. However, provision must be made for the dynamic modification, deletion and creation of descriptors. These conditions require the provision for accomodating the hardware dynamic update of descriptors and re-validation of potentially stale descriptors in the descriptor registers.

The only hardware dynamic update of the descriptors occurs with the accessed bit in segment descriptors and the busy bit task state segment descriptors. These operations are made indivisible in order to prevent any possible conflict.
Validation of on-chip descriptors is best performed upon software signal, subsequent to marking descriptors not present but prior to modifying the data structures indicated by the descriptors. Consideration has been given to providing a separate signal which forces revalidation of all active descriptors. However, the provision of a separate signal is not necessary since a simple re-entrant procedure that is invoked by an inter-processor interrupt and that causes the re-selection of all active segments is sufficient. The infrequency of this operation makes the software overhead acceptable.

SELECTORS

The deletion and creation of protected entities raises the issue of the re-usage of selectors. In general, it would be most desireable to never re-use a selector. However, this strategy is not practical due to the finite number of indices and the excessive size of the descriptor tables resulting from this strategy. PM286 does not make any explicit provisions for the re-usage of selectors. A software policy which controls the re-use of selectors to the same privilege level and the aging of selectors between protected entity destruction and creation of a new protected entity with the same selector should be sufficient for most, if not all, system requirements.

iAPX 86 - iAPX 286 COMPATIBILITY

In compatible mode the iAPX 286 is software upward compatible with the iAPX 86 except for the following cases:

1) A program that uses the iAPX 86 in a way that is not documented in the 8086 User's manual may run differently on the iAPX 286.
2) A program that is dependent on instruction execution time will work differently on the iAPX 86 than on the iAPX 286.
3) The effect of instruction pointer stepping beyond 64K is different on the 286 than on the 86.
4) In the 86 the divide error trap leaves the return link pointing to the next instruction, while in the 286 the return link points to the divide instruction.

In PM286 mode there are additional restrictions on the iAPX 86 programs that are upward compatible. Clearly, any programs that attempt to access beyond the logical bounds of a segment are incompatible with the 286 (and are likely incorrect). The other compatibility restrictions are as follows:

1) The program must not write into the code segment.
2) The program must not perform any operations on segment numbers (other than to move them).
3) The program must not use a segment register for temporary storage.

It is likely that the design of a protected system will impose additional restrictions on what programs are compatible with PM286. Almost certainly, memory management, task management and I/O will be made privileged operations. Therefore, these operations, and other operating system functions, should isolated from application level programs.

THE iAPX 286 ARCHITECTURE:

MEMORY MANAGEMENT AND PROTECTION MODEL

R. Childs
Intel Corp., Santa Clara, CA

SUMMARY

The iAPX 286 architecture provides a comprehensive system model for the new microprocessor application environments. These applications require a large number of processes concurrently in execution and highly dynamic memory allocation. Key requirements directly supported by the iAPX 286 are the management of memory space, the protection of the system from users, the protection of users from each other and the ability to rapidly service interrupts.

The architecture provides a highly efficient memory management and multi-level protection structure, based on a four ring hierarchial model. Explicit control is provided for relating a separate virtual address space to each task. The virtual address space is one gigabyte (2^{30}) per task and the real memory space is sixteen megabytes (2^{24}). Both automatic task switching and level crossing (gatekeeper) operations are directly supported for highly efficient operating system and interrupt service.

The architecture is being implemented on a single VLSI chip. Portability of 8086 based application software to the iAPX 286 is directly supported as an integral part of the architecture.

INTRODUCTION

The development the industry standard 16 bit microprocessor the 8086 opened an entirely new and unserved class of applications to microcomputer based systems. The applications in this range have performance requirements substantially above that achieveable by prior microprocessors and larger systems were not cost effective. Hence, this class of application had not been effectively served by digital systems.

Such applications are typified by highly complex computing requirements serving multiple users (either human or electronic). Performance requirements are very demanding in both the throughput and response time dimensions. A full multi-tasking environment supported by large real memory, dynamic swapping, a wide range of peripherals and comprehensive executive services is required by these applications. The reliability of the software and controlled access to data are also key issues. The processor architecture must support operating system functions within a full virtual memory protected environment. Very little performance impact is tolerable for protection enforcement.

The range of services required by the application from the operating system is extremely rich and usually organized into a hierarchy. This hierarchy may be implemented as a set of procedures supported by asynchronous tasks which are independently dedicated to various portions of the systems activities.

A wide range of other functions must be included in lower layers of the operating system to regulate the usage of the system resources by the applications. This set of services includes, at the lowest level, the system scheduling and memory management functions. Above that level, message passing, process synchronization and device scheduling services are provided. Application visible services come at the next level of the hierarchy. These include file management and file access control, system command languages, specialized report formatting and general editors.

MEMORY MANAGEMENT

The underlying idea of memory management is to allow system software to control the allocation of space in real memory (and in secondary memory) without regard to the specifics of an application program. Abstraction of program visible addresses from physical addresses into virtual addresses supports this functional requirement.

VIRTUAL ADDRESSING

The abstraction of logical (program visible) addresses from real addresses requires a separate supporting mechanism. This translation mechanism is utilized, during program execution, to map the logical addresses manipulated by the application into physical addresses.

The customary approach is to provide tables, managed by the system software, which record the mappings between the virtual addresses and the real addresses. The supporting hardware and/or software then access the translation tables to compute real addresses during application execution.

The provision of a separate logical address space for each task is highly desireable for task isolation. This permits the completely independent compilation and loading of application programs. The support of independent compilation and loading requires the provision of separate translation tables for each task.

SEGMENTATION

The organization of the iAPX 286 translation tables is based upon the concept of segments. Segmentation provides a variable size partitioning of the virtual address space. Segments are allocated on a logical basis and are directly related to the structure of the programs and data.

The logical partitioning of information into segments is very convenient for address translation, preparation of modular programs and, as we will see below, the creation of a strong protection mechanism.

ADDRESS TRANSLATION

An address pointer, at the logical (virtual) level in a segmented system consists of a segment number and an offset into the segment. The translation is then a direct computation. The real memory segment base address is obtained from the translation table. Addition of the offset to the base calculates the real address of the byte referenced by the pointer.

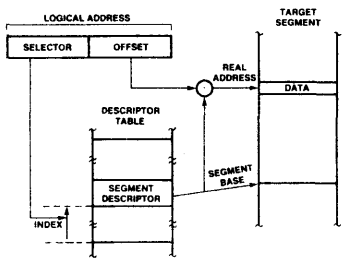


FIGURE 1
ADDRESS TRANSLATION CONCEPT

An iAPX 286 pointer is a 32 bit quantity. The segment number is represented as a 16 bit quantity called a segment selector. A maximium of 16,343 segments may reside within a single task's logical address space.

TABLES AND DESCRIPTORS

The iAPX 286 architecture uses three tables to support the address translation process. These tables are linear arrays of descriptors. The segment selector indexes directly to the desired entry in the translation table. Each table is of variable size and is accessed in response to the installation of a selector into a segment register. These tables are called the global, local and interrupt descriptor tables (GDT, LDT and IDT respectively).

Each descriptor is an 8 byte quantity. A segment's descriptor contains a 24 bit real memory base address, a 16 bit limit, and type information. The type information includes a present bit. The present bit indicates that the segment is either present in real memory or absent (presumably on secondary storage). Attempted access to a segment marked not-present generates a not-present fault interrupt.

The logical address space of a task is defined by the three tables. A single GDT and IDT are a part of every task's address space. The GDT is a separate table accessible to all tasks, thereby avoiding duplication, since many common services exist in a system. The IDT is utilized to provide vectoring for interrupt services. An LDT is an integral part of each task's state. The LDT represents the task's private descriptors. The system, of course, can have multiple LDTs. The GDT and LDT's may each have a maximium of 8K entries. A bit in the segment selector is utilized to distinguish between the LDT and GDT.

The IDT is implicitly selected by the interrupt operations and contains a maximium of 256 entries.

EXPLICIT CACHING

Logically, the translation mechanism must reference the translation tables on each memory access. The overhead of continual direct reference is a prohibitive cost in terms of performance and memory cycles. The answer to this problem is a mechanism which minimizes the frequency of references to the translation tables.

Most translation techniques rely upon some form of caching, frequently with program intervention, to minimize the hardware requirements. Common techniques include mapping registers[1] (e.g.: PDP 11/60) and associative caches[2] (e.g.: IBM 370). The mapping register approach uses a special block of external registers for address translation. System software is responsible for maintenence of the translation register contents. Fully associative caches avoid much of the software overhead of mapping registers but introduce a significant amount of associative memory and control hardware. Both techniques introduce severe instruction restart problems for handling not-present faults.

The iAPX 286 technique is called explicit caching. FIGURE 2 & 3. The basic notion is that the compiler has extensive knowledge of which portions of the logical address space will be required at any point in the program execution. The object code contains, as part of the executable flow, explicit instructions identifying the specific portions of the virtual address space that will be needed in the future.

This provision of explicit instructions to the caching and translation unit has a number of benefits. First, it is predictive, thereby relieving the caching unit of cache management responsibility and reducing the hardware requirements. Secondly, the explicit segment selection process includes immediate accessing of the translation tables. Thus, the segment not-present fault is immediately detected during the execution of a limited number of instructions. The necessity for having all instructions fully restartable for segment not-present conditions is thereby avoided. The result is a substantial economy in the amount of hardware required to support segment not-present conditions. Finally, the explicit knowledge, by the compiler, of the translation cache contents permits a major optimization in object code. Operations which do not change the contents of the cache need not contain a full pointer. Instead, they need only contain a short form of the pointer consisting of an indication, either explicit or implicit, of which cache entry is to be referenced for the translation process. This technique is the provision of a segment "nickname", instead of a segment selector, which greatly increases the density of code. In the case of the iAPX 286, the savings averages 14 bits per instruction (40%).

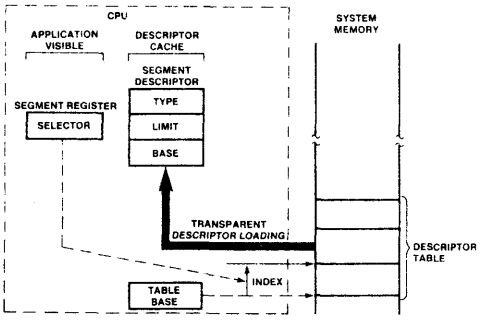
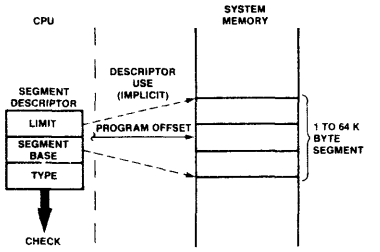


FIGURE 2
iAPX 286 DESCRIPTOR LOADING



THE RESULT:
POWERFUL MEMORY PROTECTION AND MANAGEMENT UPWARD COMPATIBLE FROM iAPX 86 WITH NO PERFORMANCE DEGRADATION

FIGURE 3
iAPX 286 ADDRESSING

The combination of the above techniques permits incorporation of memory management within a single VLSI processor chip. The result, when compared to multi-chip solutions, is substantially performance improvement.

PROTECTION

THEORY

The fundamental notion of protection is the continuous control and checking of all operations in the system based upon the rights conferred by a system protection policy. The two basic elements in protection are subjects and objects[5]. The subject of protection is the active element in the system, namely the executing task (or tasks) at any given instant. The objects of protection are all of the information in the system.

A protection policy algorithm determines access rights on the basis of the logical

properties of information for a specific domain of execution. The logical properties of information are applied to sets of information (e.g.: segments, files, etc.). The domain of execution is a controllable state of execution of a task. The access rights are then established for a domain to an object.

A protection policy can be enforced to any granularity, from single bytes to entire files. The checking of each access to objects of protection (information) by subjects (tasks) is mandatory in order to achieve comprehensive enforcement of the policy.

The basic protection concepts that must be supported by the processor architecture are the following:

- Logical grouping of information structures into objects of protection.
- Efficient control of the domain of execution.
- Recording of access rights to an object within a domain.
- Efficient and continual enforcement of the access rights.

PROTECTION MECHANISM

The iAPX 286 hardware recognizes the subject of protection as the executing task. The objects of protection are represented by descriptors in the descriptor tables.

The virtual address translation mechanism discussed above is a good starting vehicle for the creation of a powerful protection mechanism since it is highly efficient and utilized for each and every memory access. The translation mechanism contains many of the key protection support elements; the grouping of information into logical segments, the association of the translation information with a task, the continual utilization of an efficient mechanism on every access to memory, and the continual knowledge of the task's state of execution. The addition of domain and access control information to the address translation mechanism leads to a highly efficient protection mechanism.

The key issue to be faced in the development of a protection mechanism is the categories of access rights (descriptor typing) and granularity of execution state (domain) control. Many systems are based on a simple two level (user/supervisor) categorization. More sophisticated systems, such as Multics[3] or the VAX 11/780[4], offer 3 or more hierarchial levels.

The iAPX 286 provides a 4 level structure which is capable of supporting separation between kernel, executive, system services and application domains. A task is always executing in only one of these levels. Each descriptor is also assigned a level. The descriptor's level and type information are recorded in the type field of the descriptor. Additional type information for descriptors is discussed below.

The level structure creates the primary subdivision of the task's address space into domains of execution. The levels are intepreted as a fully ordered hierarchy of access. A task, at any instant, is executing at only one of the 4 levels. This level is called the current privilege level. The privilege level assigned to a descriptor is checked for visibility, in the hierarchial sense against the current privilege level. Visibility requires that the privilege level of the descriptor is at the same or a lesser privilege than the task's current privilege level. Visibility is also dependent upon the type of descriptor, as discussed below. FIGURE 4.

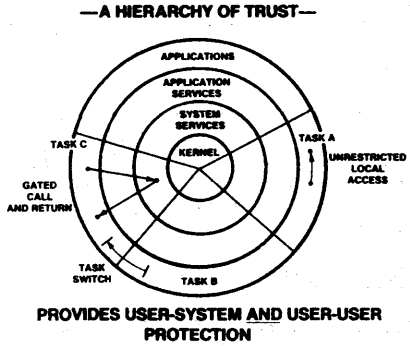


FIGURE 4
iAPX 286 PRIVILEGE LEVEL HIERARCHY

Failure to meet the visibility criteria upon an attempted access will cause a protection fault. Otherwise, the normal operation of accessing the descriptor will proceed, resulting in the installation of the base and limit values into the translation cache for subsequent usage.

The presence or absence of a descriptor for a segment in the tables is a powerful control upon the task's access to an segment. Each

task has a separate virtual address space. The creation of a descriptor in a table represents the controlled availability of the corresponding segment to that task. The lack of a descriptor in the virtual address space of a task for an segment represents complete denial of access; a direct result of the fact that a task is unable to form an address to any segment not represented in it's virtual address space.

SEGMENT TYPING

Two major components of segment typing have already been discussed; privilege level and presence. The privilege level indicates where the segment resides within the domains of a task. The presence indicator identifies whether the segment is in real or secondary storage.

Other information is required to record and enforce the permitted type of usage of the segment. The iAPX 286 supports two major type classes; segments and control descriptors.

Segments are the normal program visible portions of memory. They may contain data or executable code. Data segments are always readable and may be optionally identified as writeable. Executable segments may be classified as execute only or executable and readable.

Data segments are visible to a task only when the task is executing at an equal or greater privilege level than the segment. Executable segments are visible to a task only when the task is executing at the same level of privilege as the segment

CONTROL TRANSFERS

Control descriptors are utilized by the protection mechanism for task and privilege level transition control. Control descriptors are classified into three major categories; tables, task state segments and gates.

Descriptor tables are specially typed segments. Their contents are visible only to the hardware. The hardware will access a table as required to obtain a descriptor.

Task state segments are specially typed segments. Each task state segment maintains the entire state of a single task whenever that task is not currently executing. This information includes all of the visible register contents plus the stack pointers (one per level) and a selector for the local descriptor table (LDT).

Gates are utilized for protected entry into a new execution domain. The gate sub-types are call, task, interrupt and trap. The selection of a gate by the normal call or interrupt operations brings the task to a new state. Specialized supervisor call instructions are not required.

Call gates support call operations across privilege level boundaries. This provides direct hardware support for the classical gatekeeper operation[3]. The protection mechanism permits calls only to the same or more privileged levels. A call across a privilege level boundary involves the invocation of a more privileged code segment and a stack at the new privilege level.

The interrupt and trap gates are utilized for interrupt vectoring. They have the same privilege level constraints as call gates.

Task gates support direct task switch operations. A task gate may be utilized by call, jump or interrupt operations. The resulting task switch operation involves a complete change of the processor register contents; an operation which is performed entirely by the hardware.

CONCLUSION

The iAPX 286 is a high performance VLSI processor which provides a comphrensive and powerful memory management and protection mechanism. A large logical address space per task is directly supported. The protection mechanism incorporates complete multi-task support and a high efficiency four level protection hierarchy. The protection mechanism has been integrated within the semantics of a conventional instruction set, thereby avoiding specialized instructions, severe migration problems and excessive software overhead.

REFERENCES

1. "PDP 11/60 Processor Handbook", Digital Equipment Corp., 1977.
2. "IBM System/370 Principles of Operation", IBM Corp., August 1976
3. Organick, E. I., "The Multics System: An Examination of Its Structure", MIT Press, 1972.
4. "VAX 11/780 Architecture Handbook", Digital Equipment Corp., 1977.
5. Denning, P. J., and Graham, G. S., "Protection - Principles and Practice", Proc. SJCC, Vol. 40 (1972).

iAPX 286 MICROARCHITECTURE TO MAXIMIZE PERFORMANCE

by J. Slager, G. Louie, L. Gindraux

INTRODUCTION

Designers of previous generations of microprocessors have relied heavily on ever higher clock frequencies in order to provide increased throughput. As successive generations of microprocessors become more and more optimized, it becomes necessary to increase the use of parallelism and pipelining in order to realize significant increases in throughput. The internal circuitry of the iAPX 286 is organized in such a way that throughput is significantly increased even though major functional enhancements, which would normally be expected to reduce throughput, are also implemented.

The iAPX 286

A basic central processor can be implemented as a single logical unit with a single state machine. However instruction execution throughput will suffer since no parallelism between instructions is possible. An improvement in throughput can be obtained by providing either pipelining so that the single state machine can operate on more than one instruction at a time or by providing more state machines. The iAPX 286 is implemented with four logical units each capable of operating as an independent state machine and using pipelining. These units are called the Bus Unit, the Instruction Unit, the Execution Unit, and the Address Unit.

The Bus Unit

The Bus Unit provides the interface with memory and external input/output subsystems. It contains a bus cycle controller state machine and dedicated functional blocks for implementing code prefetch and Math Coprocessor data channel support. A bus cycle prioritizer examines bus cycle requests from four different sources and prioritizes them as follows:

1. External bus masters (HOLD Request).
2. Math Coprocessor data channel.
3. The Address Unit.
4. The Code Prefetcher.

It can be seen that the code prefetcher has lowest priority for bus cycles and, therefore, will prefetch code when there are no other demands for bus cycles.

The code prefetcher and the Math Coprocessor data channel both operate on real addresses which have been previously prepared by the Address Unit along with a real address limit required for memory protection enforcement. In this way both the prefetcher and data channel may generate bus cycles which are performed solely by the bus unit without assistance, other than initialization, by the other units.

During periods when the memory bus would otherwise be idle, the code prefetcher obtains code from memory under the assumption that the iAPX 286 is executing sequentially. Code obtained in this way is placed in a code queue where it is available for access by the Instruction Unit. Whenever the iAPX 286 ceases sequential execution; i.e., executes some form of branch, the code queue is flushed and the prefetcher is initialized with a new real address and limit. Whenever the prefetcher attempts to fetch from an illegal location, as indicated by the prefetcher limit, the Bus Unit refuses to perform the memory cycle and places a violation marker in the code queue.

The Instruction Unit

The Instruction Unit is designed to decode and format instructions in order to relieve the Execution Unit of this function so that instruction execution will be faster. The Instruction Unit obtains bytes of code from the Bus Unit code queue and prepares fully decoded instructions in its instruction queue.

The iAPX 286 instruction set consists of byte variable formats. The Instruction Unit contains a state machine which steps from state-to-state based on the value of each code byte as it is removed from the code queue. The state machine controls the filling of the instruction queue with fully decoded instruction data.

Capacity of the instruction queue is three instructions with each instruction formatted to include all the information necessary for instruction execution with the single exception of non-immediate data operands.

The Execution Unit

The Execution Unit is where actual instruction execution occurs. It contains the main

registers and ALU as well as several dedicated logic "boxes" for fast execution. The Execution Unit is controlled by a large Control ROM (CROM). Execution of a particular instruction consists of a sequence of micro instructions being supplied from the CROM. Part of the information contained in the instruction queue of the Instruction Unit is an entry point address into the CROM. As the sequence of micro instructions for any particular instruction nears completion, the CROM generates a signal causing the next CROM address to be taken from the instruction queue so that the next sequence of micro instructions begins with no time lost.

From time to time the micro instructions may specify that bus cycles should be performed for data reads or writes or for redirecting the Bus Unit prefetcher for program branches. In these case the micro instruction directs the Address Unit to compute a real address from operands supplied to it over internal data buses.

The Address Unit

The Address Unit performs three address calculations for each bus cycle. One calculation is to form the effective address as the sum of, as much as, two register contents and the address displacement from the instruction queue. This effective address is compared to the limit of the selected segment to determine if a memory protection violation has occurred. The final calculation is the addition of the effective address to the base value of the selected segment in order to form the real address.

In order to improve the speed of these address calculations, segment base, segment limit, and segment access rights are contained inside the Address Unit in an explicit cache. During the above mentioned address calculations the access rights of the selected segment are also tested for conformance with memory protection rules.

Effective address operands and command information are latched by the Address Unit so that the Execution Unit may proceed with other operations or even other instructions without waiting for the Address Unit to complete its operations.

The Address Unit is capable of continually reforming its result until the Bus Unit has finished its previous duties and is ready to accept the Address Unit result.

Instruction Processing

The focus of the iAPX 286 design was to reduce the number of micro instructions required for each instruction to bare minimum and to keep the Execution Unit constantly busy executing these micro instructions. The provision of the Instruction Unit allows advanced decoding and formatting of instructions while the Execution Unit is executing micro instructions for some previous instruction. Thus when the Execution Unit is ready to begin the next instruction, no clock cycles need be devoted to decoding or formatting. For example, the move immediate to register instruction executes in only two processor cycles on the iAPX 286, while the same instruction on the iAPX 86 requires four processor cycles. This speed up is largely due to the preformatting of the instruction in the Instruction Unit which places the immediate value in the instruction queue as part of the instruction information.

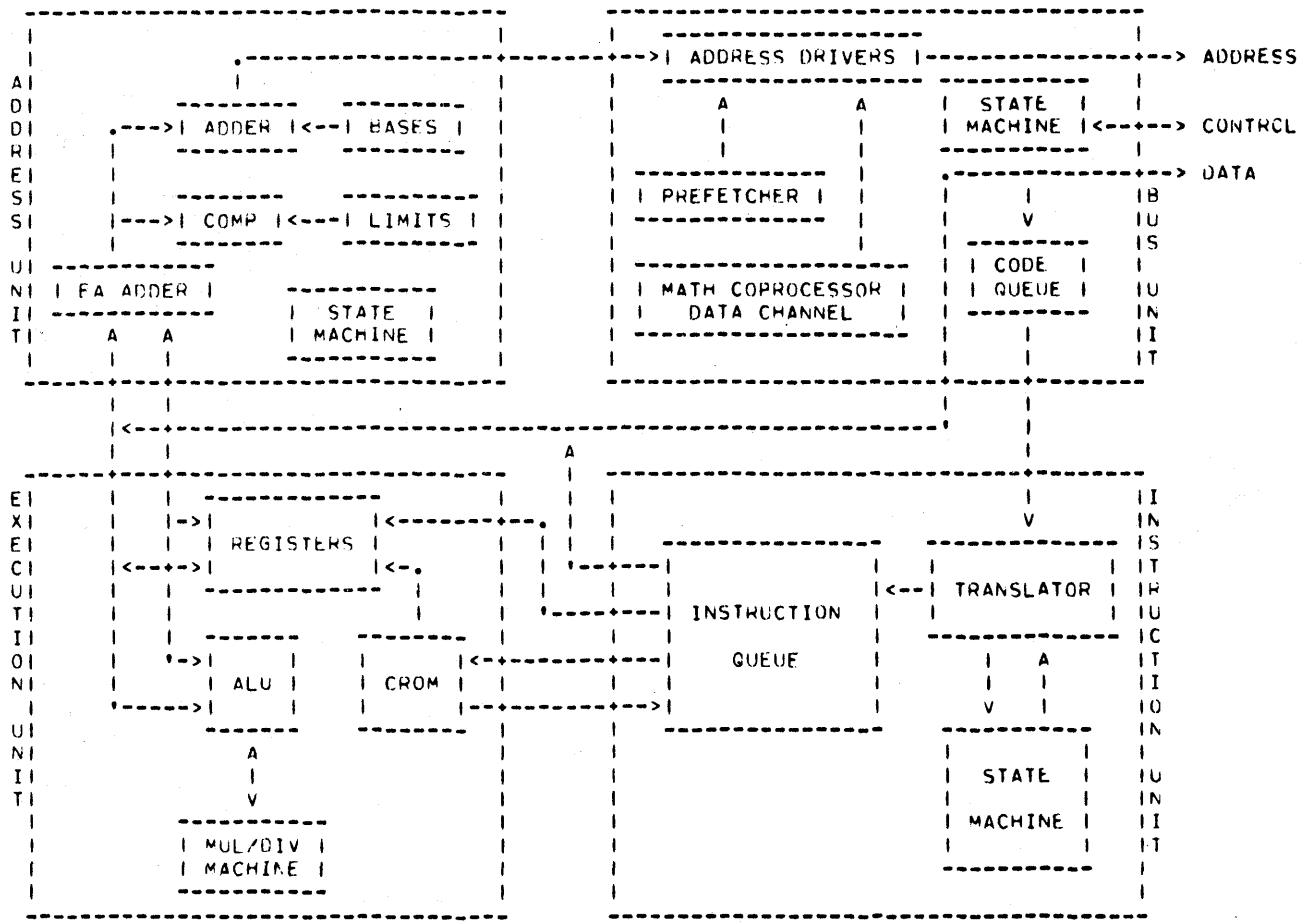
In addition, several logic boxes are provided inside the Execution Unit in order to insure fast instruction execution. For example, multiply and divide hardware allow the shift and add (or subtract) algorithms to be executed at the rate of one bit position per processor cycle as opposed to the six (or more) processor cycles required for the iAPX 86 which uses micro instruction loops.

Another large throughput enhancement is provided by the Address Unit which forms addresses and performs memory protection checking with dedicated hardware rather than micro instruction sequences. For example, data cycles on the iAPX 86 can increase by up to seven processor cycles above the minimum for complex address modes. This increase for the iAPX 286 is limited to one processor cycle.

Memory Cycles

In order for the internal logical units of the iAPX 286 to work together efficiently it is necessary that information be moved to and from memory fast enough to prevent the queues from running dry. Therefore a high bandwidth memory interface is required. The iAPX 286 features a highly optimized memory interface which pipelines bus cycles so that each bus cycle consists of three processor cycles but successive bus cycles can be performed at the rate of one bus cycle every two processor cycles. This provides a bandwidth of 8 megabytes per second while allowing memory access times of 242.5 nanoseconds from address valid to read data valid measured at the pins of the iAPX 286.

In addition to providing a high bandwidth bus, maximizing throughput also requires that the collision rate between bus cycles be minimized. The goal here is to insure that the prefetcher only runs memory cycles when the Execution Unit does not need the bus to perform data reads or writes. Naturally the prefetcher is given lowest priority when the Bus Unit prioritizes bus requests but more significant collision avoidance is provided by an advance warning sent from the CROM to the Bus Unit which informs the Bus Unit Control that no prefetch cycle should be initiated because a data cycle will soon be requested. A final bus efficiency feature of the iAPX 286 is the ability of the Instruction Unit to determine that a branch instruction has been prefetched and to inform the prefetcher that prefetching should be suspended. This feature reduces over-fetching and frees bus capacity for the Math Coprocessor data channel and for external bus masters.



I A P X 2 8 6 B L O C K D I A G R A M

The Implementation of Operating Systems with the iAPX ⁄86

Richard Markowitz
INTEL Corp.
Santa Clara, CA

The iAPX ⁄86 microprocessor is part of a new era in the development of complex system applications. Designs that formerly required the use of a sophisticated minicomputer will now become part of the expanding microprocessor world. Recognizing the present day problems of implementing such designs, the iAPX ⁄86 architecture incorporates a large repetoire of operating systems resources to support multi-user, multi-programming, memory-intensive systems. The key concepts of memory management, descriptor definition, protection, and access control are implemented in the hardware. The system designer can then concentrate on the aspects of operating system design that are compelling for his application.

iAPX ⁄86 Resources

Simplified operating systems implementation depends greatly on having a good match between the software and hardware architecture. The iAPX ⁄86 architecture implements the useful software concepts of task, segment and procedure entry point as primitive, hardware-recognized data structures. Each of these structures is represented in memory by a descriptor which is used by the processor for all addressing and control purposes . Segments are the unit of both memory management and protection. Code and data segments are distinctly typed. Memory segments have a uniform structure, regardless of type; they can range in size from 1 B to 64 KB in increments of a single byte, and can be aligned to begin and end on any arbitrary byte boundary. iAPX ⁄86 segments are both larger and more flexible than the 8KB segments available on the PDP 11/70; they are easily accessed by a simple machine instruction.

Segments can represent individual protected software structures or groups of structures which are assigned to a common segment by a translator or linker. The iAPX ⁄86 on-chip segment registers are used only for the purpose of caching descriptors rather than as general registers.

Descriptors used in addressing are always retrieved from hardware typed segments called descriptor tables. Three descriptor tables are in use at any one time. Descriptors will be generally created, managed and deleted by OS software as ordinary software data structures. After they are made valid and placed in a descriptor table, they are used by the hardware to access the information represented in the addresses they control. This operation is performed by loading a selector into a register. In programs, descriptors are referenced only with logical selectors, so that the physical memory addresses are invisible to application programs and confined to the protected tables. The operating system can easily manipulate the relocation or swapping of memory segments since physical addresses will appear neither programs nor on their stacks.

Task state image segments are hardware typed segments, recognized by the iAPX ⁄86 processor for use in task swap operations. These segments store the iAPX ⁄86 processor state of a task when it is not in execution. The task state image segment is defined by the iAPX ⁄86 for processor state only, but it is easily expanded to a larger memory segment (e.g. Task Control Block) that would contain coprocessor state and the user operating system software state. Creation, modification and deletion of task state image segments and overall scheduling of tasks are OS software functions. Actual task switching is provided in hardware.

In summary, the iAPX ∠86 architecture structure provides basic protected resources which are further developed by the operating system software. The iAPX ∠86 provides the manipulation operations that supports the system structure, however, The OS designer can straightforwardly extended the use and interpretation of the contents of segments and descriptor tables.

Protection: Hierarchy and Isolation

The general goal of protection is to allow programs or users with varying degrees of trustworthiness to coexist in the same system. As designs grow more complex and memories larger, it is increasingly important for the designer to provide for and control interactions between independent subsystems. Especially important is the protection of the operating system control program from the user applications. The simplest models of protection stop there, isolate the system program from the user programs through making certain powerful and dangerous instructions privileged. This first step allows the operating system to be written to work in an environment that is protectable from applications programs. However, today's complex system efforts involve the work of many groups over a long period of time, in many cases there will be OEM type developers who wish to take an operating system available to them and customize it for a particular applications without sacrificing the protection of their code from the user application. In a two-level system, there is only one protected domain available which will require the OEM to make a difficult choice between running his software either unprotected or in the same protection level as the base operating system. Both of these are unsatisfactory, the OEM must either trust the user software or be able to determine if the changes he has made have compromised the original base OS. A mutli-level system allows the OEM both worlds, protection from user without sacrificing the protection of the already implemented base OS and the clear delineation of responsibility.

A hierarchical implementation of the multiple levels allows the usual semantics of privileged operation to be followed, whereby the more privileged user can access the data structures of the lesser privileged, normally passed to him as pointer parameters, but is prevented from changing the data structures of the still more privileged users above him. The iAPX ∠86 architecture reflecting microprocessor applications provides four levels of hierarchy.

Complete isolation of user subsystems (separate protection domains for each task) can be obtained by allowing each task to make references only to those addresses that are specified as part of its local name space and establishing this local name space by the OS software. The iAPX ∠86 allows each task to be assigned a separate Local Name Space represented with an LDT (Tasks performing a common function e.g. multiple I/O drivers could share an LDT among themselves). Since most applications systems are designed with some shared common functionality, accessibility to a global name space via a Global Descriptor Table (GDT) is provided for representing services, data structures, or resources (tasks, buffers, mailboxes) that are specified for shared use. In addition to providing a locus for task descriptors, the separate global table reduces the need to replicate commonly used segments and entry points in each LDT, simplifying the creation and deletion of new tasks and LDTs, and thus assisting in the efficient implementation of transaction-oriented systems. When sharing is required between a limited number of applications or tasks and not a system-wide function, the OS kernel would support through primitives designed to move and copy descriptors as required among the descriptor tables of sharing tasks as defined in applications policy. In many applications the means for mutual communication and sharing can be established staticly at task initialization time with descriptors of shared areas avoiding dynamic allocation.

Protection:Access Control

System protection begins in the processor. Simple applications of protection are often the most useful: detection of stack overflow, or detection of an offset address beyond the bounds of a segment during instruction execution. Immediate faulting on these conditions will allow the operating system to inhibit corruption or unauthorized access to data before serious system problems begin to occur.

Access control by Read Write and Execute modes and the use of hardware segment typing increase the ability of the system to generate early warning of incorrect usage. One estimate is that as many as 95% of the errors detectable in run-time systems fall in this category. The alternative for the user to a protected architecture is often extensive parameter checking in software prior to use of an operation or even a completely interpretive approach. The iAPX 286 processor provides the system designer with access control for each memory reference. In addition to the above addressing errors, traps on illegal opcode and a variety of protection faults are supported as well.

The architecture of the iAPX 286 has been designed so that access errors will be detected prior to actual execution phase of the instruction, thereby avoiding the neccessity of undoing undesirable side-effects. Since every access to memory is made through the descriptors "cached" in the segment registers rather than through general purpose registers, the system audit trail will be clearer and the repair and maintenance effort will often be simplified. After the access error is detected, the user operating system interrupt handler or task for this fault type is entered with standard parametric information passed on the stack. This information will then be used by the user OS for an appropriate recovery strategy.

System Structure and Control Transfer

The iAPX 286 hardware performs major protection checking on procedure calls that actually imply protection domain changes and minor protection checks when selectors are loaded. The iAPX 286 procedure call may be intra-level within the same task, similar to the iAPX 86, in which case no protection check is required or it may cross privilege level boundaries or call a different task in which case a check is needed. Gated calls are always inward to more privileged code entry points, reflecting the philosophy of trust. The gated call both transfers control directly to a procedure entry point and and passes parameters to that procedure's stack. It is implemented as one instruction. Thus a kernel service can be directly called by a primitive, uninterruptable operation, reducing the complexity of its interface. The gated call provides a general means of crossing between levels and permits passing as parameters in the same local name space,"safe" copies of pointer arguments in the form of selectors tagged with level information which are to be copied automatically to the correct level stack of the invoked procedure. Since the iAPX 286 system is hierarchical and the saved code selector contains the privilege level of the caller, the called routine can verify that the privilege level requested in a pointer parameter is valid for that caller. Instructions are provided in the system to verify these pointers refer to accessible descriptors and adjust them without requiring OS protection trapping. Use of gates in the system structure support the concept of information hiding, as make available to the user task only the specified entry points in the service modules not the entire module. Conforming code segments fulfil the need for procedures that when called run always at the level of the caller and do not require level changes.

Use of this feature allows logical separation of function between the use of privilege for protection purposes and its use for universal availability (e.g. placing a library at level zero so that gated access is possible anywhere). The enhanced procedure call instruction is inexpensive, supporting the system designer in efforts to modularize his system and in making effective use of all privilege levels or in building a comprehensive virtual machine environment for the user task.

The iAPX 286 does not provide a hardware primitive for passing arguments between separate tasks (i.e. separate user protection domains), since the use and extent of sharing between tasks is considered a policy matter. The full operating system will implement the user-defined software protocols with a kernel procedure. Various policies may be chosen: shared segments can be provided to tasks for direct use, or an OS kernel service supporting communication between tasks can be invoked with the caller's message parameters.

OS Interface:
Synchronous/Asynchronous Calls

The key goal of a hierarchical architecture is to allow sharing of the same address space by functions at varying degrees of trust. One reason for wanting to do this is efficiency realized by synchronous OS calls. The iAPX 286 architecture supports operating system services that are invoked either synchronously (without task change) or asynchronously (with another task). Use of a synchronous call allows the OS service to be invoked uninterruptably and to be executed at the same priority as the calling user task, since the same task is still in execution. The synchronous call also does not require serialization and queue management functions if the servers is designed as re-entrant. The simplicity and efficiency of this call is especially valuable for requests with short service times, e.g. inquiry functions.

Designs could make further use of the hierachical system for protection and deadlock prevention purposes by restricting the use of critical regions of OS code or other uninterruptable (and of necessity highly trusted) functions to the level zero code common in each address space and even forbid such functions to be called from level three user code.

Asynchronous calls are more typical of longer duration OS services The iAPX 286 architecture provides an uninterruptable task switch via a task gate to perform the call and the option to mask interrupts at the point of execution of the new task. The task switch and the task backlink structure are oriented to high-speed interrupt processing. For non-interrupting situations when the target task is known to the switching task, the switch mechanism can be used to transfer directly to the serialization mechanism used by the target task, since tasks are protected by the hardware from concurrent user execution. Or the user can give up control of the processor to a master task scheduler which would perform a task dispatch.

Interrupt Structure and I/O

The iAPX 286 interrupt processing logic uses a separate interrupt (vector) decriptor table, the IDT. Since external interrupts occur independently of the executing task, and are dependent on a processor, this table is defined on a per-processor basis. It is expected however, that the designer will want uniformity in behavior for handlers of identical interrupts on each processor. The symmetry of the descriptor architecture dictates that interrupts, too, can be handled synchronously or asynchronously. Synchronous handling may be used for interrupts requiring rapid processing entry or for handling program traps and faults and servicing coprocessor exception conditions. For asynchronous interrupt processing, the processor will vector through a task switch gate in the IDT and begin execution of the separate interrupt task in its own protection domain at the defined protection level.

An asynchronous strategy is appropriate for tricky faulting conditions such as stack overflows, it might be be used also with software interrupts as well as with many external interrupts. The efficient interrupt processing on the iAPX ¿86 will not require the use of the highly privileged kernel software functions and forced execution at this highest level of privilege. This flexiblity will also encourage the OS designer to produce general purpose interrupt procedures rather attempting to take advantage of special knowledge of the local task execution context, a design tendency that can lead to large, rather obscure kernels.

The iAPX ¿86 processor state also contains an I/O privilege level register in which the user can set, on a per-task basis if required, the least privileged level allowed to do I/O. The operating system can thus force the user programs through its trusted system drivers, or when physical I/O capability is required export it to the less privileged levels. The drivers themselves can execute as synchronous services with the user task or asynchronously to it, perhaps in another protection domain.

OS Structures

The iAPX ¿86 was designed to help simplify OS design. The OS architecture that will use it most effectively will be structured in several levels rather than monolithic. It will probably use all four protection levels, although it may be designed to use fewer. Typically, its design will depend on a small highly reliable kernel function for its integrity. The operating system will comprise two basic sets of software:

The nucleus or kernel set, accessible as shared reentrant procedures within privilege level zero of each task providing the primitives that implement memory management, descriptor table management, intertask communication and synchronization. Resource management functions that must be intimate with the iAPX ¿86 physical addresses or I/O addresses e.g. interrupt control, timer and DMA peripheral coprocessor management may also appear here. Level zero privileged instructions include those that access real memory addresses and register resources, I/O devices can be physically controled at a protection level below the level of I/O privilege, set by a level zero instruction. This core portion of the OS will normally be accessed by the other levels through gates referencing the GDT.

The remainder of the operating system: programs which can be considered less trusted. They may be designed either as shared re-entrant procedure executing within the more privileged levels of each user task or as distinct (and possibly distrustful) tasks executing in independent protection domains communicating through a parameter mechanism provided by the OS. These outer layers will call the kernel primitives through the standard protection mechanisms. Most systems will probably contain a mixture of the synchronous and asynchronous implementation styles.

Sharing and Dynamic Memory Swapping

The structure of the iAPX ¿86 architecture is particularly designed for real-time applications and supports in hardware the most commonly used forms of sharing of code and data between programs. Some system designs may attempt to make use of more extensive or more parametric sharing of segments. The descriptor mechanism of the iAPX ¿86 provides a basis for such a software extensions which will copy or move the full descriptor as sharing is required. In general, the user will determine his own sharing policy, and keep track within some form of shared protection data base of the location and status of shared and possibly multiple descriptors. The iAPX ¿86 is designed for multiprocessor use so that a processor can cause itself and all the other processors in the system to revalidate their descriptors simply by causing an interrupt serviced by an interrupt task for each processor. The other processors then can be prevented from loading the descriptor undergoing modification until the change is completed.

Virtual Memory

The style of use of virtual memory with microprocessor systems is not well established at this point in time. Experience with time-shared multi-user minicomputers should not be applied to distributed microcomputer networks without careful analysis. The need for demand paging support in addition to segment swapping may not be economically justified and tends to produce a complicated system architecture. The iAPX ⁄86 has low-level resources that support the implementor of virtual memory systems. The virtual memory oriented present bit is inherent in the descriptor structure and is used to cause the OS fault used to initiate swapping. The descriptor access bit is set by the hardware when the descriptor is actually loaded on chip in order to support usage profiling of memory resident segments by the OS. These access bits can be periodically tested and reset by LRU algorithms. Complex systems applications with virtual memory requirements might use an asynchronously called virtual memory manager which might both manage the shared protection data base to support of sharing to the individual processors and the virtual memory hierarchies becoming in essence a file processor. The economic impact of microprocessor technology should increasingly relieve the working processor of need for managing large paged memories and replace it with a higher-level interface on a separate processor. Such a system would be suited for future distributed applications.

Intel Corporation
3065 Bowers Avenue
Santa Clara, California 95051
(408) 987-8080

Printed in U.S.A./T-2010/2K/0281/DG IP

32 BIT MICROPROCESSOR
- iAPX 432

INTRODUCTION TO THE iAPX 432 ARCHITECTURE

Manual Order Number: 171821-001

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

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

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

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

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

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

BXP	intel	Megachassis
CREDIT	Intelevision	Micromap
i	Intellec	Multibus
ICE	iRMX	Multimodule
iCS	iSBC	PROMPT
im	iSBX	Promware
Insite	Library Manager	RMX 80
Intel	MCS	System 2000
		UPI
		μScope

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

PREFACE

The Intel iAPX 432 represents a dramatic advance in computer architecture: it is the first computer whose architecture supports true software-transparent, multiprocessor operation; it is the first commercial system to support an object-oriented programming methodology; it is designed to be programmed entirely in high-level languages; it supports a virtual address space of over a trillion (2^{40}) bytes; and it supports on the chip itself the proposed IEEE-standard for floating-point arithmetic. Because it is so advanced, a discussion of the iAPX 432 architecture will unavoidably introduce concepts that are new to many readers.

The purpose of this document is to provide an accurate and comprehensive overview of the architecture, recognizing that not only are many of the concepts new, but that readers with widely divergent backgrounds are likely to be interested in the iAPX 432. Consequently, a considerable amount of background information will be supplied.

The Introduction is organized as follows: Chapter 1 introduces the concept of architecture and the various elements of an iAPX 432 system. Then the focus is narrowed to the architecture of the principal processing element, the General Data Processor (GDP).

Chapters 2, 3, and 4 provide an overview of three broad areas where the GDP architecture represents a significant advance over contemporary architectures, namely memory organization, data manipulation, and hardware support of state-of-the-art programming methodologies. These chapters motivate and explain most of the architectural features.

The *iAPX 432 General Data Processor Architecture Reference Manual* contains complete, detailed descriptions of all aspects of the architecture. It should be consulted whenever more information is required on any of the topics covered in this document.

CONTENTS

CHAPTER 1
COMPUTER ARCHITECTURE

PAGE

What is Computer Architecture? 1-1
The Hardware-Software Interface 1-3
Current Problem Areas 1-5
iAPX 432 Architecture 1-5
  Main Features 1-6
  Configurations 1-8
Topics Covered in this Document 1-9

CHAPTER 2
MEMORY ORGANIZATION

Fundamentals 2-1
Linear Memory 2-1
  Mapping Linear Memory 2-4
  Page-Based Mapping 2-6
  Access Rights Based on Pages 2-6
  Virtual Memory 2-7
Segmented Memory 2-7
  Mapping Segmented Memory 2-9
  Segment Types and Access Rights 2-10
  Access Control 2-10
Structured Memory 2-10
  Two-Level Mapping and Access Control 2-11
  Segment Types and Access Rights 2-13
  Virtual Memory and Dynamic Storage Allocation 2-14
  Complexes of Segments 2-15
Summary 2-17

CHAPTER 3
DATA MANIPULATION

What are Data Types? 3-1
iAPX 432 Operators and Primitive Data Types 3-1
  Characters 3-4
  Ordinals 3-4
  Integers 3-4
  Reals 3-4
Structured Data Types 3-4
Instructions 3-5
  Addressing Modes 3-6

PAGE

  Operand Stack 3-8
  Instruction Encoding 3-9
Summary 3-9

CHAPTER 4
PROGRAMMING ENVIRONMENT SUPPORT

The Software Crisis 4-1
  Modularity 4-1
  Security 4-1
  Concurrency 4-2
  Expandability 4-2
New Software Methodologies 4-3
  Type Managers 4-3
  Processes 4-3
  Architectural Support 4-4
iAPX 432 Object-Based Architecture 4-5
  System Objects 4-5
  Object Protection 4-6
Architectural Support for Type Managers 4-7
  Domain Objects 4-8
  Context Objects 4-8
  Calling Contexts 4-9
  Using the Inside Representation of Type Managers 4-9
  Using the Outside Representation of Type Managers 4-15
  User-Defined Types 4-15
Architectural Support for System Services: the Silicon OS 4-20
  Process Objects and Processor Objects 4-20
  Storage Resource Objects 4-20
  Simple Interprocess Communication Without Blocking 4-21
  Conditional and Surrogate Communications 4-23
  Process Blocking, Scheduling, and Dispatching 4-24
  Multiple-Processor Systems 4-25
Summary 4-28

CONCLUSIONS

TABLES

TABLE	TITLE	PAGE
1-1	A Comparison of the iAPX 432 Architecture and the Architecture of Conventional Mainframes	1-7
3-1	iAPX 432 Operators and Data Types	3-3
3-2	Ranges and Precisions of Short Reals, Reals, and Temporary Reals	3-4

ILLUSTRATIONS

FIGURE	TITLE	PAGE
1-1	Computer System Architectures	1-1
1-2	Configuration Architecture	1-3
1-3	Raising the Hardware-Software Interface	1-6
1-4	A Small iAPX 432 System	1-8
1-5	A Multiple-Processor System	1-9
2-1	Unmapped Linear Memory	2-2
2-2	Vulnerability of Unmapped Memory	2-3
2-3	A Simple Mapping Scheme	2-4
2-4	A Multiprogram Mapped Environment	2-5
2-5	Page-Based Protection	2-6
2-6	Comparison of Segmented and Linear Memory	2-8
2-7	Segmented, Mapped Memory	2-9
2-8	Two-Level Mapping	2-11
2-9	iAPX 432 Address Spaces	2-12
2-10	Shared Segments	2-14
2-11	A Complex of Segments	2-16
3-1	Primitive Data Types	3-2
3-2	iAPX 432 Instruction Format (3 Operands)	3-5
3-3	Addressing Modes and Structured Data Types	3-6
3-4	Using the Operand Stack	3-8
4-1	Objects and Object References	4-6
4-2	Object References	4-7
4-3	A Domain Object	4-8
4-4	A Context Object	4-10
4-5	Calling a Context	4-11
4-6	Changing the Access Environment	4-13
4-7	The Outside Representation	4-16
4-8	User-Defined Type	4-18
4-9	Basic System Objects	4-21
4-10	Sending and Receiving Messages	4-22
4-11	Surrogate Sending	4-24
4-12	Resending, Scheduling, and Dispatching	4-26
4-13	Multiple Processor System	4-27

CHAPTER 1
COMPUTER ARCHITECTURE

1.1 What is Computer Architecture?

The term *computer architecture* is often used as if it meant simply "the organization and design of computers." This rather loose definition must be made considerably more precise before we can study the differences between the architecture of the iAPX 432 and that of more conventional computers.

In practice, there are several distinct architectures within a computer system, each defined by a boundary between different levels of the system. Figure 1-1 shows an abstract picture of a computer system, with the simplest operations and functions on the bottom and the most complex, user-dependent operations on top. Each level makes use of the functions provided by the level below. The whole effect is like a stack of bricks, with each of the higher bricks supported by the ones below.

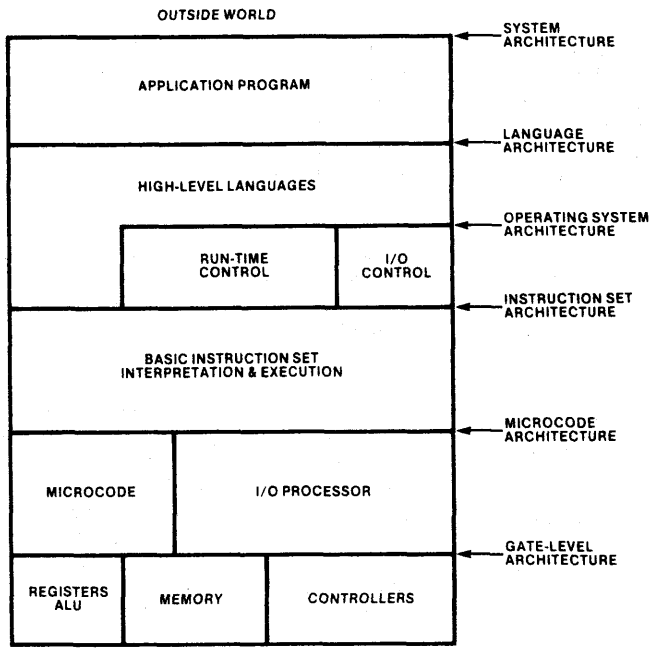


Figure 1-1. Computer System Architectures 171821-01

An architecture is the boundary or interface between two of these functional modules or bricks. It might be defined as *the functional appearance of the system below an interface to a user above the interface*. Designers above an architectural interface are generally not concerned with the details of the system below the architecture or with any still lower-level architectures. They are only concerned with the function of the system at the interface immediately below them.

In his classic book, *The Mythical Man-Month*, Frederick Brooks, project manager of the IBM System 360, summed up the notion of architecture as "the complete and detailed specification of the user interface."

The highest architecture is the interface between the whole computer system and the outside world; this can be called the *system architecture*. Somewhat lower down is the interface between the application program and the high-level programming language itself (assuming the application is written in a high-level language). This boundary can be said to define the *programming language architecture*. Since more and more programs are written in high-level languages, this architecture is the one most programmers will be concerned with. Still lower is the interface between the language and various run-time resource management functions that are usually provided by operating systems. We can call this boundary the *operating system architecture*.

The next interface is especially significant, because in conventional computers it defines the boundary between hardware and software. It is the level at which elementary, machine-recognized instructions are decoded and executed. We shall call this the *conventional instruction set architecture*. The two lowest-level architectures (*microcode architecture* and *gate-level architecture*) define even more primitive functions and are not of real concern to most programmers.

Out of this multiplicity of architectures, the one usually called *the* computer architecture is defined by the boundary between hardware and software. To be precise, the computer architecture is the level of the computer system that is seen by an assembly language programmer or compiler writer.

For each computer, the computer architecture can be seen as a kind of horizontal "slice" through the diagram in figure 1-1. Everything below the slice is performed by the hardware in this computer; everything above the slice is performed by the software. The major task of the computer architect is to decide which functions are best performed below the boundary and which above the boundary. Naturally, this boundary is not the same for all computers. Over the years, as computer technology has developed, the boundary between hardware and software has changed. When we compare the architecture of the iAPX 432 with other computer architectures, we will be discussing where this boundary between hardware and software has been drawn.

Vertical slices through the system diagram in figure 1-1 may be considered as well. A vertical slice separates components into a multiprocessor or distributed processing system. For example, a cut that separates out I/O functions defines an I/O processor (sometimes called a front-end processor). Other vertical cuts might separate several general purpose processors and so define a multiprocessor system. These vertical cuts may be said to define a *configuration architecture*. (See figure 1-2.)

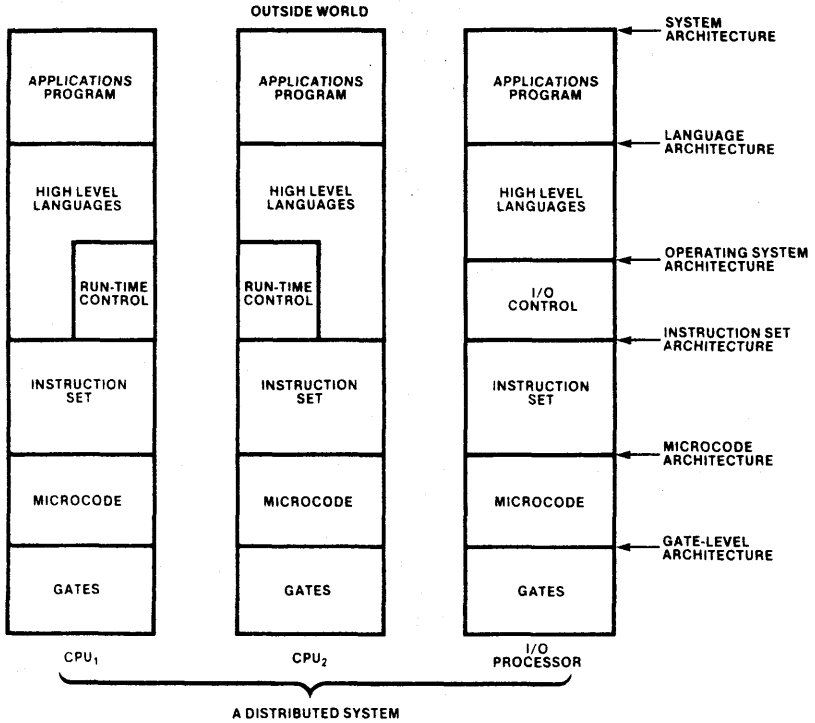


Figure 1-2. Configuration Architecture 171821-02

1.2 The Hardware-Software Interface

Thirty years ago computers were very expensive to build; the hardware filled whole rooms and a single gate might cost several hundred dollars. Because of these costs, early computers could have only very simple hardware operations, such as "Add the contents of register X to register Y." Consequently, it made good economic sense to substitute a program for a piece of equipment whenever possible. If more elaborate operations were needed, they were implemented with a library of subroutines. Thus, the architecture of these early machines represented a fairly low-level cut through the diagram in figure 1-1.

As technology has advanced, hardware costs have declined rapidly. Meanwhile, for a variety of reasons which we shall examine in Chapter 4, the overall software costs of computer systems have increased astronomically. Today, as much as 80% of the total system cost can be due to software. Thus, the architectural motivations of the 1950s and 1960s—trying to minimize hardware costs at the expense of software—make little sense today.

But a recent study concluded:

> If one compares the architecture of most current, widely-used machines (e.g., IBM System/370 and System 32, DEC PDP-10 and PDP-11, CDC 6600, Univac 1108, and Intel 8080) to . . . the first electronic stored-program computers (built in the 1940s), all the significant differences will be found to have originated in the 1950s . . . . Although current systems differ significantly in terms of cost, speed, reliability, internal organization, and circuit technology, the computer architecture . . . has not advanced beyond the concepts of the 1950s.*

This lack of advance is why we can use a single line in figure 1-1 to represent "conventional instruction set architecture" (i.e., conventional computer architecture).

The question is this: If hardware costs are not a factor, which software functions should be done in the hardware, where they can be done faster and more reliably? Some functions can be added to the hardware with little or no controversy. Instead of libraries of floating point arithmetic subroutines, manufacturers now build a floating point unit into the computer itself. But before more complicated functions can be added to the hardware, decisions must be made about high-level languages, operating systems, and even programming methodologies, since raising the hardware-software interface means that the structures and operations in the architecture will be more closely related to the structures and operations in the programming language and operating system.

Consider the following historical example. In the late fifties, dissatisfaction with Fortran led computer scientists to try to formulate a better computer language. This research culminated in the design of Algol 60. At about that time, the preliminary specifications for the Burroughs B5000 computer were being drawn up. The architects of the B5000 decided to design an architecture specifically to support Algol instead of a conventional architecture. They committed themselves to high-level language programming instead of assembly language programming, and they chose the best language available to them.

The B5000 architecture had a number of advanced features, including segmented memory to support the static block structure of Algol and a stack mechanism to support its dynamic behavior. In fact, the B5000 and its successors were among the few important commercial machines developed during the 1960s and 1970s whose architectures represented any advance over the concepts of the fifties.

The B5000 had the right approach; it attempted to raise the level of the architecture using the best available programming methodology (c. 1960), which largely reduced to "use Algol", and the architecture supported Algol very effectively. But in the 1970s and 1980s problems have arisen for which Algol and the programming methodology of the early 1960s offer no solution.

These problems have led other manufacturers, whose earlier computers had more conventional architectures, to recognize the wisdom of raising the level of the hardware-software interface. Consider, for example, the IBM System 38, IBM's most recent architecture. Not only have the designers of the System 38 followed the Burroughs approach in architectural support for high-level languages, they have also included most of the operating system in the hardware as well. It seems inevitable that the fundamental problems facing the computer industry will force more and more manufacturers to take this approach.

---

*Glenford J. Myers, *Advances in Computer Architecture* (New York: 1978)

1.3 Current Problem Areas

The basic problem facing the computer industry today is the software crisis; software cost as a fraction of total system cost has dramatically increased over the last two decades. It is an odd kind of crisis, because it is in a sense a crisis caused by good fortune: decreasing hardware costs have made possible more and more ambitious projects. Unfortunately, our ambitions greatly exceed our current programming abilities. There are four fundamental problems: the size of modern programs, the demands for security, the use of concurrency, and the need for expandable systems. These four topics are discussed in detail in Chapter 4. The software crisis has thrown a spotlight on the deficiencies of current architectures. We can now see that conventional architectures were designed to solve hardware problems that are no longer relevant. The current problems also identify three areas where the hardware-software interface can be profitably raised: memory organization, data manipulation, and support for the programming environment. (The topics that follow are discussed in much greater detail in Chapters 2, 3 and 4, where all the key terms are defined.)

Memory Organization: The logical organization of memory and the memory protection mechanisms should reflect the organization of programs, not the arbitrary limitations of the current hardware technology. Large virtual memory systems are needed to free users from concern over the size limitations of physical memory. Efficient use of memory will also require hardware support for dynamic storage allocation mechanisms, so that programs take only the memory they need, and memory that is no longer in use can be quickly reallocated.

Data Manipulation: The instruction set should support high-level language statements easily and yet produce compact code. The instructions should have a complete set of operators for a full range of data types, from bits to long floating-point numbers (typically at least 64 bits).

Support for the Programming Environment: A new programming methodology has been developed over the last decade which can make programming cheaper and more reliable. This new "object-oriented" methodology is based on the concepts of abstract data types and protection domains, concepts which are difficult to implement at an acceptable performance level unless they are supported by the architecture. In addition, many operating system mechanisms should be implemented in the architecture, where they can be handled more rapidly and securely.

1.4 iAPX 432 Architecture

The iAPX 432 represents one of the most significant advances in computer architecture since the 1950s.

Figure 1-3 shows a diagram similar to the one in figure 1-1, but with lines drawn to represent the level of the hardware-software interface in conventional micro-, mini-, and mainframe computers, and in the iAPX 432.

1.4.1 Main Features

The iAPX 432 is the first computer whose architecture supports software-transparent, multi-processor operation. Processors can be added to or removed from a system to select the desired price/performance level, without requiring any software changes. If one processor fails, the rest of the system can usually continue to operate.

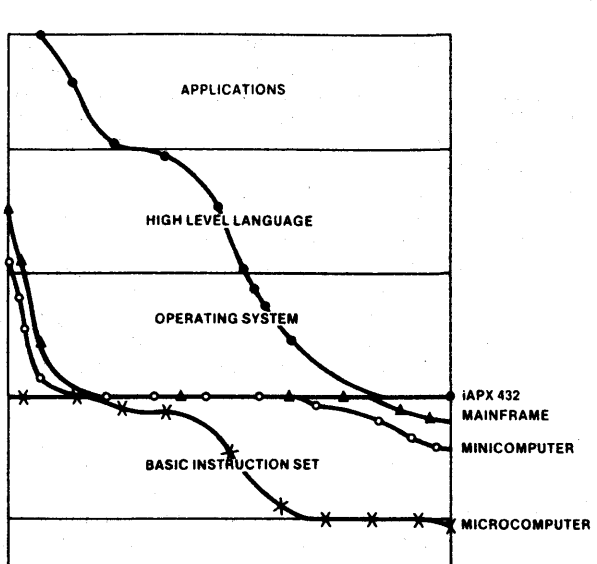


Figure 1-3. Raising the Hardware-Software Interface 171821-03

The iAPX 432 is the first commercial computer whose architecture fully supports the new object-oriented programming methodology. This new methodology can significantly reduce the enormous cost of software in contemporary systems. Both abstract data types and objects are supported by hardware recognized, hardware protected, and hardware manipulated structures. The iAPX 432 architecture also includes the Silicon Operating System, which provides the hardware mechanisms to support operations such as process scheduling, interprocess communication, and dynamic storage allocation that are implemented by the operating software in most computers. (See Chapter 4 for more details.)

The iAPX 432 is designed to be programmed entirely in high-level languages. To facilitate compiler writing, the instruction set is completely symmetric. (All addressing modes are available for every operand, and all required operators are available for every data type—see Chapter 3 for more details.) The iAPX 432 has also implemented the proposed IEEE floating-point arithmetic standard.

The iAPX 432 has an extremely large virtual address space (2^{40} bytes) and hardware-supplied mechanisms for implementing virtual memory systems that can exploit this address space. The memory architecture is segmented, with enough segments (over 16 million) for every meaningful program unit to have its own segment. Thus the memory protection mechanisms, which are based on segments, can give hardware protection to the logical structure of programs. (See Chapter 2 for more details.)

The iAPX 432 hardware can detect hundreds of different fault conditions, from attempting to divide by zero or attempting to execute data, to complex faults involving several processes. Most computers do not detect these faults at all, even at the operating system level, so it is common for the system to crash or for data to be destroyed.

On the iAPX 432, if a fault is detected, the operation is aborted, and a complete description of the fault is reported. In a multi-process system, a fault may cause the current process to suspend itself, but the other processes can continue to execute. In a multi-processor system, a fault may cause one processor to suspend itself and begin running diagnostics, but the other processors can usually keep the system operating. (Fault handling is described in Chapter 12 of the *iAPX 432 General Data Processor Architecture Reference Manual*.)

Table 1-1 summarizes the principal differences between conventional architectures and the architecture of the iAPX 432.

Table 1-1. A Comparison of the iAPX 432 Architecture and the Architecture of Conventional Mainframes

Feature of Architecture	Conventional Mainframe Architecture	iAPX 432 Architecture
MEMORY ORGANIZATION		
Organization, size	Linear $2^{24} - 2^{32}$ bytes	Structured segmented 2^{24} segments 2^{16} byte displacement 2^{40} byte virtual address space
Logical to physical address translation	Single-level map Page-based relocation and virtual memory	Two-level map Segment-based relocation and virtual memory
Protected memory unit	Fixed-size page	Individual program module or data structure
DATA MANIPULATION		
Expression evaluation	General register	Stack or memory-to-memory
Primitive data types	Characters, unsigned integers, integers, reals	Characters, unsigned integers, integers, reals temporary reals
Floating point hardware	Yes	Yes
Addressing modes	Some modes not available for all operands	Symmetrical: all modes available for every operand
PROGRAMMING ENVIRONMENT SUPPORT		
Operating system	No multi-process support	Multi-process mechanisms in hardware
	No support for dynamic storage allocation	Dynamic storage allocation mechanisms in hardware
	Very limited multi-processor operation, if any	Software-transparent, multi-processor operation
High level language	Assembly language-oriented instruction set	Oriented toward high level languages
Programming methodology	No support at all	Object-based architecture

1.4.2 Configurations

The iAPX 432 configuration architecture is defined by the components that make up an iAPX 432 system. In order to achieve high performance in both general purpose computation and input/output operation, the iAPX 432 has a distinct type of processing unit for each of these functions. The General Data Processor (GDP) handles all program decoding, computation, and address generation. The Interface Processor (IP) handles all communication with peripheral devices. Communication among the GDP, IP, and memory is provided by a packet-based interconnect bus. The IP is also connected to an interrupt-driven I/O subsystem bus, to which all peripherals are interfaced. A conventional processor, called the Attached Processor (AP), provides processing power in the I/O subsystem.

Figure 1-4 gives an overview of a small iAPX 432 system, showing GDP, IP, AP, memory, the interconnect bus, and the I/O subsystem bus.

The iAPX 432 is the first computer whose architecture allows for true software-transparent multiprocessor operation. This is perhaps the most revolutionary feature of the iAPX 432 architecture. The number of processors in a system is totally invisible to the software. Several processors (both GDPs and IPs) and memory controllers can be attached to a single interconnect bus. Figure 1-5 shows a system with two GDPs and two IPs.

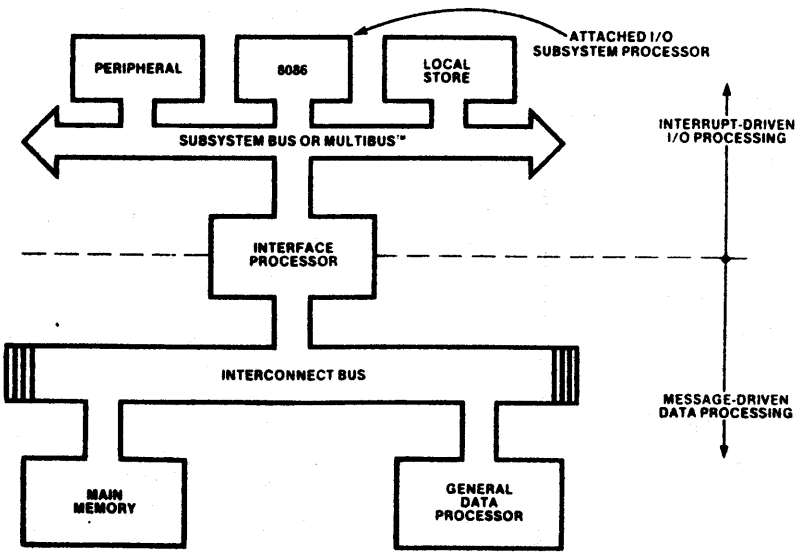


Figure 1-4. A Small iAPX 432 System 171821-04

On the iAPX 432, if a fault is detected, the operation is aborted, and a complete description of the fault is reported. In a multi-process system, a fault may cause the current process to suspend itself, but the other processes can continue to execute. In a multi-processor system, a fault may cause one processor to suspend itself and begin running diagnostics, but the other processors can usually keep the system operating. (Fault handling is described in Chapter 12 of the *iAPX 432 General Data Processor Architecture Reference Manual*.)

Table 1-1 summarizes the principal differences between conventional architectures and the architecture of the iAPX 432.

Table 1-1. A Comparison of the iAPX 432 Architecture and the Architecture of Conventional Mainframes

Feature of Architecture	Conventional Mainframe Architecture	iAPX 432 Architecture
MEMORY ORGANIZATION		
Organization, size	Linear $2^{24} - 2^{32}$ bytes	Structured segmented 2^{24} segments 2^{16} byte displacement 2^{40} byte virtual address space
Logical to physical address translation	Single-level map Page-based relocation and virtual memory	Two-level map Segment-based relocation and virtual memory
Protected memory unit	Fixed-size page	Individual program module or data structure
DATA MANIPULATION		
Expression evaluation	General register	Stack or memory-to-memory
Primitive data types	Characters, unsigned integers, integers, reals	Characters, unsigned integers, integers, reals temporary reals
Floating point hardware	Yes	Yes
Addressing modes	Some modes not available for all operands	Symmetrical: all modes available for every operand
PROGRAMMING ENVIRONMENT SUPPORT		
Operating system	No multi-process support	Multi-process mechanisms in hardware
	No support for dynamic storage allocation	Dynamic storage allocation mechanisms in hardware
	Very limited multi-processor operation, if any	Software-transparent, multi-processor operation
High level language	Assembly language-oriented instruction set	Oriented toward high level languages
Programming methodology	No support at all	Object-based architecture

1.4.2 Configurations

The iAPX 432 configuration architecture is defined by the components that make up an iAPX 432 system. In order to achieve high performance in both general purpose computation and input/output operation, the iAPX 432 has a distinct type of processing unit for each of these functions. The General Data Processor (GDP) handles all program decoding, computation, and address generation. The Interface Processor (IP) handles all communication with peripheral devices. Communication among the GDP, IP, and memory is provided by a packet-based interconnect bus. The IP is also connected to an interrupt-driven I/O subsystem bus, to which all peripherals are interfaced. A conventional processor, called the Attached Processor (AP), provides processing power in the I/O subsystem.

Figure 1-4 gives an overview of a small iAPX 432 system, showing GDP, IP, AP, memory, the interconnect bus, and the I/O subsystem bus.

The iAPX 432 is the first computer whose architecture allows for true software-transparent multiprocessor operation. This is perhaps the most revolutionary feature of the iAPX 432 architecture. The number of processors in a system is totally invisible to the software. Several processors (both GDPs and IPs) and memory controllers can be attached to a single interconnect bus. Figure 1-5 shows a system with two GDPs and two IPs.

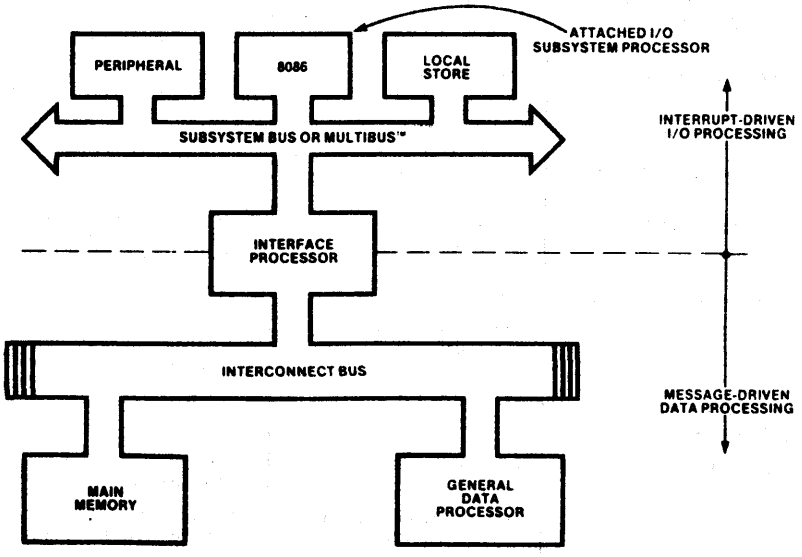


Figure 1-4. A Small iAPX 432 System 171821-04

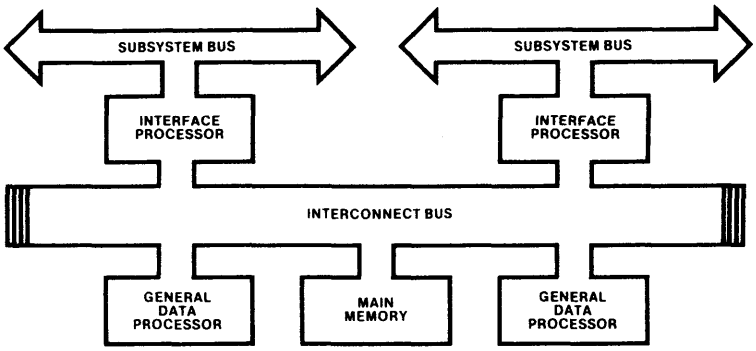


Figure 1-5. A Multiple-Processor System 171821-05

The Intel iAPX 432 computer system family covers a wide range in processing power, from the iSBC 432/100 single-board computer, which contains a single GDP (with peripheral interfacing provided by discrete components), to systems with several GDPs, IPs, and APs.

1.5 Topics Covered in this Document

Although system configuration is an important topic, this document will deal solely with the architecture of the General Data Processor, the principal component in an iAPX 432 system. System configuration, the architecture of the IP, and the related topics of communication with the outside world, initialization, program downloading, and interface to an interrupt-driven external processor will be covered in separate documents.

The next three chapters will examine three broad areas where the level of the hardware-software interface has been raised in the iAPX 432 architecture. Chapter 2 describes the memory architecture of the iAPX 432 and compares it to the memory organization of conventional computers. Chapter 3 describes the data manipulation instruction set of the iAPX 432 and shows how it supports high-level languages. Chapter 4 explains what objects are and how an object-oriented architecture can be used to support the new software methodology and the Silicon Operating System.

CHAPTER 2
MEMORY ORGANIZATION

2.1 Fundamentals

One of the most important characteristics of a computer architecture is the way memory is organized and the way information in memory is accessed.

The main memory of a computer is organized as a set of storage locations numbered consecutively, beginning with zero. The number associated with one of these physical storage locations is called a *physical address*, and the set of all physical addresses is called *physical address space*.

A *logical address*, on the other hand, is an address in an instruction—an address as used by a programmer. The *logical address space*, the set of all logical addresses, is thus the set of addresses that can be used by a program. The organization of the logical address space defines the *memory architecture*.

The organization of physical address space is determined by memory technology and by cost, but the logical address space should not necessarily be constrained by either of these considerations. Instead, the organization of logical memory should be determined by the structure of the programs that will run in memory. And in fact, the history of advanced memory architectures shows an evolution toward this goal. Initially, the logical address space was identical to the physical address space; now, in the iAPX 432, the level of the memory architecture has been raised much closer to the level of user program organization. In this chapter we will explore some of this history and show how it has led to the iAPX 432 memory architecture.

2.2 Linear Memory

The most common organization of logical address space; that is, the most common architecture for memory, is a linear, contiguous address space. In this kind of architecture addresses start at location zero and proceed in linear fashion (i.e., no holes or breaks) to the upper limit imposed by the total number of bits in a logical address. A program, often consisting of several procedures, and its data are all located within this single address space. The logical address space of a linear memory thus has the same basic organization as the physical memory.

The simplest kind of linear address space is exemplified by the 8080. 8080 programs can generate 2^{16} (65,536) distinct addresses. (See figure 2-1.) The addresses generated by 8080 programs are used directly by the memory hardware to locate data. Thus, a Load Accumulator instruction that references address 50000 will cause data to be read from physical memory location 50000.

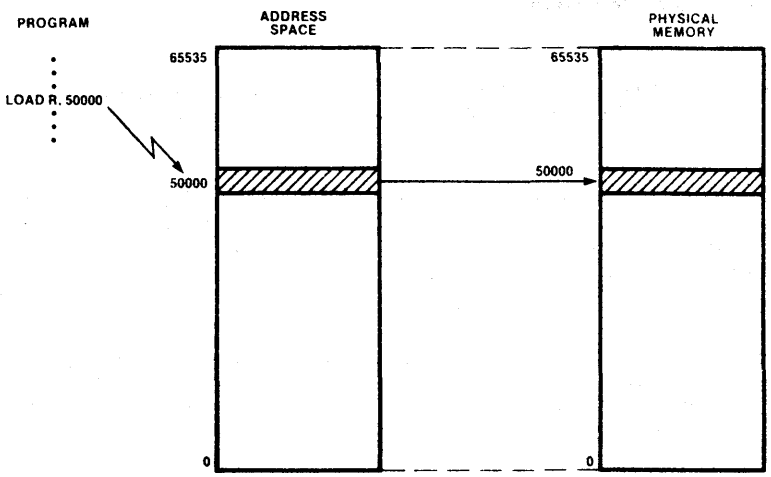


Figure 2-1. Unmapped Linear Memory 171821-06

The limitations of this memory architecture are readily apparent when several programs share the same machine in order to use efficiently the system's resources. It is difficult to implement such a multiple-program system on systems with a memory organization as simple as the 8080's.

First, nothing prevents one program from accessing or even destroying data in another program. Since all programs share the same logical address space, any program can access any location in memory, and no program is protected from unauthorized access by another program. Second, all the programs have to fit inside the relatively constricted logical address space of the 8080.

To illustrate the problem of uncontrolled access of this simple memory organization, examine figure 2-2. A program *can* reference any location in memory, even if it isn't supposed to. In this simple system nothing prevents program A from overwriting program B or other parts of its own code or data, for that matter. A simple programming error by A can wipe out B.

Thus, even though complex software partitioning schemes were employed, multiple-program systems never operated reliably or securely with this memory organization. To overcome these liabilities without abandoning linear architecture, new mechanisms for memory management were invented: logical-to-physical address translation (also called *mapping*), memory access control based on fixed-sized pages, and virtual memory. These mechanisms are explained in the next four subsections.

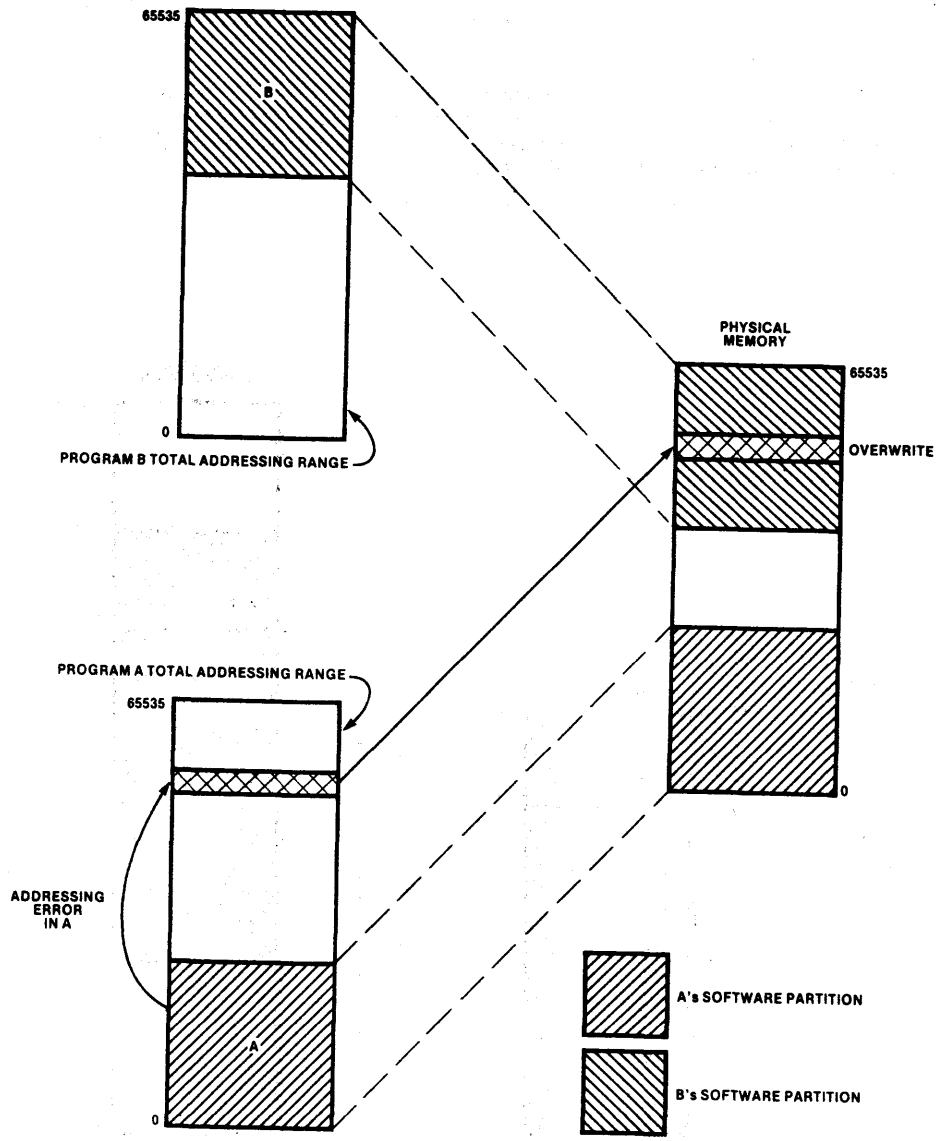


Figure 2-2. Vulnerability of Unmapped Memory

171 821-07

2.2.1 Mapping Linear Memory

Mapping is basically the process of translating logical addresses into physical addresses. In the 8080 system we considered above, logical addresses are simply equated to physical addresses; but by exploiting mapping a logical address can be assigned to an arbitrary physical address. Thus mapping is a mechanism for relocating the logical address space within the physical address space.

Figure 2-3 shows a very simple mapping operation. The entire logical address space of a program (locations 0-65535 in this case) is mapped onto physical locations 30000-95535. Thus, a program reference to location 50000 actually fetches data from physical location 80000 (50000 + 30000).

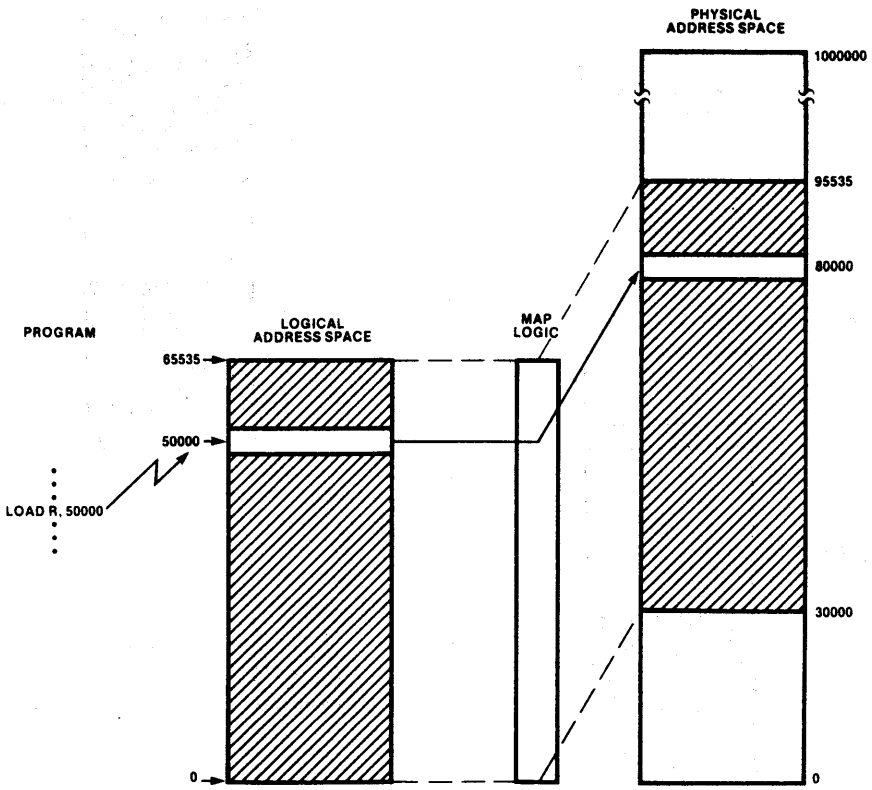


Figure 2-3. A Simple Mapping Scheme 171821-08

The utility of this operation may not be obvious, but it is extremely useful in multiprogram systems. (In fact, mapping was developed for such systems.) With this kind of mechanism, which is typical of most minicomputers and some advanced microcomputers, each program has its own logical address space which is completely independent of any other program. Thus many programs can share physical memory without any possibility of interfering with each other. The mapping unit fits all the logical address spaces into one physical memory, but the process is transparent to the programs. See figure 2-4 for a multiprogram mapped environment.

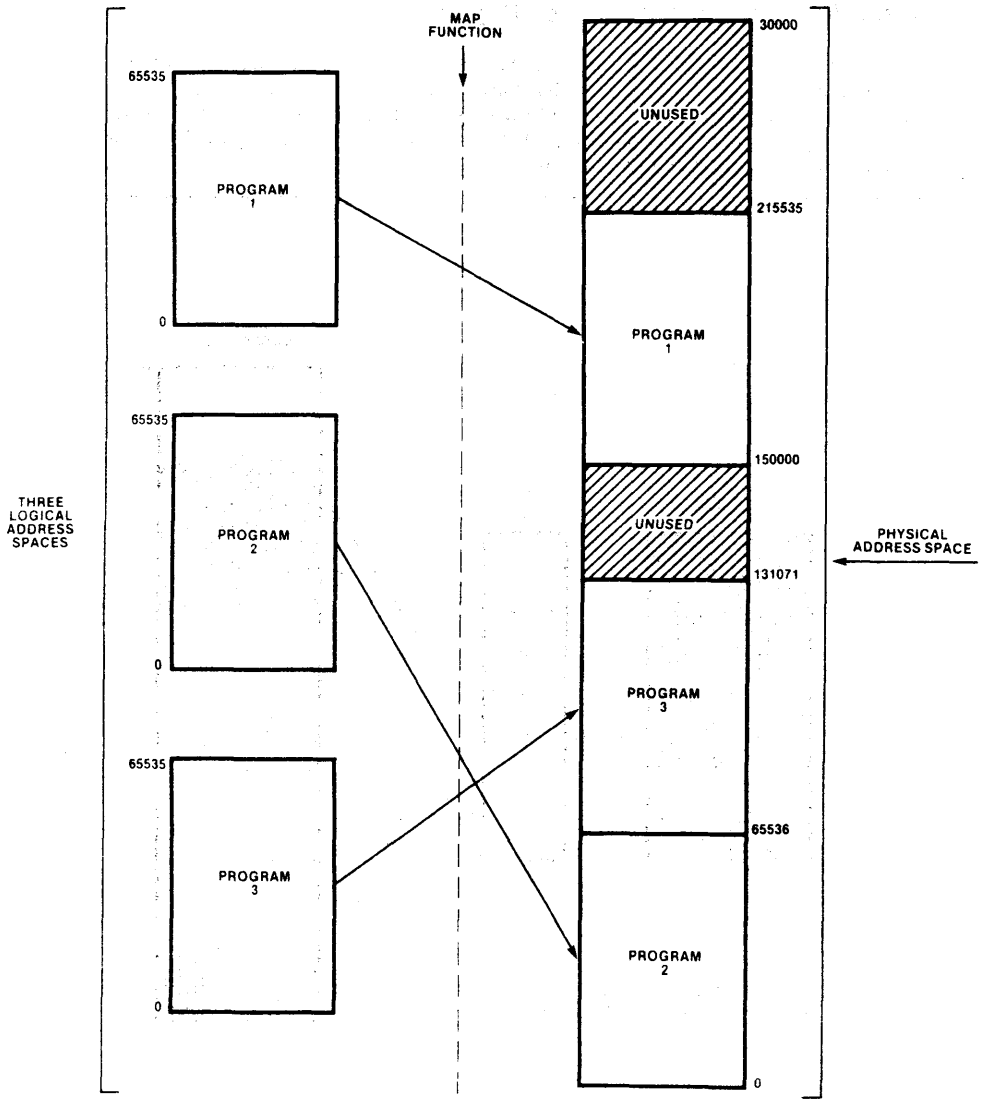


Figure 2-4. A Multiprogram Mapped Environment

171821-09

2.2.2 Page-Based Mapping

Instead of mapping the entire logical address space as a unit, as in figure 2-4, more advanced address translation mechanisms map smaller, fixed-size "pages" of logical address space onto pages of physical memory. Thus, a large program need not be relocated into one contiguous chunk of physical memory, which might be hard to find in a multiprogramming environment, but rather into several smaller sections of memory, which are more likely to be available. (It is often much easier to find twenty 1K pages than one 20K block.)

2.2.3 Access Rights Based on Pages

The page mechanism can also provide the basis for memory protection within a logical address space. Each page can have attributes associated with it (called *access rights*) that indicate how the page can be accessed. These attributes can allow reads only, reads and writes, or they can prevent any access at all. See figure 2-5 for a diagram of such a paged mapping mechanism.

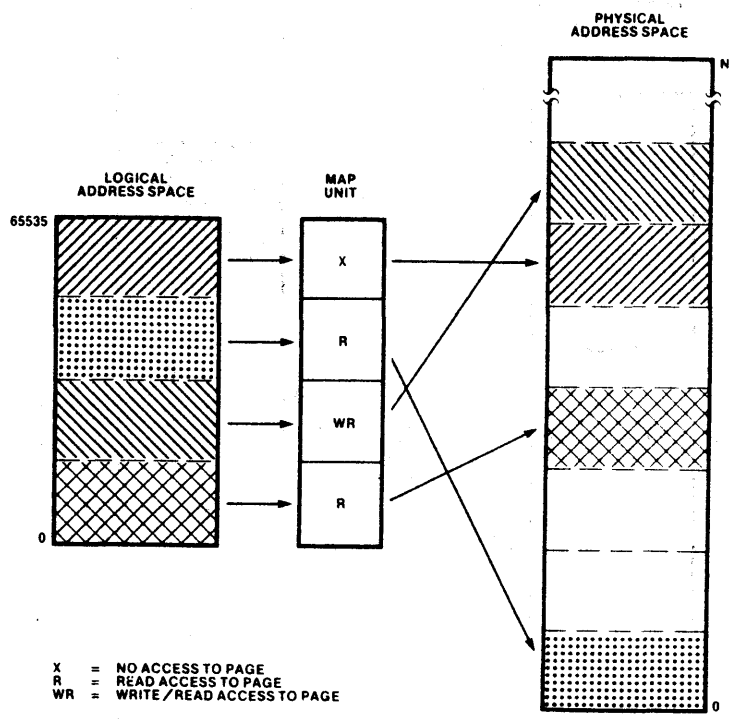


Figure 2-5. Page-Based Protection

171821-10

2.2.4 Virtual Memory

In many computer systems, the logical address space is far larger than the physical memory. *Virtual memory* is a mechanism for getting around the limits on physical memory size. Under a virtual memory system, it appears to users as if the entire logical address space were available for storage. But, in fact, at any given moment only a few pages of the logical address space are mapped onto physical space. The other pages are not present in main memory at all; instead the information in these pages is stored on a secondary-storage device, such as a disk, whose cost-per-bit is more economical.

Every time a missing page is accessed, operating system software loads the missing page from disk and stores on disk a page that has not been referenced recently. The user will have the illusion of a gigantic, although slower, physical memory.

2.3 Segmented Memory

While many systems employ linear memory architecture, with memory management mechanisms of varying complexity, a number of new processors (the Intel 8086 and the Zilog Z8000) and some older systems (the Burroughs B5000 and the Multics processor) have abandoned linear memory architecture altogether. Instead they use a form of logical memory organization called *segmented memory*.

The basic motivation for segmented memory is that programs are not written as one linear sequence of instructions and data, but rather as parcels of code and parcels of data. For example, there could be a main code section and many separate procedures. Data could be organized into arrays, or arrays of arrays, linked lists, or any number of complex data structures. Moreover, these modules of code and data could come in many different sizes.

Segmented memory architectures were developed to support this logical structure. The logical address space is broken down into many linear address spaces, each with a specified size or length. Each of these linear address spaces is called a *segment*. Each item within a segment is accessed by a two component address; the first component (the *segment selector*) specifies the segment itself, while the second component (the *displacement*) specifies the offset from the base of the segment to the item being selected.

Each segment can be used to hold a meaningful program module or data module. Thus, a program might have the main procedure in a segment, each additional procedure in its own segment, and perhaps each major data structure in its own segment. With a segmented architecture, the structure of logical address space reflects the logical organization of the program. (See figure 2-6.)

In contrast, a linear address space is by definition logically structureless. As discussed in the previous section, protection mechanisms for linear memory are usually based on fixed-length pages, whose size is determined by hardware criteria and which have no necessary relationship to the logical structure of programs.

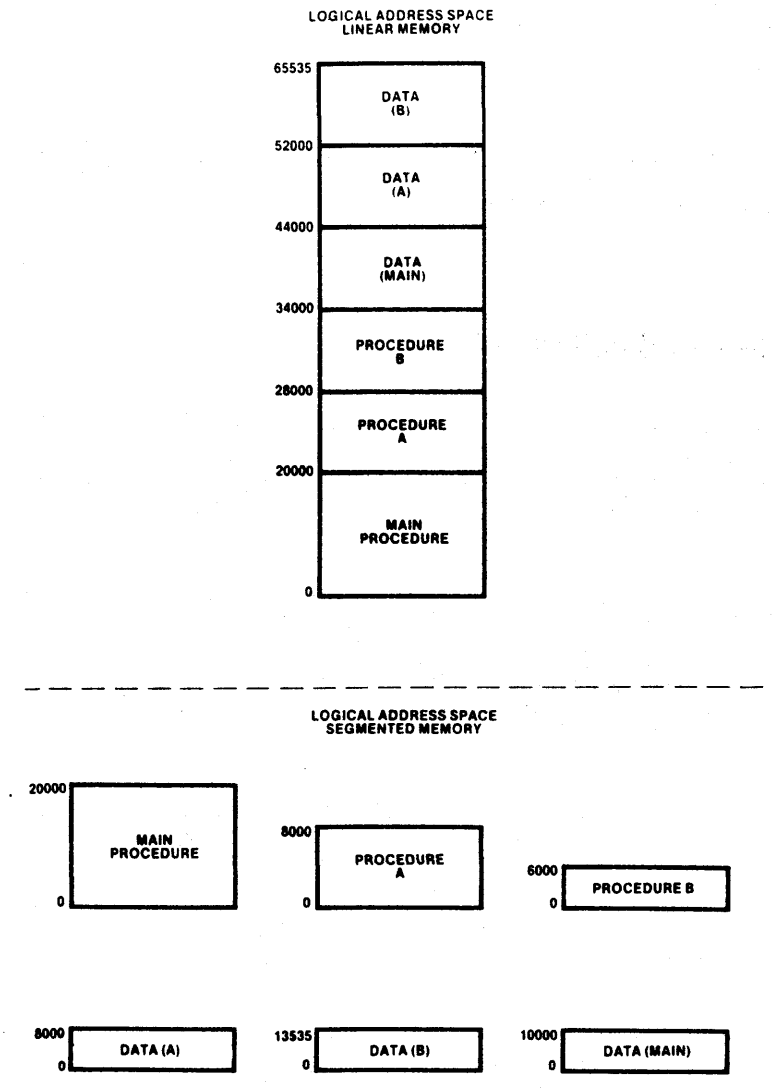


Figure 2-6. Comparison of Segmented and Linear Memory 171821-11

The problem with decomposing the logical address space into pages is that the protection mechanism cannot protect the program modules precisely; it either protects too little or too much. The basic flaw is that the page length is fixed for hardware reasons; thus it is not flexible enough for the logical structure of programs.

By contrast, any protection mechanism provided for segments naturally accrues to meaningful pieces of a program. Since each segment has a specified length, it is easy to give even a small instruction sequence its own segment and protect it from other programs, provided there are enough segments available. (Since programs can consist of hundreds or even thousands of modules, this last criterion makes clear the importance of having a large number of segments in the architecture, if segmentation is to be used properly.)

Virtual memory mechanisms can also be implemented for segmented architectures. In this case the segment is the memory unit that is swapped to and from the secondary storage device.

2.3.1 Mapping Segmented Memory

Segmented architecture is also compatible with logical to physical address translation. In this case, the segment is the unit that is mapped onto physical memory, instead of the whole logical address space or some arbitrary fixed-size unit.

Mapping in this case is implemented by a *segment table* that holds a *segment descriptor* for each segment. A segment descriptor contains the starting physical address and the length of a segment. The segment selector component of a logical address is used as an index to select a segment descriptor in the segment table. Then the displacement is added to the segment starting address to produce the physical address of the operand being referenced. (See figure 2-7.) The displacement is easily checked by the hardware to ensure that the reference does not exceed the length of the segment.

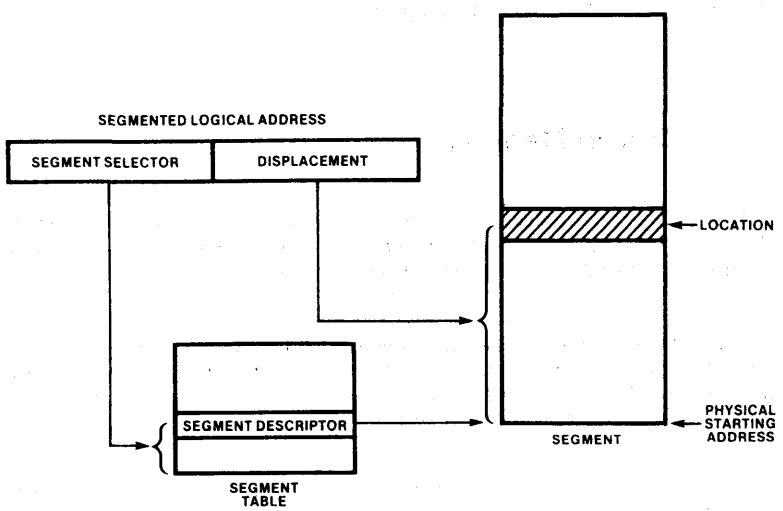


Figure 2-7. Segmented Mapped Memory 171821-12

2.3.2 Segment Types and Access Rights

Many computers with segmented architectures (such as the Z8000) include *segment type* attributes in the segment descriptors. For example, the Z8000 Memory Management Unit supports stack segments, which are defined by a particular bit in the segment descriptor. Whenever an access is made to a stack segment, the hardware checks to make sure that a stack push or pop operation is occurring. If the wrong kind of access is made to a stack segment (an instruction fetch, for example), the hardware signals a fault.

In addition, the segment descriptors often contain the access rights attributes for each segment. Notice that the access rights are therefore associated with particular segments, instead of to the program modules that reference the segments. If a segment is read-only, for example, it is read-only for *all* modules that reference it. This association of access rights with segments is a disadvantage, because we may wish to give different modules access to the same segment, but with different access rights.

2.3.3 Access Control

Another disadvantage with the segment mapping mechanism we have described is that it is difficult to limit the access by one program to another program's segments. Since the segment table contains *all* the segment descriptors, any program can access any segment simply by indexing through the segment table. However, the usual cure for this problem, multiple segment tables, may be worse than the disease itself. Under this scheme each program is given its own segment table, with each table referencing only the segments that the program needs. But this scheme implies that whenever a segment is relocated in physical memory, all the programs that share the segment will have to update their segment descriptors. This could require hundreds of entries throughout memory to be changed. There are great advantages in keeping all the segment descriptors in one central location.

2.4 Structured Memory

The iAPX 432 has a segmented memory. However, the differences between the iAPX 432 and other segmented machines are great enough to justify the use of a new term--structured memory--for the iAPX 432 memory architecture.

First, the iAPX 432 can address a very large number of segments -- 2^{24} -- or approximately 16 million segments, far more than most other computers. Furthermore, each segment can be up to 2^{16} bytes long. The total virtual address space is therefore 2^{40} bytes, that is to say, more than one *trillion* bytes!

Second, the 432 uses a two-step mapping process that separates segment relocation from access control. This two-stage mapping also allows a division of protection features into segment-specific protection (segment type checking) and user-specific protection (access rights). The two-step process is unique to the iAPX 432 and is responsible for some of its most powerful features.

2.4.1 Two-Level Mapping and Access Control

The iAPX 432 mapping is based on the simple, one-step segment mapping process described in section 2.3.1 and illustrated in figure 2-7, but another step has been added.

Each independently translated program module is supplied at run-time with a collection of segment numbers (i.e., values that can be used as indices into the segment table to select segment descriptors) for all the segments it may need to access during execution (and no others). This collection describes the *access environment* of the module, and the entire collection is stored in a set of *access segments*. Access segments form the other step in the two-step mapping process.

The access mapping step works like this: The segment selector part of a logical address is used as an index into an access segment to select one of the segment numbers. The segment number itself then acts as an index into the segment table to select a segment descriptor. This two-level process is illustrated in figure 2-8. The segment numbers are actually contained in thirty-two bit entities called *access descriptors*, which also contain access rights data.

The access environment is defined by a set of four access segments that are associated with this particular invocation of the program module (see section 4.4.2, *Context Objects*). Since the access environment restricts the logical address space available to the module, we can think of the access environment as defining the *effective address space*. Figure 2-9 illustrates the relationship among the logical address space, the effective address space, and the physical address space.

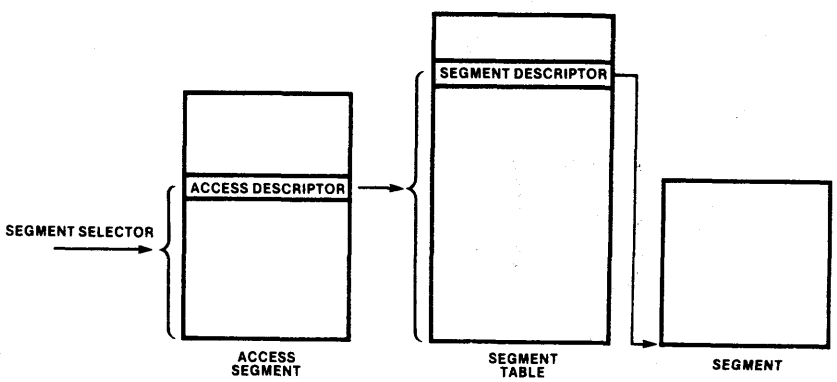


Figure 2-8. Two-Level Mapping 171821-13

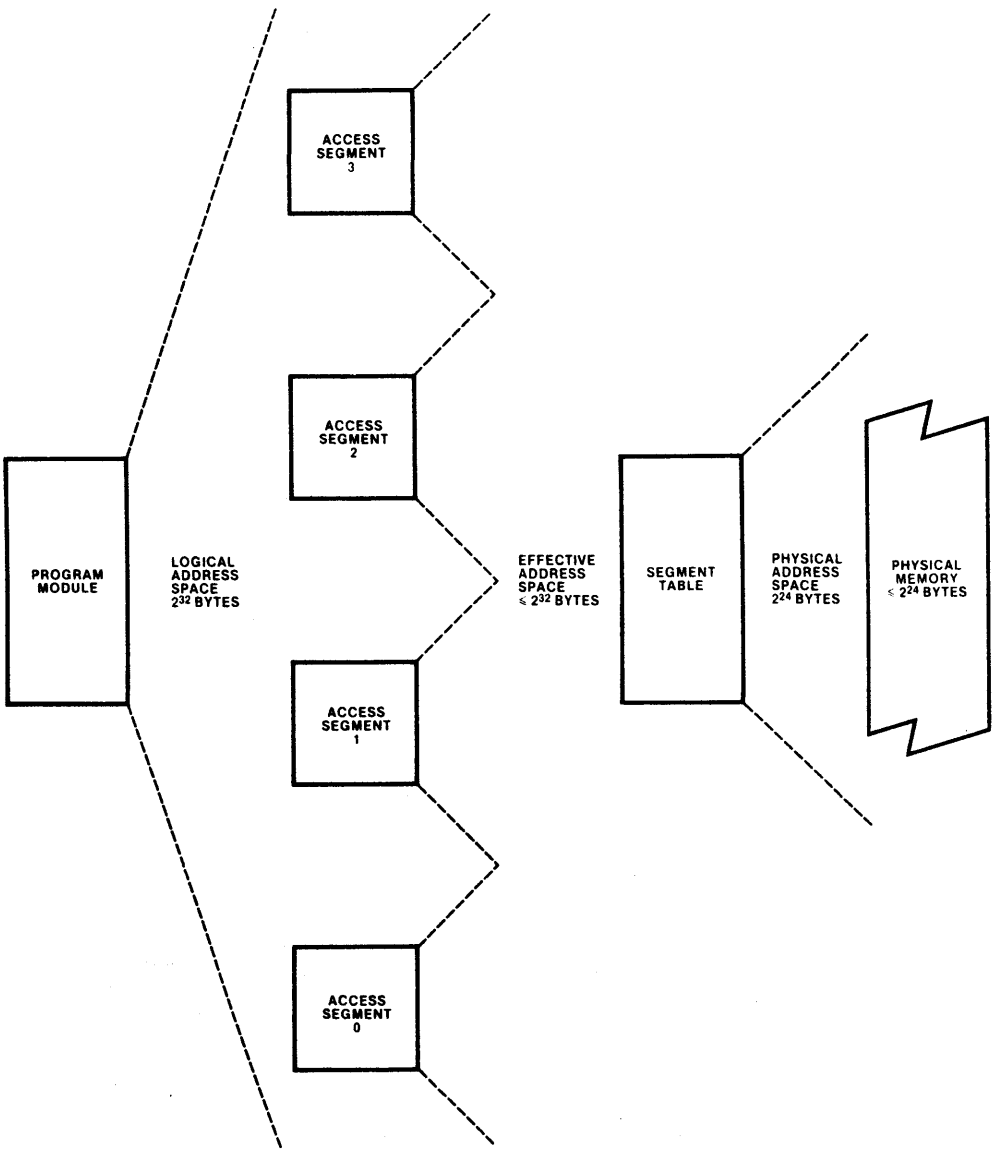


Figure 2-9. iAPX 432 Address Spaces 171821-14

The two-stage mapping process has three major advantages over a one-step map:

First, it takes fewer bits in an address to specify a particular segment. Sixteen million segments mean 16 million (2^{24}) entries in the segment table, so an index into the segment table (the segment number) must be 24 bits wide. This is far too many bits to carry around in every logical address. But only those segments in the access environment can be referenced. And, since the number of segments in the access environment will generally be much smaller than the total number of segments in memory, fewer bits are needed to specify one of them. In fact, the iAPX 432 can use as few as four bits to specify a segment.

Second, the two-step mapping makes it possible to restrict the number of segments accessible by a given program or program module. Whereas, as we indicated in the previous section, the one-step mapping allows any program to address any segment in memory, simply by indexing through the segment table.

Third, as we shall see in the next section, it allows the separation of access rights information from the segment descriptors. Access rights are instead contained in the access segments and are thus associated with program modules.

One potential problem with a two-step mapping is access time. A memory access requires both an access descriptor and a segment descriptor, each of which may have to be fetched from memory. The iAPX 432 avoids this problem by maintaining an internal associative cache to speed up the address translation process. The most recently used segment descriptors, access descriptors, and the addresses of a number of commonly accessed items (e.g., the segment table) are all stored on the chip. The result is that most accesses go directly to the addressed item.

2.4.2 Segment Types and Access Rights

In the iAPX 432, as in all computers with segmented memory, all information is contained in segments. It is apparent that if the processor could tell what *type* of information each segment contained, it could provide a powerful protection mechanism. For example, if it were known that a particular segment contained access descriptors, and a program inadvertently tried to execute the contents of the segment as code, the error could be detected, and corrective action taken before any damage was done.

Several different segment types are recognized by the iAPX 432 hardware. The two fundamental types (called the two *base types*) are data segments and access segments. Data segments are used to contain both instructions and operands (i.e., everything but access descriptors). Access segments contain access descriptors only. Type information is contained in the segment descriptors and is checked whenever an attempt is made to access a segment. If the access is not allowed for that type, a fault occurs.

The hardware maintains an absolute distinction between access segments and data segments. The contents of access segments may not be manipulated in any way by instructions which operate on data segments. Access segments have their own set of instructions which manipulate access descriptors.

Each base type is subdivided into several hardware-recognized *system types*. For example, data segments can be subcategorized as instruction segments or stack segments, to name only two possibilities. The hardware checks for inappropriate accesses to system types as well as to base types. Thus, an attempt to execute the contents of a stack segment can be detected and aborted.

The access segment is the appropriate place to put access rights information, not the segment table (compare 2.3.2), since these rights should be associated with a program module, not directly with a segment. Since access rights are stored independently of the segment descriptors, several modules can share the same segment, each with a different access right to it. See figure 2-10 for an illustration of this case.

An access rights field is part of the access descriptor. Every reference to a segment from an instruction will be checked against an access right for that segment, contained in the access descriptor. If a reference is detected that is not allowed, a fault will occur.

2.4.3 Virtual Memory and Dynamic Storage Allocation

Each program module has its own access environment, and within that environment instructions can potentially generate 2^{32} unique addresses. Since a program can dynamically modify its current access environment (for example, by calling different procedures, see section 4.4.5), the address space available to the program over the lifetime of its execution is equal to the maximum total number of segments in memory times the maximum length of each segment. This enormous figure -- 2^{40} bytes -- is called the *virtual address space*.

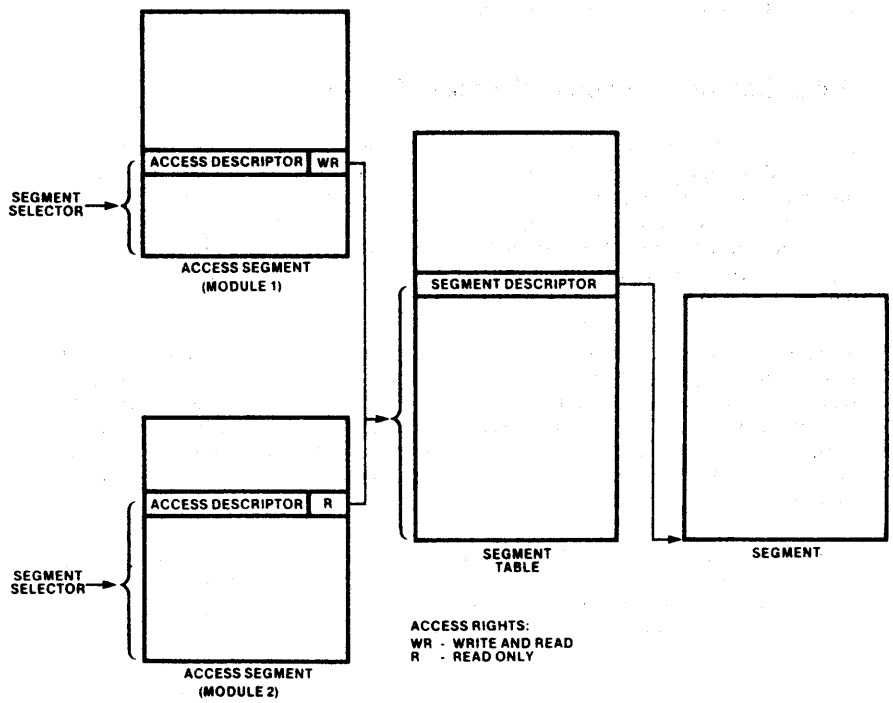


Figure 2-10. Shared Segments

171821-15

As described in section 2.2.3, virtual memory is a way of implementing a large logical address space with a much smaller physical space. Since no iAPX 432 user will attach 1 trillion bytes of main memory to his system, a virtual memory mechanism must be implemented if the gigantic virtual address space is to be used. Each segment descriptor contains the following four 1-bit fields, which are maintained by the hardware and can be used by the operating system to implement virtual memory:

- The *valid* field, which indicates whether or not the segment is currently present in main memory.
- The *storage allocated* field, which indicates whether any memory has been associated with this descriptor.
- The *accessed* field, which indicates whether this segment has been accessed by some executing process.
- The *altered* field, which indicates whether or not the information contained in the segment has been modified by some executing process.

The operating system can use the *valid* bit and the *storage allocated* bit to detect when a physical segment is not present in memory, and it can use the *accessed* and *altered* bits to decide which of the currently present segments should be swapped out or simply overwritten by the new segment. In addition, several fields in the segment descriptor can be used by the operating system to record other useful information about the segment (e.g., frequency of use) that can also be used in the swapping algorithm.

The amount of memory allocated to a module is not necessarily fixed at compile time; it may be changed dynamically. The mechanism for implementing dynamic storage allocation is described in section 4-6. The operating system can use this dynamic storage allocation mechanism to make virtual memory more efficient by only allocating segments as they are needed, thus preventing unreferenced segments from using up memory. The user can also make use of the dynamic storage allocation mechanism to reserve storage as needed.

2.4.4 Complexes of Segments

The segment table contains entries for access segments as well as data segments, so the ultimate target address specified by an access descriptor can be another access segment. A logical address in an instruction therefore can reference an access descriptor as well as an item of data. This access descriptor will in turn point to another segment, which, of course, can also be an access segment. The journey from the initial access segment, through all the intervening access segments, to the final data segment is called an *access path*.

By following these access paths, we can move through elaborate complexes of segments, in which access segments point to other access segments, and so on. To conveniently diagram these segment complexes, the two-step mapping process will be symbolized by a single arrow, which will be called an *object reference* (see figure 2-11). Figure 2-11 also shows a large segment complex.

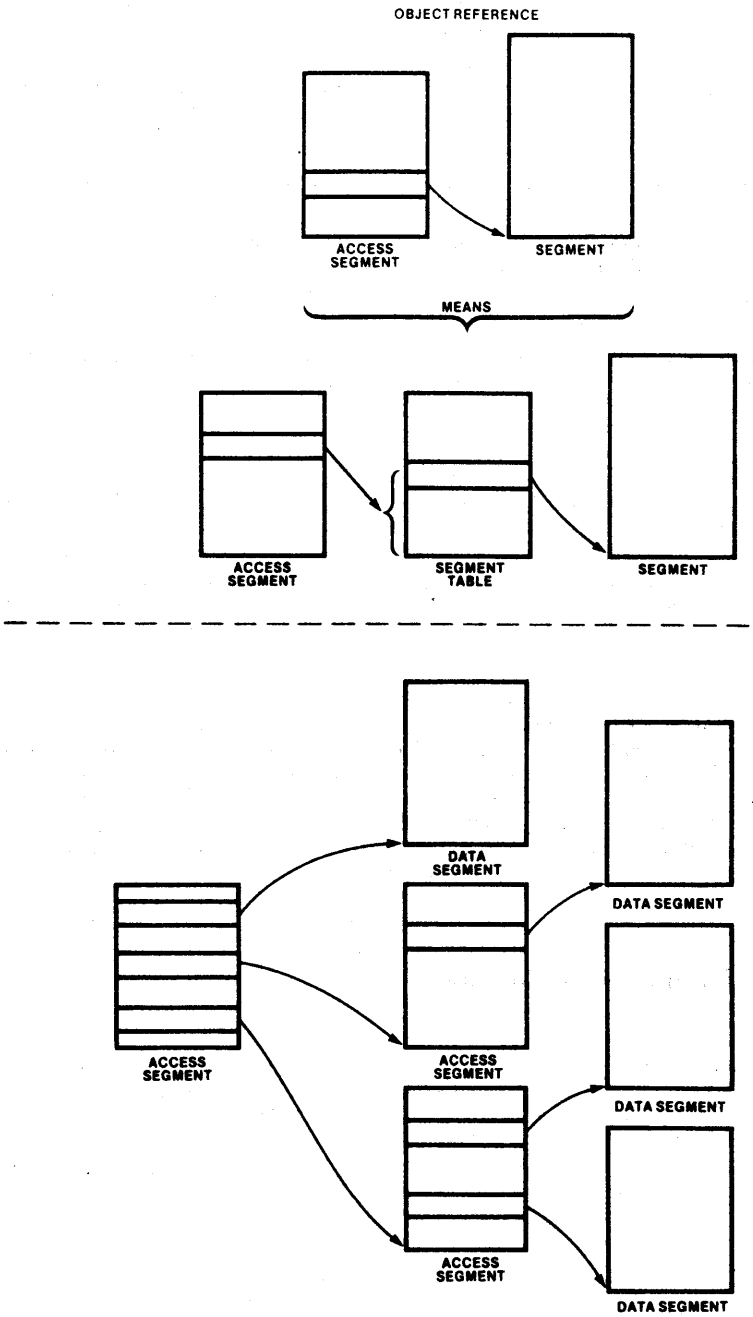


Figure 2-11. A Complex of Segments 171821-18

A complex of segments can be conceptually viewed as a single entity called an *object*. The iAPX 432 hardware recognizes a number of these objects and can manipulate them with instructions. Objects can consist of a single segment, a complex of segments, or a contiguous subsection of a segment (called a *refinement* of the segment). Objects are used to support the advanced features of the iAPX 432 architecture that are described in Chapter 4.

Viewing an object as a single entity, we can consider the segment descriptor that points to the root segment of the object as a descriptor for the whole object. We will often refer to such segment descriptors as *object descriptors* and refer to the segment table as the *object table*. The object table actually consists of several data segments. See Chapter 2 of the *iAPX 432 General Data Processor Architecture Reference Manual* for more details.

2.5 Summary

The iAPX 432 memory architecture has the following features:

- Two-step mapping process which separates the relocation mechanism from the access control mechanism.
- A unique access environment for each invocation of a program module.
- 2^{40}-byte virtual address space with mechanisms for implementing virtual memory.
- Two basic types of segments: access segments and data segments; they are recognized by the hardware and the distinction between them is rigorously enforced.
- Access rights associated with program modules not with segments.
- Mechanisms for constructing and accessing complicated data structures consisting of several segments.

CHAPTER 3
DATA MANIPULATION

3.1 What are Data Types?

All information in a computer's memory is stored as a pattern of ones and zeros. What these bits represent depends upon the interpretation given to them by the computer. The same bit pattern that encodes an ADD instruction might represent the number 17562 when interpreted as an integer. All computers have some basic data representations that are recognized by the hardware. These basic representations are called hardware-recognized *data types*.

Hardware-recognized data types have two additional characteristics. First, each instance of a type (e.g. each integer) can be addressed *as a unit* in memory. Even if the datum is several bytes long, it has a single address. Second, associated with each type is a set of operations that can be performed on the type. The instruction set of a computer is basically just the combination of all the hardware recognized operators with all the data types that can act as operands.

Early computers (and first generation microprocessors) had very simple instruction sets, with only a few operators and a few hardware-recognized data types. The arithmetic instructions were usually limited to addition and subtraction of integers. All other data types (e.g. real numbers) were manipulated by using special subroutines. Subroutines also had to be written to perform more complex operations (e.g. multiplication and division).

For instance, to multiply two real numbers on a computer lacking hardware multiplication and real data types, a number of steps had to be taken. A binary pattern for real numbers had to be devised that could be stored in the computer's memory and could adequately represent the range and accuracy desired. Second, a software subroutine had to be written that could perform the multiplication and store the result in memory. The software routine had to accomplish the multiplication using only simple operations, such as additions and subtractions of integers.

As the cost of hardware decreased, it became feasible to expand the instruction set to include operations and data types that had previously been handled by the software. Typically, real numbers, characters, and bits were added to the data types supported by the instruction set. The net result was that overall performance was increased, especially for data-type-intensive applications (e.g. scientific number crunching) and less memory was required, since whole software routines could be replaced by single instructions.

Ideally, any new hardware-supported data type should have been supported by a complete set of operators. For example, if the Add operation existed, Subtract should also have existed, and usually Multiply and Divide. Unfortunately, this ideal was not always met. Operators were often added in an *ad hoc* manner, and all the relevant operators were not supplied for each data type. The results were instruction sets that were very difficult to use.

3.2 iAPX 432 Operators and Primitive Data Types

The iAPX 432 provides a comprehensive set of operators for manipulating several different hardware recognized data types. We will call them "primitive" data types because they are used to construct more complex data structures. All required operators are available for eight primitive data types, which may be divided into four classes: character, ordinal, integer, and real. Figure 3-1 shows the formats of each of the eight types.

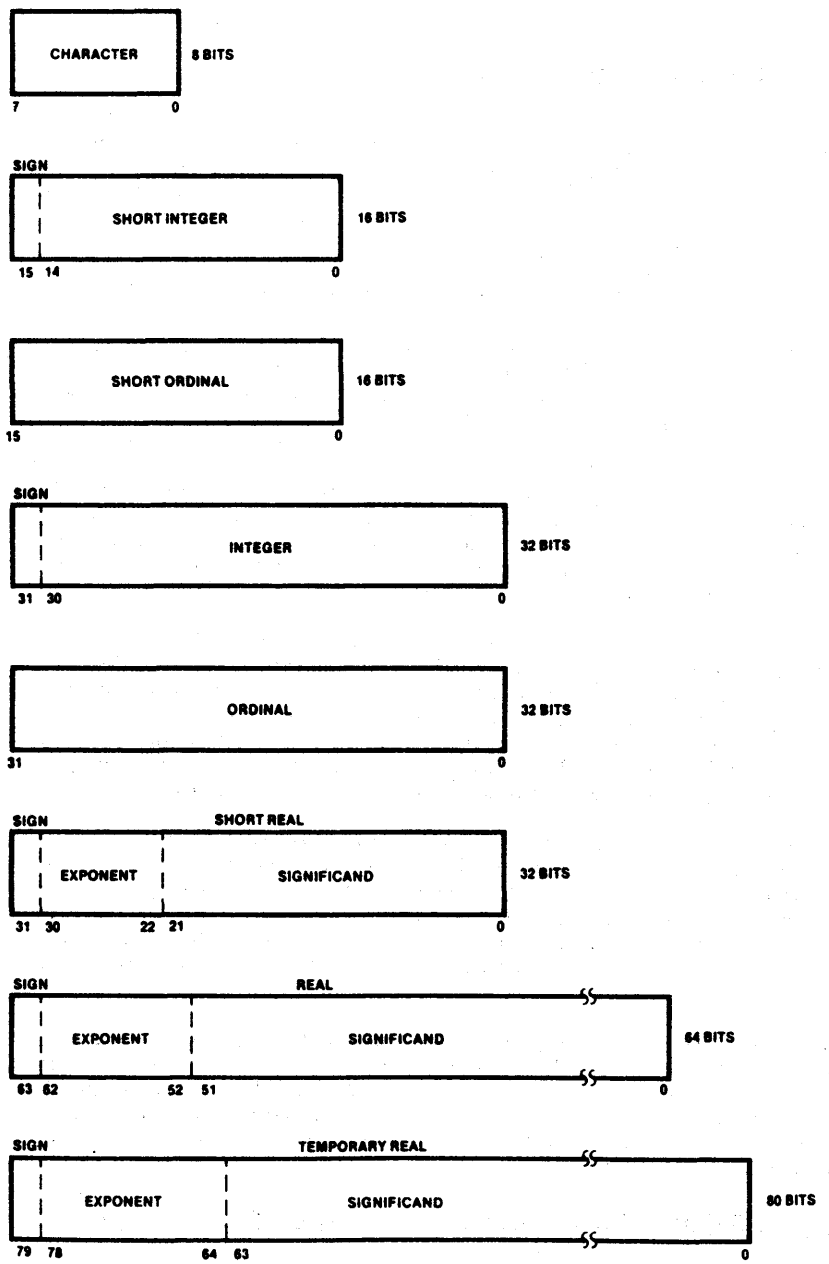


Figure 3-1. Primitive Data Types 171821-17

The operators for these data types can be divided into several broad groups: arithmetic operators (e.g. Add, Subtract, Multiply, Divide), logical operators (e.g. AND, OR, NOT, XOR), relational operators (e.g. Equals, Greater Than, Less Than), conversion operators (e.g. Convert to Character, Convert to Integer), move operators (e.g. Move, Zero, Save), and bit-field manipulation operators (e.g. Extract Bit Field, Insert Bit Field). Table 3-1 provides a chart showing all hardware recognized data types and all the operators that can be used with them. The sections that follow describe briefly the four data type classes and the operators that go with each class.

Table 3-1. iAPX 432 Operators and Data Types

	OPERATOR	CHARACTER	SHORT ORDINAL	ORDINAL	SHORT INTEGER	INTEGER	SHORT REAL	REAL	TEMPORARY REAL
MOVE OPERATORS	MOVE	x	x	x	x	x	x	x	x
	SAVE	x	x	x	x	x	x	x	x
	ZERO	x	x	x	x	x	x	x	x
	ONE	x	x	x	x	x			
LOGICAL OPERATORS	AND	x	x	x	–	–	–	–	–
	OR	x	x	x	–	–	–	–	–
	XOR & XNOR	x	x	x	–	–	–	–	–
	COMPLEMENT	x	x	x	–	–	–	–	–
ARITHMETIC OPERATORS	ADD	x	x	x	x	x	*	*	x
	SUBTRACT	x	x	x	x	x	*	*	x
	MULTIPLY		x	x	x	x	*	*	x
	DIVIDE		x	x	x	x	*	*	x
	REMAINDER		x	x	x	x			x
	INCREMENT	x	x	x	x	x	–	–	–
	DECREMENT	x	x	x	x	x	–	–	–
	NEGATE	–	–	–	x	x	x	x	x
	ABSOLUTE VALUE	–	–	–			x	x	x
	SQUARE ROOT								x
BIT FIELD OPERATORS	EXTRACT		x	x	–	–	–	–	–
	INSERT		x	x	–	–	–	–	–
	SIGNIFICANT BIT		x	x	–	–	–	–	–
RELATIONAL OPERATORS	EQUAL	x	x	x	x	x	x	x	x
	NOT EQUAL	x	x	x	x	x			
	EQUAL ZERO	x	x	x	x	x	x	x	x
	NOT EQUAL ZERO	x	x	x	x	x			
	GREATER THAN	x	x	x	x	x	x	x	x
	GREATER THAN OR EQUAL	x	x	x	x	x	x	x	x
	POSITIVE	–	–	–	x	x	x	x	x
	NEGATIVE	–	–	–	x	x	x	x	x
CONVERSION OPERATORS	CONVERT TO CHARACTER	–	x						
	SHORT ORDINAL	x	–	x					
	ORDINAL		x	–		x			x
	SHORT INTEGER				–	x			
	INTEGER			x	x	–			x
	SHORT REAL						–		x
	REAL							–	x
	TEMPORARY REAL		x	x	x	x	x	x	–

Key

X = This operator is available for the given data type.

* = This operator is available for the given data type *and* for operations where one of the operands is a temporary real.

— = This operator is not available and would not be useful if it were.

(blank) = This operator is not available.

3.2.1 Characters

Characters require one byte of memory and can represent three kinds of data: ASCII characters, unsigned integers in the range 0-255, and the boolean values TRUE or FALSE (which are stored as xxxxxxx1 and xxxxxxx0, where x means either 1 or 0). Character operators include move operators, logical operators, simple arithmetic operators (add, subtract, increment, and decrement), relational operators, and the operator Convert Character to Short Ordinal.

3.2.2 Ordinals

Ordinals are unsigned integers; they are availiable in two sizes: 16-bit (Short Ordinals) and 32-bit (Ordinals). They have values in the ranges 0 to 2^{16}-1 or 0 to 2^{32}-1. Ordinals are commonly used as indices into vectors and arrays and to hold bit fields. Ordinal operators include moves, arithmetic operators (including multiplication and division), logical operators, operators for manipulating bit fields, relational operators, and several conversion operators.

3.2.3 Integers

Integers are available in two sizes: 16-bit (Short Integers) and 32-bit (Integers). They have values in the ranges -2^{15} to 2^{15}-1 or -2^{31} to 2^{31}-1. Integer operators include moves, arithmetic operators, relational operators, and several conversion operators.

3.2.4 Reals

Real numbers have three components: an exponent, a significand (mantissa), and a sign bit. Reals are available in three sizes: 32-bit (Short Reals), 64-bit (Reals), and 80-bit (Temporary Reals). Each increase in size provides more precision (larger significand) and greater range (larger exponent). Short Reals and Reals are typically used as input and final-result operands in floating-point calculations. Temporary reals are intended for use as intermediate results, thus preserving accuracy and reducing the risk of overflow or underflow in multi-step calculations. Real operators include moves, arithmetic operators, relational operators, and several conversion operators. Table 3-2 lists the range and precision of the three sizes of reals.

Table 3-2. Range and Precision of Short Real, Real, and Temporary Real

Type	Range	Precision
Short Reals	2^{-126}—2^{127}	7 decimal digits (approx)
Reals	2^{-1022}—2^{1023}	15 decimal digits (approx)
Temporary Reals	2^{-16383}—2^{16383}	19 decimal digits (approx)

3.3 Structured Data Types

The data types discussed in section 3.1 are called *primitives*. By contrast, we will use the term *structured data types* for ordered aggregates of primitives. The iAPX 432 facilitates access to two structured data types that are commonly used in high-level languages:

- **Arrays.** An array consists of a number of components, each of the same data type. Thus we speak of arrays of integers, arrays of characters, etc.
- **Records.** A record consists of a number of components (usually called fields), that can be of different data types. Thus, a record might consist of characters, integers, and real numbers. (For instance, an employee record.)

These structured types are not supported by a set of hardware operations, but the iAPX 432 does provide a mechanism that allows structured types to be manipulated easily. Each of the primitive types may be accessed through several *addressing modes*, and these facilitate the selection of individual elements from arrays and records (see section 3.4.1). The addressing mode used to reference an operand is determined by the way the logical address is formed in the instruction that references it.

3.4 Instructions

The iAPX 432 instructions specify the operator and the operands that it acts on. Up to three operands may be required for some operators. Instructions are contained in special hardware-recognized *instruction segments* in memory (see section 2.3). The processor views an instruction segment as a continuous string of bits called the instruction stream. Instructions may contain a variable number of bits and may begin at any bit within the stream. (In the current implementation, the processor reads an instruction segment in units of 32 bits.) The general instruction format of the iAPX 432 consists of four fields arranged in the format shown in figure 3-2.

The first two instruction fields encountered by the processor in the instruction stream are called the *class* field and the *format* field. These two fields specify how many operands are in the instruction and information on how they are to be accessed. The last two fields are the *reference* field and the *operator* field. The operator field specifies the operator (e.g. ADD, SUBTRACT, NEGATE). The reference field contains the logical addresses of up to three operands. As shown in figure 2-7 of section 2.3, each logical address has two parts, a segment selector and a displacement. The segment selector identifies the segment that contains the operand, while the displacement locates the operand within the segment.

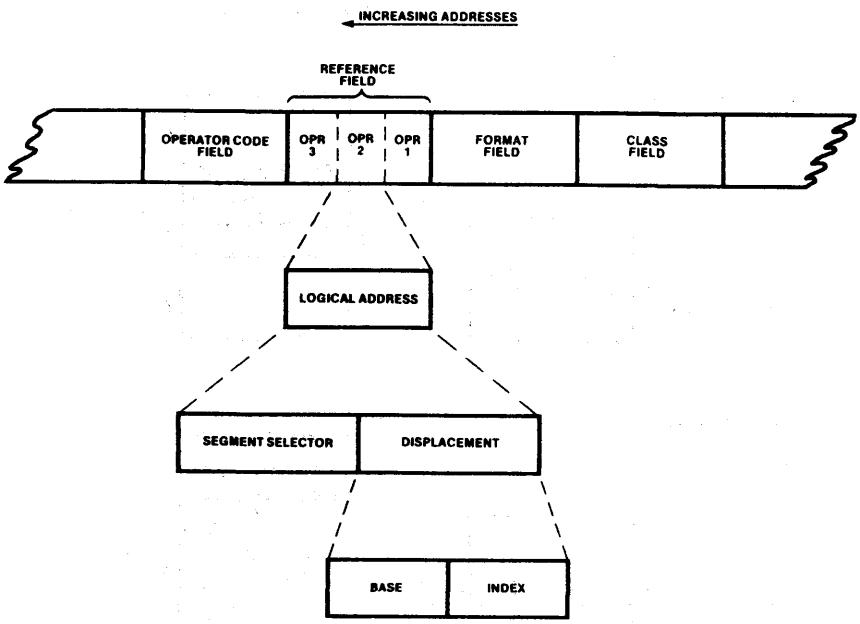


Figure 3-2. iAPX 432 Instruction Format (3 operands) 171821-18

3.4.1 Addressing Modes

The displacement really consists of two subcomponents: a base value and an index value. (See figure 3-2.) Each subcomponent can reference its value either *directly* (the subcomponent contains the value itself) or *indirectly* (the subcomponent contains a pointer to the value, which is located elsewhere in memory). A direct reference by both subcomponents is equivalent to a single-component displacement and can be used to access non-structured data (scalars). Indirect references can be combined with direct references to easily access three structured data types. The four combinations of direct and indirect reference are called *addressing modes*:

- Base and Index Direct — used to access *scalars*
- Base Indirect, Index Direct — used to access *records*
- Base Direct, Index Indirect — used to access *static arrays* (arrays whose starting location, the base, is established at compile time)
- Base and Index Indirect — used to access *dynamic arrays* (arrays whose base can be established during execution)

Figure 3-3 illustrates the way the logical address is formed in each of the four addressing modes and the structured data type that is accessed by each mode.

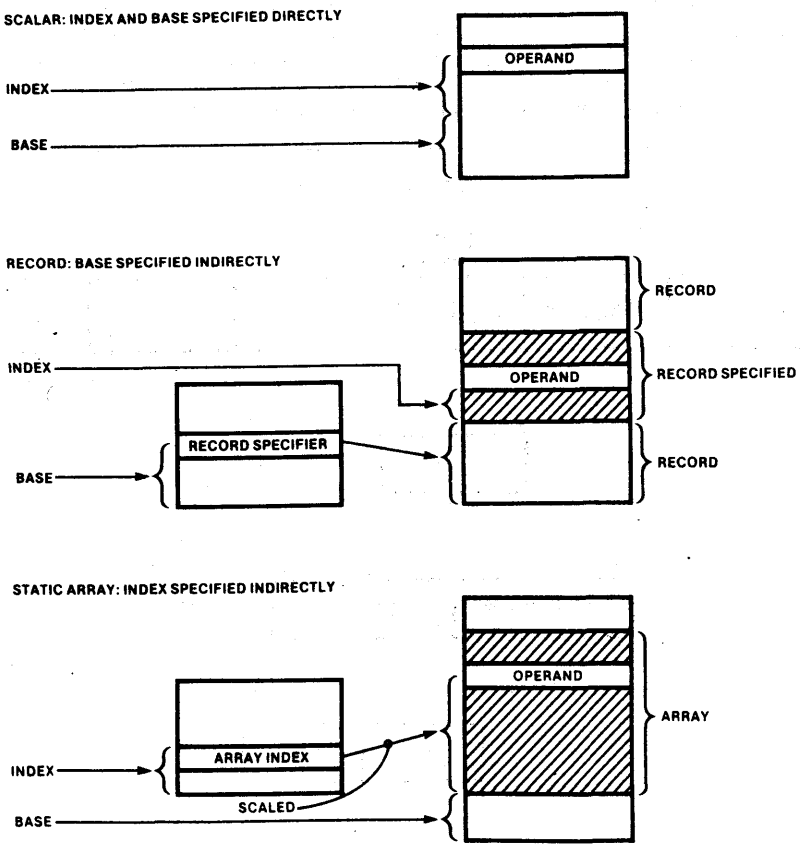


Figure 3-3. Addressing Modes and Structured Data Types (1 of 2) 171821-19

DYNAMIC ARRAY: BASE AND INDEX SPECIFIED INDIRECTLY

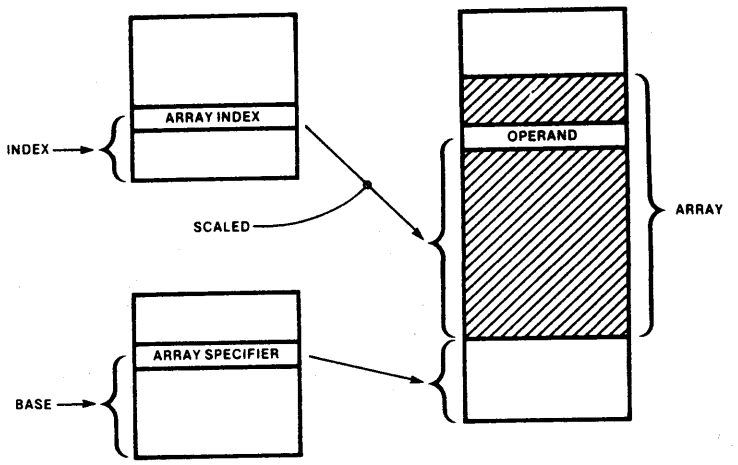


Figure 3-3. Addressing Modes and Structured Data Types (2 of 2) 171821-20

The processor automatically *scales* an index value by multiplying it by 1, 2, 4, 8, or 16 depending on whether the operand it points to occupies a byte, double byte, word, double word, or extended word. The compiler is thus freed from having to perform this calculation.

The segment selector component of the logical address can also be specified indirectly. This additional capability means that large, multi-segment arrays can be implemented easily.

It is important to stress that not only are all required operators available for every data type, but also all four addressing modes are available for any operand in an instruction. This means that iAPX 432 instruction set is completely *symmetric*. Symmetry (sometimes called regularity) is defined essentially as the degree to which all addressing modes exist for every operand and all required operators exist for every data type in the instruction set.

Symmetry is especially important for easy translation of high-level languages. If the instruction set is not symmetric (some operators are not provided for some data types, or some addressing modes are not available with some operators) then the compiler writer is faced with many special cases where software has to make up for the holes in the instruction set. Thus, the compiler is more difficult to write and more complicated.

The symmetry of the iAPX 432 instruction set and the powerful operators allow the efficient encoding of high level language statements. For example, each of the following statements is encoded by a single iAPX 432 instruction:

A = B * C	— A three-address multiply, all scalars
A [I] = B [J] / C [K]	— A three-address divide, all static array elements
P.Q = A [I] * C	— Three addresses, mixed type, "element Q of record P is assigned the product of static array A, element I, and scalar C"

3.4.2 Operand Stack

When an instruction references an operand, it can explicitly specify a logical address, in the manner described in the previous section, or it can access the top of the operand stack. The operand stack is a special data segment that is maintained by the hardware for expression evaluation. An access to an operand on the stack is called an implicit reference. Items are added to or removed from the "top" of the stack on a last-in-first-out basis. The current stack top is pointed to by a hardware-maintained stack pointer. The following arithmetic expressions, which can be performed by single iAPX 432 instructions, illustrate the flexibility of stack usage (the symbol "$" signifies that the operand is on the stack):

A = $ + B	— Add the value on the top of the stack to the value in B, then put the result in A
$ = A * $	— Multiply the value in A by the value on the top of the stack, then push the result on the stack
$ = $ + $	— Add the top two items on the stack and leave the result at the top of the stack

See figure 3-4 for an illustration of a case where the last example would occur.

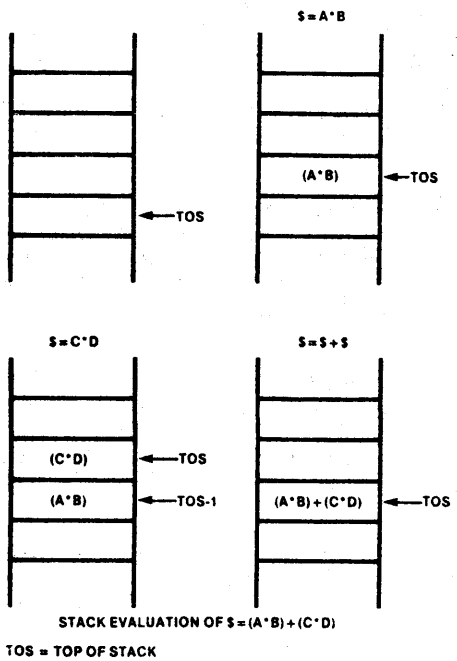


Figure 3-4. Using the Operand Stack 171821-21

Using implicit stack references instead of explicit memory references to address operands allows shorter and faster instructions. The iAPX 432, however, allows both stack arithmetic and memory-to-memory arithmetic. Research has shown that pure stack-oriented machines (i.e. machines where all arithmetic operations are between the top two elements of the stack) are not as fast at evaluating expressions, nor do they have as dense a code, as machines which allow both types of arithmetic.* But even a pure stack machine is faster and more efficient at evaluating expressions than a machine with no stack.

3.4.3 Instruction Encoding

From studies of program behavior, it is known that certain instructions are used more often than others. For instance, a push is far more frequently executed than a halt. Therefore, it makes sense to distribute the binary encoding of instructions in such a way that frequently used instructions take fewer bits to specify than infrequently occurring instructions. Most machines, however, have an instruction size that is always some multiple of the basic unit of memory storage (frequently 8 or 16 bits). This fixed-length instruction encoding tends to be inefficient and frequently forces unfavorable compromises in the design of the instruction set. Since a large percentage of a computer's time is used in fetching instructions, an improvement in instruction bit density would result in faster execution of programs and a decrease in program storage requirements.

The iAPX 432 instruction set has been designed to be very compact; the instructions are bit-variable in length and are not constrained to start or end on byte or word boundaries. If an operand reference is used more than once in an instruction, it need be specified only once. For example, in the common expression

$$A = A + B$$

the A operand need be specified only once. The format field in the instruction indicates how many times a particular operand is used.

The bit encodings were based on the frequency-of-use of operators and operand references. The shortest instruction is 6 bits long and the longest is 344 bits long.

3.5 Summary

The iAPX 432 data manipulation instruction set

- supports all required operators for 8 primitive data types
- has four addressing modes for every operand
- allows explicit or implicit access to every operand
- supports efficient coding of high-level language statements
- allows easy access to elements in arrays and records
- has a frequency-of-use bit encoding that provides compact code

*Glenford J. Myers, *Advances in Computer Architecture*, (New York: 1978).

CHAPTER 4
PROGRAMMING ENVIRONMENT SUPPORT

4.1 The Software Crisis

In his 1972 Turing Award lecture, E. W. Dijkstra, father of the "structured programming" movement, described the software crisis in the following terms:

> As long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem; and now that we have gigantic computers, programming has become an equally gigantic problem . . . . The increased power of the hardware . . . made solutions feasible that the programmer had not dared to dream about a few years before. And now, a few years later, he *had* to dream about them and, even worse, he had to transform such dreams into reality!

Decreasing hardware costs and increasing power led to more ambitious programming projects. But it proved difficult to make the programs reliable, and even more difficult to make the programming projects come in on schedule and within budget. There were four fundamental problems: the sheer complexity of many large systems, the increasing demand for system security, the increasing use of concurrent programming, and finally the demand for systems whose processing power can be expanded without affecting the existing software.

4.1.1 Modularity

Modern computer applications often require hundreds of thousands of lines of code; some data processing installations use systems that represent several million lines. Such enormous software packages are far too complicted for one person to understand completely; if they are to be developed and maintained, some form of program modularization must be used. It should be possible for different modules to be developed by different people, relatively independently, and for bugs to be fixed in one module without affecting other modules in unpredictable ways. It would also be desirable to be able to add functions by simply incorporating additional modules, without having to change the existing program.

Even though modularization seems like an easy idea to grasp, it has proved surprisingly difficult to implement correctly. A program can be modularized in a number of different ways, and some of these ways make the programming job even more difficult than if no modules were used. Consider, as an analogy, a diamond cutter. His job is to "modularize" an uncut diamond. If he chooses the correct modularization, he will create valuable gems. But if he makes a mistake, he will get a pile of dust.

The basic problem in modularization is to define correctly the interface between modules. Under many modularization schemes, some modules need to know the internal implementation details of other modules in order to operate. What is needed is a set of criteria for decomposing programs into modules in a way that minimizes the need for outside knowledge of the internal representations.

4.1.2 Security

Modern society has entrusted an enormous amount of sensitive data to computer systems, from bank accounts, credit records, and medical histories to the specifications for hydrogen bombs. Most of this information is, unfortunately, not very

securely stored. No computer system has ever successfully resisted determined efforts to defeat its security mechanisms, and many commercially available systems barely put up a fight.

Security is based on limiting access to information. The basic principle is "need to know"; users, programs, and modules within programs should have access to only that information which they absolutely need. Most modern operating systems, however, have a very crude access limitation scheme. Usually a distinction is made between a few privilege levels, and programs in the same level can do the same things. This scheme almost always results in programs being given too many privileges, since they cannot operate if they are given too few. What is needed is a method that grants each module exactly those privileges that it needs to execute properly, and no more.

4.1.3 Concurrency

As computers take on more complex tasks, they are called upon to model more and more features of the real world. One fundamental fact about the real world is that actions generally happen concurrently rather than in strict sequence. For example, in industrial control applications several jobs are usually being performed at the same time, and critical developments in one job can interrupt the work in another. Consequently, software designers have developed models of concurrent processing in order to handle these real-world situations. Concurrent processing is also often used to multiplex several programs on one processor, so that shared resources, such as memory, I/O, and the processor itself, can be more efficiently used.

Implementing concurrency requires mechanisms that permit the sharing of common resources and also mechanisms for bringing together concurrently executing modules to exchange information (communication) or to coordinate their action (synchronization). Several problems have made concurrent systems difficult to implement. The basic problem is defining the unit of concurrency. The problem is somewhat similar to the modularization problem described in section 4.1.1, except here the problem is how to modularize in the time domain. What is needed are criteria for decomposing programs into modules that can be run concurrently, and also mechanisms for communication, synchronization, and resource sharing.

4.1.4 Expandability

Project managers have always found it difficult to predict accurately just how much memory and processing power their project will require. As the project evolves, requirements tend to change (usually upward). Furthermore, over the lifetime of the product new features and functions are often added, placing more demands on power and memory. The standard approach computer manufacturers have taken to address this well-known phenomenon is to develop a family of computer systems that have a range of processing power.

Unfortunately, such a family takes years to develop, and the procesor with exactly the right performance is seldom available when it is needed. The user has to buy either too much or too little. Moreover, the operating system software is often different on different members of the family. What users need is a system that can be quickly and easily expanded in power without requiring any software changes.

The low cost of microcomputers has always presented the seductive possibility that computers might someday be constructed out of a large number of identical microprocessor components. If a system needs more power, just add another microprocessor. But the software difficulties of such multiprocessor systems have been daunting. The basic problems have been in sharing common resources and in

defining a unit of work in such a way that it could be sent to an arbitrary processor. This unit of work needs to be totally independent of particular processors and the communication between units of work has to be handled in an entirely processor-independent manner.

4.2 New Software Methodologies

In recent years, progress has been made in all four problem areas: modularity, security, concurrency, and expandability. By the late 1970s computer scientists involved in programming language and operating systems research believed that, in fact, solutions had been found to many of these problems.

4.2.1 Type Managers

A consensus seems to have emerged that the best way to modularize programs is by applying the principle of "information hiding." The term is due to D. L. Parnas, who has identified a number of criteria for correctly decomposing programs into modules. Another name for the principle is "data encapsulation." The basic idea is that a module should contain a collection of related procedures and the data structures they operate on. Procedures outside the module should not have access to the implementation details inside the module. The data can be referenced from outside the module only by a call to one of the procedures inside the module. Many names have been given to the modules that result when this principle is applied—abstract data types, extended types, Parnas modules—but in this document the term *type manager* will be used exclusively.

Until recently, the full power of the concept could not be realized, since most programming languages did not provide any direct support for the type manager construction. Users had to build their own type manager structures on top of the language, thus no compile-time checking facilities or enforcement mechanisms existed for type managers. Moreover, since no standard format for type managers existed, there was not necessarily any compatibility among different users' representations of the type manager concept, even if they used the same language.

Several modern programming languages, however, support the type manager construction (or closely related concepts): In Intel's Object Programming Language (OPL) and in Concurrent Pascal the structure is called a "class," CLU uses the term "cluster," Alphard calls its version a "form," and the new Department of Defense standard language, Ada, uses the term "package."

At the same time that programming language research was focusing on the concept of a type manager as the solution to the modularization problem, operating system research was defining a very similar structure, the *protection domain*, as the solution to the security problem. (We will treat protection domain and type-manager as synonyms.) Operating system researchers concentrated on the *data* in these domains, instead of the procedures (which were the focus of the language research), since they were concerned with security more than modularization. They gave the name *objects* to the data items associated with domains, and they called the whole information hiding methodology "object-oriented design methodology" or "the object model."

4.2.2 Processes

The fundamental concept that underlies concurrency is the *process*. The process is the module of concurrency, just as the type manager is the module of the static organization of a program. A program can be constructed out of a single process or

several processes that communicate with each other. The use of processes originated in multi-programming operating systems, where each job was a process. But in recent years multiprocessing has evolved into a general methodology for implementing concurrency, a methodology we will call the *process model*. In fact, just as Ada has the notion of a "package," corresponding to a type manager, it also has the concept of a "task," which corresponds to a process.

Since a process is an abstraction of a processor in action, breaking up programs into processes is a start toward multiple processor systems. In such systems, a process is a natural unit to allocate to an available processor, because a process is naturally processor independent. But before multiple processor systems can be implemented, several other concepts must be formalized and made processor independent as well. In particular, models have to be developed that abstract the details of the interprocess communication and synchronization mechanisms.

Much recent operating system research has focused on interprocess communication and synchronization, and progress here has been rapid. Software structures such as "communication ports" and "messages" have been developed to separate the logical character of the communications mechanism from the physical implementation. A communications port is an abstraction of a "mailbox," a place where messages can be sent or picked up. Processes send messages to ports and wait at ports to receive messages from other processes. Process scheduling can be handled in a similarly processor-independent fashion by defining the concept of a "dispatching port," where processes wait to be allocated to processors.

The object model and the process model are independent, but they can be combined very simply and elegantly. Processes and ports will be objects, and the procedures that manipulate these objects can be grouped into type managers. The HYDRA operating system for the C.mmp multiple-processor system (16 PDP-11 minicomputers), a research project at Carnegie-Mellon University, demonstrated just this combination of the object model with multiprocessing.*

4.2.3 Architectural Support

What does it mean to say that an architecture supports a software methodology? At a minimum, it means that the architecture simply provides mechanisms to help users follow the methodology; in this case we will say that the architecture is *oriented* toward the methodology. But in a more general sense it could mean that the design of the architecture itself also follows the principles of the methodology, in which case we will say that the architecture is *based* on the methodology.

For example, an *object-oriented* architecture helps users to write programs that use type managers and reflect the philosophy of information hiding. But if the architecture's design is also an example of the use of that philosophy, then we will say that it is an *object-based* architecture. An object-based architecture can be oriented to both the object model and the process model; it can support type managers, but it can also support multiprocessing, as the HYDRA system suggested. In fact, an object-based architecture provides a uniform approach toward supporting all system services.

If an architecture, a high-level language, and an operating system are all based on the same methodology, then the boundaries between them begin to blur. It becomes hard to tell where one begins and another leaves off. Functions can be moved from the operating system to the architecture, or from the language to the operating system. Basically, the whole system has a kind of geometrical integrity, constructed as it is out of a set of common building blocks.

*William Wulf, Roy Levin, and Samuel P. Harbison. *HYDRA/C.mmp: An Experimental Computer System* (New York: 1981).

The choice of which functions to provide in the architecture and which in system software is dependent on many factors—the size of the microcode, the number of gates on the chip, the presence of time-critical bottlenecks in certain frequently-used operations, and the need for security, among others. Management of any given type of object can be a responsibility shared by hardware and software. Time-critical functions will be placed in hardware while less frequently used or extremely complex functions will be left to software.

For example, type manager modularization results in many more intermodule control transfers than do more conventional modularization schemes. If the mechanism of these transfers results in high overhead, it will not be feasible to use this methodology. The same holds true for process scheduling and dispatching. The frequent process switches envisioned by the process model would be prohibitively expensive on a conventional architecture. These factors suggest that intermodule control transfer mechanisms and process scheduling and dispatching mechanisms should be provided by the architecture.

For reasons of flexibility, it is essential to maintain a separation of *policy* and *mechanism*. Resource management policies should be in software, while mechanisms should be in hardware. For example, scheduling policies (e.g. the choice between a round-robin system or a priority system) should be specified in software, since each application may have different scheduling requirements. But all scheduling policies require an efficient scheduling and dispatching mechanism (i.e. the ordering of available processes according to the policy, and the selection of one of these processes for execution), so this mechanism should be in hardware.

4.3 iAPX 432 Object-Based Architecture

The iAPX 432 architecture provides the mechanisms which make it feasible for users to write programs modularized into type managers and the mechanisms to support multiprocessing and other system services. The iAPX 432 can provide these mechanisms because it has an *object-based architecture*. This means that the designers of the iAPX 432 architecture themselves followed the object methodology described in section 4.2.1.

Therefore, since the object methodology is based on programming with type managers, it makes sense to ask what structures in the architecture correspond to type managers.

4.3.1 System Objects

In the iAPX 432 architecture, the structure that corresponds to a type manager is the group of all hardware operators that manipulate certain complexes of segments (see section 2.4.4). These hardware manipulated segment complexes are called *system objects*. System objects may consists of one segment, a refinement of one segment (see section 2.4.4), or several segments, but they are manipulated as a unit by their instructions.

System objects provide the architectural mechanisms that support type manager modularization and system services. For example, system objects are used to implement process scheduling and interprocess communications (in fact, an iAPX 432 process is a system object, as is a communication port and a dispatching port—see section 4.5 for more details).

System objects provide mechanisms that allow users to build their own type managers and define their own objects. Like system objects, user objects can be represented by one segment, a refinement of a segment, or an entire segment com-

plex. In fact, the only real difference between system objects and user objects is that the operations on system objects are implemented primarily in the microcode, whereas user objects are manipulated by software. User objects can be given hardware-enforced protection features that are similar to the ones associated with system objects.

For example, a user could create a type manager containing a telephone directory object and two procedures (look up telephone number, look up address) that manipulate this object. The iAPX 432 architecture offers mechanisms that make it impossible for any other procedure to access the directory object directly. The only way another procedure could access the telephone directory is by calling one of the two procedures in the type manager. The directory object is thus totally protected against accidental or malicious damage, and no procedure outside the type manager needs to know the internal representation of the directory.

4.3.2 Object Protection

The two-level mapping of the iAPX 432's structured memory (see section 2.4.1) provides the mechanisms for restricting access to objects. Recall the definition of object references in section 2.4.4 in terms of access descriptors and segment descriptors. These object references form the basis of object protection.

Object A is said to have an object reference for object B if an access segment in object A's segment complex contains an access descriptor for the root segment of B's complex. See figure 4-1 for an illustration.

The access rights mechanism of segmented addressing provides one level of object protection, the segment type mechanism provides another (see section 2.4.2), but the basic protection comes simply from controlling the dispersal of object references. The only way someone can get access to an object is by acquiring an object reference for it. If a procedure doesn't have an object reference for an object, the object

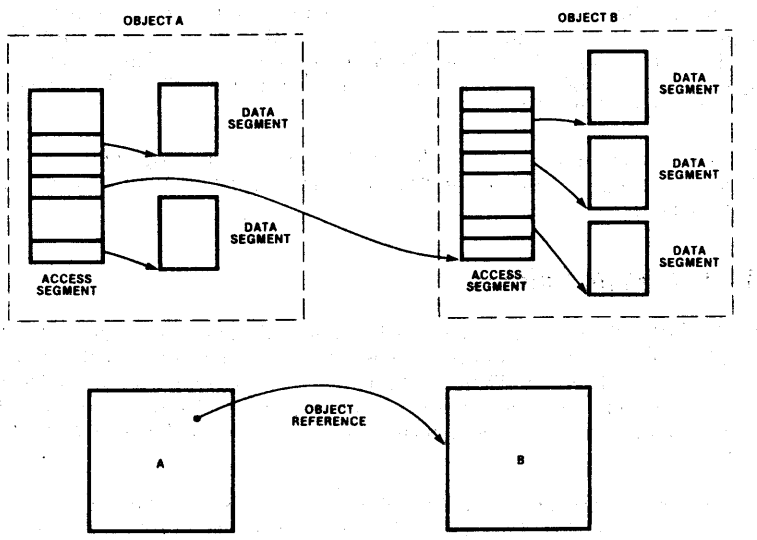


Figure 4-1. Objects and Object References 171821-22

simply doesn't exist for the procedure. In operating system research, an object reference is called a *capability*, and the whole object-oriented access mechanism is called *capability-based addressing*. Figure 4-2 shows a mulitiple-object environment and the object references of each object.

4.4 Architectural Support for Type Managers

Two aspects of the programming environment of a type manager are candidates for architectural support: its *static structure* and its *dynamic behavior* during execution. The iAPX 432 architecture provides a system object called a *domain* that implements the static structure, while the dynamic behavior is handled by a system object called a *context*.

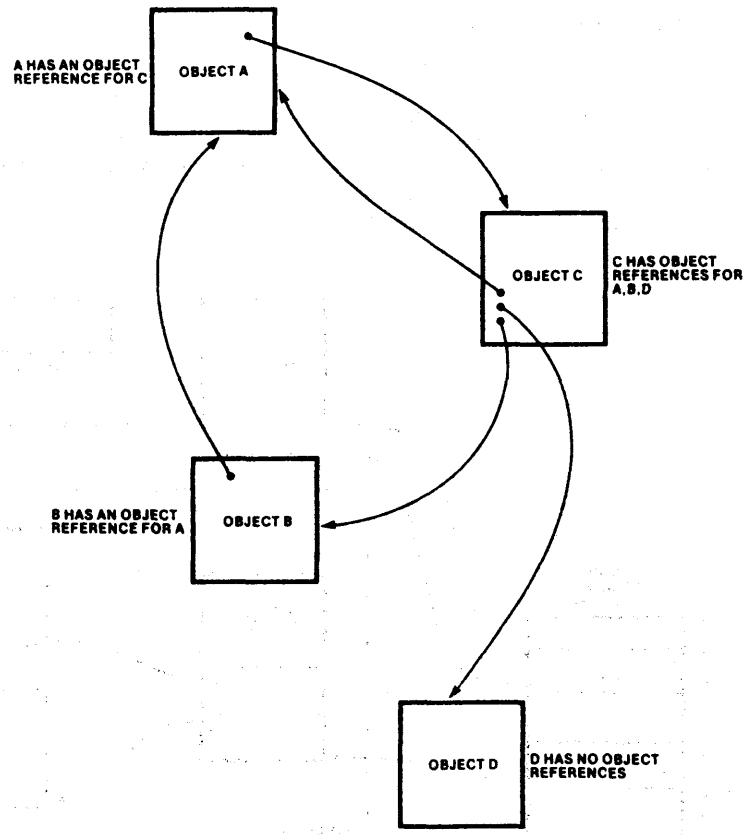


Figure 4-2. Object References 171821-23

4.4.1 Domain Objects

A type manager is represented by a segment complex called a *domain object*. The root access segment in a domain object, that is, the *domain access segment*, contains object references for all the other objects in the domain. These other objects include instruction objects (represented by instruction segments) that contain the type manager's procedures, and data objects (represented by data segments or segment complexes) containing the data that is encapsulated inside the type manager.

The domain access segment distinguishes its *public* object references from its *private* references. The public object references are contained in a refinement of the total access segment; all other references are private. Access to the list of public references will be made available to procedures outside the domain itself, but the private references are accessible only by procedures within the domain. Thus the private objects are effectively "hidden" inside the domain. This mechanism supports the information hiding philosophy of the object-oriented design methodology. Figure 4-3 shows the segments that make up a simple domain object. Notice that references to the instruction segments have been placed in the public portion of the domain access segment, while the data segment complex is hidden in the private portion.

4.4.2 Context Objects

A *context* object is the iAPX 432 hardware-recognized object that supports the run-time environment of a procedure in a type manager. Before a called procedure can be executed it must be supplied with a data structure containing run-time information that is unique to this particular instance of execution (e.g., the new instruction pointer value and the return address to the calling procedure). All this information is

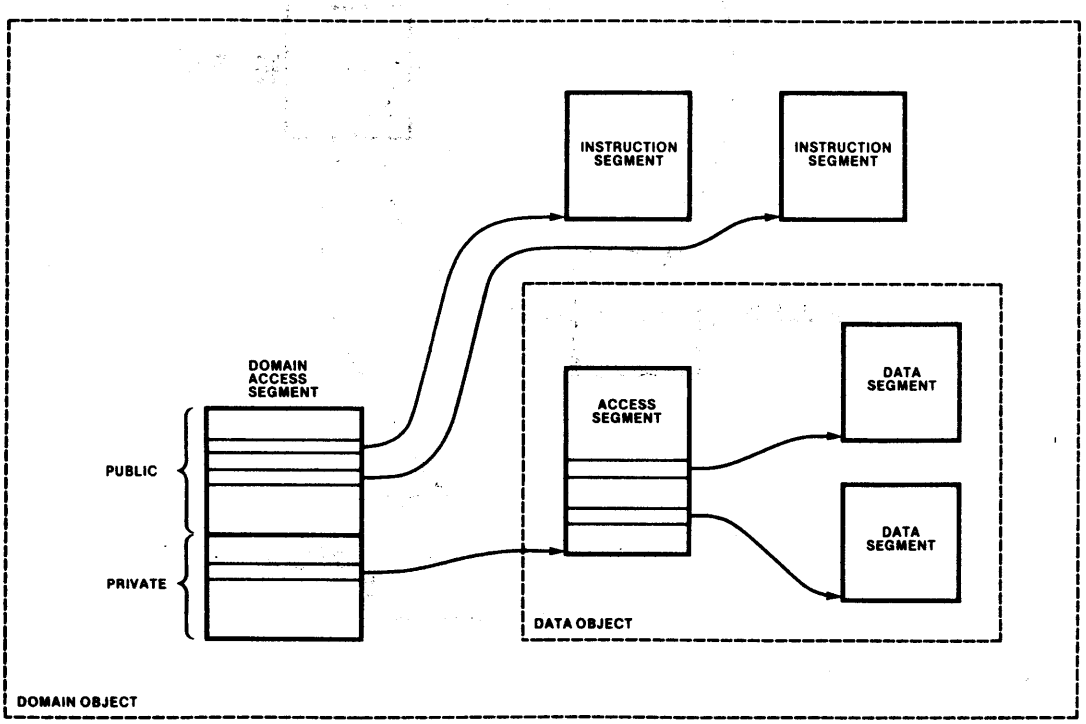


Figure 4-3. A Domain Object 171821-24

contained in a context object. It has much the same function (although in a much more protected fashion) that the *activation record* or *stack frame* has in stack-based programming language implementations.

A context object is represented by a segment complex, the root of which is called the *context access segment*. The context access segment contains object references for the following objects:

- an operand stack for expression evaluation;
- the domain access segment of the context's type manager;
- four access segments that define the instantaneous *access environment* of the called procedure (i.e. all the objects that may be referenced by the procedure);
- a message (if there is one) from the calling procedure;
- the context object of the calling procedure;
- a data segment containing local constants needed by the called procedure; and
- a data segment containing the current instruction pointer, stack pointer, and status information.

By convention, we will group into the context object the context access segment, the operand stack, the context data segments, and the four access segments that define the access environment (called the four *entry access segments*). For reasons which will become clear in the next section, one of the four entry access segments is the context access segment itself. Figure 4-4 shows the segments and object references included in a context object.

4.4.3 Calling Contexts

A new context object is automatically created when a procedure in one type manager calls a procedure in another type manager. The new context is automatically destroyed when the called procedure returns control to the calling procedure. A call instruction may reference any instruction object in the public part of any domain in the calling access environment. See figure 4-5 for an example of a call.

When a call is made, the new context has a different access environment from the calling context, so the access environment of the called procedure will be different from the access environment of the calling procedure. For example, notice that in figure 4-5 the access environment of the calling context includes only the public part of the called domain, whereas the new context's access environment includes the entire domain. Each invocation of a procedure can be given an access environment that includes only those objects it needs to access. Thus the call context mechanism helps to insure the security of the whole program.

Along with the call instructions, the iAPX 432 instruction set includes two groups of branch instructions: intrasegment branches and intersegment branches. The range of the former group is limited to a single segment, while the latter group can specify branches to any instruction segment in the domain (i.e. to any procedure in the current type manager). Branch instructions do not change the access environment.

4.4.4 Using the Inside Representation of Type Managers

The domain objects we have considered so far include both procedures and the data objects they act on. We will call this kind of domain object, in which data objects are in the domain, the *inside representation* of type managers.

The telephone directory example from section 4.3.1 can be easily implemented using the inside representation. Object references for the two procedures are placed in the public part of the domain, while the telephone directory object itself is in the private portion of the domain. Thus the procedures can be called from outside the domain, but the directory is inaccessible from outside. Whenever anyone needs to access the directory, a call must be made to one of the two procedures, which performs the operation on the directory, then returns to the caller.

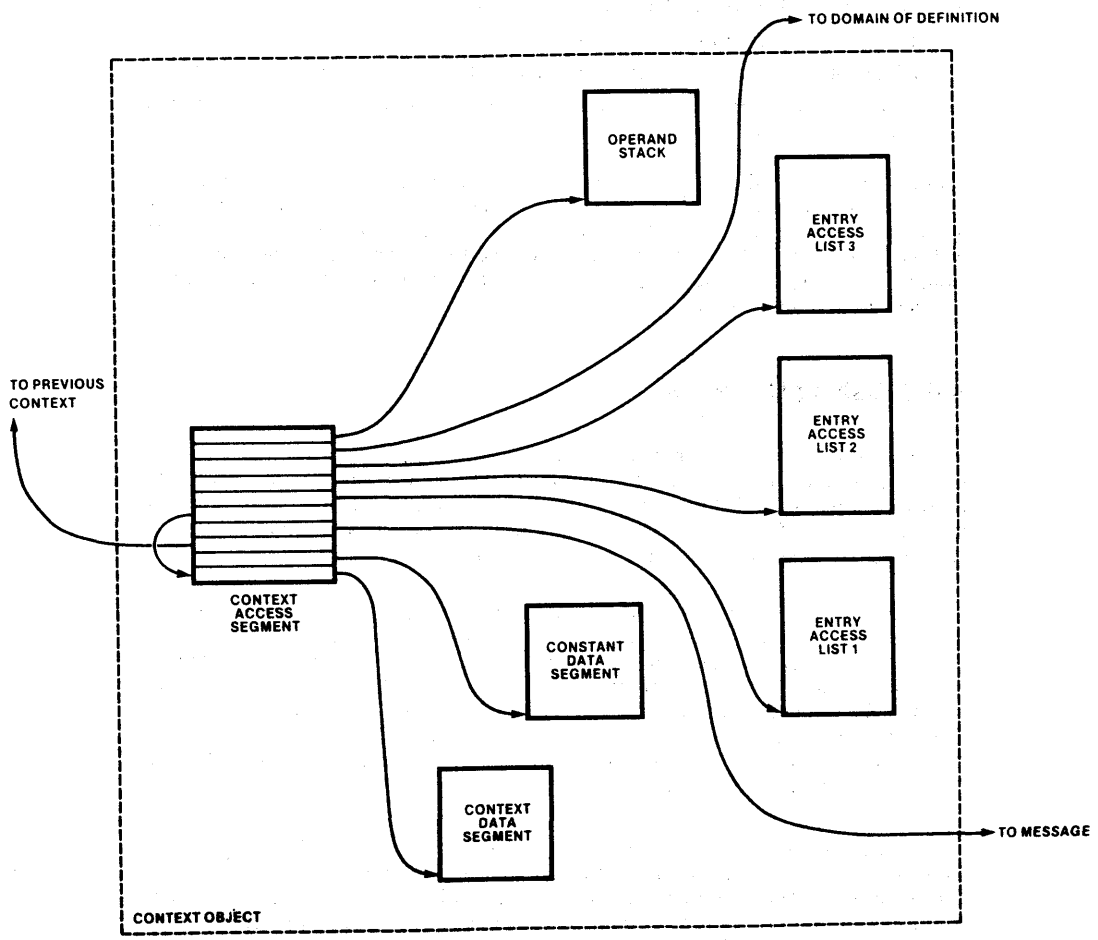
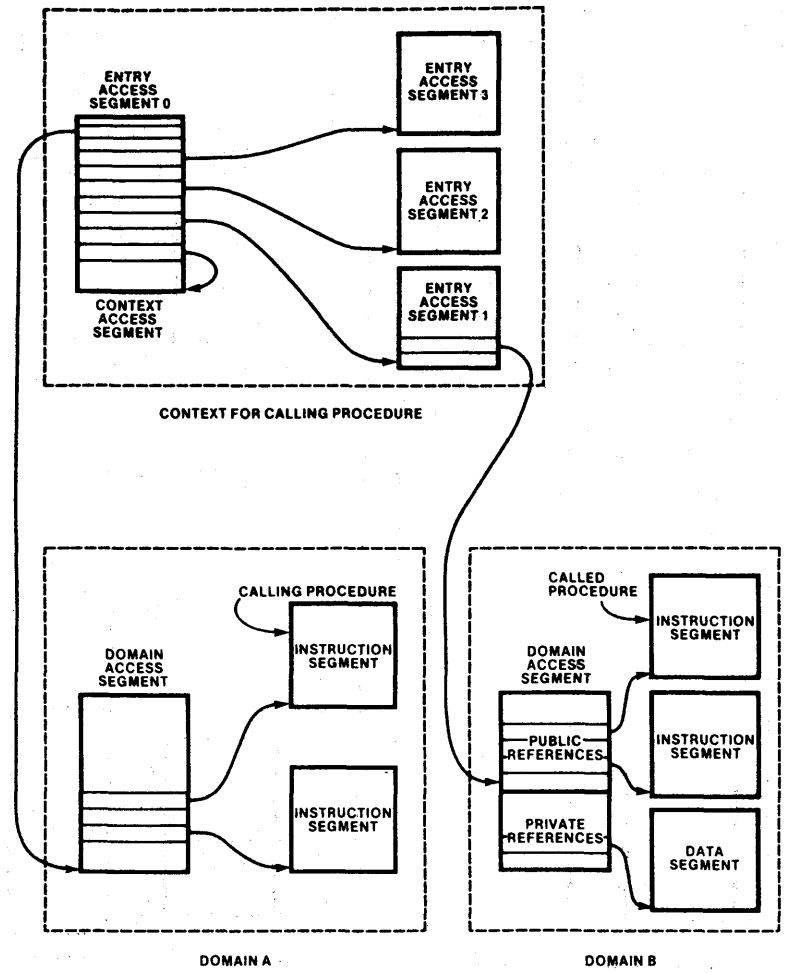


Figure 4-4. A Context Object

171821-25



INITIAL CONFIGURATION
(PROCEDURE IN DOMAIN A CALLS PROCEDURE IN DOMAIN B)

Figure 4-5. Calling a Context (1 of 2)

171821-26

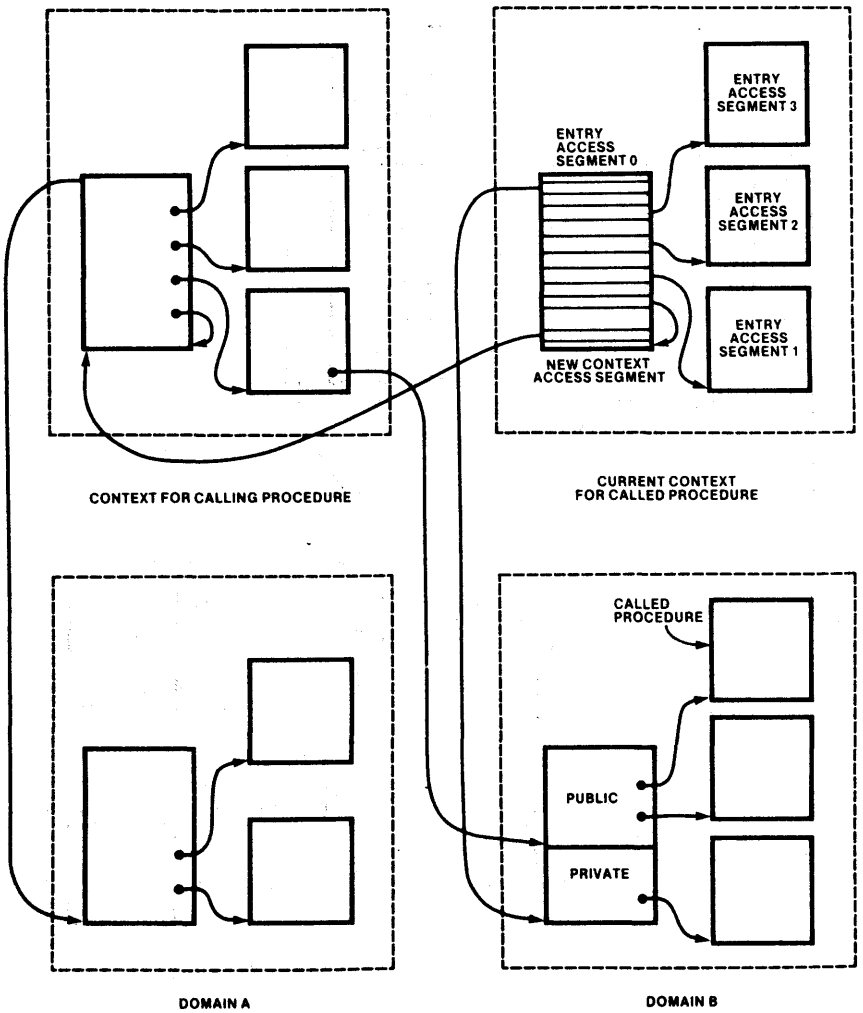


Figure 4-5. Calling a Context (2 of 2)

171821-27

Figure 4-6 displays the calling sequence for the inside representation in detail. The first drawing in figure 4-6 shows the situation that results after a call has been made to the "look up number" procedure in the telephone directory type manager. Notice that the telephone directory object is not in the immediate access environment of the procedure, because the directory is referenced only by the domain access segment, which is not one of the four entry access segments.

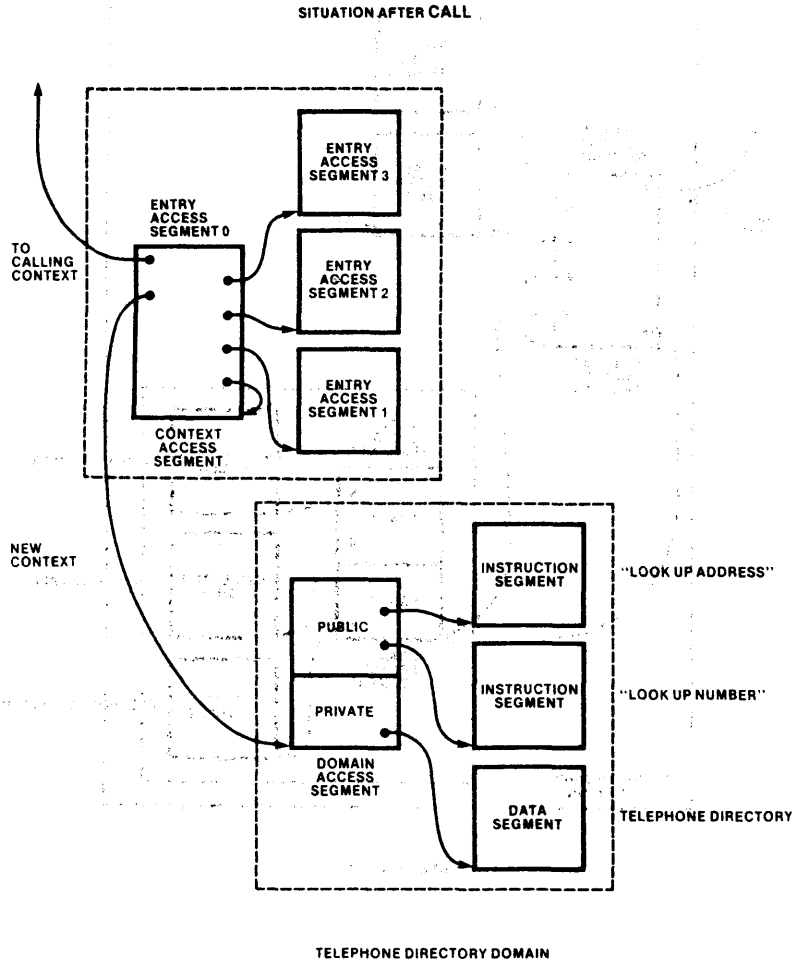


Figure 4-6. Changing the Access Environment (1 of 2) 171 821-28

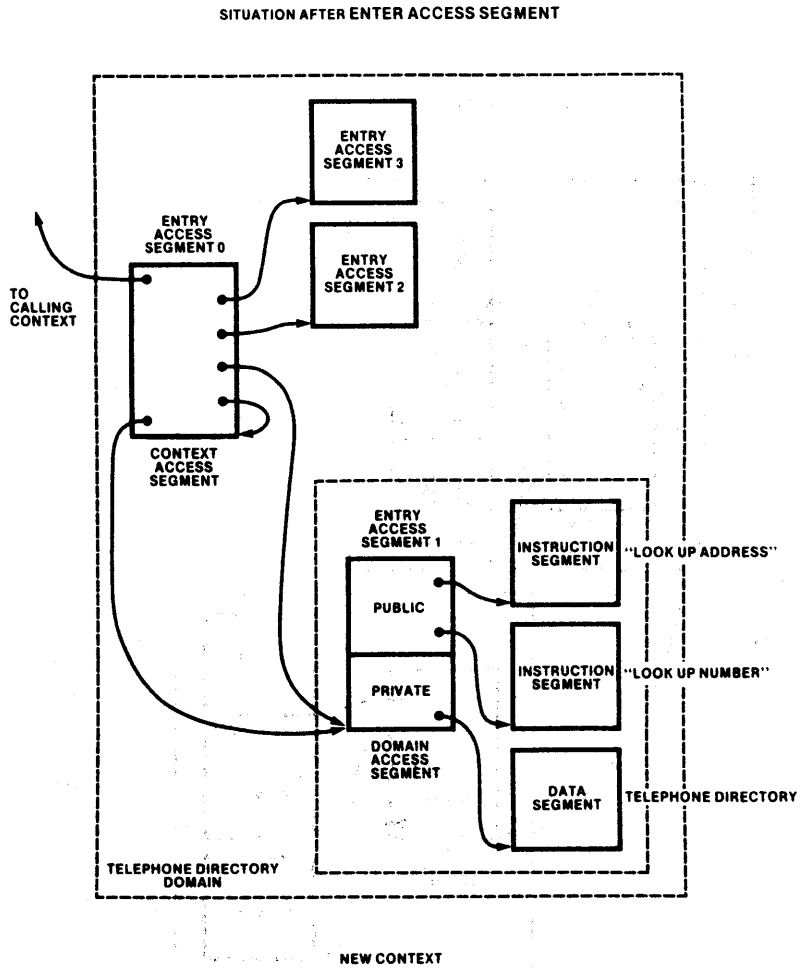


Figure 4-6. Changing the Access Environment (2 of 2) 171821-29

In order to bring the directory object into the access environment, the domain access segment must be made into one of the entry access segments. The iAPX 432 instruction set includes an instruction, ENTER ACCESS SEGMENT, which allows the user to make any access segment in the immediate access environment into an entry access segment. Since the domain access segment is referenced by entry access segment 0 (the context access segment itself), the domain access segment is in the access environment. The second drawing in figure 4-6 shows the result of executing an ENTER ACCESS SEGMENT instructon. The domain access segment has been made entry access segment 1, and the telephone directory object is now in the access environment. The procedure can now act on the object.

This example illustrates why the context access segment is one of the four entry access segments.

4.4.5 Using the Outside Representation of Type Managers

The alternative to the inside representation is for data to be kept outside the domain object; only procedures will be found inside. Whenever a procedure in the type manager is called, a reference to the data object must be passed into the domain. We will call this the *outside representation* of type managers.

The outside representation is actually more common than the inside representation, since the outside representation can be used with multiple instances of the same type of object. For example, the type managers that control system objects use the outside representation. There is only one type manager for each type of system object, but many different objects of one type can be controlled by the same type manager. The outside representation is also used for large data objects that must be passed to several type managers.

As an example of the outside representation, consider another telephone directory type manager. This type manager contains only the two procedures, look up number and look up address, and no data object. Whenever a procedure is called, a reference to some telephone directory object must be passed to the procedure. Figure 4-7 illustrates this example. The calling procedure uses the instruction CALL WITH MESSAGE, and sends the directory object as a message to the called procedure.

The problem with this simple use of the outside representation is that the data object has no protection. Any procedure with a reference for the object can manipulate it. Ideally, only procedures in the telephone directory type manager should be able to manipulate directory objects. System objects solve this problem through the mechanism of *type checking*; the only operators that are allowed to access a system object of some system type are the operators in one particular type manager. Each time an operation is attempted on a system object, the system type field in the corresponding object descriptor (see sections 2.4.2 and 2.4.4) is checked to see if the operation is allowed.

The iAPX 432 architecture provides a way to give a similar type checking mechanism to user objects.

4.4.6 User Defined Types

The iAPX 432 architecture includes several system objects and operations that can be used to create a protection mechanism for user-defined objects that is similar to the hardware protection mechanism provided for system objects by system type checking. This protection mechanism is called *user-defined type checking*.

The type definition object (TDO) is used to label user objects so they can have type checking performed on them, in much the same way that the system type fields in the object descriptors label system objects. All references to a typed object are made through a special kind of descriptor, called a *type descriptor*, in the object table (see section 2.4.4). This type descriptor points to both the typed object and the TDO.

The iAPX 432 instruction set includes several instructions for creating typed objects and for accessing them. Two varieties of types can be created, *public types* and *private types*. Public types provide no hardware protection; they merely serve to label user objects for the software. Private types, on the other hand, offer a protection mechanism. Successfully accessing a privately-typed object requires a reference to the TDO as well as a reference to the object itself. See Chapter 7 of the *iAPX 432 General Data Processor Architecture Reference Manual* for more information on how user typing is implemented.

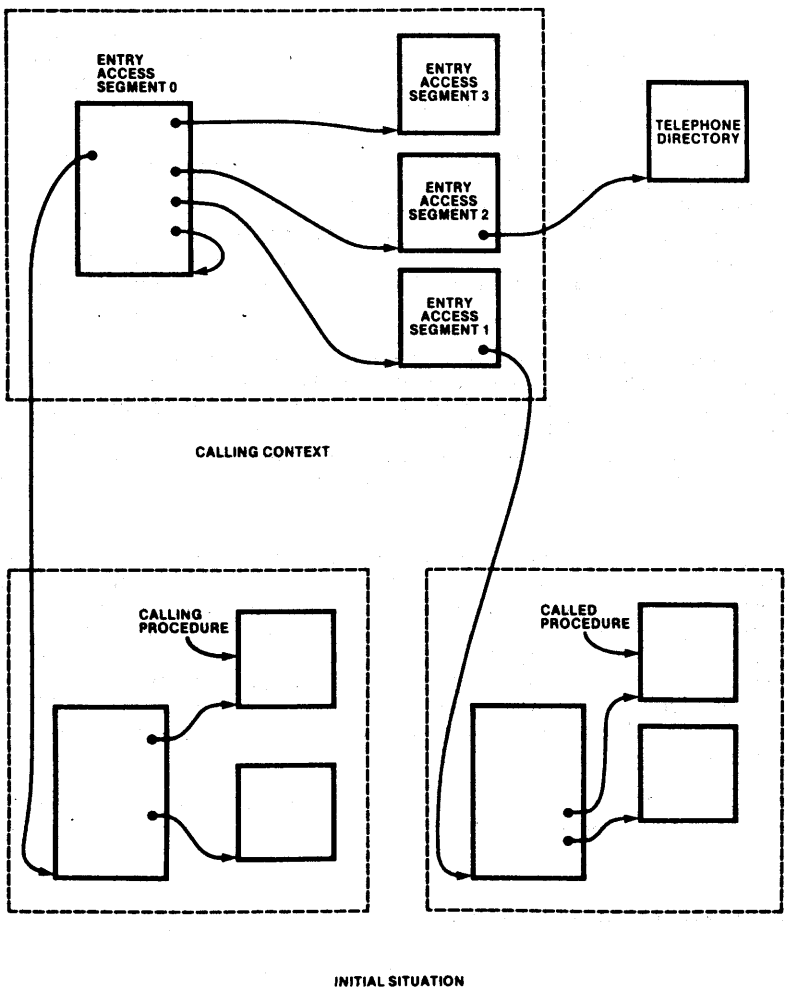


Figure 4-7. The Outside Representation (1 of 2) 171821-30

Figure 4-8 shows the same example as figure 4-7, but now the telephone directory object has been given a user-defined private type (shown by the TDO object). The domain has a matching TDO, so procedures in the domain can access the directory. No other procedures can access the directory. Notice that both a CALL CONTEXT WITH MESSAGE instruction and an ENTER ACCESS SEGMENT instruction are used in the example. The first instruction passes the typed object into the type manager. The second instruction puts the TDO object in the access environment, where it can be used in an access to the directory object.

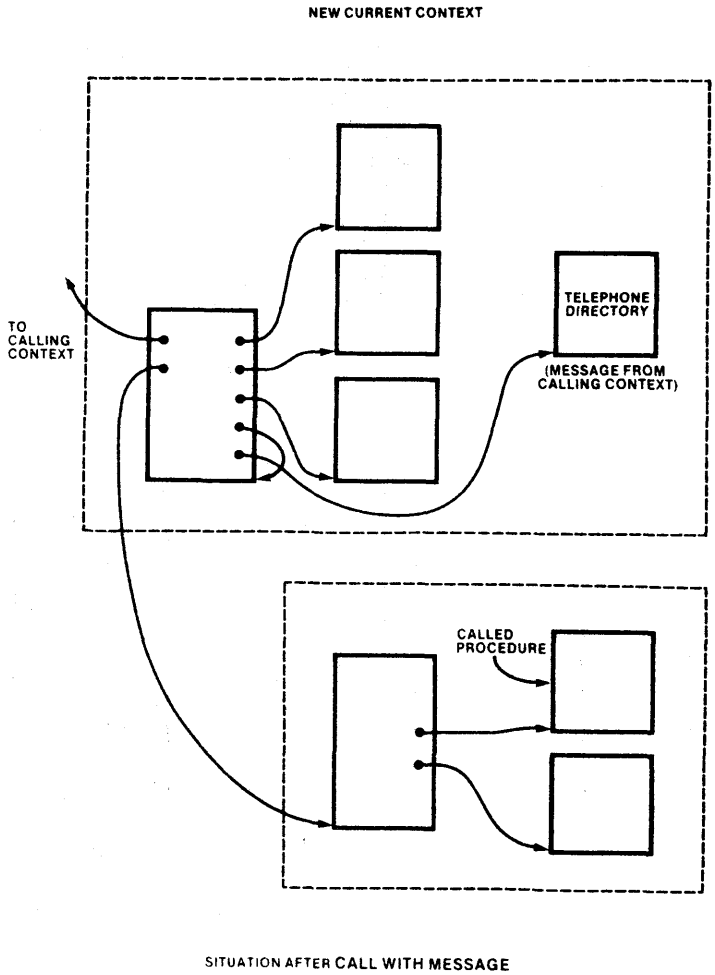


Figure 4-7. The Outside Representation (2 of 2) 171821-31

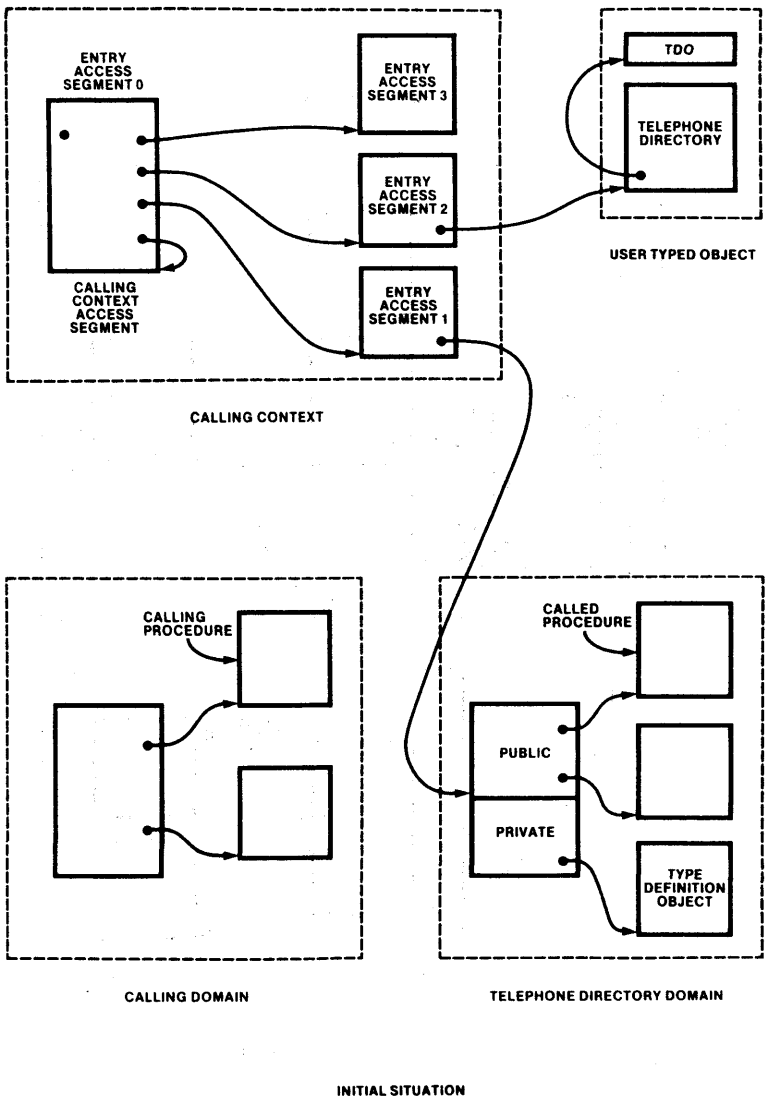
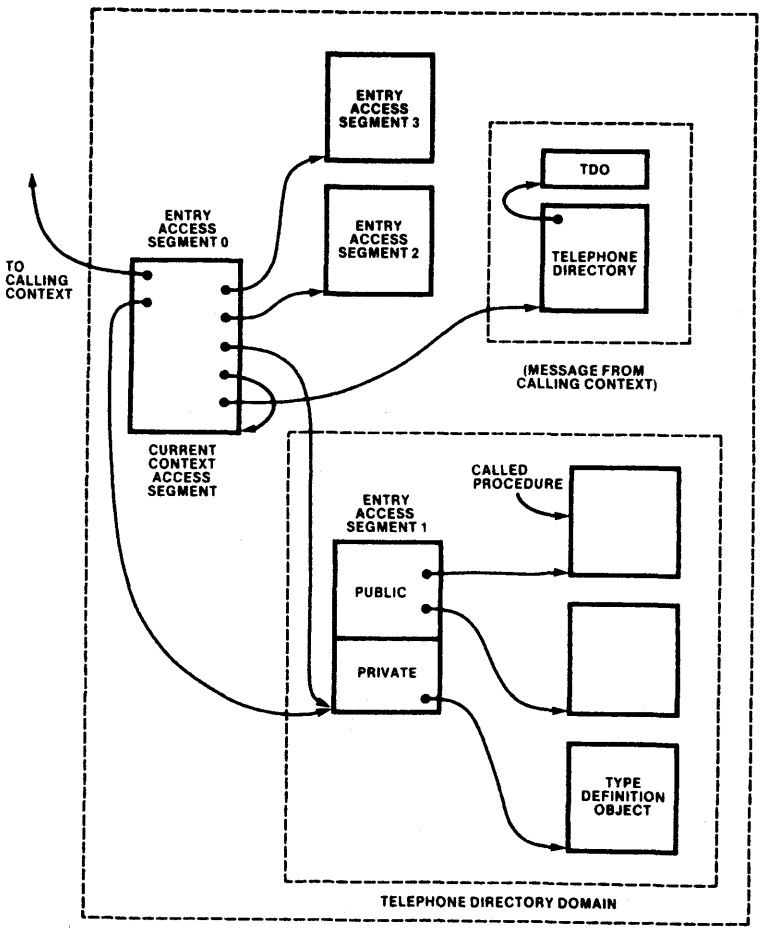


Figure 4-8. User Defined Type 171821-32



SITUATION AFTER CALL CONTEXT WITH MESSAGE AND ENTER ACCESS SEGMENT

Figure 4-8. User Defined Type (Cont'd.) 171821-33

4.5 Architectural Support for System Services: The Silicon OS

The iAPX 432 architecture implements a number of resource management mechanisms in the hardware. We call these hardware-supplied mechanisms the *Silicon Operating System*. (It is worth stressing again, as we did in section 4.2.3, that only the *mechanisms* are supplied in the hardware; the resource management policies are established by software.)

We can divide the Silicon OS into two unequal parts, one concerned with the management of *memory* resources, the other with the management of *processor* resources. By far the largest fraction is concerned with processor management. In the following sections we will explore the objects that implement dynamic storage allocation (one aspect of memory management) as well as process scheduling, dispatching and interprocess communication.

4.5.1 Process Objects and Processor Objects

A process is the unit of concurrent execution; it may also be defined as the unit of code that can be scheduled to run on a processor. The iAPX 432 architecture supports concurrent programming with several system objects: *process* objects for every process in the system, *processor* objects for every processor in the system, and *processor carrier* objects, which link processor objects to process objects. We will not describe the complete segment content of these objects, as we did for domains and contexts, but users can refer to the *iAPX 432 General Data Processor Architecture Reference Manual* for details.

We have now mentioned all the system objects that are needed for a minimal iAPX 432 system. Figure 4-9 shows the basic objects and the object references needed for a simple system. The system includes a processor object, a processor carrier object, a process object, a context object, and a domain object.

4.5.2 Storage Resource Objects

Storage Resource Object (SROs) are used to implement the dynamic storage allocation mechanisms (see section 2.4.3) of the Silicon Operating System. The SROs and the operations associated with them perform the actual binding of an unallocated descriptor in the object table to a newly created segment in memory. Several instructions, including CREATE ACCESS SEGMENT and CREATE DATA SEGMENT, require a reference to an SRO.

A special SRO associated with the process object is used implicitly whenever a call instruction is executed, since these instructions create a new context object, which must be allocated storage when it is created. Unless an SRO is used explicitly, it does not have to be included in the system. Therefore we have not included it in the simple system shown in figure 4-9.

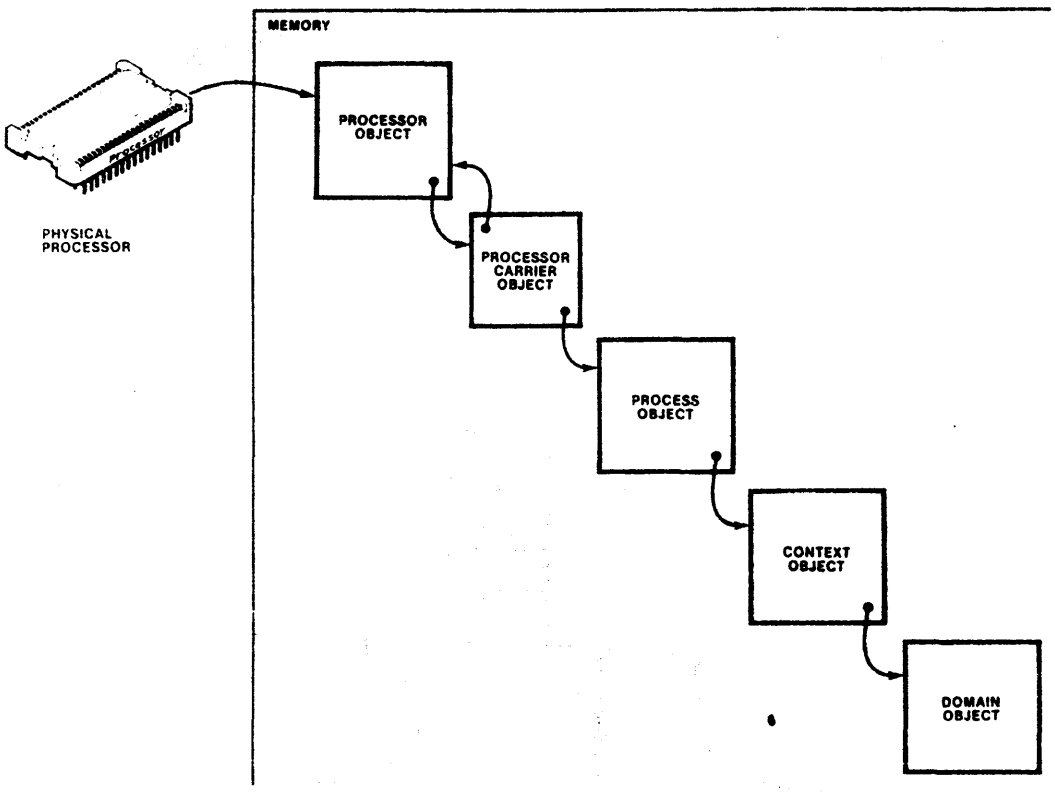


Figure 4-9. Basic System Objects 171821-34

4.5.3 Simple Interprocess Communication Without Blocking

In a multi-process environment, procedures in different processes often need to exchange information. Since the processes are asynchronous, the first one may be ready to communicate before or after the second is ready. Therefore, some kind of interprocess synchronization must be provided. In the iAPX 432 Silicon OS, a system object called a *communications port* provides the synchronization mechanism that allows asynchronous processes to communicate.

The information communicated is sent in a *message object*, which has no defined system type and in fact may be any object. For example, as we shall see in section 4.5.5, an entire process can be sent as a message.

When a procedure in one process wants to send a message to a procedure in another process, the sender specifies a message object and a communictions port as the destination of the message, then executes a SEND instruction. Similarly, the receiver specifies the same communications port as the source of a message and executes a RECEIVE instruction. Neither process need be aware of the other; neither process

knows, when it communicates with the port, if the other process is also ready to communicate. The port object contains a fixed-length buffer that holds a queue of object references for the message objects that have been sent to the port and are waiting for a receiver. As processes execute receive instructions on the port, messages are removed from the head of the queue.

Figure 4-10 shows simple examples of messages being sent and received via communications ports. Notice that the message buffer in the communications port is neither full nor empty, so both the send and receive instructions can be executed without difficulty. The message object being sent must be in the access environment of the sending context. When a message is received, its object reference is placed in the "message from calling context" slot in the context access segment.

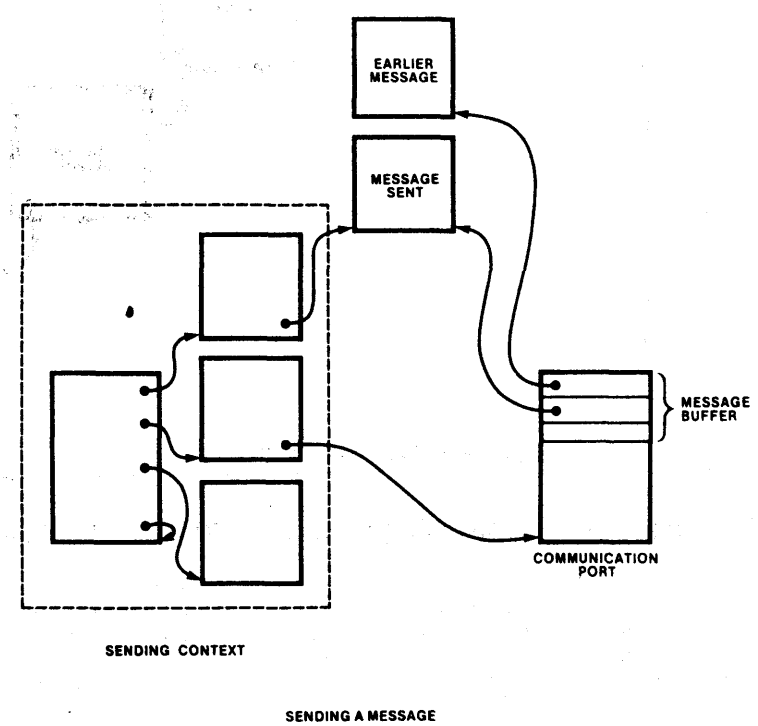


Figure 4-10. Sending and Receiving Messages 171821-35

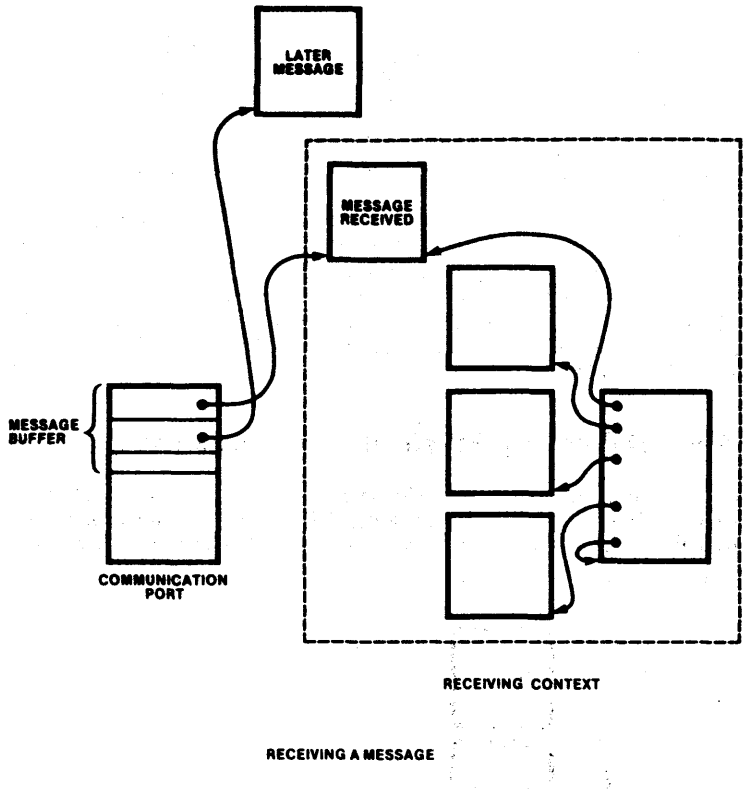


Figure 4-10. Sending and Receiving Messages (Cont'd.) 171821-36

4.5.4 Conditional and Surrogate Communication

As we shall see in the next section, when a process executes a simple SEND instruction on a port whose message buffer is full, or when it executes a simple RECEIVE on an empty port, the process is suspended while it waits for space in the buffer or for a message. Often, however, it will be unacceptable for a process to become suspended in this manner. The conditional and surrogate communication operators (CONDITIONAL SEND, CONDITIONAL RECEIVE, SURROGATE SEND, SURROGATE RECEIVE) can be used to avoid this suspension and also to implement more advanced forms of communication.

Conditional sends have exactly the same effect as simple sends, if the message buffer in the port is not full. Similarly, conditional receives are identical to simple receives, if the port has a message waiting. However, if these conditions are not satisfied, the conditional operations are not performed, whereas the simple operations are blocked and the process is suspended (see section 4.5.5). The conditional operations can be retried at a later time.

Surrogate sends and receives are more complicated than either simple or conditional operations. In surrogate operations, two ports are specified, along with a special system object called a *carrier*. When a SURROGATE SEND instruction is executed, for example, the message is sent to the first port, if the buffer is not full. If the buffer is full, the carrier object, containing a reference to the message, is enqueued

in a linked list at the port. When space becomes available in the buffer, the message object reference is inserted in the port, and the carrier is resent to the second port. Similarly, a SURROGATE RECEIVE can cause its carrier to become enqueued at a port waiting for a message. When the message is received, it and its carrier are resent to the second port. Figure 4-11 shows a queue of carriers and messages at a port.

Surrogate operations allow sophisticated communications mechanisms to be implemented. For example, priority communications channels can be implemented by specifying a priority-based port protocol and encoding priority information in the carrier object. Parallel communications channels can be multiplexed onto one port, in priority order, by executing a SURROGATE RECEIVE on several ports, but specifying the same second port in each case. A message sent to any of the several ports will be resent to the same second port. Thus, by executing a receive on the second port, one process can monitor several prioritized channels. (See Chapter 4 of the *iAPX 432 General Data Processor Architecture Reference Manual* for more information.)

4.5.5 Process Blocking, Scheduling, and Dispatching

When a process attempts to execute a simple SEND instruction on a communications port with a full buffer, or to execute a RECEIVE instruction on a communications port with an empty buffer, the process will become *blocked*. When a process is

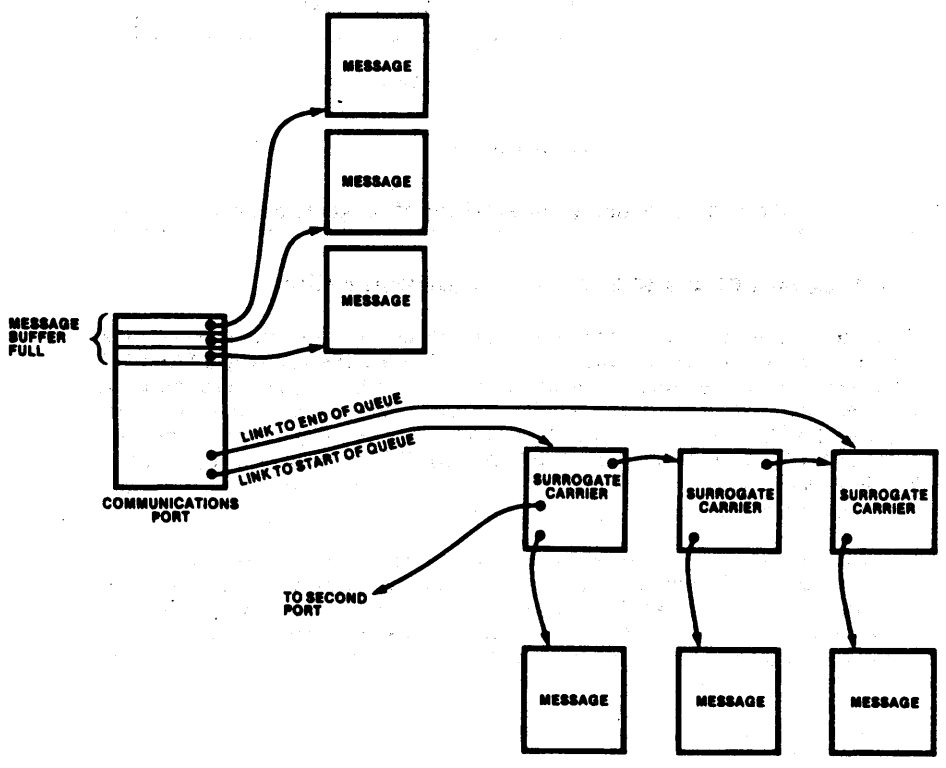


Figure 4-11. Surrogate Sending

171821-37

blocked it is enqueued at the communications port through its *process carrier*, a special carrier object associated with the process object itself. When the port buffer is available again, the process will be unblocked, and *resent* to another port object called a *dispatching port*. At the dispatching port the hardware automatically performs *scheduling* and *dispatching*.

Scheduling is the determination of the execution order of each process in a multiprocess system. Although the scheduling *policy* (e.g. round-robin or priority-based) is set by software, the *implementation* of the policy (the ordering mechanism itself) is performed by the iAPX 432 hardware. Dispatching is the assignment of a physical processor to execute a process.

Besides process objects, process carrier objects, and dispatching port objects, the hardware also uses processor objects and processor carrier objects (see section 4.5.1) to implement scheduling and dispatching. A dispatching port binds processor carriers to process objects just as a communications port binds processes to messages. (In fact, communications ports and dispatching ports are really just two slightly different varieties of a generalized port object.) When the dispatching port is created, the software specifies parameters that define the scheduling policy. (For example, one parameter defines the maximum period of time a process can run on a processor before rescheduling occurs.) The hardware executes this policy automatically for all available processes.

A process can become available for scheduling for a variety of reasons: it can be blocked while waiting to send or receive a message; it can use up its time slice on a processor; a fault can occur which causes the process to suspend itself; or it can deliberately suspend itself by executing a DELAY instruction.

Figure 4-12 shows the complete sequence of process blocking at a communications port, unblocking and resending to the dispatching port, scheduling (i.e. enqueueing at the dispatching port), and dispatching to an available processor.

Several dispatching ports can exist in a system (even a single-processor system). In this case, the ports usually implement different scheduling policies or else are dedicated to different functions. (One port might be dedicated to scheduling processes that have faulted, for example.)

4.5.6 Multiple-Processor Systems

The iAPX 432 architecture can support more than one active processor in the same system. Dispatching on a multiple processor iAPX 432 system becomes the assignment of a process to an available processor. The ability to add processors to a system and, without changing the software, improve the performance of a multi- process environment is one of the most important and powerful features of the iAPX 432.

The iAPX 432 architecture supports multiple processor dispatching as a simple extension of single-processor dispatching. Merely adding another processor object and processor carrier object for each additional physical processor enables the system to expand its power with no other changes required. Two processor carrier objects reference the same dispatching port, but otherwise the mechanism is the same. Multiple dispatching ports may be provided, as in the single-processor case.

Figure 4-13 shows a two-processor system running six processes. Two processes are executing and four are waiting at the dispatching port for a free processor.

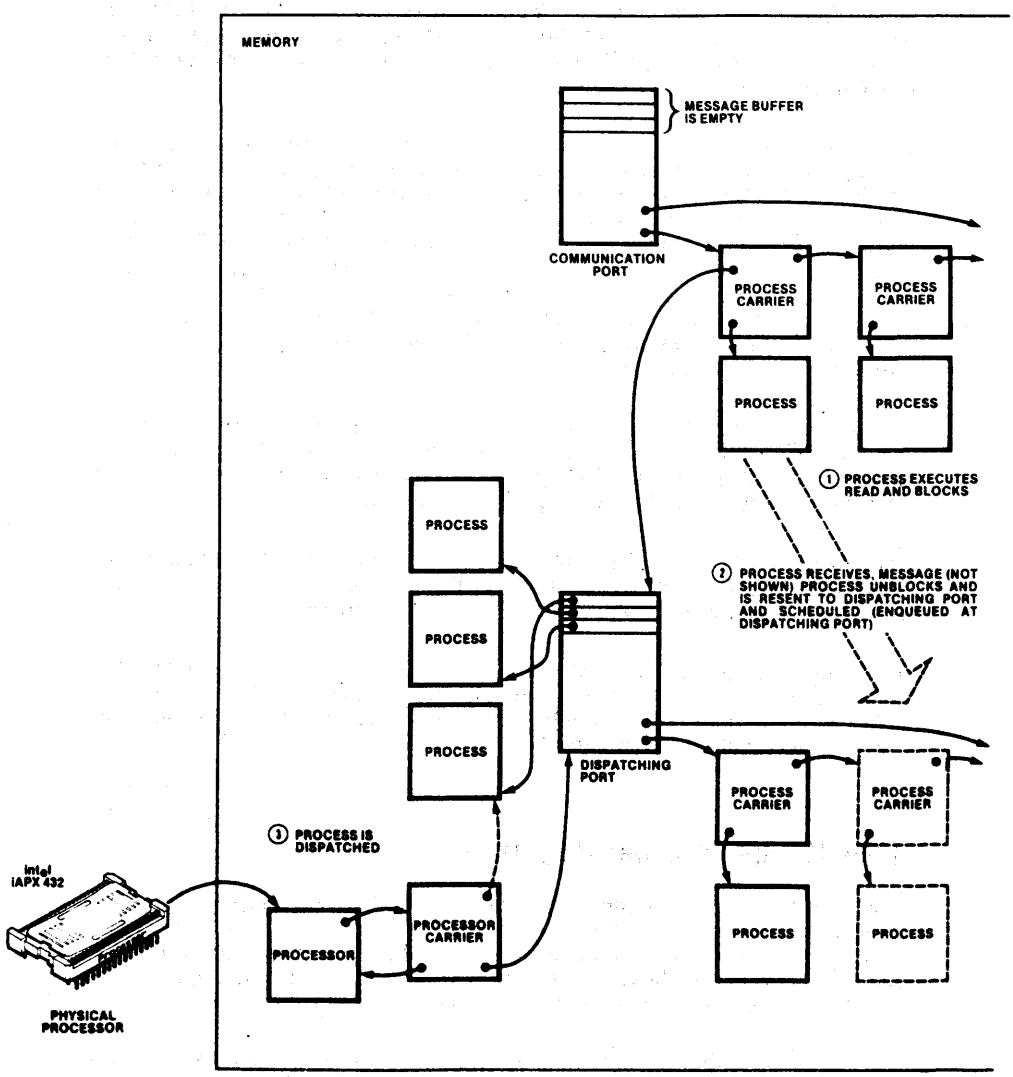


Figure 4-12. Resending, Scheduling, and Dispatching

171821-38

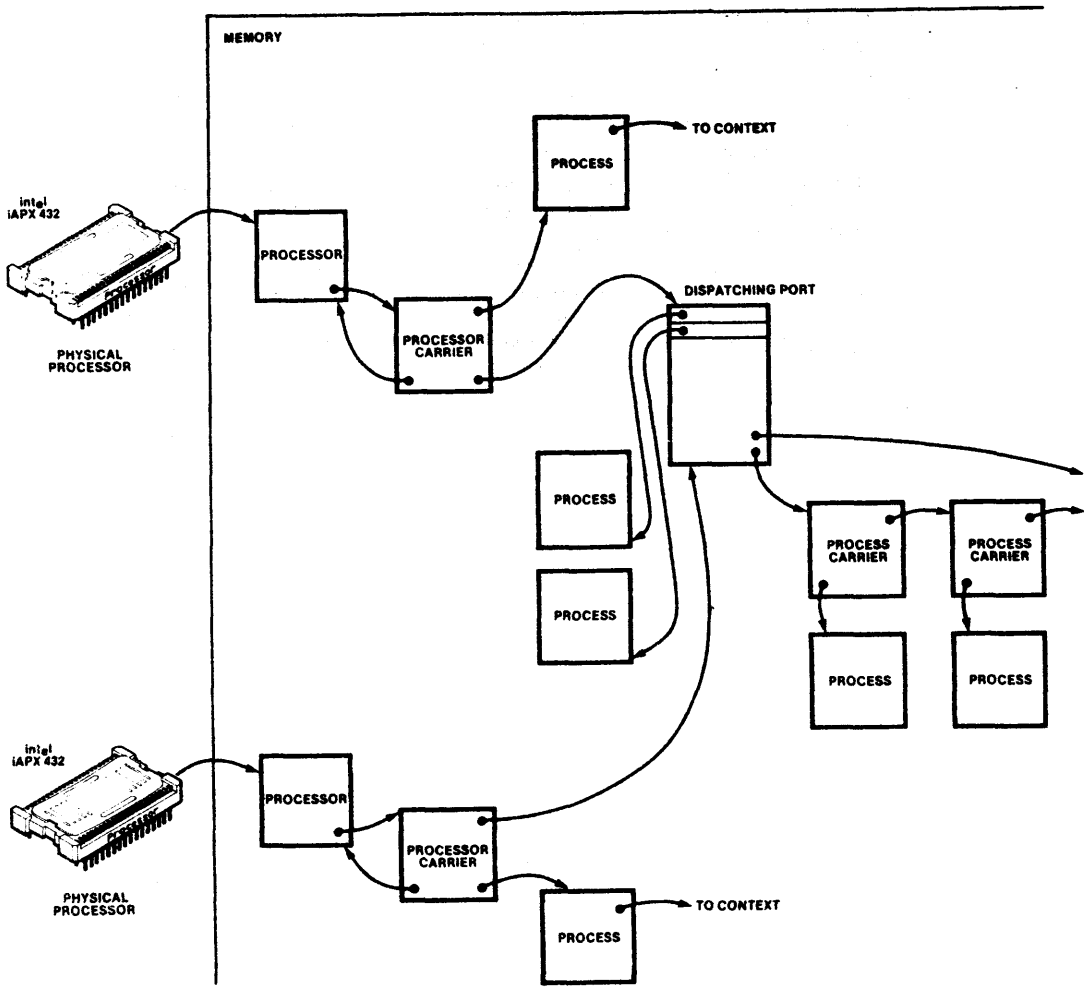


Figure 4-13. Multiple Processor System 171821-39

4.6 Summary

- The object-oriented design methodology has great promise for solving many of the current problems in software design and maintenance.
- This methodology is based on information hiding as the criterion for modularization.
- Type managers are the modules that result when the methodology is actually applied. They contain all procedures that directly manipulate objects of a given type.
- Domain objects and context objects are used to implement the static structure and dynamic behavior of type managers. The type definition object is used to give hardware protection to user objects.
- Multi-process operation can be supported by the object-based architecture of the iAPX 432, which also provdies a uniform approach to all system services (the Silicon Operating System).
- The Silicon Operating System provides a full set of hardware recognized system objects that provide support for concurrent processing (process objects, processor objects, dispatching ports, carriers), interprocess communication (communication ports), and dynamic storage allocation (storage resource objects).

CONCLUSIONS

The goal of reliable software will remain distant until computer architecture is significantly modernized to narrow the semantic gap between language concepts and hardware concepts. Given the high annual toll of people-hours lost to unreliable software, we architects have something bordering on a moral responsibility to modernize.

-Peter J. Denning
Chairman of the Computer
Science Department,
Purdue University

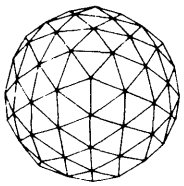
Objects, type managers, concurrent processes, and communication ports may be new to the world of computers, but these concepts are very familiar in the real world of mailboxes, locks, keys, things, and events. For too long programmers have been forced to think in concepts that are tied closely to the hardware but are distant from the world of applications. Programmers usually find that object-oriented design using modern languages, such as Ada, quickly becomes second nature, because it is so much like the anthropomorphic way we all think. These modern languages and methodologies can make programming more of a human activity, and thus capable of being performed correctly by human beings.

But the new languages and methodologies have to be supported by the architecture in order to work effectively. Unfortunately, most architectures deal with concepts that are as far below the level of Ada as Fortran is below the real world. Glenford Myers called this conceptual incompatibility between modern languages and conventional architectures the "semantic gap," a gap which the iAPX 432 architecture has bridged.

The conceptual gap between Ada and the iAPX 432 architecture is very small, because the designers of the iAPX 432 architecture followed the same object-oriented design methodology that users should follow when writing Ada programs. The same methodology is also used in iMAX, the Multifunction Applications Executive that Intel provides with Ada to handle the interface between user programs and the Silicon OS. The object methodology is thus found at every level of the iAPX 432 system.

DEVELOPMENT
SYSTEMS
- SERIES III
- NETWORK

intel



A guide to INTELLEC® SERIES III MICROCOMPUTER DEVELOPMENT SYSTEMS

121632-001

Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

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

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

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

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

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

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

BXP	intel	Megachassis
CREDIT	Intelevision	Micromap
i	Intellec	Multibus
ICE	iRMX	Multimodule
iCS	iSBC	PROMPT
im	iSBX	Promware
Insite	Library Manager	RMX/80
Intel	MCS	System 2000
		UPI
		μScope

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

PREFACE

This book is a welcome mat for the Intellec Series III Microcomputer Development System. It is also an introduction to the world of software development for micro-applications. We assume that you have some basic knowledge of microprocessors and their applications, but few demands are made on that knowledge. We do not assume that you have had exposure to any particular programming language. This book can be useful to beginners, and even sophisticated readers should find it rewarding to skim through.

This book is a tutorial for using the Series III system, especially the "8086 side" of it (the 8086 execution environment). The "8085 side" is similar to a Series II system, which is described in *A Guide to Intellec Microcomputer Development Systems* by Daniel McCracken.

We lead you through a typical software development process by providing an example of a micro-application: a climate control system for a building. To keep the example easy to understand, we only describe the software development effort, assuming that the hardware for the climate system is being developed simultaneously. In fact, we illustrate some typical problems in software development that occur as a result of changing hardware designs.

Chapter 1 gives an overall view of the Series III system and the application example. It also describes top-down design, stepwise refinement, modular programming, design considerations, and how to choose the proper software language for each module.

Chapter 2 is a step-by-step tutorial on the Series III operating system, showing typical operations.

Chapter 3 is a step-by-step tutorial on CREDIT, and it incidently shows the process of stepwise refinement of the application's main control algorithm.

Chapter 4 describes Pascal-86 programming, structured and modular design, parameter passing, data typing, and the Pascal-86 compiler.

Chapter 5 describes PL/M-86 programming, and it shows a sample PL/M-86 routine used in the application. It also briefly describes the PL/M-86 compiler and the 8086/8087/8088 Macro Assembler.

Chapter 7 describes program debugging with DEBUG-86 and hardware emulation with the ICE-88 emulator.

There is also a bibliography of related material, and a list of Intel manuals supplied with the Intellec Series III Microcomputer Development System.

CONTENTS

	Page
CHAPTER 1: THE SOFTWARE DEVELOPMENT PROCESS	1
Defining the Product's Software	3
Choosing the Software Development Tools	5
Languages	5
Modular Programming	6
Debugging and In-Circuit Emulation	7
Using Your Final Product	8
CHAPTER 2: OPERATING THE SERIES III SYSTEM	9
Turning On Your System	9
The Directory Listing	10
Formatting Disks	13
Hard Disk Subsystem Users	13
Flexible Disk Users	14
Filenames, Pathnames, and File Attributes	15
Renaming and Deleting Files	18
Copying Files to Disks and Devices	19
Executing Commands and Programs	22
Summary of the Series III Operating System	25
CHAPTER 3: TEXT EDITING	27
Creating a Text File and Inserting Text	28
Moving Around in the Text File	32
Finding Old Text and Substituting New Text	33
Macros and Command Iteration	36
Ending a Text Editing Session and Managing Backup Files	38
Displaying and Printing Text Files	39
From Text to Program	39
CHAPTER 4: PROGRAMMING IN PASCAL-86	41
Translating Pidgin Pascal to Pascal-86	41
Pascal-86 Data Types	45
Another Look at Modularizing and Hiding Information	46
Passing Data to Other Modules—Parameter Passing Techniques	48
The Interface Specification	49
Test Version of the Climate Control System	49
The Pascal-86 Compiler	55
Summary	61
CHAPTER 5: PROGRAMMING IN OTHER LANGUAGES	63
Another Look at Choosing Languages for Modules	63
Programming in PL/M-86	64
Programming in 8086/8087/8088 Assembly Language	69
Programming For the Series III Environment	70

CONTENTS (Cont'd.)

PAGE

CHAPTER 6: USING UTILITIES TO PREPARE EXECUTABLE PROGRAMS 75
Preparing a Library of Program Modules 76
Linking Modules to Form a Locatable Program 77
Locating and Running Programs 78

CHAPTER 7: DEBUGGING AND EXECUTING PROGRAMS 81
Using DEBUG-86 For Symbolic Debugging 82
Using ICE-88, an In-Circuit Emulator 92
Execution Environments 95

BIBLIOGRAPHY 97

INDEX 99

THE INTELLEC® SERIES III PUBLICATIONS LIBRARY

ILLUSTRATIONS

FIGURE	TITLE	PAGE
1-1	Developing Software on the Series III System	2
1-2	Block Diagram of Our Climate Control System	3
1-3	Nassi-Schneiderman Chart for Our Climate Control Software	4
3-1	The CREDIT Video Display	28
3-2	The Series III Keyboard	29
4-1	Algorithm for the Climate Control Main Module	42
4-2	First Try at Coding the Main Program	43
4-3	Second Try at Coding the Main Program	47
4-4	The Interface Specification	50
4-5	Test Version of Our Climate Control System	51
4-6	Lisings of Our Test Modules	57
5-1	The PL/M-86 Typed Procedure THERMOSTAT$SETTING$FROM$PORTS	64
5-2	The PL/M-86 Typed Procedures TEMP$DATA$FROM$PORTS and INTERPOLATE	66
5-3	Listing of PLMDATA with the CODE Control	71
6-1	Using Utilities to Prepare Executable Programs	75
6-2	Main Module with Subordinate Modules	77
7-1	Climate Control Program Listing and Sample Run	85
7-2	Listing of the Modified PLMDATA Module	92
7-3	Possible Execution Paths for Pascal-86 Programs	96

CHAPTER 1
THE SOFTWARE DEVELOPMENT PROCESS

"Hardware is computing *potential*; it must be harnessed and driven by software to be useful."
–Andrew S. Grove, President of Intel Corp.

The Intellec Series III Microcomputer Development System is more than a keyboard, a video display, an integral disk drive, and a box with two microprocessors. It is a useful tool for designing microcomputer software for the iAPX 86,88 processor family or for the 8080/8085 processors. You can choose the appropriate language (PL/M, FORTRAN, Pascal, macro-assembly language) for each piece of software, debug these pieces separately, and link them in different ways for different applications. The applications can then be run on this system or any other system that is based on the iAPX 86,88 or 8080/8085 families of processors.

Intel's iAPX microprocessor family provides an architecture best suited for modular software development using high-level languages. The Intellec Series III Microcomputer Development System takes full advantage of this architecture to provide a more cost effective programming environment that guarantees a shorter development cycle.

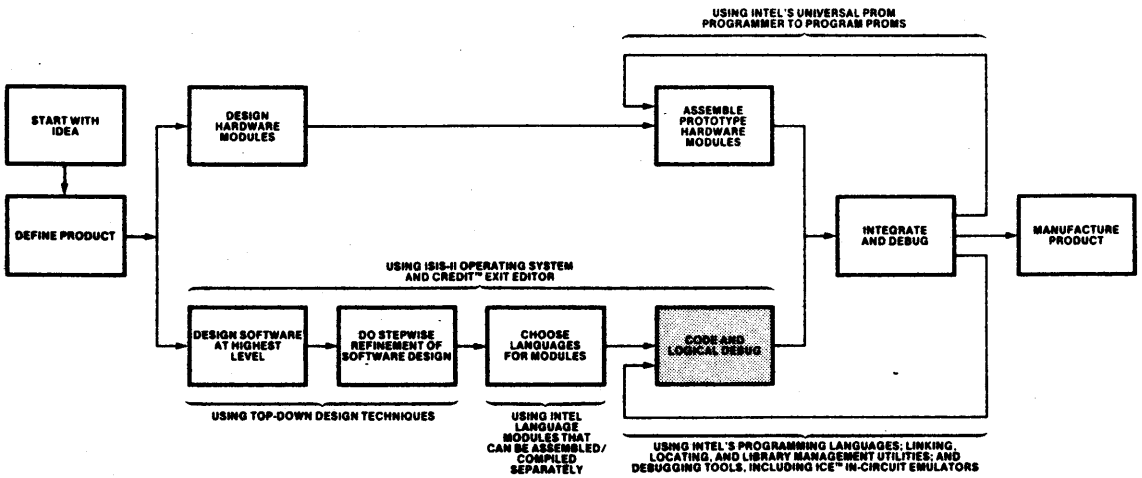
To design a product that will contain a microprocessor, you must coordinate two efforts: the design of the hardware that surrounds the microprocessor, and the design of the software that controls the microprocessor. Hardware development involves planning the interaction of the microprocessor, the associated memory and peripheral circuits, and the specialized input/output circuits and processors. Software development involves programming the microprocessor with instructions that will eventually be stored in the product's memory. These instructions must be designed to correctly perform the required tasks.

It is possible to carry out these development efforts independently--the hardware development separate from the software development. In practice, however, it takes a long time to develop error-free software on prototype hardware. To achieve good system integration and to save time, software debugging must usually begin long before prototype hardware is available to test the software.

The Intellec Series III with in-circuit emulation (ICE) is a development solution because it provides support for parallel hardware and software development efforts. Using the ICE-86 or ICE-88 emulator, you can emulate parts of your prototype hardware in order to test your software in a stable environment that resembles your final product. The ICE-86 or ICE-88 emulator also allows you to substitute memory and other resources from the Series III system for the memory and resources missing in your prototype hardware. With the ICE-86 or ICE-88 emulator, prototype hardware can be added to your product as you are designing it, and software and hardware testing can occur simultaneously (thereby speeding up the entire development process).

The chart in figure 1-1 summarizes a software development process, starting with an idea for a final product. Such a process always starts with an idea, which you refine step by step until you can define the actual product's environment, hardware, and logic.

To provide a tutorial on using the Intellec Series III system, and to show how Intel's software development tools are used in a development situation, we provide a simple software application for the iAPX 88 microsystem, at the heart of which is an 8088 microprocessor. The application is a climate control system for a building that uses a solar collector for heating and cooling, with backup methods of heating and cooling when the solar collector is not adequate. All methods use water, storage tanks, and a water-to-air exchanger or heat pump.



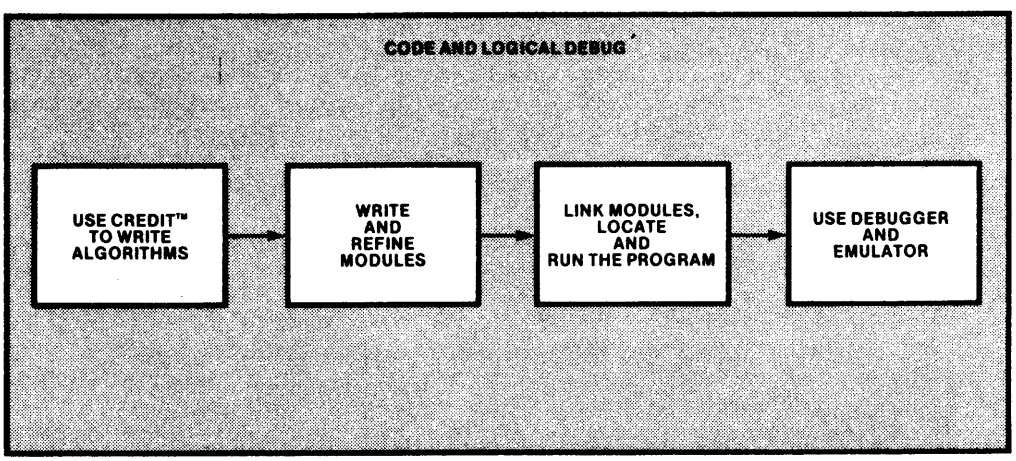


Figure 1-1. Developing Software on the Series III System 121632-1

Several decisions about this climate control system can be deferred to a later date. For example, by designing the system's software in a modular fashion, we can add more methods of heating or cooling as necessary, and we can decide how to handle water pumps and valves at a later date. We know now that the software's primary purpose is to choose a method of heating or cooling based on temperature data, and to operate the climate system's pumps and valves. Figure 1-2 is a simplified block diagram of the application.

DEFINING THE PRODUCT'S SOFTWARE

As you define your product's hardware, you must also define the *purpose* of its software. For example, the purpose of the software for our application is to gather and store the appropriate temperature data, decide on a heating or cooling method based on that data, implement the method in the climate system, and maintain the operation of the climate system. Each task can be designed as a piece of software called a *module*. By keeping tasks modular, you can change the details of any task without affecting the details of the other tasks.

Keep in mind that software provides the capability to change or add to the product. The entire product could consist of hardware and logic circuits, but then you might have to rebuild each unit to add more capabilities or change the flow of the logic. Software that is not modular and easy to maintain does not solve this problem; therefore, you must define the entire purpose of the software, with an eye to the future of your product. Keep it modular so that you can replace modules easily without rewriting modules that already work.

For our climate control system, we defined the software to be a set of modules that receive and store data, decide heating or cooling methods based on that data, and decide how to operate the hardware associated with the climate system.

At this time, we do not need a more detailed definition; in fact, more detail would hinder our process of step by step refinement. It is important to realize the order for these actions: first, the software has to start up the climate system. Once started, the software has to do several things over and over (unless the system shuts down): (1) read the various temperatures, (2) store the data for future reference, (3) decide on a heating or cooling method to use, and (4) operate the climate system to provide heating or cooling and to maintain the system (e.g., maintain heat gain).

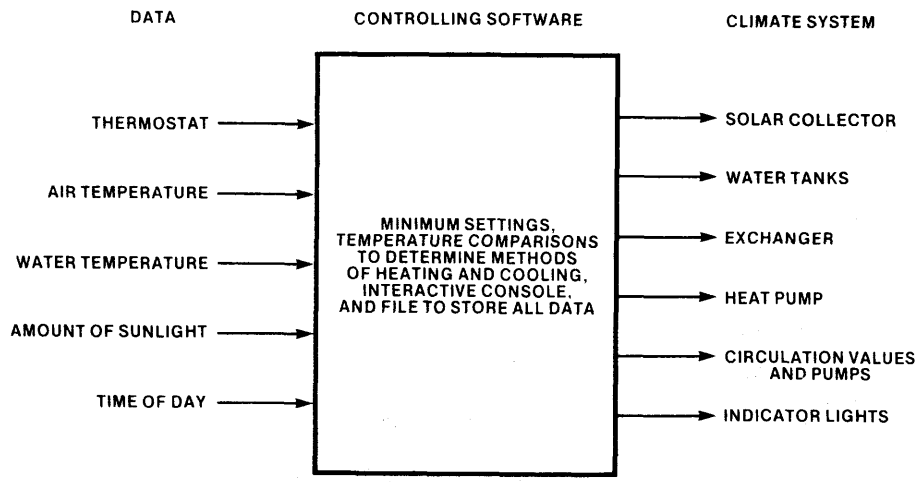
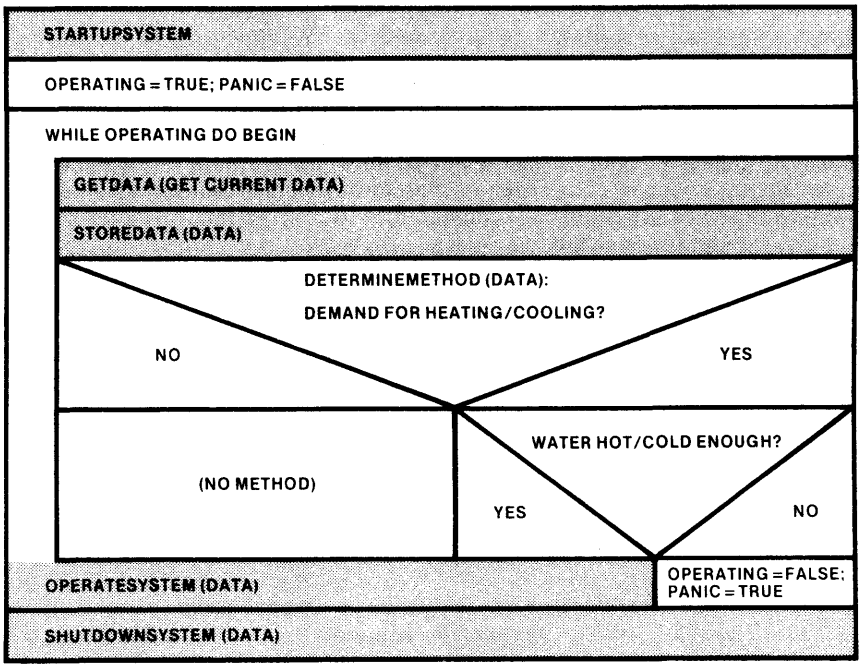


Figure 1-2. Block Diagram of Our Climate Control System 121632-2

A good software definition breaks the problem into solvable tasks. The order in which these tasks must occur also determines the *structure* of the software. If the structure is simple to understand, it will be that much easier to implement and maintain.

A *Nassi-Shneiderman Chart* (shown in figure 1-3) is useful for showing structured blocks of software. Use whatever charts you find useful, but hide many of the details so that you are not locked into doing things a certain way. In our application, we hid all details about data types, input and output, and actual heating or cooling methods. It is most important that we design each module to be self-reliant; that is, a module should not have to know about details hidden in another module, especially details that might change in the future.



OPERATION MODULE

MAIN MODULE

GETDATA MODULE

Heating Methods

Collector To Exchanger
Tank To Exchanger
Collector To Heat Pump
Tank To Heat Pump
Heated Tank To Heat Pump
No Method (No heating demand)

Data

InsideTemp
ThermostatSetting
CollectorWaterTemp
TankWaterTemp
HeatedTankTemp

AmountOfSun (for collector operation)
Hour, Minute (for collector and immersion heater operation)

Figure 1-3. Nassi-Schneiderman Chart for Our Climate Control Software 121632-3

CHOOSING THE SOFTWARE DEVELOPMENT TOOLS

It is unfortunate but common in this industry to find software development systems sorely lacking in the tools of the trade. There are some systems that force you to put together a whole program in the limited space they allow, and they don't provide the facility to create a library of canned routines that you could use with many different programs.

The Intellec Series III system provides both the ability to put together partial programs, and the facilities to build libraries of routines that you can link to different programs. The concept of *module* is inherent in this system. Whenever you build a partial program, it is a module; and the module can refer to procedures, functions, and variables in other modules found in libraries.

For our application, we do not have to decide on one programming language—the Intellec Series III system supports several, including the high-level languages PL/M-86, Pascal-86, and FORTRAN-86. We can, however, decide whether or not to use modules that already exist, and design our application with that decision, or change the decision later and create another module to replace it.

For example, we already have a module written in PL/M-86 that performs a routine to convert thermocouple voltage into degrees Celsius. We can decide now to tentatively use it, thereby saving time by not coding this routine into our main program. We can also decide, at a later time, *not* to use it, and substitute our own module to do it. The decision to link which module does not have to be made until the main program is finished!

By choosing the right tools you can save time and defer decisions on specific details until you are ready to deal with the details. By deferring such decisions, you keep your software development effort from becoming too cluttered with rigid design decisions, and you keep the effort flexible enough to accept change.

The right tools are (1) appropriate high-level languages to choose from, (2) a way to manage libraries of canned routines, (3) a linker that allows you to link finished (or unfinished) modules in different ways for different applications, (4) a locater that will locate programs in memory for you, yet give you the opportunity to specify locations for sections of the program, (5) a symbolic debugger you can use to test modules and partial programs easily, and (6) an in-circuit emulator to emulate parts of your final product before they exist. The Intellec Series III Microcomputer Development System supports all of these tools.

LANGUAGES

When designing a system using top-down techniques, you think in and express concepts in the highest-level language possible at each refinement step in order to expose the logical concepts and conceal the details.

How can you tell whether you are thinking in a language that is "high-level"? A language is "high-level" for a given application if you use it to define the overall structure of the software. You use a lower-level language to express in great detail each piece of code. With a high-level language you gain a clear understanding of the control structures of the system at the highest level, and you expose logical flaws in the structure that would have led to subtle bugs in the code. With these advantages in mind, it makes sense to start your programming with the highest-level language: English. After you have defined the software in English, you can use a high-level programming language like Pascal, which allows you to express more detailed code in a language that resembles English.

In Chapter 3, we refine our climate control algorithm step by step. We use as a language something called Pidgin Pascal, which is really a language of concise declarative English sentences. Since our control structures can be translated easily from English into Pascal, we decided to use Pascal-86 for our main climate control module (this decision does not have to be final). We do not start the translation into Pascal-86 until after we have tested the logic of our Pidgin Pascal algorithm.

Pascal is a language that resembles the control structures of human thought. We don't think in terms of GO TO branches normally; we consider a job to be a set of tasks to DO WHILE something is true, or to DO UNTIL something is done. IF something is true or false, THEN do one thing; ELSE do another thing. In the CASE of several different problems, solve each one accordingly. Occasionally we might need a disaster bail-out (GO TO a panic routine), but we should be planning our algorithm to take care of disasters elegantly.

The point is: we should think about control structures of a system as structures, not as individual branch statements. Pascal is one language that was designed to express control structures; PL/M is another, and some new versions of FORTRAN can be structured accordingly.

Another point should now be obvious: you should choose the language best suited for the algorithm. For example, our module that operates the climate system has to manipulate pumps and valves to implement a chosen heating or cooling method. This operation module receives the data from the decision-making main module written in Pascal. The operation module might use bits in a word as control signals to send to the procedure that actually interfaces to the hardware of the climate system.

We can code this operation module in assembly language or PL/M, since both languages can easily manipulate bits in a byte or word and respond with appropriate actions based on the pattern of bits. Our final decision will be made later; in the meantime, we will use a test version of the operation module, until prototype hardware for the climate system is ready. Our test version will be written in Pascal, and it will simply display appropriate test information at the console.

It is most likely that the final version of the operation module will not be written in Pascal, since Pascal does not provide bit manipulation operations. We can also guess that the module will not be written in FORTRAN--the advantage FORTRAN has over PL/M is its ability to express complex mathematical formulas. Our climate control system has no complex mathematical formulas.

We would code the final version of the operation module in assembly language if the application required the most efficient use of processor time and memory; however, PL/M programming saves development and maintenance time since it is easier to learn and the programs are easier to read and maintain. The only drawback to PL/M is that it is not as efficient in time or memory, and it cannot compute decimal or real arithmetic. These are not drawbacks in our application, since we do not need decimal or real arithmetic, nor do we have severe speed and memory constraints. The advantage of saving development time and maintenance costs with PL/M far outweigh the advantages of using assembly language.

MODULAR PROGRAMMING

The Intellec Series III system has the linking and locating tools to support modular programming, and the library utility to maintain libraries of canned routines. The decisions we make at this time are not binding, but they can be very helpful: we can decide now what types

of routines will be separated into which modules. We have several design criteria for making these decisions. We must be able to (1) write the routines of one module with little knowledge of the code in other modules, and (2) reassemble and replace any module without affecting other modules. Each module of our climate control system will contain design decisions that are hidden from the other modules so that the decisions are not binding.

With these criteria in mind, we designed our system to have several modules: the **GetData** module gets and stores our data, the **Operation** module performs the actual climate system operation (turning pumps and valves on and off, etc.), and the **Main** module makes the high-level decisions.

The only binding decisions to be made at this time concern the data passed between modules. We try to keep data passing to a minimum, or we try to enforce a data-passing standard that is easy to comply with. In our climate system, we only need to pass a *reference* to a data record to the other modules; the other modules must know what to do with the reference. This data-passing technique is one of the two we can use: pass-by-reference and pass-by-value, which are described in more detail in Chapter 4.

With the Intellec Series III system capabilities, we can design different versions of the same module, test each version, and decide at a later time which version to link with the other modules. We can also defer decisions about the physical memory locations (locations in our final product's memory) for these modules until after we have debugged our prototype hardware with an In-Circuit Emulator (ICE-86 or ICE-88). In some applications, you never have to decide physical memory locations; the Intellec Series III locater can decide them for you.

DEBUGGING AND IN-CIRCUIT EMULATION

At any time during software development, you can test a compiled module using DEBUG-86. In certain cases you will want to alter the module to be self-contained; for example, our main module needs the appropriate data from the **GetData** module, but the **GetData** module is not yet written. We can quickly write a procedure that obtains the data from an interactive session at the console, link this temporary data acquisition module to our main module, and test our main module using DEBUG-86.

When we have our modules coded and compiled, we can test the modules in an ICE-86 or ICE-88 session that can emulate the final product's processor. We can use an ICE-86 emulator if our final product will contain an 8086 processor, or use an ICE-88 emulator if it will contain an 8088 processor (remember, our iAPX 86, 88 applications software can run on either an 8086 or an 8088).

For example, our data module will read data from ports of an 8088 processor in our final product. The ICE-88 emulator can emulate those ports before we ever have a prototype of the final product. Using ICE-88, you can begin testing your software before any prototype hardware exists. As portions of your prototype become available, you can use them and still borrow resources like memory from your Series III system.

With in-circuit emulation, you control, interrogate, revise, and completely debug your product in its own environment, or in a stable environment that emulates the final product's environment. Symbolic debugging is one of the key features of Intellec microcomputer development systems and in-circuit emulators. Symbolic debugging allows you to debug your program using its own symbols and line numbers—you do not have to convert your symbols to physical memory addresses.

USING YOUR FINAL PRODUCT

Intel supplies other tools to help you put together your final product. If you designed your product to have its software in ROM (read-only memory), you can use Intel's Universal PROM Programmer (UPP) with its Universal PROM Mapper (UPM) software to create the PROM (programmable read-only memory) device to hold the software. You can then install your device in an SDK-86 or SDK-88 (System Design Kit with an 8086 or 8088 processor), or in an iSBC (Single Board Computer) system.

You can also run your software in other systems, or in dedicated application environments. For example, you could transfer your software to RAM (random-access memory) on an SDK-86 or SDK-88, or to RAM on an iSBC 86/12A (Single Board Computer system with an 86/12A board), by first using the OH86 utility described in the *iAPX 86,88 Family Utilities User's Guide for 8086-Based Development Systems* to convert the program to hexadecimal object format. You would then use an appropriate tool to load the software into your execution board (the ICE-86 In-Circuit Emulator, the SDK-C86 Software and Cable Interface, or the iSBC 957 Interface and Execution Package).

You can also use your Series III development system as the environment for your final software product. The Series III system has an 8086 processor (the "8086 side" or 8086 execution environment) to run iAPX 86, 88 applications software. As soon as you have debugged your software, it is ready to be used in any Series III system. Chapter 6 describes the run-time libraries you use to run your Pascal-86 programs in the Series III environment; you can also supply your own versions of these libraries to run these programs in another execution environment. Chapter 6 also shows how you can link modules and locate them in Series III memory in one easy step to prepare them for execution in the Series III system.

Whatever environment you choose for your final product, Intel provides the appropriate hardware and software tools to develop, debug, and produce your final product. The Intel environments that are suitable for final software products are excellent investments which can be upgraded for the software of the future.

CHAPTER 2
OPERATING THE SERIES III SYSTEM

"The benefits of using a standardized operating system should prove to be as significant as the benefits of using standardized microcomputer hardware. Development and programming costs will be reduced substantially, and you will have an upward compatible interface for future products."

—Andrew S. Grove, President of Intel Corp.

As you learned in Chapter 1, the Intellec Series III Microcomputer Development System contains the tools you need to develop software for your application. To use these tools, you must operate the system; that is, use the system's commands and utility programs.

A large multi-user computer system can serve as a useful analogy to point out the difference between *programming* the system and *using* the system. A working system usually supports both kinds of activities. In such a large system, one or more programmers might be designing programs to run on the system. One or more users (who might also be programmers) might be simply *using* the programs that have already been developed for the system. Obviously, a user does not need to know complicated details about the system to use it, whereas a programmer needs to know such details to write programs for the system.

The Series III system has most of the capabilities of larger systems, but it is only used by one person at a time. This person could be using the system and its programs, or writing programs for the system.

We wrote this chapter for people who need to know how to use the system. This information is important to anyone using or programming the system, but it is not burdened with details that a first-time user does not need. As we describe system commands and utility programs, keep in mind that we swept the more complicated details under the rug to keep the chapter easy to read. For more details on all of the system commands, see the *Intellec Series III Microcomputer Development System Console Operating Instructions*.

TURNING ON YOUR SYSTEM

Every microcomputer has something called an operating system, which is sometimes called a supervisor or monitor. It is usually the first piece of software you use—the software that is controlling your computer when you first turn it on, and the software that responds to your first typed command. Typically, you type a command to ask the computer for a list of the files you have on disk, or to execute a particular program.

When you turn on the Intellec Series III Microcomputer Development System, the following message appears on your screen:

```
SERIES II MONITOR, Vx.y
```

This message tells you that the monitor is up and running (the *x* and *y* represent version numbers). The *monitor* is a piece of software that gives you direct control of the Series III hardware. Although you can use the monitor for debugging and other operations, for your purposes the monitor performs one quick activity: it loads the operating system from the system disk in drive 0 into the computer when you push the RESET button.

To load or "boot" the system, you need a *system disk*: a disk that holds the operating system files. Intel supplies a flexible disk labeled "ISIS-II Operating System" which is your system disk. One of the first operations you will perform is a *formatting* operation to create another copy of the system disk—a copy designed to be used with the examples in this book.

You insert the "ISIS-II Operating System" disk into flexible disk drive 0, and push the RESET button (for more detailed instructions, see the *Intellec Series III Microcomputer Development System Console Operating Instructions*). The following message should appear on your screen:

```
ISIS-II, Vx.y
```

The operating system for the Intellec Series III Microcomputer Development System is called ISIS-II ("ISIS" stands for Intel Systems Implementation Supervisor). The *x* and *y* represent version numbers. ISIS-II is actually a version of the ISIS operating system that runs on older Intellec systems (the Intellec Microcomputer Development System and the Intellec Series II Microcomputer Development System).

ISIS-II manages your use of the software tools supplied with the system. Using ISIS-II, you can run utility programs like CREDIT (a text editor) to write programs, or LINK86 to link program modules into a final program. You also use ISIS-II to run compilers like the Pascal-86 and PL/M-86 compilers, and to run your own developed programs. You can also make copies of programs and control the devices attached to your system by using ISIS-II commands.

In most computer operations, you manipulate *files*, which are collections of information. Every file has a name, called a *filename*. Files and filenames are described in detail after an introduction to the most popular operating system command—the DIR command.

You use the DIR command to display a directory of filenames on a certain disk. You use other commands to perform file management operations. For example, you use the COPY command to make copies of files, or to send copies of files to the printer to be printed. You use the RENAME command to change a file's filename. This chapter explains some of these commands.

THE DIRECTORY LISTING

The first command to learn is the easiest to use—the DIR command. When your system is up and running, you will see a dash (-) on the left side of the screen. This dash is called a *prompt*, and it tells you that the system is ready for your next command. You can now type the DIR command by typing the letters "DIR", and execute the DIR command by pressing the RETURN key. The following example is shown in blue to show that it is something *you* type; the part of the example that is in ordinary black shows what the system displays. The symbol "<cr>" means that you must press the RETURN key ("cr" stands for the "carriage return" key found on most typewriters).

```
-DIR<cr>

DIRECTORY OF :F0:970003.06
NAME   .EXT  BLKS   LENGTH ATTR     NAME   .EXT  BLKS   LENGTH ATTR
SYSTEM.LIB     24     2849 WS       FPAL   .LIB    74     9125 W
PLM80 .LIB     45     5615 W
                              143
1317/4004 BLOCKS USED
```

When you execute the DIR command, the system displays a list of *filenames* called a *directory listing*. These filenames are the names of the files that are stored on the disk in drive 0. The DIR command displays the directory for the disk in drive 0 unless you specify another drive number with the command, as explained in the next paragraph. Disk drive 0 is usually the drive that holds the system disk (the disk that contains the system files), so by executing the DIR command by itself, you get a directory listing of some of the files on the system disk in drive 0 (not all, because some files are invisible).

You can use the DIR command to display the directory listing for disks in other disk drives, and for a hard disk subsystem, by specifying the drive number with the DIR command. For example, to see a directory listing for a disk in drive 1, insert the "CREDIT ISIS-II CRT-Based Text Editor" disk in drive 1, and type the following command:

```
-DIR 1<cr>

DIRECTORY OF :F1:970049.02
NAME   .EXT  BLKS    LENGTH ATTR       NAME   .EXT  BLKS    LENGTH ATTR
CREDIT        156     19470            CREDIT.HLP     25      2985 W
ADDS   .MAC     3       171 W          MICROB.MAC      3       158 W
VT52   .MAC     3       163 W          VT100 .MAC      3       190 W
1510T  .MAC     3       165 W          1510E .MAC      3       170 W
LEAR   .MAC     2       123
                                201
310/4004 BLOCKS USED
```

The *Intellec Series III Microcomputer Development System Console Operating Instructions* give details about typical hard disk and flexible disk configurations, and the associated drive numbers.

On each line of the directory listing, you are shown the name of a file. Some of these filenames are followed by a three-character *extension*, which is an optional identifier used to show types of files. These extensions are sometimes required for certain files that are used with certain programs. The section on filenames (following disk formatting) explains some of these extensions.

Following each filename entry are three columns labeled "BLKS", "LENGTH", and "ATTR". The "BLKS" column tells you how many "blocks" of space the file occupies, where one block equals 128 bytes. The "LENGTH" column gives the actual number of bytes occupied by the file. At the bottom of the listing appears a message showing how many blocks are used out of the total number of blocks available on the disk. You can use block figures to describe file sizes and to determine whether a file of a given size (in blocks) will fit on a disk.

The "ATTR" column might be empty, or it may contain a "W", "S", or both. This column tells you the *attributes* of each file—certain characteristics that govern the file's use. If a file has the "W" attribute, it is *write-protected*; that is, you cannot write to or delete (overwrite) the file. If a

file has the "S" attribute, it is a *system* file that occurs on most system disks. You use the ATTRIB command and file attributes to protect your files from inadvertant delete or write operations, and to designate certain files as system files (as shown in the next section).

There are other files not displayed through normal use of the DIR command. These files are *invisible*; that is, they have the I attribute. You can see them if you use a special form of the DIR command:

```
-DIR I<cr>

DIRECTORY OF :F0:970003.06
NAME   .EXT  BLKS   LENGTH ATTR      NAME   .EXT  BLKS   LENGTH ATTR
ISIS   .DIR    26     3200   IF      ISIS   .MAP     5      512   IF
ISIS   .TO     24     2944   IF      ISIS   .LAB    54     6784   IF
ISIS   .BIN    94    11740  SIF      ISIS   .CLI    20     2407  SIF
ATTRIB         40     4909  WSI      COPY           69     8489  WSI
DELETE         39     4824  WSI      DIR            55     6815  WSI
EDIT           58     7240  WSI      FIXMAP         52     6498  WSI
HDCOPY         48     5994  WSI      HEXOBJ         34     4133  WSI
IDISK          63     7895  WSI      FORMAT         62     7794  WSI
LIB            82    10227  WSI      LINK          105    13074  WSI
LINK   .OVL    37     4578  WSI      LOCATE        120    15021  WSI
OBJHEX         28     3337  WSI      RENAME         20     2346  WSI
SUBMIT         39     4821  WSI      SYSTEM.LIB     24     2849  WS
FPAL   .LIB    74     9125  W        PLM80  .LIB    45     5615  W
                                  1317
1317/4004 BLOCKS USED
```

The "I" is called a *switch*, and it displays files that have the invisible attribute. When you specify the "I" switch in a DIR command, the DIR command displays all of the filenames, including ones that are invisible. The files that weren't displayed during the execution of a normal DIR command are now displayed, along with their attributes (one of which is the "I" attribute). The "F" attribute is reserved for system files that are used to format the disk. These files are called *format files*, and you should never alter their attributes.

You should practice using the DIR command, and at the same time, take a look at the directory listings for each disk you received. Take each flexible disk, insert it into drive 1, and type "DIR 1" to see the directory listing.

NOTE

If you have a hard disk subsystem with only one flexible disk drive, use the following form of the DIR command:

```
-DIR P<cr>
LOAD SOURCE DISK, THEN TYPE (CR)
```

Take out the system disk and insert the disk whose directory you want to display, then press the RETURN key. After the directory listing, the following message appears:

```
LOAD SYSTEM DISK, THEN TYPE (CR)
```

Put the system disk back into the flexible drive. To see other disk directory listings, repeat these steps for each disk.

FORMATTING DISKS

We present in this section a typical scheme for storing copies of your Intel-supplied files on either hard disk or flexible disk. Subsequent examples in this book assume that certain programs reside in hard disk drives 0 and 1, or flexible disk drives 0 and 1. You may use your own scheme and distribute files over disk drives as you wish, but in order to type the examples as they are, you must use the scheme presented here. If you understand the use of pathnames as described in this chapter, and if you use the COPY command correctly, you can copy files to disks in any distribution scheme you choose, and still use the examples in this book as long as you substitute your own pathnames.

If you are using a hard disk subsystem, you must follow the procedures in the *Intellec Series III Microcomputer Development System Console Operating Instructions* to install your disk platters and prepare them for use. If you are using flexible disk drives, you should refer to the same manual for instructions on the care and insertion of flexible disks. This section shows you how to prepare one hard disk platter or one flexible disk as the system disk, and another hard disk or flexible disk as a non-system disk (to hold your files and other programs).

Hard Disk Subsystem Users

Follow the instructions in the *Intellec Series III Microcomputer Development System Console Operating Instructions* to install and power-up your hard disk subsystem. When the hard disk subsystem is ready, insert your "ISIS-II Operating System" flexible disk into the flexible disk drive of your computer, and hit the RESET button. When the dash (-) prompt is displayed (after the ISIS-II message appears), type the following command:

```
-:F4:FORMAT :F0:SYSTEM.HDK S FROM 4<cr>
```

This command prepares ("formats") drive 0 of the hard disk to be the system disk. We chose "SYSTEM.HDK" for its name, but you can use any name that has at most six characters to the right of the period, and three to the left. When this operation is finished, and the system displays the dash (-) prompt, use the following command to transfer your system files to drive 0 of the hard disk:

```
-:F4:COPY :F4:*.* TO :F0:<cr>
```

This command will copy all of the files from the "ISIS-II Operating System" disk in the flexible disk drive (called drive 4) to the hard disk drive 0. When this operation is finished, you can remove the flexible disk from drive 4 and insert another flexible disk. In addition to the "ISIS-II Operating System" disk, you should also copy files from the "CREDIT ISIS-II CRT-Based Text Editor" disk, the "Resident 8086/8087/8088 Macro Assembler" disk, and the "Resident 8086/8088 Utilities and Linkage Libraries" disk. For each flexible disk, execute the following command:

```
-COPY :F4:*.* TO :F0:<cr>
```

This command copies all files from the flexible disk to drive 0 of the hard disk. If you insert each flexible disk you received and perform this operation, you will have in drive 0 all of the files from those flexible disks. If you want to be more selective about the files you are copying to drive 0, read the "Copying Files" section in this chapter.

You should also copy files to drive 1 of the hard disk subsystem. First, you must prepare drive 1 by typing the following command:

```
-FORMAT :F1:PROG86.HDK<cr>
```

We chose "PROG86.HDK" for its name, because we intend to use it for our iAPX 86,88 application programs. If you are a Pascal-86 user, insert the "Pascal-86 Compiler and Run-Time Libraries" disk into flexible disk drive 4 and copy all of the files to the hard disk drive 1:

```
-COPY :F4:*.* TO :F1:<cr>
```

Do the same operation with your PL/M-86 flexible disk, and any other disks you have for your Series III system. The hard disk platters have plenty of room for your own files.

Flexible Disk Users

This section assumes that you have at least two double-density flexible disk drives. If you have single-density drives, you may run out of room if you try these examples. Refer to instructions in the *Intellec Series III Microcomputer Development System Console Operating Instructions* for information about flexible disks.

To bring up your system, insert the "ISIS-II Operating System" disk into drive 0 and push the RESET button. The ISIS-II message should appear, followed by the dash (-) prompt. Insert a blank disk into drive 1, and type the following FORMAT command:

```
-FORMAT :F1:SYSTEM.FLX S<cr>
```

This command creates a new system disk and automatically copies from drive 0 all files that have the "S" attribute. When this operation is finished, you can test your new system disk by removing the "ISIS-II Operating System" disk from drive 0, inserting your new system disk into drive 0, and pushing the RESET button to restart the system.

With your new system disk in drive 0, insert the "CREDIT ISIS-II CRT-Based Text Editor" disk into drive 1, and type the following command:

```
-COPY :F1:CREDIT TO :F0:<cr>
```

This command copies the program CREDIT to our system disk in drive 0. Remove the CREDIT disk from drive 1 and insert the "Resident 8086/8087/8088 Macro Assembler" disk into drive 1. Type the following command:

```
-COPY :F1:RUN.* TO :F0:<cr>
:F1:RUN COPIED TO :F0:RUN
:F1:RUN.OV0 COPIED TO :F0:RUN.OV0
```

This command copied two files at once—RUN and RUN.OV0—from drive 1 into drive 0. You need both files in order to use the special RUN command described later.

Remove the flexible disk from drive 1 and replace it with the "Resident 8086/8088 Utilities and Linkage Libraries" disk. Now type the following commands:

```
-COPY :F1:LIB86.86 TO :F0:<cr>
:F1:LIB86.86 COPIED TO :F0:LIB86.86
-COPY :F1:LOC86.86 TO :F0:<cr>
:F1:LOC86.86 COPIED TO :F0:LOC86.86
-COPY :F1:LINK86.86 TO :F0:<cr>
:F1:LINK86.86 COPIED TO :F0:LINK86.86
```

You now have a system disk that is complete for the examples in this book. You still need another disk for the Pascal-86 compiler if you have one, to hold the compiler, the run-time libraries, and your sample program used in examples in this book. To prepare a blank disk to be a non-system disk, insert the blank disk into drive 1 and type the following command:

```
-IDISK :F1:PASC86.FLX<cr>
NON-SYSTEM DISK
```

The IDISK command can prepare both system and non-system disks, but it does not automatically copy files except the ISIS-II format ("F") files.

With the new blank disk in drive 1 and the system disk in drive 0, you could put the "Pascal-86" disk in drive 2 (if you have a drive 2). The following example assumes that you only have drive 0 and 1. To copy the files from the "Pascal-86" disk, first type the following command (don't forget the "P"!):

```
-COPY *.* TO :F1: P<cr>
LOAD SOURCE DISK, THEN TYPE (CR)
```

The system pauses (because you specified a "P" at the end of the command), and waits for you to insert the "source" disk into drive 0. The "source" disk in this case is the "Pascal-86" disk, assuming you are a Pascal-86 user (if you're using PL/M-86 only, you would substitute your PL/M-86 disk in this example). Insert this flexible disk into drive 0 and press the RETURN key.

```
LOAD OUTPUT DISK, THEN TYPE (CR)
```

Since the disk to receive the Pascal-86 files is already in drive 1, you only have to press RETURN. The system should then display the above messages (along with the "COPIED" messages) every time it returns to drive 0 to copy more files—all you have to do is continue to press RETURN until the entire copy operation is finished.

You can repeat these steps with another blank disk for the PL/M-86 compiler. After these formatting and copying operations, you should have disks that match the ones we used for the examples in this book.

FILENAMES, PATHNAMES, AND FILE ATTRIBUTES

Most operating system commands manipulate files. A file can be any collection of information including text, numeric data, program instructions, and combinations of all of these. You refer to a file by using a filename, and filenames follow certain naming conventions so that the name of a file also tells you the probable use of the file.

Filenames can have six or fewer characters, followed by an optional period and three-character extension. Extensions (and filenames themselves) follow certain conventions, but there are no fixed rules. Certain extensions are necessary for certain programs, as you shall see in the following discussion. The file naming conventions are relaxed so that you have the flexibility to create your own file naming conventions for your particular application.

The term filename refers to both the name of the file and the extension, if any. Each new file you create must have a unique name for that disk (that is, you can have two files with the same name on two different disks, but not on the same disk). Some extensions have specific meanings to utility programs in the system, but these should not cramp the style with which you impose your own naming conventions. Here are the extensions that mean something to Intel-supplied programs:

- The ".BAK" extension denotes a backup of a text file created by CREDIT, the text editor described in Chapter 3. For example, LETTER.BAK is the backup file for the text file LETTER.TXT (a text file does not need a particular extension, but "TXT" helps identify it).
- The ".MAC" extension denotes a "macro" file used with certain programs like CREDIT. The ".MAC" files supplied with CREDIT enable you to use CREDIT on other non-Intel consoles connected to Intel microcomputers.
- The ".OV0" extension denotes an "overlay" file used with certain programs.
- The ".OBJ" extension denotes an *object module*, which is created by a compiler or assembler, as described in Chapters 4 and 5. Compilers and assemblers also create files with the ".LST" extension to denote program listings (also described in Chapters 4 and 5). Examples are PROG1.OBJ and PROG1.LST.
- The ".LNK" extension denotes a collection of object modules that were linked together by the linker utility program, described in Chapter 6. When you run the locater utility program on a file with the ".LNK" extension, the locater strips the ".LNK" extension away, leaving only the file's name without an extension.
- The ".LIB" extension denotes a library module that is maintained by the library utility. Library modules are described in Chapter 6.
- The ".86" extension denotes a program that should be executed in the 8086 execution mode (the "8086 side" of the system). Examples are PASC86.86 and PROG1.86.

Most completed programs have either the ".86" extension (if they run in the 8086 execution mode, described later in this chapter), or no extension. For example, the DIR command is actually a program named "DIR" without any extension.

In addition to the above extensions, we use the ".SRC" extension to denote a special text file that holds program instructions called source statements. A ".SRC" (for "source") file can be created by CREDIT and filled with assembly language, PL/M, or Pascal source statements; you can then compile this file to create an object module (".OBJ" file), as described in Chapters 4 and 5.

You can find more extensions explained in the *Intellec Series III Microcomputer Development System Console Operating Instructions*.

You can display the directory information for a single file by using a version of the DIR command, DIR FOR:

```
-DIR FOR SUBMIT<cr>
DIRECTORY OF SYSTEM.FLX
NAME   .EXT  BLKS     LENGTH   ATTR
SUBMIT         39       4821
```

The file SUBMIT is in the directory for drive 0, so the DIR command had no trouble finding it. However, if SUBMIT happened to be on another disk in drive 1, you would have to tell DIR to look in drive 1 for the file. To do this, you use a *pathname*—a filename with a directory specifier. Whenever you specify a filename for a command or utility program, if the filename is in the directory for drive 0, you only have to specify the filename. If the filename is *not* in the directory for drive 0, you have to specify a pathname.

A pathname consists of a directory specifier and a filename:

```
:F1:PROG1.SRC
```

The ":F1:" part of the pathname is the directory specifier. The "1" stands for disk drive 1. Therefore, to get the directory information for the file PROG1.SRC on the disk in drive 1, type the following command:

```
-DIR FOR :F1:PROG1.SRC<cr>
```

The number of separate disk drives depends on the configuration of your system. The format for directory specifiers is:

```
:Fn:
```

where *n* can be any drive number from 0 to 9 (0 would refer to drive 0, which does not have to be specified).

If you type a directory specifier that is incorrect, you get an error message. For example, suppose you typed the following DIR command with an incorrect directory specifier in the pathname for PROG1.SRC:

```
-DIR FOR :G1:PROG1.SRC<cr>
:G1:PROG1.SRC, UNRECOGNIZED DEVICE NAME
```

The "UNRECOGNIZED DEVICE NAME" is the ":G1:" directory specifier. ISIS-II thinks you are referring to a device by that name, and there is no device by that name. You will find legal device names in the section that describes copying files.

Another common error occurs when you forget to use the word FOR in the DIR command when you are trying to get the directory information for one file. For example, you might type this command:

```
-DIR :F1:PROG1.SRC<cr>
```

and get this error message:

```
:F1:PROG1.SRC, UNRECOGNIZED SWITCH
```

Since you forgot the word FOR, ISIS-II thinks you are trying to specify a switch to the DIR command—and it doesn't recognize ":F1:PROG1.SRC" as a switch! Switches are usually only one letter or number. Valid switches are described with the DIR command in the *Intellec Series III Microcomputer Development System Console Operating Instructions*.

The ATTRIB command assigns attributes to files. You use file attributes to protect your files from accidental deletions or modifications. You use the ATTRIB command to turn on or off the file attributes that are described below:

- "W" for write-protected (The file cannot be written to, modified, renamed, or deleted unless this attribute is turned off.)
- "I" for invisible (The file's name and information is only displayed when you execute the DIR command with the "I" switch.)
- "S" for system files (System files on a source disk are automatically copied to the output disk in a FORMAT operation with the "S" switch.)
- "F" for format files (Format files are automatically copied by the FORMAT and IDISK commands to format disks. *Do not* turn off this attribute, nor use it with your own files, unless you've read its description in the *Intellec Series III Microcomputer Development System Console Operating Instructions*.)

For example, to turn on the "W" (write-protected) attribute for file CREDIT, type the following command:

```
-ATTRIB CREDIT W1<cr>
    FILE                CURRENT ATTRIBUTES
:F0:CREDIT                      W
```

To turn off the "W" attribute, specify a 0 instead of a 1:

```
-ATTRIB CREDIT W0<cr>
    FILE                CURRENT ATTRIBUTES
:F0:CREDIT
```

RENAMING AND DELETING FILES

Occasionally you will have a file with a name that needs to be changed for some reason. For example, you could rename LIB86.86 to LIBRY.86 by using the RENAME command:

```
-RENAME LIB86.86 TO LIBRY.86<cr>
```

The RENAME command always expects to see the original name first, followed by a space, then the keyword TO, followed by another space, and finally the new name. Both names must be legal Series III filenames or pathnames. You cannot RENAME a file in one directory to a name with a different directory specifier—the new name must have the same directory specifier.

If you actually renamed LIB86.86 to LIBRY.86, you should rename it back to LIB86.86, the name used for it in future examples.

The DELETE command is very simple. Since there are no files in your system disk that you can delete, we'll use in this example a file that doesn't actually exist on your disk:

```
-DELETE SAMPLE.TXT<cr>
:F0:SAMPLE.TXT DELETED
```

The DELETE command will irrevocably delete the file you specify. When you DELETE a file, you cannot retrieve it—it's gone. Use DELETE with caution.

To delete a file on a disk in a drive other than drive 1, simply use the appropriate pathname for the file. For example, suppose the file JUNK.TXT existed on the disk in drive 1. You would use this command to delete it:

```
-DELETE :F1:JUNK.TXT<cr>
:F1:JUNK.TXT DELETED
```

You can also delete many files at once. As you will see in Chapter 3, CREDIT creates a backup file with a ".BAK" extension for every file you edit. At some point (to utilize disk space), you might want to delete all of your backup files at once. For example, to delete all of the backup files on the disk in drive 1, use the following command:

```
-DELETE :F1:*.BAK<cr>
```

The asterisk (*) matches any name, and the ".BAK" extension matches only filenames with that extension. This "filename matching" technique is shown in more examples to come.

COPYING FILES TO DISKS AND DEVICES

The COPY command is useful for several operations:

- To make a copy of a file
- To send a copy of a file to another hard or flexible disk
- To send a copy of a file to a device like a printer or a paper tape punch
- To receive a file from a sending device like a paper tape reader
- To copy all non-system files from one disk to another

The COPY command is similar to the RENAME command. For example:

```
-COPY :F1:PROG1.SRC TO :F1:PROG1.BAK<cr>
COPIED :F1:PROG1.SRC TO :F1:PROG1.BAK
```

The COPY command expects to see the name of the *source* file or device (the source of the information to be copied, not to be confused with a *source file* of program source statements), followed by a space, then the keyword TO, followed by a space, and then the name of the *output* file or device (the destination of the copied information). After the output (or destination) file or device name, you can optionally specify a *switch* like "P" for pause, or "U" for update (these switches are explained in the *Intellec Series III Microcomputer Development System Console Operating Instructions*).

More frequent uses of the COPY command are copying to devices, copying files onto other disks, and copying all the files in one disk to another disk. In all of these cases, you must specify the device or disk that is the source of the information, and the device or disk that will receive the copied information. For example, the RENAME program is in the directory for drive 0. If you want to put a copy of it on the disk in drive 1, you would type the following command:

```
-COPY RENAME TO :F1:<cr>
COPIED :F0:RENAME TO :F1:RENAME
```

When the source file is in the directory for drive 0, you don't have to specify :F0: for the file, because :F0: is the default directory if none is specified. When the destination is a disk in another drive, you have to specify the directory (in this case, :F1:) for the destination file.

In the example above, the file RENAME was copied to the disk in drive 1, and the drive 1 version has the same name. When you make copies of files for other disks, you will probably want the files on the other disks to have the same name. To keep the same name for the copy, specify only the destination directory without a new filename, as we did in the above example.

To change the name for the copy, specify a different name with the destination directory. For example, if you want to use RNAM as the name of the new copy in drive 1, use this command:

```
-COPY RENAME TO :F1:RNAM<cr>
:F0:RENAME COPIED TO :F1:RNAM
```

In the last two examples, the RENAME file was copied from drive 0 to drive 1. To demonstrate another feature of the COPY command, we will first copy a file from drive 1 to drive 0 (you should also try this example):

```
-COPY :F1:PROG1.SRC TO :F0:<cr>
:F1:PROG1.SRC COPIED TO :F0:PROG1.SRC
```

Let's suppose that we used CREDIT (described in the next chapter) to modify the new copy of PROG1.SRC on the disk in drive 0 (:F0:PROG1.SRC), and that we want to copy the newly-modified PROG1.SRC to the disk in drive 1. Since we already have a PROG1.SRC on the disk in drive 1, we might want to DELETE that one first, then COPY the newly-modified one to the disk in drive 1. However, let's suppose that we forgot to DELETE the old one first, or that we don't even know the old one exists on the disk in drive 1. We would type the following COPY command:

```
-COPY PROG1.SRC TO :F1:<cr>
:F1:PROG1.SRC FILE ALREADY EXISTS
DELETE?
```

The COPY command found :F1:PROG1.SRC, and now it is asking us if we want to delete it in order to replace it with the newly-modified :F0:PROG1.SRC. We type a "Y" (or "y") to delete the old one and replace it with the new one, or type an "N" (or "n" or any other letter) to keep the old one and abort the copy operation. In this case, we want to replace the old :F1:PROG1.SRC with the newly-modified :F0:PROG1.SRC, so we type a "Y" followed by the RETURN key:

```
DELETE?Y<cr>
:F0:PROG1.SRC COPIED TO :F1:PROG1.SRC
```

By using this feature of the COPY command, you can selectively update existing files by typing "Y" (or "y") for the ones you want to update, and "N" (or "n") for the ones you don't want to update.

To send a file to a device like a line printer or a paper tape punch, specify the device name as the destination device:

```
-COPY PROG1.SRC TO :LP:<cr>
:F0:PROG1.SRC COPIED TO :LP:
-COPY PROG1.SRC TO :HP:<cr>
:F0:PROG1.SRC COPIED TO :HP:
```

The device name :LP: is the name of the line printer. The second example sends the file to the high-speed paper tape punch, whose device name is :HP:. For other device names, see the *Intellec Series III Microcomputer Development System Console Operating Instructions*.

The COPY command can also be used to copy several files at once, if you make use of *wild card filenames*. You can specify a wild card filename with the DELETE, RENAME, COPY, DIR, HDCOPY, and ATTRIB commands. A wild card filename matches a group of filenames in order to perform the action on several files at once. For example:

```
-COPY *.* TO :F1:<cr>
```

This command copies all of the files in directory :F0: to directory :F1:.

NOTE

To perform this example, you should insert a new disk into drive 1 and use the IDISK command (described in the next section in more detail) to prepare the new disk in drive 1 before copying all of the files in drive 0 to it.

The above example demonstrates use of the wild card filename "*.*". The first asterisk will match any number of characters in the name, and the second asterisk will match any three characters in the extension of the filename.

You can also specify some of the characters of a filename in a wild card filename. For example, if you wanted to copy all of the files that have ".SRC" as an extension, you would use this COPY command:

```
-COPY *.SRC TO :F1:<cr>
```

There are other wild card filenames that are described in detail in the *Intellec Series III Microcomputer Development System Console Operating Instructions*.

There are several ways to copy entire disks. The IDISK and FORMAT commands are used to prepare ("format") disks for use in the Series III system. The IDISK command will simply format the disk as a system or non-system disk, copying over to the new disk only the format ("F") files that are needed to format the new disk as a system or non-system disk.

The FORMAT command will format a new disk as a system or non-system disk depending on the source disk implied (drive 0) or specified (using FROM). The FORMAT command will also copy certain files from the source disk. FORMAT with the "S" switch copies all files from the source drive that have the "S" (system) attribute. See the previous section on formatting disks for an example of the FORMAT command with the "S" switch.

To copy *all* of the files from the source disk, use the "A" switch. To show an example of the FORMAT command with the "A" switch, insert another blank disk into drive 1, and type this command:

```
-FORMAT :F1:SYSTEM.BAK A<cr>
SYSTEM DISK
.
.
.
```

In this example, the FORMAT command formats the disk in drive 1 as a system disk because the source disk in drive 0 is a system disk. The new disk is called SYSTEM.BAK. The "A" switch specified that *all* files from the source disk in drive 0 should be copied to SYSTEM.BAK in drive 1.

So far, all of the FORMAT examples used the disk in drive 0 as the source disk. If, for example, you have more disk drives and you want to FORMAT a disk in drive 1 using as a source a disk in drive 2, you would specify FROM 2 as in the following example:

```
-FORMAT :F1:MYDISK A FROM 2<cr>
```

In a previous section on formatting disks, we described formatting operations to prepare your disks in a way that conforms with our examples. We do not necessarily recommend this formatting scheme for your particular configuration. Chapter 2 of the *Intellec Series III Microcomputer Development System Console Operating Instructions* gives you the specific details for setting up disks in any Series III configuration, using the IDISK, FORMAT, and HDCOPY commands.

EXECUTING COMMANDS AND PROGRAMS

The Series III has two processors: the 8085 (8-bit processor), and the 8086 (16-bit processor). You can execute 8085-based programs in the 8085 environment (which is called the *8085 execution mode* or "8085 side"), and execute 8086-based (or 8088-based) programs in the 8086 environment (which is called the *8086 execution mode* or "8086 side"). When you type the filename of a program as a command (as you have been doing when you type "DIR" or "COPY"), you are executing the program in the 8085 execution mode. In order to execute a program or command in the 8086 execution mode, you must prefix the command RUN to the program or command you wish to execute. For example, if you want to execute LIB86.86, the 8086 librarian (which can only be run in the 8086 execution mode), you would type the following:

```
-RUN LIB86<cr>
```

You should notice two things that are different: the use of the command RUN, and the fact that you do not have to supply the ".86" extension for LIB86.86 when you RUN it.

The RUN command is actually a program supplied on the system disk that activates the 8086 execution mode. When you supply a filename with no extension, the RUN command automatically attaches the ".86" extension to the name you supplied, and looks for the file by that name (i.e., the name with the ".86" extension). This is a protection feature you can use for your 8085 and 8086 programs: you can use the same name for both, with an ".86" extension for the 8086 program and no extension for the 8085 program. When you specify filenames without extensions, RUN only looks for files that are *supposed* to run in the 8086 execution mode; i.e., files with the ".86" extension. This protection feature assumes that you would put the ".86" extension on files that are meant to run in the 8086 execution mode.

If you actually executed LIB86.86, the following would appear on your screen:

```
-RUN LIB86<cr>
SERIES III 8086 LIBRARIAN, Vx.y
*
```

The LIB86.86 program is now in control. To leave this program and return to the operating system, use the librarian's EXIT command:

```
*EXIT<cr>
-
```

You can, of course, supply an extension with the filename you supply to the RUN command, and the RUN command would not supply the ".86" extension. For example, if you have an 8086 program called MYPROG.PRG and you want to run it in the 8086 execution mode (on the "8086 side"), you would type the following:

```
-RUN MYPROG.PRG<cr>
```

You can also RUN a program whose filename has no extension. However, you must supply a period to show that there should be no extension (i.e., to keep RUN from supplying the ".86" extension). The following is an example, assuming that the name of the program is not MYPROG.PRG but is MYPROG:

```
-RUN MYPROG.<cr>
```

When you use RUN as a command (as in the examples above), the 8086 execution mode is turned on for program execution, and when the program terminates, the 8086 execution mode also terminates, returning you to the 8085 execution mode. The 8085 execution mode is controlled by ISIS-II, which signifies its control by displaying the dash (-) prompt. You can, however, use RUN as a program and *stay in the 8086 execution mode*.

To use RUN as a program, simply type RUN:

```
-RUN<cr>
ISIS-II RUN 8086, Vx.y
>
```

The RUN program displays a sign-on message and its own prompt, the right angle (>) bracket, to signify that the RUN program is now in control. This mode is called the *interactive 8086 mode*, and you use it to execute more than one program. When you get the angle (>) prompt, you can type the filename of a program in order to execute the program. The rule stated above about the use of the ".86" extension still applies.

The following example shows the execution of several 8086-based programs: PASC86.86 (the Pascal-86 compiler), LINK86.86 (the linker and binder), and PROG1.86 (the assembled, linked, and bound program). If your Pascal-86 disk (containing the compiler, run-time libraries, and sample PROG1.SRC) is in drive 1, and LINK86.86 is on the disk in drive 0, you can type this example exactly as presented. The LINK86.86 command line shows use of the "&" continuation character that allows you to type commands over several lines without executing them. The EXIT command turns off the 8086 execution mode and returns you to ISIS-II and the 8085 execution mode:

```
-RUN<cr>
ISIS-II RUN 8086, V1.0
>:F1:PASC86 :F1:PROG1.SRC<cr>

.                       This executes PASC86.86 to compile
.                       PROG1.SRC, both of which are on the
.                       disk in drive 1. The Pascal-86
                        compiler displays some information.

>LINK86 :F1:PROG1.OBJ,:F1:P86RN0.LIB,:F1:P86RN1.LIB,&<cr>
>>:F1:P86RN2.LIB,:F1:P86RN3.LIB,:F1:E8087.LIB,&<cr>
>>:F1:E8087,:F1:LARGE.LIB TO :F1:PROG1.86 BIND<cr>

.                       This example shows use of the ''&''
.                       command line continuation character.
.                       The entire command links the object
                        program with the run-time libraries,
                        and binds the linked module to form
                        an executable program called
                        PROG1.86
```

```
>:F1:PROG1<cr>              This command executes :F1:PROG1.86

Farenheit temperature is:72<cr>
.
.                          PROG1.86 is a temperature conversion
.                          program. When it asks for more
                           input, you can stop the program by
                           typing ''N''.

Another temperature input?   :N<cr>

>EXIT<cr>                  This terminates the 8086 execution
                           mode and returns you to the 8085
                           execution mode.
```

In the above examples, commands and programs were executed directly by typing the command name or program name (sometimes followed by the name of a file to be acted upon by the command or program; this is called an *actual parameter*). When you execute a program or command directly, it is called *interactive execution*.

In many cases you will want to submit a batch of commands or programs to be executed in a sequence. The SUBMIT command allows you to submit a file of commands as a job to be handled by the system without any interaction on your part. This is called *non-interactive execution*.

You use SUBMIT by first providing a file of commands to be executed—a *command sequence definition file*, which has the extension ".CSD". This file can contain operating system commands, parameters for the commands, and *comments*—any line starting with a semicolon (;), or any text following a semicolon, is a comment, not an executable command. In the following example, note that the first command is the RUN command, which starts the 8086 execution mode, and that blank spaces in command lines are allowed to improve readability:

```
;
; CSD file to SUBMIT for linking a Pascal-86 object module
; to run-time support libraries (with E8087 emulator)
;
;      Parameter 0 = :fn:myprog
;      Parameter 1 = drive containing run-time libraries
;
;

RUN
LINK86 %0.OBJ, &
       :F%1:P86RN0.LIB, &
       :F%1:P86RN1.LIB, &
       :F%1:P86RN2.LIB, &
       :F%1:P86RN3.LIB, &
       :F%1:E8087.LIB, &
       :F%1:E8087, &
       :F%1:LARGE.LIB TO %0.86 BIND

EXIT

;
; Returns to 8085 execution mode.
;
```

When this file, called LNKBND.CSD, is submitted, the system will execute the RUN command, then the LINK86.86 program (the 8086 linker), and finally the EXIT command to leave the "8086 side" and return to the 8085 execution mode.

The percent symbols (%) indicate the use of *formal parameters*. When you execute the SUBMIT command, you supply *actual parameters* for each of these formal parameters. The following is an example:

```
-SUBMIT LNKBND(:F1:PROG1,1)<cr>
```

The SUBMIT command first looks for LNKBND.CSD (it supplies the missing ".CSD" extension), and then it substitutes the first actual parameter (:F1:PROG1) for "%0" and the second actual parameter (1) for "%1". The resulting command sequence is as follows:

```
RUN
LINK86 :F1:PROG1.OBJ, &
       :F1:P86RN0.LIB, &
       :F1:P86RN1.LIB, &
       :F1:P86RN2.LIB, &
       :F1:P86RN3.LIB, &
       :F1:E8087.LIB, &
       :F1:E8087, &
       :F1:LARGE.LIB TO :F1:PROG1.86 BIND

EXIT
```

The SUBMIT program creates another file with a ".CS" extension to hold the resulting command sequence. You should not modify this file.

SUMMARY OF THE SERIES III OPERATING SYSTEM

The Series III system has two execution environments: the 8085 environment (the "8085 side") where the ISIS-II commands and 8080/8085 programs run, and the 8086 environment (the "8086 side"), activated by the RUN program, where 8086 and 8088 programs run. Programs that run in the "8086 side" usually have an ".86" extension in their filenames. The RUN program actually looks for an ".86" extension if you specify a filename without an extension.

You use the Series III operating system and its tools to develop software, but there is another way to use it: your software products can use it as an execution vehicle. The *Intellec Series III Programmer's Reference Manual* describes the "innards" of the Series III operating system and how your programs can make use of sections of the system. By using PL/M or assembly language, you can call operating system procedures directly (as described in the summary of Chapter 5) from your program. Pascal-86 programs automatically use the operating system procedures by using run-time libraries which interface between the Pascal-86 programs and the Series III system.

The Series III system has a standard set of procedures that your software products can use; by using this standard programming interface, you can be sure that your future programs will remain compatible with present and future Intel operating systems.

There are a few more commands that were not introduced because they are used infrequently, or they are easy to use. These commands are adequately introduced in the *Intellec Series III Microcomputer Development System Console Operating Instructions*. You should consult this manual anyway, to be aware of details not mentioned in this tutorial.

CHAPTER 3
TEXT EDITING

"A good workman is known by his tools." —proverb

You use a text editing program to create any kind of document, including a document that consists of program instructions. CREDIT is a text editor that takes advantage of the capabilities you have with CRT screens to:

- Display and scroll text on the screen
- Rewrite text by typing new letters over old ones
- Rearrange lines of text and insert new lines of text between old lines
- Move to any position in the text file or to any point on the screen instantly
- Correct typing mistakes easily as you type

To simplify everyday text editing operations, CREDIT also provides features that allow you to:

- Save both the newly edited version and the old unedited version of the same document
- Find any string of characters and substitute another string of characters automatically
- Copy sections of a document to use in another document
- Create macros to execute several commands at once, or to repeat sets of commands (command iteration)

CREDIT is useful for all documents; for example, we used CREDIT to write this book. CREDIT is more often used to write the Pascal statements, PL/M statements, and assembly language instructions that comprise program modules. Text files created by CREDIT are called documents if they are meant to be read by humans; and they are also called *source programs* if they consist of program statements (Pascal, PL/M, or FORTRAN statements, or assembly language instructions) that are read by compilers or assemblers in order to be compiled or assembled into working programs.

In the following examples, we show you how to create and edit a text file consisting of English sentences that you can translate (eventually) into Pascal-86 statements. As described in Chapter 1, this step of generating Pidgin Pascal (English sentences describing a Pascal program) is a very important one in program development, because the program logic expressed in English sentences closely resembles human thinking and is therefore easier to debug.

CREATING A TEXT FILE AND INSERTING TEXT

When you run CREDIT, you also specify the name of a file to be edited. If CREDIT can find the file you are specifying, it will open the file for editing. If CREDIT cannot find the file you are specifying, it will create a new file by that name.

For example, let's create a file called PIDGIN.TXT in the directory for drive 0. To create and edit PIDGIN.TXT, type the following command:

```
-CREDIT PIDGIN.TXT<cr>
```

CREDIT responds with:

```
ISIS-II CRT-BASED TEXT EDITOR V2.0
NEW FILE     1982 FREE DISK BLOCKS
```

The number of free disk blocks depends on the number of files on your disk and the size of your disk—you do not have to worry about it unless you get a "DISK FULL" warning. If you get such a warning, terminate your CREDIT session by typing "EX" (followed by RETURN), and specify another disk (like :F1:) for your text file.

The screen is partitioned into two areas, as shown in figure 3-1.

Notice the vertical line " | " in the text area? This symbol marks the end of the text in the file. Since the file is new and holds no text yet, this symbol appears at the beginning of the file. As you type text into the file, the symbol moves and continues to mark the end of the text.

The blinking cursor sits underneath this vertical line. You can immediately type text into the file:

```
just type!|
```

As you type, you can make changes to the text you already typed by moving the cursor back to the previous text and typing over it. Move the cursor by using the cursor control keys that surround the HOME key (do not use the HOME key yet!). If you inadvertantly typed the HOME key, hold down the CNTL key and type a V (Control-V) to return the cursor to the text body.

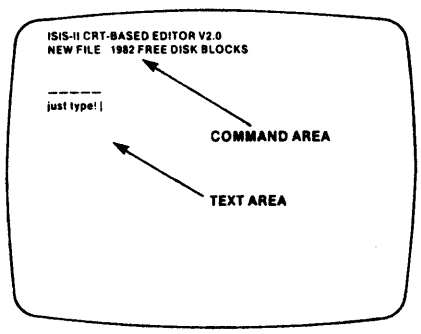


Figure 3-1. The CREDIT™ Video Display 121632-4

You end each line of text with a carriage return by typing the RETURN key, just like a typewriter. CREDIT displays the carriage return operation as an uparrow (↑). The carriage return operation is actually performed by two ASCII codes: one for the carriage return character, and one for the line feed character. CREDIT uses one symbol, the uparrow (↑), for both characters. This symbol is called the *line terminator*.

There are several keys used often in editing:

- The RPT (Repeat) repeats whatever key you hold down. For example, hold down both the RPT key and a cursor movement key, and the cursor will move faster. Use the RPT key with RUBOUT to erase a line.
- The TPWR (Typewriter) key, when in the down position, displays every character in lower case like a typewriter. When in the up position, all characters are in upper case.
- The ESC (Escape) key will cause a *break* in an executing command.

If your keyboard seems to freeze and does not display characters, you probably moved the cursor past the vertical line. You cannot type any characters after the vertical line, since the vertical line marks the end of the text. You can move the cursor to the vertical line and type, thereby moving the vertical line, or you can move the cursor to any position before the vertical line and type over the previously typed characters.

The *ISIS-II CREDIT (CRT-Based Text Editor) User's Guide* contains an interesting tutorial session. Before you learn how to use some of CREDIT's powerful editing commands, you should learn how to end a CREDIT session properly, without losing the edits you have made. Figure 3-1 shows the layout of the screen; in the Command Area you'll find an asterisk. To move the cursor to this asterisk, press the HOME key.

Figure 3-2. The Series III Keyboard 121632-5

When the cursor appears after the asterisk, you can type a CREDIT command. The EX (exit) command terminates a CREDIT session properly. Type "EX", followed by RETURN:

```
*EX<cr>
```

CREDIT will clear the screen and display the following message:

```
EDITED TO :F0:PIDGIN.TXT
```

By using the DIR command, you can see that the directory listing for :F0: now contains a new file called PIDGIN.TXT.

You can use CREDIT to edit PIDGIN.TXT by invoking CREDIT again:

```
-CREDIT PIDGIN.TXT<cr>
```

This time, CREDIT does not have to create the file, because it already exists in the directory for the disk in drive 0. CREDIT responds with:

```
ISIS-II CRT-BASED TEXT EDITOR V2.0
OLD FILE   SIZE=2       1980 FREE DISK BLOCKS
```

The text you typed during the last example appears under this message. The message tells you that PIDGIN.TXT is an old file, its size is 2 blocks (a block is 128 bytes), and you have 1980 free disk blocks in which to expand the file (these numbers vary depending on the space taken up by files on your disk). If you do not have enough free disk blocks to expand the file, CREDIT displays a warning message.

You can continue to experiment with CREDIT by typing and retyping lines of text. When you are finished typing extraneous characters and are ready to type meaningful English sentences, move the cursor to the first character or screen position, and type CNTL and Z (hold down the CNTL key and type Z). The @ symbol will replace the character. Now move the cursor to the end of the text—to the vertical line—and type CNTL and Z together again. ZAP! You just deleted all the text you typed. All of the characters between the two @ symbols were deleted, and you are left with an empty text file again.

Let's start with the simple problem definition. Type the following sentence:

```
Maintain the climate of a building using a system comprised of↑
heating and cooling methods.↑
↑
```

NOTE

The uparrow (↑) stands for the RETURN key. Use the RETURN key to end each line of text, including blank lines.

Using CREDIT, you can type the sentences you know you need, and insert more sentences later. For example, you know you need the following sentences:

```
Based on temperature data, see if there is a demand,↑
and determine the type of demand.↑
If there is no demand, simply continue operating the climate system.↑
If there is a demand for heat, determine the heating method,↑
     and operate the system with this method.↑
If there is a demand for cold, determine the cooling method,↑
     and operate the system with this method.↑
↑
```

Two questions should immediately come to mind: what information does the program need, and what else would a climate control program do? Let's add new sentences by using CREDIT's insert capability. Here is our text file so far:

```
Maintain the climate of a building using a system comprised of↑
heating and cooling methods.↑
↑
Based on temperature data, see if there is a demand,↑
and determine the type of demand.↑
If there is no demand, simply continue operating the climate system.↑
If there is a demand for heat, determine the heating method,↑
     and operate the system with this method.↑
If there is a demand for cold, determine the cooling method,↑
     and operate the system with this method.↑
↑
```

Move the cursor to the beginning of the fourth line of text (the line that begins with "Based on ..."). Hold down the CNTL key and type A (CNTL-A). The rest of the text disappears, and you are now able to type sentences. Type the following sentences and blank lines:

```
↑
Startup the climate system.↑
↑
While the system is operating, do (and repeat) the following:↑
↑
Get the data needed for each pass: the time, the temperatures,↑
the weather, the state of the solar collector, etc.↑
Store this data.↑
↑
```

Now type CNTL and A together again. The rest of the text file reappears with the new lines inserted in their proper places. By typing CNTL and A together, you turn on the Add Text Mode; by typing them again, you turn off Add Text mode. Here is your text file with the newly inserted lines:

```
Maintain the climate of a building using a system comprised of↑
heating and cooling methods.↑
↑
Startup the climate system.↑
↑
While the system is operating, do (and repeat) the following:↑
↑
Get the data needed for each pass: the time, the temperatures,↑
the weather, the state of the solar collector, etc.↑
Store this data.↑
↑
Based on temperature data, see if there is a demand,↑
and determine the type of demand.↑
If there is no demand, simply continue operating the climate system.↑
If there is a demand for heat, determine the heating method,↑
     and operate the system with this method.↑
If there is a demand for cold, determine the cooling method,↑
     and operate the system with this method.↑
↑
```

MOVING AROUND IN THE TEXT FILE

Using the cursor movement keys, you can move the cursor anywhere within a screenful of text. If the text file is larger than twenty lines, it will not fit entirely in one screen—you have to *scroll* the screen by using the scrolling commands.

The scrolling commands only operate when the cursor is in the text area (when the cursor is in the text area, you are in *screen mode*). Refer to figure 3-1 for an illustration of the text and command areas. When the cursor is in the command area (*command mode*), you can only execute command mode commands; to move into screen mode (and move the cursor to the text area), type CNTL-V (hold down the CNTL key and type V).

The scrolling commands are CNTL-N (to see the next screenful of text), and CNTL-P (to see the previous screenful of text). You can also use CNTL-V to see which character CREDIT is currently pointing to, and to move into screen mode. If you experiment with CNTL-V, you will notice that the first use of CNTL-V moves the cursor to the character that CREDIT's *pointer* is pointing to (more on this pointer in the next paragraph). Subsequent executions of CNTL-V rearrange the lines of text so that the line that contains the character pointed to by CREDIT becomes the third line in a screenful of text. This is very useful for partial scrolling. You can use these commands without fear, since they do not modify the text.

NOTE

The CNTL key, when used with another key (e.g., CNTL-V), is sometimes represented by the uparrow (↑) symbol in our manuals and pocket references. For example, CNTL-V is shown as ↑V, and CNTL-A is shown as ↑A.

The *character pointer* mentioned above is a reference point for all of CREDIT's commands. In screen mode, the cursor represents the pointer. It is sometimes called "the CP". Most text editors have some pointer or marker that points to a place in the file, and commands that insert characters, delete characters, or search for characters use this pointer to find the place in the file to perform their operations. CREDIT's pointer points to a single character, and this pointer moves whenever you move the cursor within the text area. When you type the HOME key and move the cursor to the command area, the pointer stays where it is, pointing to the character in the text area. The CNTL-V command moves the cursor back to the text area and back to the character pointed to.

In screen mode, the cursor movement keys and the scrolling commands move this pointer. In command mode, all editing changes are made relative to the position of this pointer. Deleting a single character, for example, erases the character pointed to by the pointer, and moves the pointer to the next character. When you insert text, the text is inserted preceding this pointer. Many other commands also move this pointer. CNTL-V will always move the cursor to the character pointed to by this pointer.

One command that always moves the pointer is the J (jump) command. You can only use the J command in command mode, and the J command leaves you in command mode (so you have to do another CNTL-V to get into screen mode). You can jump to any location relative to the pointer, or you can jump to specific locations called *tags*. You can set your own tags by using the TS (tag set) command, and delete tags by using the TD (tag delete) command. There are two permanent tags that need not be set, and that cannot be deleted: the beginning of the file, known as TT (tag for top), and the end of the file, known as TE (tag for end). For example:

```
*JTT<cr>
```

This command moves the pointer to the top of the text file (the beginning of the first line).

Since CREDIT usually puts the pointer at the top of the file when you start an editing session, the JTT command (jump to tag for top of file) is more useful during an edit session. The JTE command (jump to tag for end of file) is useful at any time:

```
*JTE<cr>
```

After using the J command, you are still left in command mode. Use the CNTL-V command to return to screen mode, and to return the cursor to the character pointed to by the CP.

Use the JTE command (jump to tag for end of file), followed by a CNTL-V command to return to screen mode, in order to add a sentence to the end of our Pidgin Pascal software definition. The following example shows the execution of the JTE command, followed by a CNTL-V (displayed as ↑V), followed by a display of the current text file with the added sentence at the end:

```
*JTE<cr>
*↑V


Maintain the climate of a building using a system comprised of↑
heating and cooling methods.↑
↑
Startup the climate system.↑
↑
While the system is operating, do (and repeat) the following:↑
↑
Get the data needed for each pass: the time, the temperatures,↑
the weather, the state of the solar collector, etc.↑
Store this data.↑
↑
Based on temperature data, see if there is a demand,↑
and determine the type of demand.↑
If there is no demand, simply continue operating the climate system.↑
If there is a demand for heat, determine the heating method,↑
     and operate the system with this method.↑
If there is a demand for cold, determine the cooling method,↑
     and operate the system with this method.↑
↑
If no method is possible (abnormal conditions),↑
shut down the climate system.↑
↑
```

FINDING OLD TEXT AND SUBSTITUTING NEW TEXT

There will be numerous occasions when you will want to find a specific word or group of words, and move the character pointer at the same time. There will also be times when you will want to substitute a new word or group of words for an old one. For example, you might write a program that displays the name of your product (for example, "ACME Solar Controller") in several different places. Sometime later, you find out that the marketing people changed the name to "ACME Climate Controller". With one simple CREDIT command, you could substitute the new word ("Climate") for the old word ("Solar") wherever the old word occurs in the text file. You would only have to make sure that you specified the old name using upper and lower case characters as they appear in the file, and that the new name looks exactly as it should look.

The F (Find) command finds any *string* (group of characters) you specify. The S (Substitute) command finds the old string you specify and substitutes the new string you specify, and rearranges the text so that spaces aren't introduced into the file. The SQ (Substitute after Query) command finds the old string you specify, then asks you for a yes-or-no answer: a yes tells CREDIT to substitute the new string you specified, and a no tells CREDIT not to substitute the new string. If you executed the SQ command iteratively (see next section), CREDIT would continue looking for more instances of the old string.

To show examples of these commands, we'll return to our Pidgin Pascal text file to add some new text and substitute a new word for an old one. Use the F command to find the string "simply continue":

```
*F/simply continue/<cr>
```

When the F command finishes, it displays the asterisk once again. To see where it put the character pointer, type CNTL-V. The cursor should be under the space after the last letter of "continue." We want to rewrite the sentence so that "no method" is one of the methods used to operate the climate system.

At this point, it is easy to move the cursor to the appropriate place to insert new text. You can type over the old text, and use the RETURN key to continue typing a line. You can also use the CNTL-A combination to insert a lot of text. After inserting the new text, the sentences should read as follows:

```
If there is no demand, choose ''no method'' as the method,↑
     and operate the system with this method.↑
If there is a demand for heat, determine the heating method,↑
     and operate the system with this method.↑
If there is a demand for cold, determine the cooling method,↑
     and operate the system with this method.↑
↑
If no method is possible (abnormal conditions),↑
shut down the climate system.↑
```

To illustrate use of the S and SQ commands, we will substitute the word "request" for "demand" throughout our text file (and thereby make our program more polite). First, use the JTT command to move the character pointer to the beginning of the file; then use the S command in the following manner:

```
*JTT<cr>
*S/demand/request/<cr>
```

The S command found the first instance of "demand" and substituted "request" for it. Check to see the result by typing CNTL-V:

```
Based on temperature data, see if there is a request,↑
and determine the type of demand.↑
```

Note that S command only found the first instance of "demand" and substituted "request" for it only. Note also that the S command must have moved the comma after the first "demand" in order to fit the word "request" in that place.

To execute the S or SQ commands repeatedly, you would use a form of command iteration. The SQ command performs the same operation as the S command if you answer with a yes; on a no answer, the SQ command does not substitute text. Here is an example of the SQ command, with a sneak preview of the easiest form of command iteration:

```
*!<SQ/demand/request/><cr>
```

The angle brackets around the entire SQ command, in conjunction with the exclamation point, cause the SQ command to be executed repeatedly until the command reaches the end of the text file. Do not be confused by the angle brackets surrounding the "cr"—that is the symbol depicting use of the RETURN key. Angle brackets delimit the command to be executed iteratively, and the exclamation point replaces a number that would specify the number of iterative executions. This is explained in more detail in the next section.

The SQ command displays the line that contains the old text, and then displays a question mark. You must respond with a "Y" for yes, or an "N" for no:

```
and determine the type of demand.↑
?Y
```

You do not have to type RETURN after typing the "Y" or "N", because CREDIT is expecting such an answer. The SQ command goes on to find more instances of "demand":

```
If there is no demand, choose ''no method'' as the method,↑
?Y
If there is a demand for heat, determine the heating method,↑
?Y
If there is a demand for cold, determine the cooling method,↑
?Y

*
```

When the SQ command has found the end of the text file, the condition for iterative execution is satisfied, and the execution ends. Type CNTL-V, followed by a CNTL-P, to see the previous page and the substitutions:

```
Maintain the climate of a building using a system comprised of↑
heating and cooling methods.↑
↑
Startup the climate system.↑
↑
While the system is operating, do (and repeat) the following:↑
↑
Get the data needed for each pass: the time, the temperatures,↑
the weather, the state of the solar collector, etc.↑
Store this data.↑
↑
Based on temperature data, see if there is a request,↑
and determine the type of request.↑
If there is no request, choose ''no method'' as the method,↑
     and operate the system with this method.↑
If there is a request for heat, determine the heating method,↑
     and operate the system with this method.↑
If there is a request for cold, determine the cooling method,↑
     and operate the system with this method.↑
↑
If no method is possible (abnormal conditions),↑
shut down the climate system.↑
↑
```

MACROS AND COMMAND ITERATION

Our Pidgin Pascal program still needs a modification that will vastly improve its readability. We need to indent all of the sentences that occur after the sentence "While the system is operating, do (and repeat) the following:". This will improve the readability, since it will then be obvious that the indented sentences are the actions that have to be repeated.

To make this improvement, we will define a macro to hold several editing commands that will be executed iteratively. Rather than explain the process of defining macros and the syntax for command iteration, we'll show you the steps to take to make this improvement to our text file, and then we'll explain them.

To begin, move the cursor (in screen mode, so that it also moves the character pointer) to the beginning of the blank line that follows the sentence "While the system is operating...". Your cursor (and the character pointer) should now be at the beginning of this blank line, as shown by the underscore "_" symbol:

```
Maintain the climate of a building using a system comprised of↑
heating and cooling methods.↑
↑
Startup the climate system.↑
↑
While the system is operating, do (and repeat) the following:↑
_
```

Now type the HOME key to return to command mode.

You can only define macros while in command mode. We will now define a macro called X that will make the improvement. Here is our definition of X:

```
*MSX@L0;%<L;I/□□□□□/;P>@<cr>
```

Type this macro definition exactly as it is shown above (the "□" symbol stands for a space—type five spaces—and the "<cr>" stands for the RETURN key). When CREDIT finishes digesting this definition, it displays the asterisk again. Now type the following command:

```
*MFX(13)<cr>
```

You will see each line following the sentence "While the system is operating..." being acted upon by macro X. All of the lines following that sentence are now indented by five spaces, as shown below:

```
Maintain the climate of a building using a system comprised of↑
heating and cooling methods.↑
↑
Startup the climate system.↑
↑
While the system is operating, do (and repeat) the following:↑
↑
     Get the data needed for each pass: the time, the temperatures,↑
     the weather, the state of the solar collector, etc.↑
     Store this data.↑
     ↑
     Based on temperature data, see if there is a request,↑
     and determine the type of request.↑
     If there is no request, choose ''no method'' as the method,↑
          and operate the system with this method.↑
```

```
     If there is a request for heat, determine the heating method,↑
          and operate the system with this method.↑
     If there is a request for cold, determine the cooling method,↑
          and operate the system with this method.↑
     ↑
If no method is possible (abnormal conditions),↑
shut down the climate system.↑
↑
```

"What was that gibberish I typed?" To answer your question, display the macro definition again by typing the following command in command mode:

```
*?M<cr>
XL0;%<L;I/      /;P>
```

The display is exactly what you typed before, without the "@" symbols. The "@" symbols were used to define the macro—they delimit the actual text of the macro, so they are not necessary in this display (therefore CREDIT does not display them).

The macro starts with its name: X. Following the name is the first command in the macro: the L0 command, which moves the character pointer (called CP from now on) to the beginning of the current line (the line that the CP is currently sitting on—this line changes every time the entire macro executes). Following the L0 command is a semicolon, which separates one command from the next.

The next command in macro X is a set of commands to be executed repeatedly, or *iteratively*. The "%" (percent) symbol is a special one in this case, and is explained in the next paragraph. You saw the angle brackets in the last section—they delimit the set of commands to be executed repeatedly. The commands to be executed repeatedly are the L command (move CP to the beginning of the next line), the I command (insert the text that is specified within the following "/" (slash) symbols), and the P command (display the current line). The text within the "/" (slash) symbols consists of the five spaces we need to insert in order to indent the lines.

Now we'll explain the "%" (percent) symbol. When you typed the command to execute macro X, you typed:

```
*MFX(13)<cr>
```

The "MF" command executes the macro "X". The number 13 in parentheses is called a *parameter*—the MF command substitutes the parameter you specify (in this case, 13) for the "%" (percent) symbol. The result is that the iterative command set ("<L;I/ /;P>") is executed 13 times. Why did we pick 13? Because there were only 13 lines ahead of the character pointer to be indented. We could have specified a "!" (exclamation point) symbol to execute the iterative command set over and over until it reached the end of the file, but we chose this method to show you the use of parameters in macros, and to make sure that the last two lines would not be indented.

We also chose to use the "@" symbol to delimit the entire macro, but you can use any symbol that is not used within the macro.

Obviously, there are more details you should learn before using macros and other advanced editing features. However, you do not need these features to perform simple editing operations. They exist only to make your text editing sessions easier, if you first learn how to use them.

ENDING A TEXT EDITING SESSION AND MANAGING BACKUP FILES

When you are editing text, the edited text is in the computer's memory, but it is not yet on disk. To update the text file with the edited text, you have to end your text editing session properly. The EX command properly replaces the old text in the file with the newly edited text, and it also saves the old version of the file in *another* file called a *backup* file. When the edit session ends, control of the system returns to the operating system (ISIS-II).

If you have been following the examples in this chapter, you are now in CREDIT with your edited text, and you are ready to end your edit session. If you are not in command mode (with the asterisk prompt), use the HOME key to get into command mode. When you are in command mode, type the following command:

```
*EX<cr>
```

The text on the screen should disappear, and another message should appear:

```
EDITED TO :F0:PIDGIN.TXT
```

The file PIDGIN.TXT, on the disk in drive 0, now contains the newly edited text. Use the DIR command to see the directory for drive 0. In the directory listing, you should see the filename PIDGIN.BAK. CREDIT created this file to be the backup file that contains the old unedited version of PIDGIN.TXT. Every time you use the EX command by itself, CREDIT automatically creates a backup file with the ".BAK" extension to contain the previous version of the file.

NOTE

You should not execute CREDIT to edit a backup file (a file with the ".BAK" extension), because the EX command would first put the new edits into the ".BAK" file, then it would overwrite the ".BAK" file with the previous (unedited) version of it. Use the REN (rename) command to rename the file before editing it, or use the filename option with the EX command, described in the next paragraph.

You can optionally specify a filename with the EX command, so that the newly edited text becomes a new text file, and the old text file remains unedited. For example, if you had typed the following version of the EX command (rather than the preceding version):

```
*EX NEW.TXT<cr>
```

the following message would appear:

```
EDITED TO :F0:NEW.TXT
```

The file NEW.TXT would contain the newly edited text, and the old PIDGIN.TXT would not have been updated (and PIDGIN.BAK would not have been created as a backup file).

NOTE

If you type "EXIT" rather than "EX", CREDIT assumes that you want to store the newly edited text in the file IT (:F0:IT).

Another way to end an edit session is to use the EQ (quit) command. The EQ command will keep all files unchanged, as if nothing had happened. All edits you made while in the edit session vanish, and the text and backup files (if they exist) remain unchanged. If you use the EQ command in a session that created the text file, the new text file would not exist.

To keep you from making a mistake, the EQ command will first ask you for a Y or N answer before it ends the session:

```
*EQ<cr>
QUIT?Y
```

If you reply with anything other than a "Y" or "y", the EQ does not end the edit session.

DISPLAYING AND PRINTING TEXT FILES

Now that you have a text file, we can show you how to display the file on your screen and copy the file to a printing device. You can display the text file on your screen without using CREDIT by using the COPY command and the device name :CO: (for "console output"):

```
-COPY PIDGIN.TXT TO :CO:<cr>
```

To print the text file PIDGIN.TXT, you use the COPY command to copy the file to the device name :LP: (for "line printer"), or to the device name :TO: (for "teletype output"). You can COPY a file to any output device. You can find a complete list of device names in the *Intellec Series III Microcomputer Development System Console Operating Instructions*. This example assumes that you have an :LP: device to receive a copy of the file :F0:PIDGIN.TXT:

```
-COPY PIDGIN.TXT TO :LP:<cr>
```

FROM TEXT TO PROGRAM

Your PIDGIN.TXT text file now contains an algorithm that is actually a program in disguise (the disguise is English, or Pidgin Pascal). You should keep a copy of it somewhere, perhaps on another disk (or leave it in drive 0 if you have a hard disk subsystem). To use this algorithm in the next chapter, you will want to copy the text file to the Pascal-86 disk, and call it MAIN.SRC, since it will become the source file of your main control algorithm:

```
-COPY PIDGIN.TXT TO :F1:MAIN.SRC<cr>
:F0:PIDGIN.TXT COPIED TO :F1:MAIN.SRC
```

The examples in this and subsequent chapters assume that you either have a hard disk subsystem, or at least two double-density flexible disk drives. If you have single-density flexible disk drives, you should have more than two of them; in this case, you should put your program on the third disk, since it probably will not fit on the Pascal-86 disk in drive 1.

Although MAIN.SRC only has Pidgin Pascal statements at this time, you will edit them to make them real Pascal-86 statements as they appear in figures in Chapter 4.

CHAPTER 4
PROGRAMMING IN PASCAL-86

"One of the most important aspects of any computing tool is its influence on the thinking habits of those who try to use it..."

—E. W. Dijkstra

The Series III Microcomputer Development System was designed to support a variety of programming techniques with several programming languages. The preceding chapters give you the background you need to use this system wisely, and this and the following chapters help you decide the criteria for decomposing your application into modules and picking the appropriate language to use for each module.

One popular approach to programming is the top-down approach, where you define the problem completely, design an abstract algorithm to solve the problem, and refine this algorithm into self-supporting modules that can be coded and compiled separately. Typically, the *main module* would contain the most abstract algorithm—the control algorithm at the top of the design that solves the entire problem. The subordinate modules perform the procedures dictated by the main module.

Pascal-86 is a language that is ideal for the main module of such a modular solution. Using Intel's Pascal-86 compiler, you can decompose a program into modules that can be compiled separately, whereas other Pascal compilers only compile whole programs which have to be tailored to fit into microprocessor environments.

Perhaps the most important reason for Pascal's wide acceptance is the fact that it is a language that closely resembles English. In the past, programmers had to keep their algorithm designs well within the constraints of programming languages that were designed to express mathematical equations. In other words, at the outset they had to think in terms of the programming language available to them. This approach reinforced the practice of giving implementation ("how to") information in the problem definition (for example, "the input to this program is to be formatted on cards in columns 0-15 ...").

In Chapter 3 we designed an algorithm for the climate control of a building using English, which we jokingly call Pidgin Pascal. Since our control structures are in English, we have been able to communicate this algorithm easily and test its logic before translating it into Pascal-86. By now we should have a complete problem definition and a clean algorithm that could be translated into *any* programming language.

TRANSLATING PIDGIN PASCAL TO PASCAL-86

Figure 4-1 shows the main climate control algorithm, described in Pidgin Pascal. Several assumptions are made: that another subordinate module will operate the climate system, that yet another module will access and store the data, and that the data itself will be in the form of a record, which will be available to this algorithm.

As in typical development situations, a change has just occurred in our climate system that we software engineers have to accomodate: the first version of our climate system will not have cooling methods—only heating methods. We must design the software to make room for cooling methods in the future.

We made another change to the algorithm to accomodate a "panic" condition. An algorithm is not complete unless it can handle *any* situation; remember, Murphy said that if it can go wrong, it will. Therefore, we added a test to see if the climate system can handle the request for heat. If neither the collector water nor the tank water is hot enough to heat the building, a panic condition occurs that stops the normal operation of the climate system. At this time, it is sufficient to simply stop the program and output warning messages; later, we can add more procedures to handle such panics.

Using CREDIT (as described in Chapter 3), you can change this algorithm into Pascal-86 by adding the Pascal-86 statements and using *comment symbols* to turn the English sentences into program comments. The (* and *) symbols tell the Pascal-86 compiler to ignore whatever is between them. Some comments are only a few words surrounded by the (* and *) symbols, but comments can take up many lines, as shown at the end of the program in figure 4-2. As soon as the compiler sees the (* symbols, it ignores the characters and lines following it until it sees the *) closing symbols.

These comments are carried over with the program statements to the listing file produced by the compiler. You use the listing file as documentation for the program. You'll see a listing file later in this chapter.

Figure 4-2 shows the same algorithm expressed in Pascal-86 statements, with the English sentences masquerading as program comments. We also added more comments. It is a good practice to write the comments of a program before writing the actual program statements.

```
Maintain the climate of a building using a system comprised of
heating methods.

Startup the climate system.

While the system is operating, do (and repeat) the following:

     Get the data needed for each pass: the time, the temperatures,
     the weather, the state of the solar collector, etc.
     Store this data.

     Based on temperature data, see if there is a request
     for heat.
     If there is no request, choose ''no method'' as the method,
          and operate the system with this method.
     If there is a request for heat, determine whether the
          system can handle the request. If not, cause a panic.
          Otherwise, determine the heating method,
          and operate the system with this method.

If no method is possible (panic or abnormal conditions),
shut down the climate system.
```

Figure 4-1. Algorithm for the Climate Control Main Module

```
(**Type and variable declarations to be supplied later**)
(**Public procedures external to this program will be supplied later**)

PROGRAM MainControl(INPUT,OUTPUT);

BEGIN (**Main Control Algorithm**)
StartUpSystem; (*procedure to start up the climate system*)
Operating:=TRUE;
Panic:=FALSE;
   WHILE Operating DO (*While the system is operating, do
                       (and repeat) the following:*)
      BEGIN
         GetData(CurrentData);
             (*Get data needed for each pass: temps, time, etc.*)
         StoreData(CurrentData); (*Store this data as record*)

  (**If there is a request for heat, determine whether the system
         can handle the request.  If not, cause a panic.
         Otherwise, determine the heating method,
         and operate the system with this method.
     If there is no request for heat, choose ''no method,''
         and operate the system with this method.        **)

         WITH CurrentData DO
             BEGIN
                IF InsideTemp<ThermostatSetting THEN (*if request*)
                 BEGIN
                   IF CollectorWaterTemp>MinimumForExchanger THEN
                      BEGIN ChosenMethod:=CollectorToExchanger;
                            OperateSystem(CurrentData);
                      END
                   ELSE IF CollectorWaterTemp>MimimumForHeatPump THEN
                      BEGIN ChosenMethod:=CollectorToHeatPump;
                            OperateSystem(CurrentData);
                      END
                   ELSE IF TankWaterTemp>MinimumForExchanger THEN
                      BEGIN ChosenMethod:=TankToExchanger;
                            OperateSystem(CurrentData);
                      END
                   ELSE IF TankWaterTemp>MimimumForHeatPump THEN
                      BEGIN ChosenMethod:=TankToHeatPump;
                            OperateSystem(CurrentData);
                      END
                   ELSE IF HeatedTankTemp>MinimumForHeatPump THEN
                      BEGIN ChosenMethod:=HeatedTankToHeatPump;
                            OperateSystem(CurrentData);
                      END
                   ELSE Panic:=TRUE; Operating:=FALSE;
                 END (*if heating request*)
                ELSE (*no heating request*)
                   BEGIN ChosenMethod:=NoMethod;
                         OperateSystem(CurrentData);
                   END;
             END; (*End routine WITH CurrentData*)
      END; (*While operating*)
ShutDownSystem(CurrentData); (* panic or abnormal condition *)
END.
```

Figure 4-2. First Try at Coding the Main Program

```
(**The following are only comments:

   The following procedures will be coded in another module
   called the Operation Module:

   PROCEDURE OperateSystem(CurrentData)

   This procedure will operate the system and constantly
   maintain heat gain in the system. Depending on the
   heating method chosen, it opens certain valves and
   closes others, and turns on certain pumps and turns
   off others. It also maintains a flow of heated water
   to the storage tank.

   NOTE: for our testing purposes, a dummy OperateSystem
   procedure will only display messages telling us what
   heating method was chosen, and temperature data.


   PROCEDURE ShutDownSystem(CurrentData)

   This procedure will perform an orderly shut down if a
   panic or abnormal condition occurs.  The shut down must
   keep warm water flowing through the solar collector and
   close any extraneous valves, etc.  It must also send a
   warning messages to the console, advising manual operation
   of the furnace, etc.

   NOTE: for our testing purposes, a dummy ShutDownSystem
   procedure will only display data and a shutdown message.


   PROCEDURE StartUpSystem

   This procedure will start the climate system (cold start,
   or after ShutDownSystem occurs), open necessary valves,
   etc. It will also display a startup message.

   NOTE: for our testing purposes, a dummy StartUpSystem
   procedure will only display a startup message.


   The following procedures will be coded in another module
   called the GetData Module:

   PROCEDURE GetData(CurrentData)

   This procedure will obtain the data from a PL/M-86 module
   called PLMDATA that accesses ports to obtain temperature
   data.  Data other than temperature data will come from
   ports via port input/output procedures in this Pascal-86
   module.

   NOTE: for our testing purposes, a dummy GetData procedure
   will obtain all data from the console.
```

Figure 4-2. First Try at Coding the Main Program (Cont'd.)

```
PROCEDURE StoreData(CurrentData)

This procedure will store the data record CurrentData in a
file (the file would probably reside in non-volatile bubble
memory).

NOTE: for our testing purposes, a dummy StoreData procedure
will simply output the data to the console.


This is the last line of comments.**)
```

Figure 4-2. First Try at Coding the Main Program (Cont'd.)

This guide cannot possibly explain Pascal syntax—there are several books mentioned in the Bibliography that can give you the background you need, and the *Pascal-86 User's Guide* provides all the information you need to use Intel's extensions to standard Pascal.

PASCAL-86 DATA TYPES

A major advantage that Pascal has over other high-level languages is its strong type checking mechanisms that enforce data typing. By using Pascal's data types, you avoid some of the classic causes of errors in programs—the ambiguities involved with using simple X and Y variables to hold truly non-numeric data, the mistakes that occur when you attach arbitrary meanings to numeric data, and the complexities that are magnified by ambiguous variable names.

An example is worth a thousand explanations. In the Pascal-86 algorithm in figure 4-2, we make assignments like this one:

```
ChosenMethod:=TankToExchanger;
```

If **ChosenMethod** and **TankToExchanger** are declared properly in the module heading (not shown in figure 4-2, but shown later in this section), the Pascal-86 compiler will know their meanings. When you read this assignment, you know exactly what heating method has been chosen. The data type is a type of heating method, not an integer representing a method. In other programming languages you might be able to have a variable named "ChosenMethod" and another variable named "TankToExchanger", but you would also have to be sure to assign proper numeric or string values to them. A typical way of expressing the above assignment in PL/M would be:

```
     CHOSEN$METHOD = 2
/*where 2 is the appropriate method*/
```

or

```
     CHOSEN$METHOD = T$TO$EXCH
/*where T$TO$EXCH has already been assigned the appropriate
value*/
```

In both cases, you have to know a numeric code for the heating method. In Pascal, however, you only have to define a set of heating methods, and pick one for the assignment. Here is an example of such a definition:

```
TYPE HeatingMethods = (CollectorToExchanger,
                       CollectorToHeatPump,
                       TankToExchanger,
                       TankToHeatPump,
                       HeatedTankToHeatPump,
                       NoMethod);

(*The above defines the data type HeatingMethods, which is
  used to define the variable ChosenMethod below.*)

VAR ChosenMethod  : HeatingMethods;
```

In order to assign a value to **ChosenMethod**, the program must assign one of the methods in the set of type **HeatingMethods**. Any other assignment would cause a compiler error message to occur. By enforcing this data typing mechanism, the Pascal-86 compiler reduces the number of run-time errors by flagging the data type errors early in the game.

There are several standard Pascal data types that are useful. For example, you can define a variable to be of type BOOLEAN, which means that the only values that can be assigned to the variable are the values TRUE and FALSE. The variables **Operating** and **Panic** are of type BOOLEAN; they are either TRUE or FALSE.

Note that **ChosenMethod** and **CHOSENMETHOD** would refer to the same variable, since lower case characters are treated as upper case. This feature allows you to create long identifying names with combinations of upper and lower case characters that are easy to read and understand.

ANOTHER LOOK AT MODULARIZING AND HIDING INFORMATION

The programming technique called information-hiding is *not* an excuse for designers to withhold information from their documentors—it is more akin to a technique we use to hold a lot of information in our minds. When we have to interface with several different organizations within a company in order to get a job done, we don't pay attention to the inner workings of an organization; we simply assume that the organization will do its job, and we define our interface with the organization. Their organization is one module, and ours is another module; the job gets done because the modules know how to communicate to each other without interfering in each other's details.

Most logical algorithms are designed with assumptions about working modules. In our algorithm in figure 4-2, we assume that the procedures **StartUpSystem**, **GetData**, **StoreData**, **OperateSystem** and **ShutDownSystem** will work as planned, even though they are not yet written. We also assume that another group may write them. We can make these assumptions because we designed our main module to hide most of the details about choosing heating methods.

So far, the main module's algorithm decides the appropriate heating method based on a set of data. Once the algorithm is written, it may never change; and if it had to change, its change should not affect the other modules. However, we could change the control algorithm so that changes to the heating methods, or additional heating methods, would not even affect the main control algorithm. A simple way to do this would be to turn the heating method determina-

tion algorithm into an independent procedure called **DetermineMethod**, extract from this algorithm the calls to **OperateSystem**, and put the call to **OperateSystem** in the control algorithm.

The resulting main module is shown in figure 4-3. We added the module heading, but we still need the interface specification and variable declarations (shown later).

```
MODULE MainControl;

(* Interface specification goes here, to be supplied later. *)
(* Type definitions and variable declarations to be supplied later. *)

PROGRAM MainControl(INPUT,OUTPUT);


PROCEDURE DetermineMethod(VAR CurrentData : SystemData);
   BEGIN
     WITH CurrentData DO
          BEGIN
            IF InsideTemp<ThermostatSetting THEN
               BEGIN
                  IF CollectorWaterTemp>MinimumForExchanger THEN
                     ChosenMethod:=CollectorToExchanger
                  ELSE IF CollectorWaterTemp>MinimumForHeatPump THEN
                     ChosenMethod:=CollectorToHeatPump
                  ELSE IF TankWaterTemp>MinimumForExchanger THEN
                     ChosenMethod:=TankToExchanger
                  ELSE IF TankWaterTemp>MinimumForHeatPump THEN
                     ChosenMethod:=TankToHeatPump
                  ELSE IF HeatedTankTemp>MinimumForHeatPump THEN
                     ChosenMethod:=HeatedTankToHeatPump
                  ELSE Panic:=TRUE; Operating:=FALSE;
               END
            ELSE (*no heating request*)ChosenMethod:=NoMethod
          END; (*With CurrentData*)
   END; (*DetermineMethod)

(************** MAIN PROGRAM ************************************)

BEGIN
   StartUpSystem;
   Operating:=TRUE;
   Panic:=FALSE;
   WHILE Operating DO (*while system is operating, do:*)
      BEGIN
         GetData(CurrentData); (*Get the temps, time, etc.*)
         StoreData(CurrentData); (*Store this data as record*)
         DetermineMethod(CurrentData); (*this detects a panic*)
         OperateSystem(CurrentData);
      END; (*while operating*)
   ShutDownSystem(CurrentData);
END. (*Main Control Algorithm*)
```

Figure 4-3. Second Try at Coding the Main Program

The **DetermineMethod** procedure now hides all the information about choosing the appropriate heating method. We could also rewrite it to include cooling methods, or to change the heating methods. The procedure expects to receive the record **CurrentData**, and it only changes the value of the variable **ChosenMethod**.

The **OperateSystem** procedure will not be written until more facts are known about the hardware of the actual climate system. However, we already know that if we make a decision about a chosen method, include that method in the data record **CurrentData**, and send that data record to the **OperateSystem** procedure properly, the **OperateSystem** procedure will know how to operate the system. We defer these details to a later time when we'll have a prototype system to operate.

PASSING DATA TO OTHER MODULES—PARAMETER PASSING TECHNIQUES

We designed our main module so that it will receive a record of information. A Pascal-86 *record* is much like a PL/M-86 *structure* which can be defined to hold certain data types. We have to define this record in order to write the main module, but we can defer decisions about obtaining the data in order to preserve our options.

A PL/M-86 procedure could easily obtain the data and build the structure according to interface specifications; so could an 8086/8087/8088 Macro Assembly Language program, or an 8089 Assembly Language program. In fact, we might be able to use existing routines to obtain the data, and simply write another routine to structure the data accordingly.

In any case, we only have to pass the *address* of the structure to the Pascal-86 main module, which knows what to do with it. This parameter-passing technique is known as *pass by reference*, because the main module only needs a reference to the address of the structure in order to treat the structure as a Pascal record.

Another parameter-passing technique is *pass by value*, where a procedure calls another procedure and sends it a value rather than an address. We don't use this technique in our application, since our procedures need to access data in the record. We decided against passing specific values from this data record, and decided instead to make the entire data record available to the appropriate procedures.

To define the data record properly and still provide the ability to change it easily, we created a data type for the record:

```
TYPE (*definitions publicly defined in this module*)
.
.
.
     SystemData        = RECORD
                           ChosenMethod          : HeatingMethods;
                           InsideTemp,
                           ThermostatSetting     : AirTemperature;
                           CollectorWaterTemp,
                           TankWaterTemp,
                           HeatedTankTemp        : WaterTemperature;
                           AmountOfSunlight      : Integer;
                           Hour                  : 00..24;
                           Minute                : 00..59;
                         END (*SystemData*);
```

Using the data type **SystemData**, we defined the variable **CurrentData** to be of that type:

```
VAR
   CurrentData                : SystemData;
.
.
.
```

We pass the variable **CurrentData** to other procedures. If we were to change the data fields in the record, we would only have to change the definition of **SystemData**; we would not have to change any of the calls that pass the variable **CurrentData**. If the data fields kept the same names (**ChosenMethod**, **ThermostatSetting**, etc.), we would not have to change the routines that use those data fields.

THE INTERFACE SPECIFICATION

A module that calls a procedure in another module must have some information about where the other procedure is, and it must provide some information to the other procedure about the data being passed. Intel's Pascal-86 provides a mechanism for supplying the appropriate information to all modules that are to be linked to form a program—it is called an *interface specification*.

The interface specification typically holds the type definitions and variable declarations that are needed by all modules, and it also contains the names of procedures (with their parameters) that are public to other modules; that is, they can be called from other modules. Each module of the entire program contains this information. Figure 4-4 shows the interface specification for our program.

In addition to PUBLIC definitions in the interface specification, a module can have PRIVATE type definitions and variable declarations for variables used only within the module. Our **Operation** module will have a PRIVATE section for all variables that are only used within the module, but our **MainControl** main module does not need one. In Pascal-86, a PRIVATE heading is used in non-main modules instead of a PROGRAM heading.

Several *enumerated types* are defined in our program: **AirTemperature** and **WaterTemperature** are defined as types that can only have values in the ranges specified. The variables **Hour** and **Minute** are also of enumerated types, but since their ranges do not change, their types are not defined separately. By defining the temperature types separately, we can easily change their ranges without affecting the data record.

By defining the data record as type **SystemData**, we can easily change data fields in the record without changing the **CurrentData** declaration. By defining and declaring types and variables in the interface specification, we can maintain the interface specification separately (and change definitions and declarations) without affecting the procedures.

TEST VERSION OF THE CLIMATE CONTROL SYSTEM

Since our hardware designs are not yet firm, we should put together a test version of our system that does not interact with any prototype hardware. This version should include dummy procedures for the procedures that would normally rely on 8088 ports and other hardware.

```
PUBLIC MainControl; (*section of interface specification*)

   CONST (*declarations declared publicly in this module*)

     MinimumForExchanger    =  35;(*degrees Celsius*)
     MinimumForHeatPump     =  13;

   TYPE (*definitions publicly defined in this module*)

     AirTemperature    =-20..120;(*degrees in Celsius*)
     WaterTemperature  =0..120;
     HeatingMethods    =(CollectorToExchanger,
                         CollectorToHeatPump,
                         TankToExchanger,
                         TankToHeatPump,
                         HeatedTankToHeatPump,
                         NoMethod);
     SystemData        = RECORD
                           ChosenMethod          : HeatingMethods;
                           InsideTemp,
                           ThermostatSetting     : AirTemperature;
                           CollectorWaterTemp,
                           TankWaterTemp,
                           HeatedTankTemp        : WaterTemperature;
                           AmountOfSunlight      : Integer;
                           Hour                  : 00..24;
                           Minute                : 00..59;
                         END (*SystemData*);

   VAR (*variables publicly defined in this module.*)

     CurrentData                 : SystemData;
     Operating, Panic            : BOOLEAN;

PUBLIC GetData; (*GetData Module containing GetData & StoreData*)

   PROCEDURE GetData(VAR CurrentData:SystemData);
   PROCEDURE StoreData(VAR CurrentData:SystemData);

PUBLIC Operation; (*Operation Module containing OperateSystem,
                    StartUpSystem and ShutDownSystem*)
   PROCEDURE StartUpSystem;
   PROCEDURE OperateSystem(VAR CurrentData:SystemData);
   PROCEDURE ShutDownSystem(VAR CurrentData:SystemData);
```

Figure 4-4. The Interface Specification

The dummy versions are shown in figure 4-5. You should type these versions exactly as you see them, with the **StartUpSystem**, **OperateSystem**, and **ShutDownSystem** dummy procedures in the dummy **Operation** module stored in the file :F1:DUMOP.SRC, and **GetData** and **StoreData** dummy procedures in the dummy **GetData** module stored in the file :F1:DUMDAT.SRC. The **MainControl** module should be stored in :F1:MAIN.SRC, and the *interface specification* shown in figure 4-4 should be typed into the file :F1:INSPEC.SRC. If you cannot fit all of these files on the disk in drive 1, you should put all of them on another disk—and use your own pathname (:F*n*:) for the ".SRC" files. For our examples we assume that these files are on the disk in drive 1, along with the Pascal-86 compiler and run-time libraries.

Figure 4-5 shows each module and the dummy procedures. Since the interface specification is repeated in each module, we use a shortcut when compiling the modules by putting the common interface specification in a seperate file (:F1:INSPEC.SRC), and we use the INCLUDE control in a *control line* as shown:

```
$INCLUDE(:F1:INSPEC.SRC)
```

If you put INSPEC.SRC on a disk in a drive other than drive 1, use your own pathname instead of :F1:INSPEC.SRC.

```
MODULE MainControl;

(* Interface specification common to all modules *)

$INCLUDE(:F1:INSPEC.SRC)

PROGRAM MainControl(INPUT,OUTPUT);

(* end of interface specification *)

PROCEDURE DetermineMethod(VAR CurrentData : SystemData);
   BEGIN
     WITH CurrentData DO
          BEGIN
            IF InsideTemp<ThermostatSetting THEN
               BEGIN
                  IF CollectorWaterTemp>MinimumForExchanger THEN
                     ChosenMethod:=CollectorToExchanger
                  ELSE IF CollectorWaterTemp>MinimumForHeatPump THEN
                     ChosenMethod:=CollectorToHeatPump
                  ELSE IF TankWaterTemp>MinimumForExchanger THEN
                     ChosenMethod:=TankToExchanger
                  ELSE IF TankWaterTemp>MinimumForHeatPump THEN
                     ChosenMethod:=TankToHeatPump
                  ELSE IF HeatedTankTemp>MinimumForHeatPump THEN
                     ChosenMethod:=HeatedTankToHeatPump
                  ELSE Panic:=TRUE; Operating:=FALSE;
               END
            ELSE (*no heating request*)ChosenMethod:=NoMethod
          END; (*With CurrentData*)
   END; (*DetermineMethod)
```

Figure 4-5. Test Version of Our Climate Control System

```
(************** MAIN PROGRAM ************************************)

BEGIN
   StartUpSystem;
   Operating:=TRUE;
   Panic:=FALSE;
   WHILE Operating DO (*while system is operating, do:*)
      BEGIN
         GetData(CurrentData); (*Get the temps, time, etc.*)
         StoreData(CurrentData); (*Store this data as record*)
         DetermineMethod(CurrentData); (*this detects a panic*)
         OperateSystem(CurrentData);
      END; (*while operating*)
   ShutDownSystem(CurrentData);
END. (*Main Control Algorithm*)


(*This is a dummy GetData module, with dummy GetData
and StoreData procedures, for use with MainControl
module in testing phases. It only performs console
input to get Celsius temperatures, the time of day,
and the amount of sunlight (insolation) for the
solar collector. Use PLMDATA module for real
application.*)

MODULE GetData;

(* Interface specification common to all modules *)

$INCLUDE(:F1:INSPEC.SRC)

PRIVATE GetData;
(* end of interface specification *)


PROCEDURE GetData(VAR CurrentData:SystemData);
BEGIN
   WITH CurrentData DO BEGIN
      WRITE('Type the thermostat setting in degrees Celsius:');
      READLN(ThermostatSetting); WRITELN;
      WRITE('Type the inside temperature reading in Celsius:');
      READLN(InsideTemp); WRITELN;
      WRITE('Type the temperature of the collector water in Celsius:');
      READLN(CollectorWaterTemp); WRITELN;
      WRITE('Type the temperature of the tank water in Celsius:');
      READLN(TankWaterTemp); WRITELN;
      WRITE('Type the temperature of the heated tank water in Celsius:');
      READLN(HeatedTankTemp); WRITELN;
      WRITE('Type the hour of day, as in 04 or 24:');
      READLN(Hour); WRITELN;
```

Figure 4-5. Test Version of Our Climate Control System (Cont'd.)

```
      WRITE('Type the minute of the hour, as in 01 or 59: ');
      READLN(Minute); WRITELN;
      WRITE('Type the amount of sunlight, any integer will do for now:');
      READLN(AmountOfSunlight); WRITELN;
   END;  (*with CurrentData*)
END;
PROCEDURE StoreData(VAR CurrentData:SystemData);
BEGIN
(*Dummy procedure, eventually will store CurrentData in a file*)
   WITH CurrentData DO BEGIN
      WRITELN('--------------------------');
      WRITELN('CURRENT DATA IS AS FOLLOWS:');
      WRITELN('--------------------------');
      WRITELN('Thermostat Setting is ',ThermostatSetting,'C');
      WRITELN('Inside temperature is ',InsideTemp,'C');
      WRITELN('Temperature of collector water is ',CollectorWaterTemp,'C');
      WRITELN('Temperature of tank water is ',TankWaterTemp,'C');
      WRITELN('Temperature of the heated tank water is ',HeatedTankTemp,'C');
      WRITELN('Time of day is ',Hour,':',Minute);
      WRITELN('Amount of sunlight is ',AmountOfSunlight);
      WRITELN; (*a blank line*)
   END; (*with CurrentData*)
END.

(*This is a dummy Operation module, with dummy StartUpSystem,
ShutDownSystem, and OperateSystem procedures,
for use with MainControl module in testing phases.*)

MODULE Operation;

(* Interface specification common to all modules *)

$INCLUDE(:F1:INSPEC.SRC)

PRIVATE Operation;
(* end of interface specification *)


PROCEDURE StartUpSystem;
BEGIN
     WRITELN ('Climate system is now on.');
     WRITELN ('-------------------------');
     WRITELN;
END;


PROCEDURE OperateSystem(VAR CurrentData:SystemData);
BEGIN
  WITH CurrentData DO BEGIN
    WRITELN('==========================================');
    WRITELN('The Climate System is now operating.');
```

Figure 4-5. Test Version of Our Climate Control System (Cont'd.)

```
      WRITELN;
      WRITELN('The Time is ',Hour,':',Minute);
      WRITELN('The inside temperature is ',InsideTemp,'C');
      WRITELN('The thermostat setting is ',ThermostatSetting,'C');
      WRITE('Method chosen to heat the building: ');
      CASE ChosenMethod OF
         CollectorToExchanger: WRITE('Solar Collector to Exchanger');
         CollectorToHeatPump : WRITE('Solar Collector to Heat Pump');
         TankToExchanger     : WRITE('Tank to Exchanger');
         TankToHeatPump      : WRITE('Tank to Heat Pump');
         HeatedTankToHeatPump: WRITE('Heated Tank to Heat Pump');
         NoMethod            : WRITE('No heat required');
      END;
      WRITELN;
      WRITELN('============================================');
      WRITELN; (*write a blank line.*)
    END;
  END;(*OperateSystem*)


  PROCEDURE ShutDownSystem(VAR CurrentData:SystemData);
  BEGIN
  WRITELN('::::::::::::::::::::::::::::::::::::::::::::::::::' );
     IF Panic THEN WRITELN('PANIC occurred, NORMAL shutdown.')
              ELSE WRITELN('No panic occurred, ABNORMAL shutdown.');
     WRITELN('::::::::::::::::::::::::::::::::::::::::::::::::::') ;
     WITH CurrentData DO BEGIN
        WRITE('Last chosen heating method was: ');
        CASE ChosenMethod OF
           CollectorToExchanger: WRITE('Solar Collector to Exchanger');
           CollectorToHeatPump : WRITE('Solar Collector to Heat Pump');
           TankToExchanger     : WRITE('Tank to Exchanger');
           TankToHeatPump      : WRITE('Tank to Heat Pump');
           HeatedTankToHeatPump: WRITE('Heated Tank to Heat Pump');
           NoMethod            : WRITE('No heat required');
        END;
        WRITELN;
        WRITELN('Thermostat Setting is ',ThermostatSetting,'C');
        WRITELN('Inside temperature is ',InsideTemp,'C');
        WRITELN('Temperature of collector water is ',CollectorWaterTemp,'C');
        WRITELN('Temperature of tank water is ',TankWaterTemp,'C');
        WRITELN('Temperature of the heated tank water is ',HeatedTankTemp,'C');
        WRITELN('Time of day is ',Hour,':',Minute);
        WRITELN('Amount of sunlight is ',AmountOfSunlight);
     END;(*with CurrentData*)
     WRITELN('::::::::::::::::::::::::::::::::::::::::::::::::::: :::');
     WRITELN;
     WRITELN('Goodnight, Irene...');
  END.(*ShutDownSystem*)
```

Figure 4-5. Test Version of Our Climate Control System (Cont'd.)

THE PASCAL-86 COMPILER

A *compiler* is a program that translates your high-level language statements into machine code. Machine code, sometimes called *object code*, consists of the instructions that machines understand, whereas high-level language statements are instructions that humans understand. You must translate your high-level language statements into machine code by *compiling* your high-level language program.

The Pascal-86 statements we typed (using CREDIT) are *program source statements*. We now have three *source files*: :F1:MAIN.SRC (for the **MainControl** module), :F1:DUMDAT.SRC (for the dummy **GetData** module), and :F1:DUMOP.SRC (for the dummy **Operation** module). To translate these source statement modules into object code modules, we must compile each source module separately.

The Pascal-86 compiler is supplied as the file PASC86.86. You invoke the compiler by using the RUN command to load and execute it in the "8086 side" (8086 execution mode) of the Series III system. The compiler usually produces two files: a *listing file* that contains a listing of the source program as the compiler saw it, and an *object file* that contains the actual machine code. The listing file usually contains a listing of the source statements, additional information about the compilation, and any errors that occurred during the compilation.

For example, assume that MAIN.SRC, the first module, is in directory :F1:. The Pascal-86 Compiler, PASC86.86, is also in directory :F1:. To compile this module, use the following command:

```
-RUN :F1:PASC86 :F1:MAIN.SRC DEBUG<cr>
```

Let's analyze this command line. RUN is the command used to execute the program in the "8086 side" of the system. :F1:PASC86 is the pathname (without the ".86" extension) of the Pascal-86 Compiler (RUN supplies the ".86" extension). :F1:MAIN.SRC is the pathname for the MAIN.SRC module. Finally, DEBUG is a *compiler control* which tells the compiler to do something special (described later).

This compilation produced two files: :F1:MAIN.LST is the listing file, and :F1:MAIN.OBJ is the object file that contains the object code. :F1:MAIN.SRC is still in drive 1. All of these files are in directory :F1:, since that is where MAIN.SRC resides.

Compiler controls tell the compiler to perform certain operations. Most controls have *default settings* that you do not have to specify. For example, the PRINT control is always on unless you specify NOPRINT. The PRINT control tells the compiler to produce a listing file, and use the name of the source file with an ".LST" extension (e.g., :F1:MAIN.LST). We could have used this version of the PRINT control:

```
-RUN :F1:PASC86 :F1:MAIN.SRC DEBUG PRINT(:LP:)<cr>
```

This version of the PRINT control sends the listing to the line printer (:LP:), rather than creating :F1:MAIN.LST as a listing file.

Most of the compiler control default settings are useful for everyday compiling; that is, there is no need to learn how to use the compiler controls unless you want to do something special. For example, if you want the compiler to issue a warning message whenever it sees a non-standard Pascal statement (an Intel extension to the standard Pascal language), use the NOEXTENSIONS control.

We used the DEBUG control for a good reason: we want to do symbolic debugging while the program is running (using DEBUG-86, described in Chapter 7). You will want to do symbolic debugging during the first run of your program. Use the DEBUG control to prepare your program for symbolic debugging unless your program is extremely large.

Most compiler controls can be specified in the invocation line as we show above. Most compiler controls can also be imbedded in the source file—as *control lines*. For example, we used the INCLUDE control in a control line:

```
$INCLUDE(:F1:INSPEC.SRC)
```

The INCLUDE control allows you to insert source statements from another file into this compilation. In this case, we wanted to insert the common interface specification (in :F1:INSPEC.SRC) into our compilation. The INCLUDE control saved us from typing the same interface specification for all three source files.

We also need to compile the other modules separately. The following invocation line compiles our **GetData** module in DUMDAT.SRC:

```
-RUN :F1:PASC86 :F1:DUMDAT.SRC DEBUG<cr>
SERIES III Pascal-86 V1.0
PARSE
  68   99  0  0
***WARNING, input:  ''END ''
***was repaired to: ''END ;''
END PARSE(1), ANALYZE(0), NOXREF, OBJECT

    COMPILATION OF GETDATA COMPLETED, 1 ERROR DETECTED.
    END OF Pascal-86 COMPILATION.
```

The compiler displays a sign-on message, then the word "PARSE" to show that it is parsing the program statements. During the parsing phase, the compiler discovered an error—the "END " statement was not punctuated correctly. The compiler repairs our error, and continues to compile. Each phase of compiling is displayed with a number in parentheses—the number of errors detected during the phase. The compiler only detected that one error, and since the error was easily repaired, the compilation was successful. We now have :F1:DUMDAT.OBJ containing the object module.

To compile our dummy **Operation** module, we use the following invocation line:

```
-RUN :F1:PASC86 :F1:DUMOP.SRC DEBUG<cr>
.
.
.
```

The listing files :F1:MAIN.LST, :F1:DUMDAT.LST, and :F1:DUMOP.LST are shown in figure 4-6.

```
SERIES-III Pascal-86, X031                                   09/01/80      PAGE  1
                                                                           MAINCONTROL

Source File: :F1:MAIN.SRC
Object File: :F1:MAIN.OBJ
Controls Specified:  DEBUG.

STMT LINE NESTING      SOURCE TEXT: :F1:MAIN.SRC
   1    1  0  0       MODULE MainControl;
   2    2  0  0

                      (* Interface specification common to all modules *)

                      $INCLUDE(:F1:INSPEC.SRC)
                      PUBLIC MainControl; (*section of interface specification*)
   3    2  0  0

                         CONST (*declarations declared publicly in this module*)

                           MinimumForExchanger    =  35;(*degrees Celsius*)
   4    6  0  0           MinimumForHeatPump     =  13;
   5    7  0  0

                         TYPE (*definitions publicly defined in this module*)

                           AirTemperature    =-20..120;(*degrees in Celsius*)
   6   11  0  0           WaterTemperature =0..120;
   7   12  0  0           HeatingMethods    =(CollectorToExchanger,
                                               CollectorToHeatPump,
                                               TankToExchanger,
                                               TankToHeatPump,
                                               HeatedTankToHeatPump,
                                               NoMethod);
   8   18  0  0

                           SystemData        = RECORD
   8   20  0  1                                 ChosenMethod          : HeatingMethods;
   9   21  0  1                                 InsideTemp,
                                                 ThermostatSetting     : AirTemperature;
  10   23  0  1                                 CollectorWaterTemp,
                                                 TankWaterTemp,
                                                 HeatedTankTemp        : WaterTemperature;
  11   26  0  1                                 AmountOfSunlight      : Integer;
  12   27  0  1                                 Hour                  : 00..24;
  13   28  0  1                                 Minute                : 00..59;
  14   29  0  1                               END (*SystemData*);
  15   30  0  0

                         VAR (*variables publicly defined in this module.*)

                           CurrentData                 : SystemData;
  16   34  0  0           Operating, Panic            : BOOLEAN;
  17   35  0  0

                      PUBLIC GetData; (*GetData Module containing GetData & StoreData*)
  18   37  0  0

                         PROCEDURE GetData(VAR CurrentData:SystemData);
  19   39  0  0           PROCEDURE StoreData(VAR CurrentData:SystemData);
  20   40  0  0

                      PUBLIC Operation; (*Operation Module containing OperateSystem,
  21   42  0  0                          StartUpSystem and ShutDownSystem*)
                         PROCEDURE StartUpSystem;
  22   44  0  0           PROCEDURE OperateSystem(VAR CurrentData:SystemData);
  23   45  0  0           PROCEDURE ShutDownSystem(VAR CurrentData:SystemData);
  24   46  0  0

  25    8  0  0      PROGRAM MainControl(INPUT,OUTPUT);

                      (* end of interface specification *)

                      PROCEDURE DetermineMethod(VAR CurrentData : SystemData);
  26   13  1  0         BEGIN
  26   14  1  1           WITH CurrentData DO
  27   15  1  1                BEGIN
  27   16  1  2               IF InsideTemp<ThermostatSetting THEN
  28   17  1  2                  BEGIN
  28   18  1  3                    IF CollectorWaterTemp>MinimumForExchanger THEN
  29   19  1  3                       ChosenMethod:=CollectorToExchanger
                                  ELSE IF CollectorWaterTemp>MinimumForHeatPump THEN
  31   21  1  3                       ChosenMethod:=CollectorToHeatPump
                                  ELSE IF TankWaterTemp>MinimumForExchanger THEN
  33   23  1  3                       ChosenMethod:=TankToExchanger
                                  ELSE IF TankWaterTemp>MinimumForHeatPump THEN
  35   25  1  3                       ChosenMethod:=TankToHeatPump
                                  ELSE IF HeatedTankTemp>MinimumForHeatPump THEN
  37   27  1  3                       ChosenMethod:=HeatedTankToHeatPump
                                  ELSE Panic:=TRUE; Operating:=FALSE;
  40   29  1  3                  END
  41   30  1  2               ELSE (*no heating request*)ChosenMethod:=NoMethod
  42   31  1  2           END; (*With CurrentData*)
  44   32  1  1         END; (*DetermineMethod)
  45   33  0  0
```

Figure 4-6. Listings of Our Test Modules

```
                        (************** MAIN PROGRAM ************************************)

                        BEGIN
 45   37  0  1             StartUpSystem;
 46   38  0  1             Operating:=TRUE;
 47   39  0  1             Panic:=FALSE;
 48   40  0  1             WHILE Operating DO (*while system is operating, do:*)
 49   41  0  1                BEGIN
 49   42  0  2                   GetData(CurrentData); (*Get the temps, time, etc.*)
 50   43  0  2                   StoreData(CurrentData); (*Store this data as record*)
 51   44  0  2                   DetermineMethod(CurrentData); (*this detects a panic*)
 52   45  0  2                   OperateSystem(CurrentData);
 53   46  0  2                END; (*while operating*)
 55   47  0  1             ShutDownSystem(CurrentData);
 56   48  0  1          END. (*Main Control Algorithm*)


Summary Information:

PROCEDURE                       OFFSET    CODE SIZE     DATA SIZE    STACK SIZE
DETERMINEMETHOD                 0011H  008FH   143D                  0006H     6D

Total                                  0147H   327D  0016H    22D    004CH    76D

  94 Lines Read.
   0 Errors Detected.


33% Utilization of Memory.
```

```
SERIES-III Pascal-86, X031                                          09/01/80        PAGE   1


Source File: :F1:DUMDAT.SRC
Object File: :F1:DUMDAT.OBJ
Controls Specified:  DEBUG.

STMT LINE NESTING       SOURCE TEXT: :F1:DUMDAT.SRC
   1    1  0  0         (*This is a dummy GetData module, with dummy GetData
                        and StoreData procedures, for use with MainControl
                        module in testing phases.  It only performs console
                        input to get Celsius temperatures, the time of day,
                        and the amount of sunlight (insolation) for the
                        solar collector.  Use PLM86$DATA module for real
                        application.*)

                        MODULE GetData;

   2   10  0  0
                        (* Interface specification, common to all modules *)

                        $INCLUDE(:F1:INSPEC.SRC)
                  =1    PUBLIC MainControl; (*section of interface specification*)
   3    2  0  0 =1
                  =1       CONST (*declarations declared publicly in this module*)
                  =1
                  =1         MinimumForExchanger   =  35;(*degrees Celsius*)
   4    6  0  0 =1         MinimumForHeatPump    =  13;
   5    7  0  0 =1
                  =1       TYPE (*definitions publicly defined in this module*)
                  =1
                  =1         AirTemperature    =-20..120;(*degrees in Celsius*)
   6   11  0  0 =1         WaterTemperature =0..120;
   7   12  0  0 =1         HeatingMethods    =(CollectorToExchanger,
                  =1                          CollectorToHeatPump,
                  =1                          TankToExchanger,
                  =1                          TankToHeatPump,
                  =1                          HeatedTankToHeatPump,
                  =1                          NoMethod);
   8   18  0  0 =1
                  =1         SystemData        = RECORD
   8   20  0  1 =1                               ChosenMethod          : HeatingMethods;
   9   21  0  1 =1                               InsideTemp,
                  =1                            ThermostatSetting     : AirTemperature;
  10   23  0  1 =1                               CollectorWaterTemp,
                  =1                            TankWaterTemp,
                  =1                            HeatedTankTemp        : WaterTemperature;
  11   26  0  1 =1                               AmountOfSunlight      : Integer;
  12   27  0  1 =1                               Hour                  : 00..24;
  13   28  0  1 =1                               Minute                : 00..59;
  14   29  0  1 =1                            END (*SystemData*);
```

Figure 4-6. Listings of Our Test Modules (Cont'd.)

```
  15   30   0   0 =1
                 =1          VAR (*variables publicly defined in this module.*)
                 =1
                 =1            CurrentData                : SystemData;
  16   34   0   0 =1            Operating, Panic           : BOOLEAN;
  17   35   0   0 =1
                 =1        PUBLIC GetData; (*GetData Module containing GetData & StoreData*)
  18   37   0   0 =1
                 =1           PROCEDURE GetData(VAR CurrentData:SystemData);
  19   39   0   0 =1           PROCEDURE StoreData(VAR CurrentData:SystemData);
  20   40   0   0 =1
                 =1        PUBLIC Operation; (*Operation Module containing OperateSystem,
  21   42   0   0 =1                           StartUpSystem and ShutDownSystem*)
                 =1           PROCEDURE StartUpSystem;
  22   44   0   0 =1           PROCEDURE OperateSystem(VAR CurrantData:SystemData);
  23   45   0   0 =1           PROCEDURE ShutDownSystem(VAR CurrentData:SystemData);
  24   46   0   0 =1

                        PRIVATE GetData;
  25   16   0   0
                        (* end of interface specification *)


                        PROCEDURE GetData(VAR CurrentData:SystemData);
  26   21   1   0        BEGIN
  26   22   1   1           WITH CurrentData DO BEGIN
  27   23   1   2              WRITE('Type the thermostat setting in degrees Celsius:');
  28   24   1   2              READLN(ThermostatSetting); WRITELN;
  30   25   1   2              WRITE('Type the inside temperature reading in Celsius:');
  31   26   1   2              READLN(InsideTemp); WRITELN;
  33   27   1   2              WRITE('Type the temperature of the collector water in Celsius:');
  34   28   1   2              READLN(CollectorWaterTemp); WRITELN;
  36   29   1   2              WRITE('Type the temperature of the tank water in Celsius:');
  37   30   1   2              READLN(TankWaterTemp); WRITELN;
  39   31   1   2              WRITE('Type the temperature of the heated tank water in Celsius:');
  40   32   1   2              READLN(HeatedTankTemp); WRITELN;
  42   33   1   2              WRITE('Type the hour of day, as in 04 or 24:');
  43   34   1   2              READLN(Hour); WRITELN;
  45   35   1   2              WRITE('Type the minute of the hour, as in 01 or 59: ');
  46   36   1   2              READLN(Minute); WRITELN;
  48   37   1   2              WRITE('Type the amount of sunlight, any integer will do for now:');
  49   38   1   2              READLN(AmountOfSunlight); WRITELN;
  51   39   1   2           END;  (*with CurrentData*)
  53   40   1   1        END;
  54   41   0   0
                        PROCEDURE StoreData(VAR CurrentData:SystemData);
  55   43   1   0        BEGIN
  55   44   1   1        (*Dummy procedure, eventually will store CurrentData in a file*)
                           WITH CurrentData DO BEGIN
  56   46   1   2              WRITELN('----------------------------');
  57   47   1   2              WRITELN('CURRENT DATA IS AS FOLLOWS:');
  58   48   1   2              WRITELN('----------------------------');
  59   49   1   2              WRITELN('Thermostat Setting is ',ThermostatSetting,'C');
  60   50   1   2              WRITELN('Inside temperature is ',InsideTemp,'C');
  61   51   1   2              WRITELN('Temperature of collector water is ',CollectorWaterTemp,'C');
  62   52   1   2              WRITELN('Temperature of tank water is ',TankWaterTemp,'C');
  63   53   1   2              WRITELN('Temperature of the heated tank water is ',HeatedTankTemp,'C');
  64   54   1   2              WRITELN('Time of day is ',Hour,':',Minute);
  65   55   1   2              WRITELN('Amount of sunlight is ',AmountOfSunlight);
  66   56   1   2              WRITELN; (*a blank line*)
  67   57   1   2           END; (*with CurrentData*)
  69   58   1   1        END
***WARNING, input: "END "
***was repaired to "END ; "
  70   58   0   0           .


Summary Information:

PROCEDURE                    OFFSET    CODE SIZE     DATA SIZE     STACK SIZE
STOREDATA                     0405H   0222H   546D                  0010H    16D
GETDATA                       0294H   0241H   577D                  0010H    16D

Total                                 06F7H  1783D  0000H     0D   0020H    32D

 104 Lines Read.
   1 Error Detected.
 33% Utilization of Memory.
```

Figure 4-6. Listings of Our Test Modules (Cont'd.)

```
SERIES-III Pascal-86, X031                                   09/01/80        PAGE   1


Source File: :F1:DUMOP.SRC
Object File: :F1:DUMOP.OBJ
Controls Specified:  DEBUG.

STMT LINE NESTING       SOURCE TEXT: :F1:DUMOP.SRC
   1    1  0  0         (*This is a dummy Operation module, with dummy StartUpSystem,
                        ShutDownSystem, and OperateSystem procedures,
                        for use with MainControl module in testing phases.
                        *)

                        MODULE Operation;
   2    7  0  0
                        (* Interface specification common to all modules *)

                        $INCLUDE(:F1:INSPEC.SRC)
                    =1  PUBLIC MainControl; (*section of interface specification*)
   3    2  0  0 =1
                    =1      CONST (*declarations declared publicly in this module*)
                    =1
                    =1        MinimumForExchanger   =  35;(*degrees Celsius*)
   4    6  0  0 =1           MinimumForHeatPump    =  13;
   5    7  0  0 =1
                    =1      TYPE (*definitions publicly defined in this module*)
                    =1
                    =1        AirTemperature   =-20..120;(*degrees in Celsius*)
   6   11  0  0 =1           WaterTemperature =0..120;
   7   12  0  0 =1           HeatingMethods   =(CollectorToExchanger,
                    =1                           CollectorToHeatPump,
                    =1                           TankToExchanger,
                    =1                           TankToHeatPump,
                    =1                           HeatedTankToHeatPump,
                    =1                           NoMethod);
   8   18  0  0 =1
                    =1        SystemData       = RECORD
   8   20  0  1 =1                                ChosenMethod          : HeatingMethods;
   9   21  0  1 =1                                InsideTemp,
                    =1                           ThermostatSetting     : AirTemperature;
  10   23  0  1 =1                                CollectorWaterTemp,
                    =1                           TankWaterTemp,
                    =1                           HeatedTankTemp        : WaterTemperature;
  11   26  0  1 =1                                AmountOfSunlight      : Integer;
  12   27  0  1 =1                                Hour                  : 00..24;
  13   28  0  1 =1                                Minute                : 00..59;
  14   29  0  1 =1                              END (*SystemData*);
  15   30  0  0 =1
                    =1      VAR (*variables publicly defined in this module.*)
                    =1
                    =1        CurrentData                 : SystemData;
  16   34  0  0 =1           Operating, Panic            : BOOLEAN;
  17   35  0  0 =1
                    =1      PUBLIC GetData; (*GetData Module containing GetData & StoreData*)
  18   37  0  0 =1
                    =1        PROCEDURE GetData(VAR CurrentData:SystemData);
  19   39  0  0 =1           PROCEDURE StoreData(VAR CurrentData:SystemData);
  20   40  0  0 =1
                    =1      PUBLIC Operation; (*Operation Module containing OperateSystem,
  21   42  0  0 =1                           StartUpSystem and ShutDownSystem*)
                    =1        PROCEDURE StartUpSystem;
  22   44  0  0 =1           PROCEDURE OperateSystem(VAR CurrentData:SystemData);
  23   45  0  0 =1           PROCEDURE ShutDownSystem(VAR CurrentData:SystemData);
  24   46  0  0 =1

                        PRIVATE Operation;
  25   13  0  0
                        (* end of interface specification *)


                        PROCEDURE StartUpSystem;
  26   18  1  0         BEGIN
  26   19  1  1              WRITELN ('Climate system is now on.');
  27   20  1  1              WRITELN ('--------------------------');
  28   21  1  1              WRITELN;
  29   22  1  1         END;
  30   23  0  0

                        PROCEDURE OperateSystem(VAR CurrentData:SystemData);
  31   26  1  0         BEGIN
  31   27  1  1           WITH CurrentData DO BEGIN
  32   28  1  2             WRITELN('=========================================');
  33   29  1  2             WRITELN('The Climate System is now operating.');
  34   30  1  2             WRITELN;
  35   31  1  2             WRITELN('The Time is ',Hour,':',Minute);
  36   32  1  2             WRITELN('The inside temperature is ',InsideTemp,'C');
  37   33  1  2             WRITELN('The thermostat setting is ',ThermostatSetting,'C');
  38   34  1  2             WRITE('Method chosen to heat the building: ');
```

Figure 4-6. Listings of Our Test Modules (Cont'd.)

```
  39   35  1  2            CASE ChosenMethod OF
  40   36  1  3               CollectorToExchanger: WRITE('Solar Collector to Exchanger');
  41   37  1  3               CollectorToHeatPump : WRITE('Solar Collector to Heat Pump');
  42   38  1  3               TankToExchanger     : WRITE('Tank to Exchanger');
  43   39  1  3               TankToHeatPump      : WRITE('Tank to Heat Pump');
  44   40  1  3               HeatedTankToHeatPump: WRITE('Heated Tank to Heat Pump');
  45   41  1  3               NoMethod            : WRITE('No heat required');
  46   42  1  3            END;
  48   43  1  2            WRITELN;
  49   44  1  2            WRITELN('==========================================');
  50   45  1  2            WRITELN; (*write a blank line.*)
  51   46  1  2         END;
  53   47  1  1       END;(*OperateSystem*)
  54   48  0  0
                       PROCEDURE ShutDownSystem(VAR CurrentData:SystemData);
  55   51  1  0       BEGIN
  55   52  1  1       WRITELN('::::::::::::::::::::::::::::::::::::::::::::::::::');
  56   53  1  1          IF Panic THEN WRITELN('PANIC occurred, NORMAL shutdown.')
  57   54  1  1                   ELSE WRITELN('No panic occurred, ABNORMAL shutdown.');
  59   55  1  1          WRITELN('::::::::::::::::::::::::::::::::::::::::::::::::::');
  60   56  1  1          WITH CurrentData DO BEGIN
  61   57  1  2             WRITE('Last chosen heating method was: ');
  62   58  1  2             CASE ChosenMethod OF
  63   59  1  3                CollectorToExchanger: WRITE('Solar Collector to Exchanger');
  64   60  1  3                CollectorToHeatPump : WRITE('Solar Collector to Heat Pump');
  65   61  1  3                TankToExchanger     : WRITE('Tank to Exchanger');
  66   62  1  3                TankToHeatPump      : WRITE('Tank to Heat Pump');
  67   63  1  3                HeatedTankToHeatPump: WRITE('Heated Tank to Heat Pump');
  68   64  1  3                NoMethod            : WRITE('No heat required');
  69   65  1  3             END;
  71   66  1  2             WRITELN;
  72   67  1  2             WRITELN('Thermostat Setting is ',ThermostatSetting,'C');
  73   68  1  2             WRITELN('Inside temperature is ',InsideTemp,'C');
  74   69  1  2             WRITELN('Temperature of collector water is ',CollectorWaterTemp,'C');
  75   70  1  2             WRITELN('Temperature of tank water is ',TankWaterTemp,'C');
  76   71  1  2             WRITELN('Temperature of the heated tank water is ',HeatedTankTemp,'C');
  77   72  1  2             WRITELN('Time of day is ',Hour,':',Minute);
  78   73  1  2             WRITELN('Amount of sunlight is ',AmountOfSunlight);
  79   74  1  2          END;(*with CurrentData*)
  81   75  1  1          WRITELN('::::::::::::::::::::::::::::::::::::::::::::::::::');
  82   76  1  1          WRITELN;
  83   77  1  1          WRITELN('Goodnight, Irene...');
  84   78  1  1       END
***WARNING, input: "END "
***was repaired to "END ; "
  85   78  0  0          .(*ShutDownSystem*)

Summary Information:

PROCEDURE                    OFFSET   CODE SIZE     DATA SIZE    STACK SIZE
SHUTDOWNSYSTEM               0698H  0398H   920D                 0010H    16D
OPERATESYSTEM                0440H  0258H   600D                 0010H    16D
STARTUPSYSTEM                03E2H  005EH    94D                 0010H    16D

Total                               0A30H  2608D  0000H     0D  0030H    48D

 125 Lines Read.
   1 Error Detected.
 33% Utilization of Memory.
```

Figure 4-6. Listings of Our Test Modules (Cont'd.)

Summary

We now have three Pascal-86 modules: **MainControl**, **GetData**, and **Operation**. Two of these modules, **GetData** and **Operation**, are dummy versions that do not interact with any hardware except the Series III system; we will use them for examples in subsequent chapters. The **MainControl** module can remain unchanged even in our final application.

The final application will probably use an 8088 processor. In Chapter 5, we show PL/M-86 procedures we can use to obtain data from thermocouples via the input/output ports on the 8088. Since you probably do not have prototype climate control hardware with an 8088, we will not include these procedures in our test versions for execution on a Series III system; nevertheless, they are good examples of PL/M-86 procedures.

Our Pascal-86 modules cannot run by themselves on a Series III. Certain built-in procedures (like WRITELN and READLN) rely on *run-time support software*, which consists of public modules that contain the software needed to perform console input/output and other operations. In Chapter 6, we'll show you how to link the run-time support libraries to these Pascal-86 modules to make them executable on a Series III system.

CHAPTER 5
PROGRAMMING IN OTHER LANGUAGES

"It is possible by ingenuity and at the expense of clarity ... [to do almost anything in any language]. However, the fact that it is possible to push a pea up a mountain with your nose does not mean that this is a sensible way of getting it there."

—Christopher Strachey
NATO Summer School in Programming

The Intellec Series III system also supports PL/M, FORTRAN, and assembly language programming for iAPX 86,88 and 8080/8085 applications. In the previous chapter we used Pascal-86 for our main control algorithm, but a modular strategy might take advantage of other languages for other modules.

ANOTHER LOOK AT CHOOSING LANGUAGES FOR MODULES

In the best of all possible worlds, would we all speak the same language? Tower of Babel enthusiasts would have us coding our entire program in one language—but which? We do not want to return to the stone age and lose cultural variety and language diversity. There are expressions that can only be expressed in Chinese characters, and there are problem-solving statements that are better expressed in PL/M than in FORTRAN. A good carpenter should have more tools than nails and a hammer; a good programmer should be fluent in several programming languages.

When you design an algorithm, design it using a comfortable language. You will find the algorithm easier to debug, and you will notice the paradigm inherent in your design. When you are ready to translate your algorithm into code for a computer, use the language best suited for the paradigm.

If you have many algorithms that must work together, you should keep the communication among them simple. With simple interfaces, you can code each algorithm in the language best suited for the algorithm. In some cases, you can use an algorithm that has already been developed for use in another application—another reason for keeping your algorithms and interfaces simple. With the Intellec Series III development tools, you can link these algorithms in different configurations to form several applications.

For example, we chose to write our main control algorithm for the climate system in Pascal-86. We still need an algorithm for retrieving the data and converting it to Celsius temperatures. We decided to write a simple Pascal-86 routine for testing purposes only (this routine only retrieves the data from the Series III console); however, our final product will rely on thermocouples and other sources of data, and we will need an algorithm to convert thermocouple voltages to degrees Celsius. Fortunately, we already have a routine in PL/M-86 that performs this activity, and we can save development time and money by using it.

PROGRAMMING IN PL/M-86

PL/M is renowned for its structure and versatility. PL/M is one of the only structured high-level languages that allow you to manipulate bits with AND, OR, and shift (SHR for "shift right" and SHL for "shift left") operations. PL/M's data types are not as strictly enforced as Pascal's—PL/M's BYTE, WORD (ADDRESS), and POINTER types have loose definitions so that you can use them for different kinds of data. For this reason, PL/M is easier to use for system programming (designing computer systems or control mechanisms), yet harder to use in application programming where enforced data typing makes it easier to write error-free programs.

In our climate control system, there are two routines whose paradigms lend themselves easily to PL/M: the routine to get BCD digits from a thermostat device and convert them to a Celsius temperature, and the routine to get voltage data from a thermocouple and convert the voltages to Celsius temperatures. Figure 5-1 shows the algorithm and the actual PL/M-86 code for the routine to retrieve data from a thermostat device. You are already familiar with comments in Pascal programs that occur between the (* and *) symbols; in PL/M, comments occur between the /* and */ symbols.

A PL/M *typed procedure* is like a Pascal or FORTRAN function: it is called in an assignment statement (as in X:=FUNCTION(Y)), and it returns a value (X receives the value of FUNCTION(Y)). In figure 5-1, the typed procedure THERMOSTAT$SETTING$FROM$PORTS returns the value of THERMO$SETTING, which it computes by accessing the two ports HIGH and LOW and converting the BCD digits to a Celsius temperature. The value of THERMO$SETTING is assigned to **ThermostatSetting** in the **GetData** procedure's assignment statement:

```
ThermostatSetting:=THERMOSTAT$SETTING$FROM$PORTS(StatPort1,Stat Port2);
```

```
PLMDATA: DO;

/* This module holds the procedures THERMOSTAT$SETTING$FROM$PORTS,
   TEMP$DATA$FROM$PORTS, and INTERPOLATE (a procedure used by
   TEMP$DATA$FROM$PORTS). These will be used in the final testing
   phase of the climate control system (when prototype hardware is
   available). For intermediate testing (and examples in this book),
   do not use this module; use the dummy GetData module. */

/*
The algorithm for getting a Celsius temperature from a thermostat
device that can send BCD digits to two ports of the 8088:

   The final version of the GetData procedure (to be
   written in Pascal-86) will use this statement to get
   the thermostat reading:

ThermostatSetting:=THERMOSTAT$SETTING$FROM$PORTS(StatPort1,StatPort2);

   A PL/M-86 typed procedure called THERMOSTAT$SETTING$FROM$PORTS
   receives two port numbers from GetData: StatPort1 and StatPort2.
   THERMOSTAT$SETTING$FROM$PORTS must access these ports, convert
   the BCD digits to a Celsius temperature, and return the
   temperature.
```

Figure 5-1. The PL/M-86 Typed Procedure THERMOSTAT$SETTING$FROM$PORTS

```
INPUTS
------

THERMOSTAT$SETTING$FROM$PORTS:

Formal parameters HIGH and LOW receive port numbers as actual
parameters.

Input ports HIGH and LOW: Three BCD digits for the thermostat setting:

                    Port HIGH, bits 3-0: hundred's digit
                    Port LOW, bits 7-4: ten's digit
                    Port LOW, bits 3-0: unit's digit


OUTPUT
------

THERMOSTAT$SETTING$FROM$PORTS: Return WORD with Celsius temperature
*/

/*Here is the typed procedure THERMOSTAT$SETTING$FROM$PORTS:*/

THERMOSTAT$SETTING$FROM$PORTS:
     PROCEDURE (HIGH, LOW) WORD;
        DECLARE (HIGH, LOW) WORD;
        DECLARE (IN$PORT$HIGH, IN$PORT$LOW) BYTE;
        DECLARE THERMO$SETTING WORD;
        DECLARE (HUNDREDS, TENS, UNITS) BYTE;

        IN$PORT$HIGH = INPUT(HIGH);
        IN$PORT$LOW = INPUT(LOW);
        HUNDREDS = IN$PORT$HIGH AND 00001111B;
        TENS = SHR(IN$PORT$LOW, 4);
        UNITS = IN$PORT$LOW AND 00001111B;
        THERMO$SETTING = UNITS + 10*TENS + 100*HUNDREDS;

        RETURN THERMO$SETTING; /*this returns the temperature*/
END THERMOSTAT$SETTING$FROM$PORTS;

.
.
.
More procedures follow--see figure 5-2.
```

Figure 5-1. The PL/M-86 Typed Procedure THERMOSTAT$SETTING$FROM$PORTS (Cont'd.)

There are notable similarities between Pascal and PL/M. Most notable are the logical structures that can occur in both languages—both have the DO WHILE and IF-THEN-ELSE constructs. The languages are lexically and syntactically different in data declarations, procedure headings, and other constructs, but they are logically similar. By conforming to a logical structure, you make your program readable and easier to debug.

The data declarations in both languages are very similar. In both languages, you must declare your data identifiers to be of some type before using the identifiers. In Pascal, you define data types or use predefined Pascal data types. In PL/M, you are restricted to the acceptable PL/M data types, but they are loosely defined. A BYTE can be any value expressed in eight bits, and a WORD in PL/M-86 (ADDRESS in PL/M-80) can be any value expressed in sixteen bits; both types of values are treated as unsigned integers. PL/M-86 offers several more types: INTEGER (for signed integers), REAL (for floating point numbers), and POINTER (for 8086 and 8088 addressing).

PL/M-86 offers many features useful for system programming— arrays and structures, based variables, easy type conversion, built-in procedures for manipulating strings, setting and disabling interrupts, accessing the 8086 or 8088 hardware stack pointer and base registers, and performing bit shift and rotate operations. We use the AND and SHR operators in the THERMOSTAT$SETTING$FROM$PORTS procedure in figure 5-1.

We also use PL/M-86 procedures to obtain temperature data from thermocouple voltage data. The **GetData** procedure in our **GetData** module (written in Pascal-86) calls the PL/M-86 procedure TEMP$DATA$FROM$PORTS to obtain the temperatures **InsideTemp**, **CollectorWaterTemp**, **TankWaterTemp**, and **HeatedTankTemp** using these assignment statements:

```
InsideTemp:=TEMP$DATA$FROM$PORTS(InsidePort1,InsidePort2);
CollectorWaterTemp:=TEMP$DATA$FROM$PORTS(CollectPort1,CollectPort2);
TankWaterTemp:=TEMP$DATA$FROM$PORTS(TankPort1,TankPort2);
HeatedTankTemp:=TEMP$DATA$FROM$PORTS(HotTankPort1,HotTankPort2);
```

In all four assignments, the Pascal-86 identifiers on the left side of the := symbols receive the values from the PL/M-86 typed procedure TEMP$DATA$FROM$PORTS. Figure 5-2 shows TEMP$DATA$FROM$PORTS.

```
.
.
.
/*
The typed procedure TEMP$DATA$FROM$PORTS receives two port numbers:
HIGH and LOW. These ports are accessed for the binary ADC output
from a thermocouple device: HIGH gets the high-order 8 bits, and
LOW gets the low-order 8 bits. TEMP$DATA$FROM$PORTS then uses
the typed procedure INTERPOLATE, a routine that interpolates a
Celsius temperature from a thermocouple voltage using two tables.
TEMP$DATA$FROM$PORTS sends a WORD with the input voltage to
INTERPOLATE. INTERPOLATE returns a Celsius temperature, which
is returned to the GetData procedure (written in Pascal-86).

INPUTS
------

TEMP$DATA$FROM$PORTS:

Formal parameters HIGH and LOW receive port numbers as actual
parameters.

Input port HIGH: Binary ADC output of thermocouple, high-order 8 bits
Input port LOW: Binary ADC output of thermocouple, low-order 8 bits
```

Figure 5-2. The PL/M-86 Typed Procedures TEMP$DATA$FROM$PORTS and INTERPOLATE

```
OUTPUT
------

TEMP$DATA$FROM$PORTS: Return WORD with temperature in Celsius

*/

/* INTERPOLATE is declared first, then its calling procedure
   TEMP$DATA$FROM$PORTS is declared. */

/* INTERPOLATE is a typed procedure that receives thermocouple
   voltage and returns temperature in Celsius using an
   interpolation routine. */

INTERPOLATE:
   PROCEDURE(VOLT$IN) WORD;
      DECLARE VOLTS(*) WORD DATA(0,51,102,154,206,258,365,472);
      DECLARE T$CEL(*) WORD DATA(0,10,20,30,40,50,70,90);
      DECLARE (I, VOLT$IN, NUMERATOR) WORD;

      I = 0;
      IF VOLT$IN=0 THEN RETURN T$CEL(I);
      DO WHILE VOLT$IN > VOLTS(I);
         I = I + 1;
      END;
      /* Shift for rounding, and return Celsius temperature */
      NUMERATOR = SHL((VOLT$IN-VOLTS(I-1))*(T$CEL(I)- T$CEL(I-1)), 1);
      RETURN T$CEL(I-1) + SHR(NUMERATOR/(VOLTS(I)- VOLTS(I-1))+1, 1);

END INTERPOLATE;

/*************************************************************************/
/* TEMP$DATA$FROM$PORTS */

TEMP$DATA$FROM$PORTS:
   PROCEDURE(HIGH,LOW) WORD;
      DECLARE (HIGH,LOW) WORD;
      DECLARE IN$PORT$HIGH WORD;
      DECLARE IN$PORT$LOW BYTE;
      DECLARE (THERMOCOUPLE$OUTPUT, TEMPERATURE) WORD;

      IN$PORT$HIGH = INPUT(HIGH);
      IN$PORT$LOW = INPUT(LOW);
      THERMOCOUPLE$OUTPUT = SHL(IN$PORT$HIGH, 8) OR IN$PORT$LOW;
      TEMPERATURE = INTERPOLATE(THERMOCOUPLE$OUTPUT);
      RETURN TEMPERATURE;

END TEMP$DATA$FROM$PORTS;
```

Figure 5-2. The PLM-86 Typed Procedures TEMP$DATA$FROM$PORTS and INTERPOLATE (Cont'd.)

Let's look closely at the following statement, which appears in the procedure TEMP$DATA$FROM$PORTS shown in figure 5-2:

```
THERMOCOUPLE$OUTPUT = SHL(IN$PORT$HIGH, 8) OR IN$PORT$LOW;
```

The variable IN$PORT$HIGH was declared as a WORD with 16 bits, and the variable IN$PORT$LOW was declared as a BYTE with 8 bits. The thermocouple voltage data from an analog-to-digital converter can have up to 13 bits, but our procedure was originally written to access 8-bit ports. To assemble the 13 bits, we use the SHL (shift to the left) built-in procedure to shift the rightmost 8 bits of IN$PORT$HIGH to the left, and we OR this shifted value with IN$PORT$LOW.

The INTERPOLATE procedure uses a more complicated expression that includes both the SHL (shift to the left) and SHR (shift to the right) built-in procedures. The INTERPOLATE procedure also uses two tables, or *arrays*. They are declared in the following statements:

```
DECLARE VOLTS(*) WORD DATA(0,51,102,154,206,258,365,472);
DECLARE T$CEL(*) WORD DATA(0,10,20,30,40,50,70,90);
```

In both declarations, the arrays VOLTS and T$CEL are assigned actual values through the use of DATA initializations. The DATA initialization allocates storage for the array and assigns the values specified in parentheses after the word DATA in a single step.

The values chosen for the VOLTS array are from the National Bureau of Standards; they represent the output (in millivolts) of type J thermocouples that corresponds to the Celsius temperature range assigned to the T$CEL array. For example, a thermocouple output of 102 millivolts should correspond roughly to 20 degrees Celsius. The INTERPOLATE procedure uses these tables to arrive at an approximate Celsius temperature value for a known thermocouple output value. Our calculations would be more accurate if the ranges between values within each table were smaller.

Compiling a PL/M-86 program is very similar to compiling a Pascal-86 program. We execute the PL/M-86 compiler in the 8086 execution environment by using the Series III RUN command. The PL/M-86 compiler is supplied in the file PLM86.86 on the PL/M-86 disk. We inserted a copy of this disk into drive 1 (we also put our source program on the same disk). In the following example, we execute the PLM86.86 using the RUN command. We do not have to supply the ".86" extension, since RUN already assumes that the file you specify has that extension. The following example shows the RUN command line for compiling the module PLMDATA, which is in a source file called PLMDAT.SRC (PLMDAT.SRC and the PL/M-86 compiler are both in directory :F1:):

```
-RUN :F1:PLM86 :F1:PLMDAT.SRC CODE LARGE<cr>
```

This compiler invocation produces two output files: :F1:PLMDAT.OBJ to hold the compiled object module, and :F1:PLMDAT.LST to hold the listing. The listing is shown in figure 5-3 (in the next section).

Two *compiler controls*, CODE and LARGE, were also specified. The CODE control tells the compiler to list the approximate assembly language instructions that would be necessary to implement the PL/M-86 statements—this is useful for many reasons, some of which are described in the next section. The LARGE control is needed here because Pascal-86 modules are compiled with a default size control that is equivalent to the PL/M-86 LARGE model. All modules of a program must conform to the same size control, so our PL/M-86 module must be compiled as a LARGE module to conform to the default size of Pascal-86 modules.

You need to know more about the PL/M-86 language and compiler than the brief introduction provided in this guide. Intel provides a manual for the PL/M-86 language and compiler (*PL/M-86 User's Guide for 8086-Based Development Systems*). Intel also supplies two manuals for PL/M-80 program development (*PL/M-80 Programming Manual* and *ISIS-II PL/M-80 Compiler Operator's Manual*) in the 8085 execution environment of the Series III. For tutorial information on the PL/M language, see *A Guide to PL/M Programming For Microcomputer Applications* by Daniel McCracken (listed in the Bibliography).

PROGRAMMING IN 8086/8087/8088 ASSEMBLY LANGUAGE

So far we have described high-level languages that are translated by compilers into machine code; namely, Pascal-86 and PL/M-86. Another high-level language not described in this book is FORTRAN, which is also translated by a compiler into machine code. Other high-level languages like BASIC-80 are translated by interpreters into machine code.

An *assembly language* program is translated into machine code by an *assembler*. Intel provides the 8086/8087/8088 Macro Assembler, which is described in this section, to translate 8086/8087/8088 Assembly Language programs. It is called a macro assembler because it will also translate *macro definitions* written in the Macro Processor Language, which is described with the 8086/8087/8088 Assembly Language.

The common denominator of all these languages is the *machine code*, which is the binary language of ones and zeros that only the processor can "speak" well. The following is an example of machine code, with comments to explain what action each code performs:

Memory Address (Hexadecimal)	Memory Contents (Binary)	Comments (English)
00000	11100101	Read word into reg. AX...
00001	00000101	...from input port 5.
00002	01000000	Increment contents of AX.
00003	11100111	Write word from reg. AX...
00004	00000010	...to output port 2.
00005	11101011	Repeat by jumping...
00006	11111001	...back seven bytes.
00007	...	

This machine code (sometimes called *object code*) is the code that the processor executes. All languages are eventually translated into this type of code.

A program written in assembly language is a symbolic representation of machine code. The relation between assembly language instructions and the resulting machine code is usually very obvious; the relation between statements in a high-level language and the resulting machine code is often not obvious (with some exceptions in the PL/M-86 language).

Assembly language gives you complete control over the resulting machine code and thereby allows you to generate very efficient machine code. There are times when this control is desirable, and other times when you want to be free of such details. Most high-level language compilers are efficient enough for microcomputer applications; in fact, some compilers are more efficient than most humans.

Assembly language is the closest language to machine code, but it does allow you to use symbolic names. Here is a rewrite of the machine code instructions shown before, using 8086/8087/8088 Assembly Language (comments follow the semicolons):

```
CYCLE:    IN      AX,5        ;Read word from port 5 into reg. AX.
          INC     AX          ;Increment the contents of reg. AX.
          OUT     2,AX        ;Write word from reg. AX to port 2.
          JMP     CYCLE       ;Jump to beginning and repeat.
```

This program fragment is simpler to read because it uses symbolic names like CYCLE instead of binary and hexadecimal numbers. The 8086/8087/8088 Assembly Language also provides sophisticated code and data structuring mechanisms usually found only in high-level languages. The assembler enforces some consistency in data types to prevent inadvertent errors, yet it also allows some deliberate ways to override data types.

The 8086/8087/8088 Macro Assembler also processes macro definitions. A *macro* is a shorthand function name for a string of instructions. First you define a macro, using the Macro Processing Language, to be a sequence of assembly language instructions. Once defined, you can specify the macro name in an assembly language program, and the macro assembler automatically replaces the macro name with the actual sequence of instructions from the definition. Using this facility you can create many macros and use them in many programs.

There are many times when an assembly language version of a routine runs faster and takes up less space than a high-level language version. Intel's compilers can produce a listing of the assembly language instructions that are approximately the ones you would use to implement the compiled high-level language routine in assembly language. For example, we used the CODE control in the previous section when we compiled the PL/M-86 program PLMDAT.SRC. The CODE control produced the listing shown in figure 5-3.

PROGRAMMING FOR THE SERIES III ENVIRONMENT

Assembly language and PL/M give you more direct control over the processor's operation; however, the Series III system gives you a set of operating system procedures called *primitives* (or *service routines*) that your programs can use to perform standard operations. By using these primitives, you save development time in two ways: first, you save time by not implementing the standard operations yourself; and second, you save time in the future by writing programs that will always be compatible with future Intel operating systems.

In Pascal-86, you are supplied with built-in procedures that automatically call these primitives. You link the run-time libraries that contain the primitives to your Pascal-86 modules, and you're all set to run the program on a Series III system. The program will also run on future Intel operating systems, since the only changes it would need would be contained in the run-time libraries; that is, you would have a different set of run-time libraries for each system, but your basic program modules would remain unchanged.

To have this modularity in PL/M or assembly language, you simply use the set of primitives described in the *Intellec Series III Microcomputer Development System Programmer's Reference Manual*, and then link in the appropriate system libraries to your PL/M or assembly language modules (as described in the *iAPX 86,88 Family Utilities User's Guide*). For future Intel operating systems, you only need to use a different set of system libraries. Your basic program modules would remain unchanged.

The next chapter describes the linking and locating operations for our Pascal-86 modules.

```
PL/M-86 COMPILER    PLMDATA                                          PAGE   1


ISIS-II PL/M-86 V121 COMPILATION OF MODULE PLMDATA
OBJECT MODULE PLACED IN PLMEX.OBJ
COMPILER INVOKED BY:  PLM86 PLMEX.SRC CODE


   1          PLMDATA: DO;

              /*
              INPUTS
              ------

              THERMOSTAT$SETTING$FROM$PORTS:

              Formal parameters HIGH and LOW receive port numbers as actual parameters.

              Input ports HIGH and LOW: Three BCD digits for the thermostat setting:

                                  Port HIGH, bits 3-0: hundred's digit
                                  Port LOW, bits 7-4: ten's digit
                                  Port LOW, bits 3-0: unit's digit


              TEMP$DATA$FROM$PORTS:

              Formal parameters HIGH and LOW receive port numbers as actual parameters.

              Input port HIGH: Binary ADC output of thermocouple, high-order 8 bits
              Input port LOW: Binary ADC output of thermocouple, low-order 8 bits


              OUTPUTS
              -------

              THERMOSTAT$SETTING$FROM$PORTS:  Return WORD with setting in Celsius

              TEMP$DATA$FROM$PORTS:  Return WORD with temperature in Celsius

              */

   2    1     THERMOSTAT$SETTING$FROM$PORTS:
                                                        ; STATEMENT # 2
                        THERMOSTATSETTINGFROMPORTS      PROC NEAR
            0000  55                PUSH    BP
            0001  8BEC              MOV     BP,SP
              PROCEDURE (HIGH, LOW) WORD;
   3    2        DECLARE (HIGH, LOW) WORD;
   4    2        DECLARE (IN$PORT$HIGH, IN$PORT$LOW) BYTE;
   5    2        DECLARE THERMO$SETTING WORD;
   6    2        DECLARE (HUNDREDS, TENS, UNITS) BYTE;

   7    2        IN$PORT$HIGH = INPUT(HIGH);
                                                        ; STATEMENT # 7
            0003  8B5606            MOV     DX,[BP].HIGH
            0006  EC                IN      DX
            0007  88060C00          MOV     INPORTHIGH,AL
   8    2        IN$PORT$LOW = INPUT(LOW);
                                                        ; STATEMENT # 8
            0008  8B5604            MOV     DX,[BP].LOW
            000E  EC                IN      DX
            000F  88060D00          MOV     INPORTLOW,AL
   9    2        HUNDREDS = IN$PORT$HIGH AND 00001111B;
                                                        ; STATEMENT # 9
            0013  8A060C00          MOV     AL,INPORTHIGH
            0017  80E00F            AND     AL,0FH
            001A  88060E00          MOV     HUNDREDS,AL
  10    2        TENS = SHR(IN$PORT$LOW, 4);
                                                        ; STATEMENT # 10
            001E  8A060D00          MOV     AL,INPORTLOW
            0022  B104              MOV     CL,4H
            0024  D2E8              SHR     AL,CL
            0026  88060F00          MOV     TENS,AL
  11    2        UNITS = IN$PORT$LOW AND 00001111B;
                                                        ; STATEMENT # 11
            002A  8A060D00          MOV     AL,INPORTLOW
            002E  80E00F            AND     AL,0FH
            0031  88061000          MOV     UNITS,AL
```

Figure 5-3. Listing of PLMDATA with the CODE Control

```
12  2       THERMOSSETTING = UNITS + 10*TENS + 100*HUNDREDS;
                                                ; STATEMENT # 12

        0035  8A060F00        MOV     AL,TENS
        0039  B10A            MOV     CL,0AH
        003B  F6E1            MUL     CL
        003D  8A0E1000        MOV     CL,UNITS
        0041  8500            MOV     CH,0H
        0043  03C1            ADD     AX,CX
        0045  50              PUSH    AX      ; 1
        0046  8A060E00        MOV     AL,HUNDREDS
        004A  B164            MOV     CL,64H
        004C  F6E1            MUL     CL
        004E  59              POP     CX      ; 1
        004F  03C1            ADD     AX,CX
        0051  89060000        MOV     THERMOSETTING,AX
13  2       RETURN THERMOSSETTING;
                                                ; STATEMENT # 13
        0055  5D              POP     BP
        0056  C20400          RET     4H
14  2   END THERMOSTATSSETTINGSFROMSPORTS;
                                                ; STATEMENT # 14
                    THERMOSTATSETTINGFROMPORTS      ENDP


      /* Another typed procedure to return temperature data, which
         uses the INTERPOLATE typed procedure. */

      /* INTERPOLATE is a typed procedure that receives thermocouple
         voltage and returns temperature in Celsius using an
         interpolation routine */

15  1   INTERPOLATE:
                                                ; STATEMENT # 15
                INTERPOLATE     PROC NEAR
        0059  55              PUSH    BP
        005A  8BEC            MOV     BP,SP
    .   PROCEDURE(VOLTSIN) WORD;
16  2       DECLARE VOLTS(*) WORD DATA(0,51,102,154,206,258,365,472);
17  2       DECLARE TSCEL(*) WORD DATA(0,10,20,30,40,50,70,90);
18  2       DECLARE (I, VOLTSIN, NUMERATOR) WORD;

19  2       I = 0;
                                                ; STATEMENT # 19
        005C  C706020000000   MOV     I,0H
20  2       IF VOLTSIN<1 THEN RETURN TSCEL(I);
                                                ; STATEMENT # 20
        0062  817E040100      CMP     [BP].VOLTIN,1H
        0067  7203            JB      $+5H
        0069  E90300          JMP     @1
                                                ; STATEMENT # 21
        006C  8B0000          MOV     BX,0H
        006F  D1E3            SHL     BX,1
        0071  8B871000        MOV     AX,TCEL[BX]
        0075  5D              POP     BP
        0076  C20200          RET     2H
                @1:
22  2       DO WHILE VOLTSIN > VOLTS(I);
                                                ; STATEMENT # 22
                @2:
        0079  8B1E0200        MOV     BX,I
        007D  D1E3            SHL     BX,1
        007F  8B4604          MOV     AX,[BP].VOLTIN
        0082  3B870000        CMP     AX,VOLTS[BX]
        0086  7703            JA      $+5H
        0088  E90700          JMP     @3
23  3           I = I + 1;
                                                ; STATEMENT # 23
        008B  FF060200        INC     I
24  3       END;
                                                ; STATEMENT # 24
        008F  E9E7FF          JMP     @2
                @3:

      /* Shift for rounding, and return Celsius temperature */
25  2       NUMERATOR = SHL((VOLTSIN-VOLTS(I-1))*(TSCEL(I)-TSCEL(I-1)), 1);
                                                ; STATEMENT # 25
        0092  8B1E0200        MOV     BX,I
        0096  4B              DEC     BX
        0097  D1E3            SHL     BX,1
        0099  8B870000        MOV     AX,VOLTS[BX]
        009D  8B4E04          MOV     CX,[BP].VOLTIN
        00A0  2BC8            SUB     CX,AX
        00A2  8B360200        MOV     SI,I
        00A6  D1E6            SHL     SI,1
```

Figure 5-3. Listing of PLMDATA with the CODE Control (Cont'd.)

```
                00A8  8B971000          MOV     DX,TCEL[BX]
                00AC  8B8C1000          MOV     BX,TCEL[SI]
                00B0  2BDA              SUB     BX,DX
                00B2  50                PUSH    AX      ; 1
                00B3  89C8              MOV     AX,CX
                00B5  52                PUSH    DX      ; 2
                00B6  F7E3              MUL     BX
                00B8  D1E0              SHL     AX,1

                00BA  89060400          MOV     NUMERATOR,AX
  26    2           RETURN T$CEL(I-1) + SHR(NUMERATOR/(VOLTS(I)-VOLTS(I-1))+1, 1);
                                                ; STATEMENT # 26

                00BE  8B8C0000          MOV     BX,VOLTS[SI]
                00C2  5A                POP     DX      ; 2
                00C3  59                POP     CX      ; 1
                00C4  2BD9              SUB     BX,CX
                00C6  52                PUSH    DX      ; 1
                00C7  31D2              XOR     DX,DX
                00C9  F7F3              DIV     BX
                00CB  40                INC     AX
                00CC  D1E8              SHR     AX,1
                00CE  59                POP     CX      ; 1
                00CF  03C1              ADD     AX,CX
                00D1  5D                POP     BP
                00D2  C20200            RET     2H
  27    2       END INTERPOLATE;
                                                ; STATEMENT # 27
                        INTERPOLATE     ENDP


  28    1       TEMP$DATA$FROM$PORTS:
                                                ; STATEMENT # 28
                        TEMPDATAFROMPORTS       PROC NEAR
                00D5  55                PUSH    BP
                00D6  8BEC              MOV     BP,SP
             PROCEDURE(HIGH,LOW) WORD;
  29    2           DECLARE (HIGH,LOW) WORD;
  30    2           DECLARE IN$PORT$HIGH WORD; /* ADDRESS in McCracken's book */
  31    2           DECLARE IN$PORT$LOW BYTE;
  32    2           DECLARE (THERMOCOUPLE$OUTPUT, TEMPERATURE) WORD;

  33    2           IN$PORT$HIGH = INPUT(HIGH);
                                                ; STATEMENT # 33
                00D8  8B5606            MOV     DX,[BP].HIGH
                00DB  EC                IN      DX
                00DC  8A00              MOV     AH,0H
                00DE  89060600          MOV     INPORTHIGH,AX
  34    2           IN$PORT$LOW = INPUT(LOW);
                                                ; STATEMENT # 34
                00E2  8B5604            MOV     DX,[BP].LOW
                00E5  EC                IN      DX
                00E6  88061100          MOV     INPORTLOW,AL
  35    2           THERMOCOUPLE$OUTPUT = SHL(IN$PORT$HIGH, 8) OR IN$PORT$LOW;
                                                ; STATEMENT # 35
                00EA  8B060600          MOV     AX,INPORTHIGH
                00EE  B108              MOV     CL,8H
                00F0  D3E0              SHL     AX,CL
                00F2  8A0E1100          MOV     CL,INPORTLOW
                00F6  B500              MOV     CH,0H
                00F8  0BC1              OR      AX,CX
                00FA  89060800          MOV     THERMOCOUPLEOUTPUT,AX
  36    2           TEMPERATURE = INTERPOLATE(THERMOCOUPLE$OUTPUT);
                                                ; STATEMENT # 36
                00FE  50                PUSH    AX      ; 1
                00FF  E857FF            CALL    INTERPOLATE
                0102  89060A00          MOV     TEMPERATURE,AX
  37    2           RETURN TEMPERATURE;
                                                ; STATEMENT # 37

                0106  8B060A00          MOV     AX,TEMPERATURE
                010A  5D                POP     BP
                010B  C20400            RET     4H
  38    2       END TEMP$DATA$FROM$PORTS;
                                                ; STATEMENT # 38
                        TEMPDATAFROMPORTS       ENDP

  39    1       END PLMDATA;
                                                ; STATEMENT # 39


MODULE INFORMATION:

     CODE AREA SIZE     = 010EH    2700
     CONSTANT AREA SIZE = 0020H     320
     VARIABLE AREA SIZE = 0012H     180
     MAXIMUM STACK SIZE = 0010H     160
     95 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

Figure 5-3. Listing of PLMDATA with the CODE Control (Cont'd.)

CHAPTER 6
USING UTILITIES TO PREPARE EXECUTABLE PROGRAMS

"Three things are to be looked to in a building: that it stand on the right spot; that it be securely founded; that it be successfully executed."

—Johann Wolfgang Von Goethe

You must do three things to prepare and execute programs successfully: link program modules to resolve external references, locate the linked modules by binding them to memory addresses, and run the program in the appropriate operating environment.

These things are easy to do for most high-level language programs. Easy-to-use *utility programs* perform these operations for you. They are also flexible enough to allow you to perform more complicated linking and locating operations for programs that refer to physical memory addresses. The compilers for high-level languages usually produce programs that do not refer to physical memory addresses; these programs can be linked and located in one easy step.

The diagram in figure 6-1 shows the process of linking and locating (binding to addresses) modules to prepare a program that can be RUN on the Series III system (or debugged via DEBUG-86, described in Chapter 7).

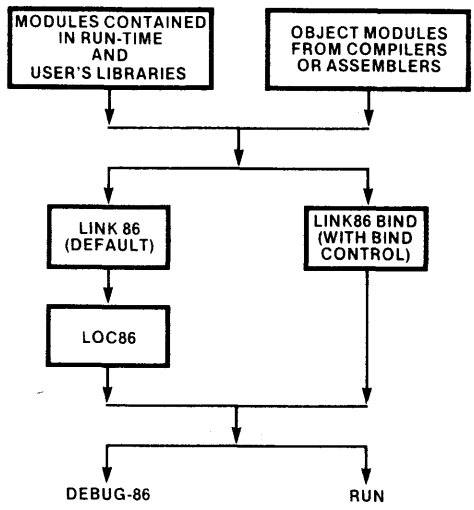


Figure 6-1. Using Utilities to Prepare Executable Programs 121632-6

CHAPTER 6

PREPARING A LIBRARY OF PROGRAM MODULES

A *library* is any collection of public modules—modules that contain public procedures that can be used by programs. Some libraries are supplied by Intel; for example, the *run-time system* used with Pascal-86 programs is supplied as several library files. Pascal-86 has a number of built-in procedures that you can use in any Pascal-86 program—these procedures can be found in the supplied run-time libraries. To use any of the built-in procedures, you must link the run-time libraries to your program modules using the LINK86 utility.

You can also build your own library files using the LIB86 utility. With the LIB86 utility you can create a library, add modules from another library, add new modules, delete modules in a library, and list the names of modules in a library.

The following example shows a session with LIB86. We create a new library called TONY.LIB, and then we ADD to it some of the modules from a supplied run-time library called P86RN2.LIB. We then LIST the modules in TONY.LIB, and EXIT from the LIB86 utility:

```
-RUN LIB86<cr>
SERIES-III 8086 LIBRARIAN V1.0
*CREATE :F1:TONY.LIB<cr>
*ADD :F1:P86RN2.LIB(MOD1, MOD4, MOD7) TO :F1:TONY.LIB<cr>
*LIST :F1:TONY.LIB<cr>
TONY.LIB
     MOD1
     MOD4
     MOD7
*EXIT<cr>
-
```

The linker (LINK86) treats library files in a special way. As shown in the next section, you specify your program modules first in the LINK86 command line, then you specify the appropriate libraries. You must be sure to link these modules in the proper sequence.

Why? Remember that your program's main module refers to procedures that exist only in other modules—*external procedures*. The linker must be able to find the external procedures. LINK86 remembers the references to external procedures in the first module, and looks in the subsequent modules for those external procedures. If it cannot find the external procedures in subsequent modules (maybe because you erroneously specified the library *before* specifying the program modules in the LINK86 command line), LINK86 will generate an error message.

The built-in procedures supplied with Pascal-86 (READLN, WRITELN, etc.) are external procedures contained in the modules included with the run-time support libraries. LINK86 will first see the reference to these procedures in your program modules, and then it will look in the libraries for the modules that will satisfy those references. LINK86 will only link in those modules that are needed to satisfy external references; it will not link in the entire library of modules unless your program modules refer to all of the library modules.

Since you do not need to use LIB86 to handle run-time libraries supplied with Pascal-86, you only need LIB86 to handle your own libraries. Why would you set up your own libraries? To manage sets of repetitive modules. In many software development labs, modules useful to many different programs would either be lost or repeated. Libraries are sets of modules that are easily maintained through use of the LIB86 librarian. The LINK86 utility is capable of searching such a library and only checking out the modules needed for the linked program.

LINKING MODULES TO FORM A LOCATABLE PROGRAM

Most modular programs have a main module that calls procedures in other subordinate modules. Although a subordinate module can call a procedure in the main module, most calls are from the main module to a subordinate module, and the modular structure resembles an upside-down tree, as shown in figure 6-2. We included the run-time system modules in this tree, since the program modules rely on the built-in procedures and operating environment calls found in the run-time system.

When you link these modules together using the LINK86 utility, you allow LINK86 to see the main module first, because the main module is the most abstract; that is, it has the highest level of abstraction, and it calls procedures in lower levels to perform each activity. You should then allow LINK86 to see the next subordinate level of modules, and so on.

The *last* group of modules for LINK86 should be any run-time system libraries that are needed to perform the built-in procedures (READLN, WRITELN, etc.). The run-time system libraries contain modules that are at the lowest level of abstraction—these are the modules that call procedures in the operating environment of your system (the operating environment is usually invisible to you, but not to your Pascal program).

For an example, we will link together the modules needed to test our main program in the Series III environment. We start with our main module MAIN.OBJ, which holds the **MainControl** module. We link to it the test versions of the modules **GetData** found in DUMDAT.OBJ, and **Operation** found in DUMOP.OBJ. Finally, we link in the modules we need from the run-time system libraries P86RN0.LIB, P86RN1.LIB, P86RN2.LIB, P86RN3.LIB, 87NULL.LIB, and LARGE.LIB (we explain these libraries after the example):

```
-RUN LINK86 :F1:MAIN.OBJ,:F1:DUMDAT.OBJ,:F1:DUMOP.OBJ,&<cr>
>>:F1:P86RN0.LIB,:F1:P86RN1.LIB,:F1:P86RN2.LIB,&<cr>
>>:F1:P86RN 3.LIB,:F1:87NULL.LIB,:F1:LARGE.LIB&<cr>
>>TO :F1:PROGRM.86 BIND<cr>
```

Let's explain this example. We used the RUN command to run the LINK86.86 utility in the "8086 side" of the Series III. We specified the three object modules of our program, and then we specified all of the run-time libraries needed to run our program in the Series III environment.

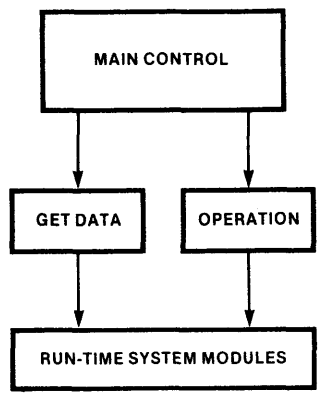


Figure 6-2. Main Module with Subordinate Modules 121632-7

We directed the output to a file called PROGRM.86. Finally, we specified the BIND control in order to make PROGRM.86 an LTL (load-time locatable) program. The BIND control is the easiest way to make a program locatable in the Series III environment (more on BIND in the next section). PROGRM.86 can now be loaded and executed (via the RUN command) in the Series III system.

The run-time system libraries P86RN0.LIB and P86RN1.LIB are required for any Pascal-86 program that needs run-time support. The libraries P86RN2.LIB and P86RN3.LIB are required for any Pascal-86 program that uses the file input/output procedures and other operations performed by the operating system. The 87NULL.LIB is needed for programs that do *not* use either the 8087 processor or emulator to perform real arithmetic. Since our program has no REAL data types, it does not perform real arithmetic; therefore, it needs the 87NULL.LIB library (if it did perform real arithmetic, it would need other libraries). Finally, to run any Pascal-86 program on the Series III system, you need to link the LARGE.LIB library to the program. The LARGE.LIB library contains the primitives (service routines) used to perform operations in the Series III environment.

The run-time system is separated into several libraries so that you can customize your run-time environment if you so wish. The libraries described above are the default libraries used to run programs on the Series III system, if your programs do not use REAL data types. If your programs use REAL data types, you would not use 87NULL.LIB; instead, you would use the 8087 processor with the library 8087.LIB, or the 8087 software emulator with the libraries E8087 and E8087.LIB. Consult the *Pascal-86 User's Guide* for specific information about the run-time system libraries.

LOCATING AND RUNNING PROGRAMS

A program must reside in actual memory before it can run. The *locating* process assigns actual (physical) memory addresses to a program. There are two ways to accomplish the locating process:

1. Using LINK86 with the BIND control to produce an LTL module (discussed in the next paragraph), which can be located, loaded, and executed automatically by the RUN command
2. Using LINK86 (without the BIND control) to produce a linked module, then LOC86 to locate the module in an area of memory you specify, and finally RUN to load and execute the program

The simplest locating operation involves using the LINK86 utility with the BIND control, as shown in the previous example. This simpler process, called *binding*, binds modules to logical segments, which can be located in actual memory by the RUN command in one fast step. Modules produced by the LINK86 utility with the BIND control are called *load-time locatable* modules (LTL modules). An LTL module is a module that can be located almost anywhere in memory, and so the RUN loader can easily locate it in the Series III environment for you. The DEBUG-86 debugger can also locate an LTL module for you, as we will show in the next chapter.

For example, our LINK86 example in the previous section bound the modules properly to form the program PROGRM.86. Now, in one step, you can locate this bound program in actual memory, load it into memory, and execute it in the "8086 side" (8086 execution mode) of the Series III system by using the RUN command:

```
-RUN :F1:PROGRM<cr>
```

Note that we did not type PROGRM.86, only PROGRM. The RUN loader looks for the ".86" extension automatically, unless you specify another extension or a period at the end of the name (the period signifies no extension).

High-level language programmers usually do not burden themselves with more details about locating programs. However, assembly language programs and some PL/M programs frequently refer to physical addresses rather than symbolic (logical) addresses. These program modules are called *absolute* modules because they use absolute physical addresses. Absolute modules cannot be located automatically by the RUN command—they must be relocated first by the LOC86 utility.

There is a case when even the simplest program must be located by LOC86: if you intend to debug your program using an ICE-86 or ICE-88 emulator, you must locate the program with LOC86 to make it an absolute module. The ICE (In-Circuit Emulation) loaders can only load absolute modules.

For an example, we will link our new PL/M-86 module PLMDAT.OBJ to our program, along with another version of our **GetData** module in DATA.OBJ, and produce the linked module MAIN.LNK:

```
-RUN LINK86 :F1:MAIN.OBJ,:F1:DATA.OBJ,:F1:PLMDAT.OBJ,&<cr>
>>:F1:DUMOP.OBJ,:F1:P86RN0.LIB,:F1:P86RN1.LIB,:F1:P86RN2.LIB,&< cr>
>>:F1:P86RN3.LIB,:F1:87NULL.LIB,:F1:LARGE.LIB<cr>
```

Since we did not specify a new filename with a "TO" clause, the LINK86 utility directed the linked output to the file :F1:MAIN.LNK (LINK86 takes the name of the first object module MAIN.OBJ, and changes its extension to LNK to make MAIN.LNK). Now we are ready to locate :F1:MAIN.LNK with the LOC86 utility:

```
-RUN LOC86 :F1:MAIN.LNK TO :F1:PROGRM.86 RESERVE(200H TO 77FFH)<cr>
```

We used the RESERVE control with LOC86 to reserve an area of memory for the Series III operating system. LOC86 will not locate any program segments in the area between addresses 200H and 77FFH ("H" is for hexadecimal). You must leave room for the Series III operating system to execute programs in the Series III environment.

The LINK86, LOC86, and LIB86 utilities are described in detail in the *iAPX 86,88 Family Utilities User's Guide for 8086-Based Development Systems*. This manual describes all of the utilities for iAPX 86 or iAPX 88 applications development. These utilities are designed to be used in 8086-based development systems like the Series III.

We can now RUN our program on the Series III system:

```
-RUN PROGRM<cr>

Climate system is now on.
-------------------------
Type the thermostat setting in degrees Celsius:24<cr>

Type the inside temperature reading in Celsius:21<cr>

Type the temperature of the collector water in Celsius:50<cr>

Type the temperature of the tank water in Celsius:60<cr>
```

```
Type the temperature of the heated tank water in Celsius:70<cr>

Type the hour of day, as in 04 or 24:13<cr>

Type the minute of the hour, as in 01 or 59:25<cr>

Type the amount of sunlight, any integer will do for now:1<cr>

--------------------------
CURRENT DATA IS AS FOLLOWS:
--------------------------
Thermostat Setting is 24C
Inside temperature is 21C
Temperature of collector water is 60C
Temperature of tank water is 60C
Temperature of the heated tank water is 70C
Time of day is 13:25
Amount of sunlight is 1

==============================================
The Climate System is now operating.

The time is 13:25
The inside temperature is 21C
The thermostat setting is 24C
Method chosen to heat the building: Solar Collector to Exchanger
==============================================

::::::::::::::::::::::::::::::::::::::::::::::::::
No panic occurred, ABNORMAL shutdown.
::::::::::::::::::::::::::::::::::::::::::::::::::
Last chosen heating method was: Solar Collector to Exchanger
Thermostat Setting is 24C
Inside temperature is 21C
Temperature of collector water is 60C
Temperature of tank water is 60C
Temperature of the heated tank water is 70C
Time of day is 13:25
Amount of sunlight is 1
::::::::::::::::::::::::::::::::::::::::::::::::::

Goodnight, Irene...
-
```

What happened!? Our program stopped with an abnormal shutdown! It had no trouble picking the right heating method; however, something caused the variable **Operating** to become FALSE, without setting **Panic** to TRUE. We will have to debug the program in the next chapter...

CHAPTER 7
DEBUGGING AND EXECUTING PROGRAMS

"Appearances are often deceiving." —Aesop

We have written program modules and they compiled correctly. We used the LINK86 utility with BIND to link the modules into a program and prepare it for execution. By all appearances, it looks like it will work as planned.

However, when we loaded and executed (via the RUN command) the program, something mysterious happened—the "hidden glitch" struck!

We now have to *debug* our program—find all of the "hidden glitches" and fix them. Debugging is so essential to programming that in most software development efforts, more time is alotted to debugging than to any other activity.

Debuggers are programs that monitor a program's execution and allow you to stop execution and check details. Usually a debugger is part of an *execution vehicle*—a piece of hardware your program executes on, or a piece of software your program executes in. The debugger monitors the activity in the execution environment.

For example, the Series III provides two execution environments or execution vehicles: the 8085 execution mode and the 8086 execution mode. To debug programs that run in the 8085 execution mode, you type the DEBUG command on the "8085 side" (8085 execution mode) to start the Monitor. To debug programs that run in the 8086 execution mode, you RUN the DEBUG command on the "8086 side" (8086 execution mode) to start the DEBUG-86 debugger. Both debuggers reside in ROM.

With a debugger you can load and execute a program (rather than using RUN), and stop the execution to check the values of variables, the contents of registers, and other details. You can also change such values, and perform other activities that monitor a program's execution. You can even execute your program one step at a time.

Since the Monitor (for 8085 execution mode) is described in several Intel documents (McCracken's *Guide to Intellec Microcomputer Development Systems* and the *Intellec Series III Microcomputer Development System Console Operating Instructions*), we will devote this chapter to describing DEBUG-86 and the ICE-88 emulator (a version of the ICE-86 emulator) for debugging programs in the 8086 execution mode for iAPX 86, 88 applications.

CHAPTER 7

USING DEBUG-86 FOR SYMBOLIC DEBUGGING

Debuggers are useful because they load and execute a program for you, and they allow you to stop execution at any point you specify. If you are an assembly language programmer, your program directly uses registers and memory locations; you would want to check the contents of these registers and memory locations. If you are a high-level language programmer (PL/M, Pascal, FORTRAN, etc.), you do not directly refer to registers and memory locations; you would rather check the values of your variables, or *symbols*.

For example, we know we have a problem with our program: the variable **Operating** should always have the boolean value of TRUE as long as **Panic** is FALSE. If **Panic** is set to TRUE, **Operating** should then become FALSE; however, **Operating** is somehow set to FALSE while **Panic** remains FALSE. During the execution of our program, we should be able to stop execution and check the value of **Operating**.

One way to do this is to find the address of **Operating** and check its contents. A better way would be to simply ask for the value of **Operating** in a command such as:

```
*BOOLEAN BYTE ..MAINCONTROL.OPERATING<cr>
FALSE
```

In DEBUG-86, you can execute our program and stop its execution anywhere. You can then use the above command to find the boolean (true or false) value of **Operating** without knowing where **Operating** is located.

In our program there is only one symbol named **Operating**. If there were more than one, we would specify the symbol by specifying both the module name and the symbol name (e.g., ..MAINCONTROL.OPERATING), as we did in the above example. We would have obtained the same result by specifying only ".OPERATING". You use two periods as a prefix to module names, and one period as a prefix to symbol names.

To do symbolic debugging using the symbols in your program, you must compile your program using the DEBUG compiler control (see Chapter 4 for the Pascal-86 compiler, and Chapter 5 for the PL/M-86 compiler). The DEBUG compiler control produces a symbol table for your program, which can be loaded by DEBUG-86. If you do not use the DEBUG compiler control while compiling, you can still do symbolic debugging by defining all of your symbols from within DEBUG-86.

To invoke DEBUG-86, use the following command:

```
-RUN DEBUG<cr>
DEBUG 8086, V1.0
*
```

DEBUG-86 is now in control, as shown by the asterisk (*) prompt. You can now type DEBUG-86 commands. To load PROGRM.86 and its symbol table, type the following command:

```
*LOAD :F1:PROGRM.86<cr>
```

You can check the symbol table by typing the SYMBOLS command:

```
*SYMBOLS<cr>
.
.
.
```

The symbol table contains each module name and symbol name, with their addresses. You can find the address of a single symbol by typing the name of the symbol. For example, we want to know the address of the call (in module **MainControl**) to the procedure **OperateSystem** (which is in the module **Operation**):

```
*..MAINCONTROL.OPERATESYSTEM<cr>
..MAINCONTROL.OPERATESYSTEM=0481:06CCH
```

DEBUG-86 knows where to start executing the program by looking at the contents of the CS and IP (CS:IP) registers. The CS register holds the starting address of the program's code segment, and the IP register holds the address within the code segment where the program should start execution. CS and IP change as the program executes; therefore, to preserve the starting location of the program, we create a new symbol called .START to hold the starting address:

```
*DEFINE .START=CS:IP<cr>
```

We could also check the contents of CS:IP:

```
*CS<cr>
CS=0481H
*IP<cr>
IP=0FD6H
```

You should familiarize yourself with addresses. The 8086 and 8088 processors use 20-bit physical addresses separated into two words: the segment base address and the offset value. The CS register holds the code segment base address, and the IP register holds the offset value. All addresses are in hexadecimal notation (the "H" stands for hexadecimal). All addresses are displayed with the segment base, followed by a colon, followed by the offset.

The EVALUATE (abbreviated "EVA") command is useful for evaluating numeric and character values and addresses. For example, the number 4142H can be represented in several ways:

```
*EVA 4142<cr>
1000001010000Y  40502Q   16706T  4142H  'AB'
```

When you type a number by itself, DEBUG-86 assumes the number is in hexadecimal notation. The number 4142H is equivalent to the decimal number 16706 ("T" denotes decimal notation), the octal number 40502Q ("Q" denotes octal notation), the binary number 1000001010000Y ("Y" denotes binary notation), and the ASCII characters "AB" (41H is the ASCII code for "A" and 42H is the ASCII code for "B").

The EVA command will also find the closest symbol that has the address you specify. The keyword SYM tells the EVA command to evaluate the address symbolically:

```
*EVA 481:6CC SYM<cr>
..MAINCONTROL.OPERATESYSTEM
```

We will now execute our program and stop its execution before it executes the **OperateSystem** procedure. The GO command will start executing at the beginning of the program, which it knows by looking at the CS and IP registers (CS:IP). You could specify an actual address with GO to start from, or the line number of a program statement, or a statement label; however, we do not use statement labels in our program. We want our program to stop while it is executing the main (WHILE **Operating**) loop, *not* at the start of the **OperateSystem** proced e in the

Operation module (since procedure definitions do not actually execute). Our program will GO until it reaches the starting address, in the **MainControl** module, of the call to the **OperateSystem** procedure:

```
*GO TILL ..MAINCONTROL.OPERATESYSTEM<cr>
Climate System is now on.
-------------------------
.
.                    During execution, we type in the
.                    temperatures and other data. The program
                     stops and DEBUG-86 displays the next
                     instruction to be executed (in assembly
                     language form).

0481:06CCH          PUSH        BP
**
```

At this point, we can check the boolean values of both **Panic** and **Operating**:

```
**BOOL BYTE .PANIC<cr>
FALSE
*BOOL BYTE .OPERATING<cr>
FALSE
```

To make sure the program stopped at the right place, we evaluate the CS and IP registers symbolically:

```
*EVA CS:IP SYM<cr>
..MAINCONTROL.OPERATESYSTEM
```

Another useful DEBUG-86 command is the STEP command. From any point of program execution, you can execute the program step by step—one machine instruction at a time. The STEP command also displays the next machine instruction to be executed. This display is in "disassembled" form—the machine instruction is translated back into assembly language:

```
*STEP<cr>
0481:06CDH              MOV       BP,SP
*
```

STEP executes one machine instruction and displays the next one disassembled. Another STEP would execute the displayed instruction.

We will now change the value of **Operating** to be TRUE. A boolean variable is TRUE if its numeric value is odd; FALSE if its numeric value is even. Why? Because a boolean is TRUE if its rightmost bit (in a binary representation) is 1, and FALSE if its rightmost bit is 0. All even values written in binary form end with a 0 in the rightmost bit, and all odd values written in binary form end with a 1.

To change the value of **Operating**, we must also specify BYTE, since it is only one byte long:

```
*BYTE .OPERATING=1<cr>
*BOOL BYTE .OPERATING<cr>
TRUE
```

We can now continue execution with a GO command:

```
*GO<cr>
================================================
The Climate System is now operating.
.
.
.
```

Execution continues correctly, and the program loops back to the **GetData** *procedure to obtain more data. As we continue execution, the program once again erroneously changes* **Operating** *to FALSE.*

We can use our .START symbol to start execution once again at the beginning of the program:

```
*GO FROM .START TILL ..MAINCONTROL.OPERATESYSTEM<cr>
```

After repeating these debugging operations, we are sure that our error occurred at the statement where **Operating** is set to FALSE while **Panic** is set to TRUE.

Refer to the listing in Chapter 4. The Pascal-86 statement numbers are in the "STMT" column of the listing. The "LINE" column shows the source file line numbers (you can use the line numbers with CREDIT to display, edit, move, or copy particular lines).

Between statements 37 and 40 (source lines 68 and 70), we have an ELSE clause that should only execute if the temperatures are not greater than the minimums necessary to heat the building. **Panic** should be set to TRUE and **Operating** should be set to FALSE, if this ELSE clause executes.

However, the "ELSE **Panic**:=TRUE" is not executing, but the "**Operating**:=FALSE" is *always* executing! The error is one of omission: to have two statements execute as part of an ELSE clause, they must begin with BEGIN and end with END, as shown:

```
ELSE BEGIN
       Panic:=TRUE; Operating:=FALSE;
     END;
```

To make this change, we must re-edit MAIN.SRC, re-compile MAIN.SRC to obtain a new MAIN.OBJ, and re-link the modules using LINK86 (with BIND, as shown in Chapter 6). Figure 7-1 shows the revised listing, with the corrected ELSE clause shaded, and a sample run of the program.

```
SERIES-III Pascal-86, X031                                  09/01/80      PAGE   1
                                                                          MAINCONTROL

Source File: :F1:MAIN.SRC
Object File: :F1:MAIN.OBJ
Controls Specified:  DEBUG.

STMT LINE NESTING       SOURCE TEXT: :F1:MAIN.SRC
   1    1  0  0         MODULE MainControl;
   2    2  0  0
                       (* Interface specification common to all modules *)

                       $INCLUDE(:F1:INSPEC.SRC)
              =1       PUBLIC MainControl; (*section of interface specification*)
```

Figure 7-1. Climate Control Program Listing and Sample Run

```
 3    2  0  0 =1
                =1          CONST (*declarations declared publicly in this module*)
                =1
                =1            MinimumForExchanger   =  35;(*degrees Celsius*)
 4    6  0  0 =1            MinimumForHeatPump    =  13;
 5    7  0  0 =1
                =1          TYPE (*definitions publicly defined in this module*)
                =1
                =1            AirTemperature    =-20..120;(*degrees in Celsius*)
 6   11  0  0 =1            WaterTemperature  =0..120;
 7   12  0  0 =1            HeatingMethods    =(CollectorToExchanger,
                =1                              CollectorToHeatPump,
                =1                              TankToExchanger,
                =1                              TankToHeatPump,
                =1                              HeatedTankToHeatPump,
                =1                              NoMethod);
 8   18  0  0 =1
                =1            SystemData        = RECORD
 8   20  0  1 =1                                   ChosenMethod          : HeatingMethods;
 9   21  0  1 =1                                   InsideTemp,
                =1                                 ThermostatSetting     : AirTemperature;
10   23  0  1 =1                                   CollectorWaterTemp,
                =1                                 TankWaterTemp,
                =1                                 HeatedTankTemp        : WaterTemperature;
11   26  0  1 =1                                   AmountOfSunlight      : Integer;
12   27  0  1 =1                                   Hour                  : 00..24;
13   28  0  1 =1                                   Minute                : 00..59;
14   29  0  1 =1                                 END (*SystemData*);
15   30  0  0 =1
                =1          VAR (*variables publicly defined in this module.*)
                =1
                =1            CurrentData                 : SystemData;
16   34  0  0 =1            Operating, Panic            : BOOLEAN;
17   35  0  0 =1
                =1        PUBLIC GetData; (*GetData Module containing GetData & StoreData*)
18   37  0  0 =1
                =1          PROCEDURE GetData(VAR CurrentData:SystemData);
19   39  0  0 =1          PROCEDURE StoreData(VAR CurrentData:SystemData);
20   40  0  0 =1
                =1        PUBLIC Operation; (*Operation Module containing OperateSystem,
21   42  0  0 =1                           StartUpSystem and ShutDownSystem*)
                =1          PROCEDURE StartUpSystem;
22   44  0  0 =1          PROCEDURE OperateSystem(VAR CurrentData:SystemData);
23   45  0  0 =1          PROCEDURE ShutDownSystem(VAR CurrentData:SystemData);
24   46  0  0 =1

                        PROGRAM MainControl(INPUT,OUTPUT);
25    8  0  0

                        (* end of interface specification *)


                        PROCEDURE DetermineMethod(VAR CurrentData : SystemData);
26   13  1  0              BEGIN
26   14  1  1                WITH CurrentData DO
27   15  1  1                      BEGIN
27   16  1  2                       IF InsideTemp<ThermostatSetting THEN
28   17  1  2                          BEGIN
28   18  1  3                             IF CollectorWaterTemp>MinimumForExchanger THEN
29   19  1  3                                ChosenMethod:=CollectorToExchanger
                                          ELSE IF CollectorWaterTemp>MinimumForHeatPump THEN
31   21  1  3                                ChosenMethod:=CollectorToHeatPump
                                          ELSE IF TankWaterTemp>MinimumForExchanger THEN
33   23  1  3                                ChosenMethod:=TankToExchanger
                                          ELSE IF TankWaterTemp>MinimumForHeatPump THEN
35   25  1  3                                ChosenMethod:=TankToHeatPump
                                          ELSE IF HeatedTankTemp>MinimumForHeatPump THEN
37   27  1  3                                ChosenMethod:=HeatedTankToHeatPump
                                          ELSE BEGIN
38   29  1  4                                Panic:=TRUE; Operating:=FALSE;
40   30  1  4                                END
41   31  1  3                          END
42   32  1  2                       ELSE (*no heating request*)ChosenMethod:=NoMethod
43   33  1  2                   END; (*With CurrentData*)
45   34  1  1               END; (*DetermineMethod)
46   35  0  0

                        (************** MAIN PROGRAM ***********************************)

                        BEGIN
46   39  0  1              StartUpSystem;
47   40  0  1              Operating:=TRUE;
48   41  0  1              Panic:=FALSE;
49   42  0  1              WHILE Operating DO (*while system is operating, do:*)
50   43  0  1                 BEGIN
50   44  0  2                    GetData(CurrentData); (*Get the temps, time, etc.*)
51   45  0  2                    StoreData(CurrentData); (*Store this data as record*)
52   46  0  2                    DetermineMethod(CurrentData); (*this detects a panic*)
53   47  0  2                    OperateSystem(CurrentData);
54   48  0  2                 END; (*while operating*)
56   49  0  1              ShutDownSystem(CurrentData);
57   50  0  1        END. (*Main Control Algorithm*)
```

Figure 7-1. Climate Control Program Listing and Sample Run (Cont'd.)

```
Summary Information:

PROCEDURE                          OFFSET    CODE SIZE     DATA SIZE     STACK SIZE
DETERMINEMETHOD                    0011H  003FH   143D                   0006H     6D

Total                                     0147H   327D  0016H    22D  004CH    76D

  96 Lines Read.
   0 Errors Detected.
 33% Utilization of Memory.



SERIES-III Pascal-86/ V031                                          09/01/80        PAGE   1


Source File: :F1:DUMDAT.SRC
Object File: :F1:DUMDAT.OBJ
Controls Specified:  DEBUG.

STMT LINE NESTING       SOURCE TEXT: :F1:DUMDAT.SRC
   1    1  0  0         (*This is a dummy GetData module, with dummy GetData
                        and StoreData procedures, for use with MainControl
                        module in testing phases.  It only performs console
                        input to get Celsius temperatures, the time of day,
                        and the amount of sunlight (insolation) for the
                        solar collector.  Use PLM86$DATA module for real
                        application.*)

                        MODULE GetData;
   2   10  0  0
                        (* Interface specification common to all modules *)

                        $INCLUDE(:F1:INSPEC.SRC)
                    =1  PUBLIC MainControl; (*section of interface specification*)
   3    2  0  0 =1
                    =1     CONST (*declarations declared publicly in this module*)
                    =1
                    =1       MinimumForExchanger   =  35;(*degrees Celsius*)
   4    6  0  0 =1           MinimumForHeatPump    =  13;
   5    7  0  0 =1
                    =1     TYPE (*definitions publicly defined in this module*)
                    =1
                    =1       AirTemperature   =-20..120;(*degrees in Celsius*)
   6   11  0  0 =1           WaterTemperature =0..120;
   7   12  0  0 =1           HeatingMethods   =(CollectorToExchanger,
                    =1                           CollectorToHeatPump,
                    =1                           TankToExchanger,
                    =1                           TankToHeatPump,
                    =1                           HeatedTankToHeatPump,
                    =1                           NoMethod);
   8   18  0  0 =1
                    =1           SystemData       = RECORD
   8   20  0  1 =1                                    ChosenMethod          : HeatingMethods;
   9   21  0  1 =1                                    InsideTemp,
                    =1                              ThermostatSetting     : AirTemperature;
  10   23  0  1 =1                                    CollectorWaterTemp,
                    =1                              TankWaterTemp,
                    =1                              HeatedTankTemp        : WaterTemperature;
  11   26  0  1 =1                                    AmountOfSunlight      : Integer;
  12   27  0  1 =1                                    Hour                  : 00..24;
  13   28  0  1 =1                                    Minute                : 00..59;
  14   29  0  1 =1                                  END (*SystemData*);
  15   30  0  0 =1
                    =1     VAR (*variables publicly defined in this module.*)
                    =1
                    =1       CurrentData                 : SystemData;
  16   34  0  0 =1           Operating, Panic            : BOOLEAN;
  17   35  0  0 =1
                    =1   PUBLIC GetData; (*GetData Module containing GetData & StoreData*)
  18   37  0  0 =1
                    =1     PROCEDURE GetData(VAR CurrentData:SystemData);
  19   39  0  0 =1           PROCEDURE StoreData(VAR CurrentData:SystemData);
  20   40  0  0 =1
                    =1   PUBLIC Operation; (*Operation Module containing OperateSystem,
  21   42  0  0 =1                           StartUpSystem and ShutDownSystem*)
                    =1     PROCEDURE StartUpSystem;
  22   44  0  0 =1           PROCEDURE OperateSystem(VAR CurrentData:SystemData);
  23   45  0  0 =1           PROCEDURE ShutDownSystem(VAR CurrentData:SystemData);
  24   46  0  0 =1


                        PRIVATE GetData;
  25   16  0  0
                        (* end of interface specification *)
```

Figure 7-1. Climate Control Program Listing and Sample Run (Cont'd.)

```
                        PROCEDURE GetData(VAR CurrentData:SystemData);
  26   21  1  0          BEGIN
  26   22  1  1             WITH CurrentData DO BEGIN
  27   23  1  2                WRITE('Type the thermostat setting in degrees Celsius:');
  28   24  1  2                READLN(ThermostatSetting); WRITELN;
  30   25  1  2                WRITE('Type the inside temperature reading in Celsius:');
  31   26  1  2                READLN(InsideTemp); WRITELN;
  33   27  1  2                WRITE('Type the temperature of the collector water in Celsius:');
  34   28  1  2                READLN(CollectorWaterTemp); WRITELN;
  36   29  1  2                WRITE('Type the temperature of the tank water in Celsius:');
  37   30  1  2                READLN(TankWaterTemp); WRITELN;
  39   31  1  2                WRITE('Type the temperature of the heated tank water in Celsius:');
  40   32  1  2                READLN(HeatedTankTemp); WRITELN;
  42   33  1  2                WRITE('Type the hour of day, as in 04 or 24:');
  43   34  1  2                READLN(Hour); WRITELN;
  45   35  1  2                WRITE('Type the minute of the hour, as in 01 or 59: ');
  46   36  1  2                READLN(Minute); WRITELN;
  48   37  1  2                WRITE('Type the amount of sunlight, any integer will do for now:');
  49   38  1  2                READLN(AmountOfSunlight); WRITELN;
  51   39  1  2             END;  (*with CurrentData*)
  53   40  1  1          END;
  54   41  0  0
                        PROCEDURE StoreData(VAR CurrentData:SystemData);
  55   43  1  0          BEGIN
  55   44  1  1          (*Dummy procedure, eventually will store CurrentData in a file*)
                             WITH CurrentData DO BEGIN
  56   46  1  2                WRITELN('----------------------------');
  57   47  1  2                WRITELN('CURRENT DATA IS AS FOLLOWS:');
  58   48  1  2                WRITELN('----------------------------');
  59   49  1  2                WRITELN('Thermostat Setting is ',ThermostatSetting,'C');
  60   50  1  2                WRITELN('Inside temperature is ',InsideTemp,'C');
  61   51  1  2                WRITELN('Temperature of collector water is ',CollectorWaterTemp,'C');
  62   52  1  2                WRITELN('Temperature of tank water is ',TankWaterTemp,'C');
  63   53  1  2                WRITELN('Temperature of the heated tank water is ',HeatedTankTemp,'C');
  64   54  1  2                WRITELN('Time of day is ',Hour,':',Minute);
  65   55  1  2                WRITELN('Amount of sunlight is ',AmountOfSunlight);
  66   56  1  2                WRITELN; (*a blank line*)
  67   57  1  2             END; (*with CurrentData*)
  69   58  1  1          END
***WARNING, input: "END "
***was repaired to "END ; "
  70   58  0  0

Summary Information:

PROCEDURE                       OFFSET    CODE SIZE     DATA SIZE    STACK SIZE
STOREDATA                        0405H   0222H   546D                0010H    16D
GETDATA                          0294H   0241H   577D                0010H    16D

Total                                    06F7H  1783D  0000H     0D  0020H    32D

 104 Lines Read.
   1 Error Detected.
 33% Utilization of Memory.


SERIES-III Pascal-86, X031                                          09/01/80          PAGE   1


Source File: :F1:DUMOP.SRC
Object File: :F1:DUMOP.OBJ
Controls Specified:  DEBUG.

STMT LINE NESTING       SOURCE TEXT: :F1:DUMOP.SRC
   1    1  0  0         (*This is a dummy Operation module, with dummy StartUpSystem,
                        ShutDownSystem, and OperateSystem procedures,
                        for use with MainControl module in testing phases.
                        *)

                        MODULE Operation;
   2    7  0  0
                        (* Interface specification common to all modules *)

                        $INCLUDE(:F1:INSPEC.SRC)
                  =1    PUBLIC MainControl; (*section of interface specification*)
   3    2  0  0 =1
                  =1       CONST (*declarations declared publicly in this module*)
                  =1
                  =1         MinimumForExchanger   =  35;(*degrees Celsius*)
   4    6  0  0 =1           MinimumForHeatPump    =  13;
   5    7  0  0 =1
                  =1       TYPE (*definitions publicly defined in this module*)
                  =1
                  =1         AirTemperature   =-20..120;(*degrees in Celsius*)
   6   11  0  0 =1           WaterTemperature =0..120;
```

Figure 7-1. Climate Control Program Listing and Sample Run (Cont'd.)

```
 7   12  0  0 =1          HeatingMethods   =(CollectorToExchanger,
                          =1                 CollectorToHeatPump,
                          =1                 TankToExchanger,
                          =1                 TankToHeatPump,
                          =1                 HeatedTankToHeatPump,
                          =1                 NoMethod);
 8   18  0  0 =1
                          =1          SystemData       = RECORD
 8   20  0  1 =1                                          ChosenMethod          : HeatingMethods;
 9   21  0  1 =1                                          InsideTemp,
                          =1                             ThermostatSetting     : AirTemperature;
10   23  0  1 =1                                          CollectorWaterTemp,
                          =1                             TankWaterTemp,
                          =1                             HeatedTankTemp        : WaterTemperature;
11   26  0  1 =1                                          AmountOfSunlight      : Integer;
12   27  0  1 =1                                          Hour                  : 00..24;
13   28  0  1 =1                                          Minute                : 00..59;
14   29  0  1 =1                                        END (*SystemData*);
15   30  0  0 =1
                          =1       VAR (*variables publicly defined in this module.*)
                          =1
                          =1         CurrentData                 : SystemData;
16   34  0  0 =1            Operating, Panic            : BOOLEAN;
17   35  0  0 =1
                          =1     PUBLIC GetData; (*GetData Module containing GetData & StoreData*)
18   37  0  0 =1
                          =1       PROCEDURE GetData(VAR CurrentData:SystemData);
19   39  0  0 =1            PROCEDURE StoreData(VAR CurrentData:SystemData);
20   40  0  0 =1
                          =1     PUBLIC Operation; (*Operation Module containing OperateSystem,
21   42  0  0 =1                                 StartUpSystem and ShutDownSystem*)
                          =1       PROCEDURE StartUpSystem;
22   44  0  0 =1            PROCEDURE OperateSystem(VAR CurrentData:SystemData);
23   45  0  0 =1            PROCEDURE ShutDownSystem(VAR CurrentData:SystemData);
24   46  0  0 =1

                          PRIVATE Operation;
25   13  0  0
                          (* end of interface specification *)


                          PROCEDURE StartUpSystem;
26   18  1  0             BEGIN
26   19  1  1                  WRITELN ('Climate system is now on.');
27   20  1  1                  WRITELN ('--------------------------');
28   21  1  1                  WRITELN;
29   22  1  1             END;
30   23  0  0

                          PROCEDURE OperateSystem(VAR CurrentData:SystemData);
31   26  1  0             BEGIN
31   27  1  1               WITH CurrentData DO BEGIN
32   28  1  2                 WRITELN('=============================================');
33   29  1  2                 WRITELN('The Climate System is now operating.');
34   30  1  2                 WRITELN;
35   31  1  2                 WRITELN('The Time is ',Hour,':',Minute);
36   32  1  2                 WRITELN('The inside temperature is ',InsideTemp,'C');
37   33  1  2                 WRITELN('The thermostat setting is ',ThermostatSetting,'C');
38   34  1  2                 WRITE('Method chosen to heat the building: ');
39   35  1  2                 CASE ChosenMethod OF
40   36  1  3                    CollectorToExchanger: WRITE('Solar Collector to Exchanger');
41   37  1  3                    CollectorToHeatPump : WRITE('Solar Collector to Heat Pump');
42   38  1  3                    TankToExchanger     : WRITE('Tank to Exchanger');
43   39  1  3                    TankToHeatPump      : WRITE('Tank to Heat Pump');
44   40  1  3                    HeatedTankToHeatPump: WRITE('Heated Tank to Heat Pump');
45   41  1  3                    NoMethod            : WRITE('No heat required');
46   42  1  3                 END;
48   43  1  2                 WRITELN;
49   44  1  2                 WRITELN('=============================================');
50   45  1  2                 WRITELN; (*write a blank line.*)
51   46  1  2               END;
53   47  1  1             END;(*OperateSystem*)
54   48  0  0

                          PROCEDURE ShutDownSystem(VAR CurrentData:SystemData);
55   51  1  0             BEGIN
55   52  1  1             WRITELN('::::::::::::::::::::::::::::::::::::::::::::::::::::');
56   53  1  1                IF Panic THEN WRITELN('PANIC occurred, NORMAL shutdown.')
57   54  1  1                        ELSE WRITELN('No panic occurred, ABNORMAL shutdown.');
59   55  1  1                WRITELN('::::::::::::::::::::::::::::::::::::::::::::::::::::');
60   56  1  1                WITH CurrentData DO BEGIN
61   57  1  2                   WRITE('Last chosen heating method was: ');
62   58  1  2                   CASE ChosenMethod OF
63   59  1  3                      CollectorToExchanger: WRITE('Solar Collector to Exchanger');
64   60  1  3                      CollectorToHeatPump : WRITE('Solar Collector to Heat Pump');
65   61  1  3                      TankToExchanger     : WRITE('Tank to Exchanger');
66   62  1  3                      TankToHeatPump      : WRITE('Tank to Heat Pump');
67   63  1  3                      HeatedTankToHeatPump: WRITE('Heated Tank to Heat Pump');
68   64  1  3                      NoMethod            : WRITE('No heat required');
```

Figure 7-1. Climate Control Program Listing and Sample Run (Cont'd.)

```
  69   65  1  3              END;
  71   66  1  2              WRITELN;
  72   67  1  2              WRITELN('Thermostat Setting is ',ThermostatSetting,'C');
  73   68  1  2              WRITELN('Inside temperature is ',InsideTemp,'C');
  74   69  1  2              WRITELN('Temperature of collector water is ',CollectorWaterTemp,'C');
  75   70  1  2              WRITELN('Temperature of tank water is ',TankWaterTemp,'C');
  76   71  1  2              WRITELN('Temperature of the heated tank water is ',HeatedTankTemp,'C');
  77   72  1  2              WRITELN('Time of day is ',Hour,':',Minute);
  78   73  1  2              WRITELN('Amount of sunlight is ',AmountOfSunlight);
  79   74  1  2            END;(*with CurrentData*)
  81   75  1  1            WRITELN('::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::');
  82   76  1  1            WRITELN;
  83   77  1  1            WRITELN('Goodnight, Irene...');
  84   78  1  1          END
***WARNING, input: "END "
***was repaired to "END ; "
  85   78  0  0          .(*ShutDownSystem*)


Summary Information:

PROCEDURE                    OFFSET   CODE SIZE     DATA SIZE    STACK SIZE
SHUTDOWNSYSTEM               0698H  0398H   920D                 0010H   16D
OPERATESYSTEM                0440H  0258H   600D                 0010H   16D
STARTUPSYSTEM                03E2H  005EH    94D                 0010H   16D

Total                               0A30H  2608D  0000H     0D  0030H   48D

 125 Lines Read.
   1 Error Detected.
 33% Utilization of Memory.
```

```
-RUN PROGRM<cr>

Climate system is now on.
-------------------------
Type the thermostat setting in degrees Celsius:24<cr>

Type the inside temperature reading in Celsius:21<cr>

Type the temperature of the collector water in Celsius:60<cr>

Type the temperature of the tank water in Celsius:60<cr>

Type the temperature of the heated tank water in Celsius:70<cr>

Type the hour of day, as in 04 or 24:13<cr>

Type the minute of the hour, as in 01 or 59:25<cr>

Type the amount of sunlight, any integer will do for now:1<cr>

---------------------------
CURRENT DATA IS AS FOLLOWS:
---------------------------
Thermostat Setting is 24C
Inside temperature is 21C
Temperature of collector water is 60C
Temperature of tank water is 60C
Temperature of the heated tank water is 70C
Time of day is 13:25
Amount of sunlight is 1
```

Figure 7-1. Climate Control Program Listing and Sample Run (Cont'd.)

```
==============================================
The Climate System is now operating.

The time is 13:25
The inside temperature is 21C
The thermostat setting is 24C
Method chosen to heat the building: Solar Collector to Exchanger
==============================================

Type the thermostat setting in degrees Celsius:24<cr>

Type the inside temperature reading in Celsius:22<cr>

Type the temperature of the collector water in Celsius:10<cr>

Type the temperature of the tank water in Celsius:10<cr>

Type the temperature of the heated tank water in Celsius:10<cr>

Type the hour of day, as in 04 or 24:13<cr>

Type the minute of the hour, as in 01 or 59:30<cr>

Type the amount of sunlight, any integer will do for now:0<cr>
--------------------------
CURRENT DATA IS AS FOLLOWS:
--------------------------
Thermostat Setting is 24C
Inside temperature is 22C
Temperature of collector water is 10C
Temperature of tank water is 10C
Temperature of the heated tank water is 10C
Time of day is 13:30
Amount of sunlight is 0
==============================================
The Climate System is now operating.

The time is 13:30
The inside temperature is 22C
The thermostat setting is 24C
Method chosen to heat the building: Solar Collector To Exchanger
==============================================
::::::::::::::::::::::::::::::::::::::::::::::::::
PANIC occurred, NORMAL shutdown.
::::::::::::::::::::::::::::::::::::::::::::::::::
Last chosen heating method was: Solar Collector to Exchanger
Thermostat Setting is 24C
Inside temperature is 22C
Temperature of collector water is 10C
Temperature of tank water is 10C
Temperature of the heated tank water is 10C
Time of day is 13:30
Amount of sunlight is 0
::::::::::::::::::::::::::::::::::::::::::::::::::

Goodnight, Irene...
-
```

Figure 7-1. Climate Control Program Listing and Sample Run (Cont'd.)

Obviously this book only introduces DEBUG-86. For a complete description, see the *Intellec Series III Microcomputer Development System Console Operating Instructions*.

CHAPTER 7

USING ICE-88™, AN IN-CIRCUIT EMULATOR

The ICE-88 emulator consists of three circuit boards, a cable, a buffer box, and software on disk. The three circuit boards fit inside your Intellec Series III development system, and the cable and buffer box assembly connects your Series III system to your prototype hardware. A forty-pin plug at the end of the cable plugs into your prototype system in place of your prototype's CPU, allowing the in-circuit emulator hardware to emulate all functions of your prototype's CPU. If you do not have any prototype hardware, keep the plug's protector closed to prevent damage to its pins.

In-circuit emulation shortens your development time in two ways. It provides symbolic debugging capabilities and diagnostic hardware debugging capabilities, and it lets you borrow resources (like memory) from your Series III system until your prototype system is complete.

Your program can be loaded into borrowed memory, prototype memory, or any combination of the two, and run as if it were resident in the prototype. You can emulate program execution at real-time speed or in single- or multiple-instruction steps. You can also stop emulation manually at any time to examine system status, or you can specify breakpoints as you can with DEBUG-86.

We will debug the **PLMDATA** module (first shown in Chapter 5) to see if it performs its port input and interpolations correctly. We added statements (shaded in figure 7-2) to the module that will execute the procedures. Figure 7-2 shows the listing of the compiled **PLMDATA** module.

```
PL/M-86 COMPILER     PLMDATA                                                PAGE   1

ISIS-II PL/M-86 V121 COMPILATION OF MODULE PLMDATA
OBJECT MODULE PLACED IN PLMDEB.OBJ
COMPILER INVOKED BY:  PLM86 PLMDEB.SRC DEBUG


   1          PLMDATA: DO;

   2   1      DECLARE START LABEL PUBLIC;
   3   1      DECLARE (SETTING, TEMPERATURE) WORD;

              /*
              INPUTS
              ------

              THERMOSTAT$SETTING$FROM$PORTS:

              Formal parameters HIGH and LOW receive port numbers as actual parameters.

              Input ports HIGH and LOW: Three BCD digits for the thermostat setting:

                                  Port HIGH, bits 3-0: hundred's digit
                                  Port LOW, bits 7-4: ten's digit
                                  Port LOW, bits 3-0: unit's digit


              TEMP$DATA$FROM$PORTS:

              Formal parameters HIGH and LOW receive port numbers as actual parameters.

              Input port HIGH: Binary ADC output of thermocouple, high-order 8 bits
              Input port LOW: Binary ADC output of thermocouple, low-order 8 bits


              OUTPUTS
              -------

              THERMOSTAT$SETTING$FROM$PORTS:  Return WORD with setting in Celsius

              TEMP$DATA$FROM$PORTS:  Return WORD with temperature in Celsius

              */
```

Figure 7-2. Listing of the Modified PLMDATA Module

```
 4   1      THERMOSTAT$SETTINGS$FROM$PORTS:
            PROCEDURE (HIGH, LOW) WORD;
 5   2         DECLARE (HIGH, LOW) WORD;
 6   2         DECLARE (IN$PORT$HIGH, IN$PORT$LOW) BYTE;
 7   2         DECLARE THERMO$SETTING WORD;
 8   2         DECLARE (HUNDREDS, TENS, UNITS) BYTE;

 9   2         IN$PORT$HIGH = INPUT(HIGH);
10   2         IN$PORT$LOW = INPUT(LOW);
11   2         HUNDREDS = IN$PORT$HIGH AND 00000111B;
12   2         TENS = SHR(IN$PORT$LOW, 4);
13   2         UNITS = IN$PORT$LOW AND 00001111B;
14   2         THERMO$SETTING = UNITS + 10*TENS + 100*HUNDREDS;
15   2         RETURN THERMO$SETTING;
16   2      END THERMOSTAT$SETTINGS$FROM$PORTS;


            /* Another typed procedure to return temperature data, which
               uses the INTERPOLATE typed procedure. */

            /* INTERPOLATE is a typed procedure that receives thermocouple
               voltage and returns temperature in Celsius using an
               interpolation routine */

17   1      INTERPOLATE:
            PROCEDURE(VOLT$IN) WORD;
18   2         DECLARE VOLTS(*) WORD DATA(0,51,102,154,206,258,365,472);
19   2         DECLARE T$CEL(*) WORD DATA(0,10,20,30,40,50,70,90);
20   2         DECLARE (I, VOLT$IN, NUMERATOR) WORD;

21   2         I = 0;
22   2         DO WHILE VOLT$IN > VOLTS(I);
23   3            I = I + 1;
24   3         END;

               /* Shift for rounding, and return Celsius temperature */
25   2         NUMERATOR = SHL((VOLT$IN-VOLTS(I-1))*(T$CEL(I)-T$CEL(I-1)), 1);
26   2         RETURN T$CEL(I-1) + SHR(NUMERATOR/(VOLTS(I)-VOLTS(I-1))+1, 1);

27   2      END INTERPOLATE;


28   1      TEMP$DATA$FROM$PORTS:
            PROCEDURE(HIGH,LOW) WORD;
29   2         DECLARE (HIGH,LOW) WORD;
30   2         DECLARE IN$PORT$HIGH WORD; /* ADDRESS in McCracken's book */
31   2         DECLARE IN$PORT$LOW BYTE;
32   2         DECLARE (THERMOCOUPLE$OUTPUT, TEMPERATURE) WORD;

33   2         IN$PORT$HIGH = INPUT(HIGH);
34   2         IN$PORT$LOW = INPUT(LOW);
35   2         THERMOCOUPLE$OUTPUT = SHL(IN$PORT$HIGH, 8) OR IN$PORT$LOW;
36   2         TEMPERATURE = INTERPOLATE(THERMOCOUPLE$OUTPUT);
37   2         RETURN TEMPERATURE;

38   2      END TEMP$DATA$FROMSPORTS;

39   1      START: DO;
40   2             SETTING = THERMOSTAT$SETTING$FROM$PORTS(2000,1000);
41   2             TEMPERATURE = TEMP$DATA$FROM$PORTS(200,100);
42   2             END START;
43   1      END PLMDATA;


MODULE INFORMATION:

     CODE AREA SIZE     = 012CH    3000
     CONSTANT AREA SIZE = 0020H     320
     VARIABLE AREA SIZE = 0016H     220
     MAXIMUM STACK SIZE = 0012H     180
     101 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

Figure 7-2. Listing of the Modified PLMDATA Module (Cont'd.)

We also have to link the LARGE.LIB run-time library to this module, and locate the final module:

```
-RUN LINK86 :F1:PLMDAT.OBJ, :F1:LARGE.LIB TO :F1:PLMDAT.LNK<cr>
     .
     .
     .
-RUN LOC86 :F1:PLMDAT.LNK RESERVE(200H TO 77FFH) TO :F1:PLMDAT.86<cr>
     .
     .
     .
```

To use the ICE-88 emulator, copy the ICE88 program from its disk onto a disk in your system (for example, the system disk in drive 0), and type the following command:

```
-ICE88<cr>
```

You can now use the ICE-88 LOAD command to load the module and its symbols:

```
*LOAD :F1:PLMDAT.86<cr>
*SYMBOLS<cr>
```

The SYMBOLS command displays the entire symbol table. We can start emulation with the GO command, specifying both a starting location and a breakpoint. We can use the .START label as a starting location, and specify line #8 as a breakpoint:

```
*GO FROM .START TILL #8 EXECUTED<cr>
EMULATION BEGUN
.
.
.
EMULATION TERMINATED, CS:IP=0781:0040H
```

At this point, we want to change the contents of the 8088 ports so that the THERMOSTAT$SETTING$FROM$PORTS procedure can pick them up:

```
*PORT 2000=0  ;Hundred's digit of BCD thermostat setting.<cr>
*PORT 1000=24 ;Tens and units of BCD setting.<cr>
*GO FROM CS:IP TILL .SETTING WRITTEN<cr>
EMULATION BEGUN
.
.
EMULATION TERMINATED, CS:IP=0781:002CH
```

Note that a semicolon and comment is allowed on an ICE-88 command line. We assigned 24H to port 1000 to obtain a 2 as the tens digit and a 4 as the units digit (to represent a thermostat setting of 24 degrees Celsius). Then we resumed emulation from the program counter (PC) until a value was obtained for the .SETTING variable. At this point, we can check the contents of .SETTING:

```
*WORD .SETTING<cr>
WOR 0794:0024H = 0018H
                        18 hexadecimal is 24 in decimal.
```

Since .SETTING is correct, we can set the contents of the other two ports and continue emulation:

```
*PORT 200=0  ;High order 8 bits are zero.<cr>
*PORT 100=66 ;Low order equal 66H, or 102 in decimal.<cr>
*GO FROM CS:IP TILL #43 EXECUTED<cr>
EMULATION BEGUN
.
.
.
EMULATION TERMINATED, CS:IP=0781:003BH
*WORD .TEMPERATURE<cr>
WOR 0794:0026H = 0014H
*
```

14 hexadecimal is 20 in decimal.

We have a temperature reading of 20 degrees Celsius.

Obviously this session only introduces the ICE-88 emulator. The *ICE-88 In-Circuit Emulator Operating Instructions for ISIS-II Users* contains both tutorial and reference information on the ICE-88 emulator. The similarities between DEBUG-86 and in-circuit emulators enhance their usefulness in software development efforts, since both provide symbolic debugging. In-circuit emulation also lets you emulate all of your prototype CPU functions, even though your prototype CPU is not installed, and even if your prototype has not been built. It is a powerful debugging and diagnostic tool for both the hardware and software of your final product.

EXECUTION ENVIRONMENTS

The Intellec Series III system provides an 8086 execution environment and operating system support—the support your program needs to be able to access devices and files. When you link the run-time support libraries to your Pascal-86 program, you are providing the software your program needs to "talk" to the Series III operating system.

You can also run Pascal-86 programs in other systems, or in dedicated application environments, as long as you provide the run-time support software. For example, you could transfer your program to RAM on an SDK-86 (System Design Kit with an 8086), or to RAM on an iSBC 86/12A Single Board Computer system, by first using the OH86 utility described in the *iAPX 86,88 Family Utilities User's Guide for 8086-Based Development Systems* to convert the program to hexadecimal object format, and then using an appropriate tool to load the program into your execution board (the ICE-86 In-Circuit Emulator, the SDK-C86 Software and Cable Interface, or the iSBC 957 Interface and Execution Package).

You could also transfer your program to ROM on an SDK-86 kit, iSBC Single Board Computer system, or your own custom-designed hardware, by using the Universal PROM Programmer (UPP) with its Universal PROM Mapper (UPM) software.

Figure 7-3 shows possible execution paths for Pascal-86 programs.

The Series III operating system has a standard set of primitives (service routines) that any program can use. Intel supplies *run-time support libraries* that act as an interface between your Pascal-86 program and the Series III system. By replacing this interface with your own custom-designed interface, you can use the same Pascal-86 programs on other non-Intel systems. With each future Intel system, Intel will supply the appropriate run-time interface so that your present programs will also run in future Intel systems.

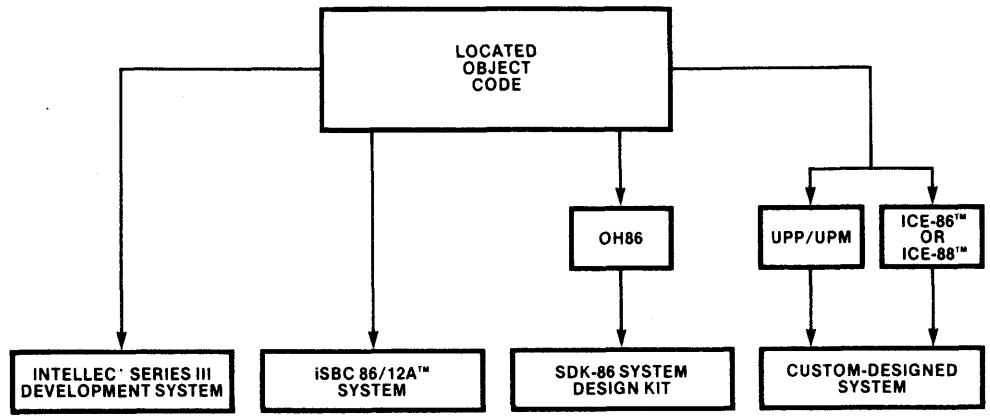


Figure 7-3. Possible Execution Paths for Pascal-86 Programs 121632-8

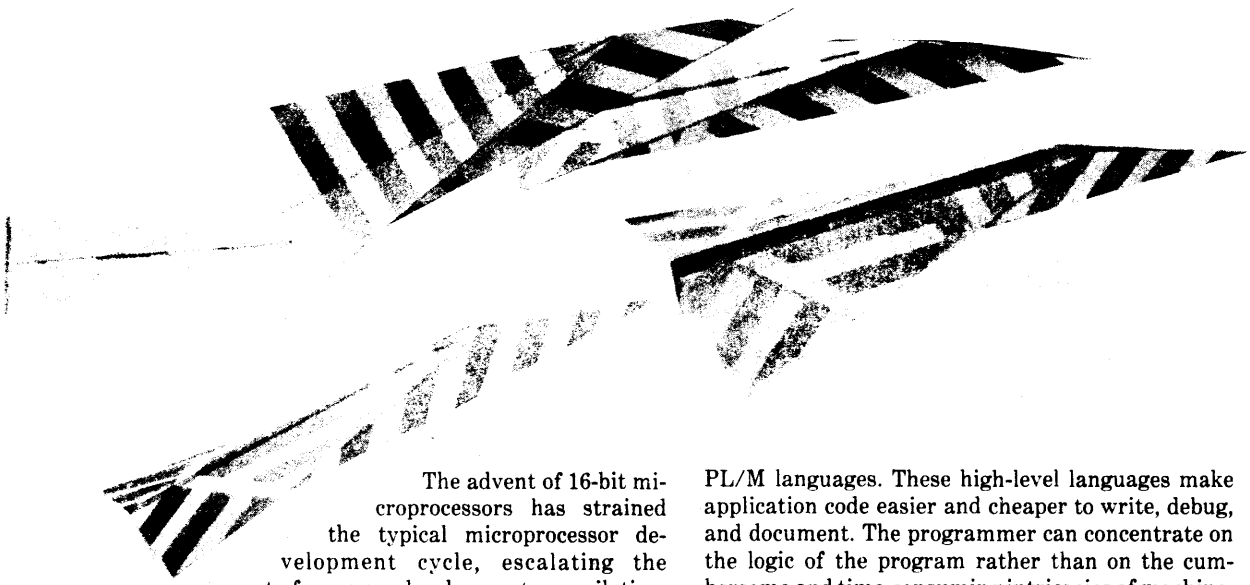
A system library is also supplied for PL/M and assembly language programs; this library (or set of libraries) also acts as an interface between your programs and the Series III operating system. By supplying your own interface, you can also use these programs on other systems.

The Series III system was designed in this modular fashion to provide operating system support without necessarily binding programs to that particular system. It was designed to be *used* as part of your application (as the operating environment), but it was also designed to be useful for the development of applications that do not need full-blown operating system support. With several layers of interfacing between the system and your program, you can choose exactly how much system you want in your final application, and you can preserve your software investment with an eye to the future.

Design

Program development for 16-bit microprocessors creates problems that only a 16-bit system can solve. The Intellec Series III applies high-level languages and in-circuit emulation to expedite programming and debugging.

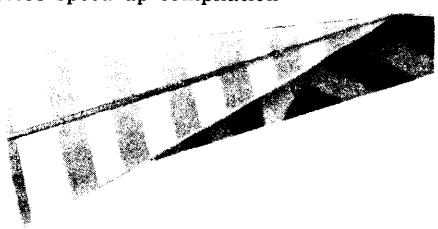
System simplifies program development for 16-bitters



The advent of 16-bit microprocessors has strained the typical microprocessor development cycle, escalating the cost of program development, compilation, and debug and the time required to complete a project that may involve many independent programs. The Intellec Series III microcomputer development system directly addresses the crisis by employing high-level languages (including Pascal), minimizing the amount of assembly code generated per project, and incorporating a 16-bit microprocessor as one of its two host CPUs. The Series III is a stand-alone system that also contains a CRT monitor with detached keyboard, an integral floppy-disk drive and internal power supply, fans, and cables (Fig. 1).

The new Intellec system supports an iAPX 88/86 resident assembler and the Pascal, Fortran, and PL/M languages. These high-level languages make application code easier and cheaper to write, debug, and document. The programmer can concentrate on the logic of the program rather than on the cumbersome and time-consuming intricacies of machine-dependent assemblers.

Two host microprocessors speed up compilation and 8086/8088 code execution within the III. The 8085 handles 8-bit and the 8086 μP handles 16-bit applications and translations. High-level language compilers run faster with the up to 1-Mbyte memory resident in the Series III than with the 64-kbyte memory to which other systems are limited. The two CPUs bring more processing power to the Series III.

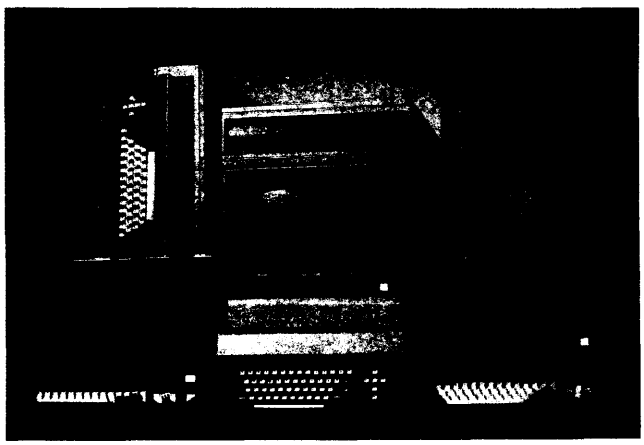


Any Intellec development system, including the Model 800, can be upgraded to Series III performance and features with the Model 556 upgrade kit. It

James Schwabe, Product Marketing Manager
David M. Miller, Product Marketing Manager
Intel Corp.
3065 Bowers Ave., Santa Clara, CA 95051

consists of the 8086 resident processor board, a 64-kbyte RAM memory card, the PROM-based application debugger, and the 8086 resident utilities and assembler. Thus, the full value of previous Intellec hardware is retained.

A Disk File Sharing System eases the bottlenecks involved in programming as much as 1 Mbyte of code. The multiple users working on older Model-800 and Intellec Series II, as well as on the newer Intellec Series III, can tie together and share the information stored in a common disk resource.



1. Program development for 16-bit microprocessors is faster and easier with the Intellec Series III microprocessor development system. The Series III includes a new set of high-level languages and adds a fast 16-bit microprocessor to the 8-bit microprocessor found in earlier versions of the development system.

By today's estimates, 60 to 70% of the cost and time in microcomputer development can be attributed to the debugging and maintenance of application code. To improve debugging, the Series III has an applications debugger, in addition to two software emulators.

As the cost of program development has escalated, the programmer has been forced to improve his productivity to keep the final system cost competitive. One way of improving productivity is through the use of high-level languages.

A highly structured, block-oriented programming language, Pascal has become very popular as a microcomputer applications language. Its rigid structure encourages and enforces good programming techniques that, combined with its high level of readability, help to produce more reliable and maintainable software. A major benefit of the language, maintainability, results because the Type mechanism is both strictly enforced and user extendable.

Another benefit of the language is the absence of machine-specific constructs. Thus, the user need never concern himself with the architecture of the microprocessor for which he is writing code. In addition, Pascal programs "talk" to the resident operating system. Therefore, the development operating system can be replaced with the user's own interface in the final application.

Program modules compiled by the Series III under Pascal-86 are compatible and linkable with modules written in PL/M88/86 ASM88/86, or Fortran88/86. The programmer can implement each program module of an overall system in the language appropriate for the task at hand. Extensions of the language (including predefined procedures for interrupt handling and direct port I/O) make it possible to code an entire application in Pascal without compromising the program's portability.

Compiler options available with the language let the user control the program listings and object modules during compilation. During debug, the user can generate additional information, such as the symbol record information, which is useful for debugging with an in-circuit emulator. After debugging, this additional information can be removed to reduce the memory requirements of the target system.

Fortran offers many of the same benefits as Pascal and a few benefits peculiar to Fortran. All programs already written in Fortran, including the earlier Fortran-80 language, can be executed on the new compiler. Because of the way Intel has implemented the language, it can directly manage byte or word-oriented port I/O operations, reentrant procedures, interrupt procedures, and fixed-point, floating-point, or I/O errors.

Along with Pascal and Fortran, the new system offers PL/M-88/86, a sophisticated, block-structured programming language that maximizes the advantages of the iAPX 86 and iAPX 88 microprocessors. PL/M is Intel's system implementation language, designed for programming machine-intimate routines without run-time support from the operating system. Since PL/M is a high-level language, a user can manipulate hardware directly without using the machine-dependent instructions common to assemblers. The programmer gets help from support or absolute addressing, interrupt handling, reentrant procedures, direct port I/O, and memory-mapped I/O.

For this high-level language, the Series III also has a high-speed option that checks the source code for correct language construction and syntax. The programmer can quickly weed out simple syntax errors, without delaying to generate object code from a full compile. Software development is easier and takes less time.

The resident relocating iAPX 88/86 macro-assembler, ASM88/86, is a "strongly typed" assembler that encourages well-designed and well-structured programs. The iAPX 88/86 assembly language includes approximately 100 mnemonics from which more than 3800 machine-language instructions can be generated. The few mnemonics ease the designer's task in learning assembly language and developing software—for a significant cut in program development time.

The various addressing modes of each operation are differentiated by the assembler, which evaluates the "type" of operands within the instruction. This ability ("type checking") ensures that the most efficient machine instruction is generated. For example, the assembler looks at the operands of a MOV instruction to determine whether the appropriate machine instruction is an 8-bit or 16-bit MOV and whether it should be indexed, indirect, or immediate, and so on.

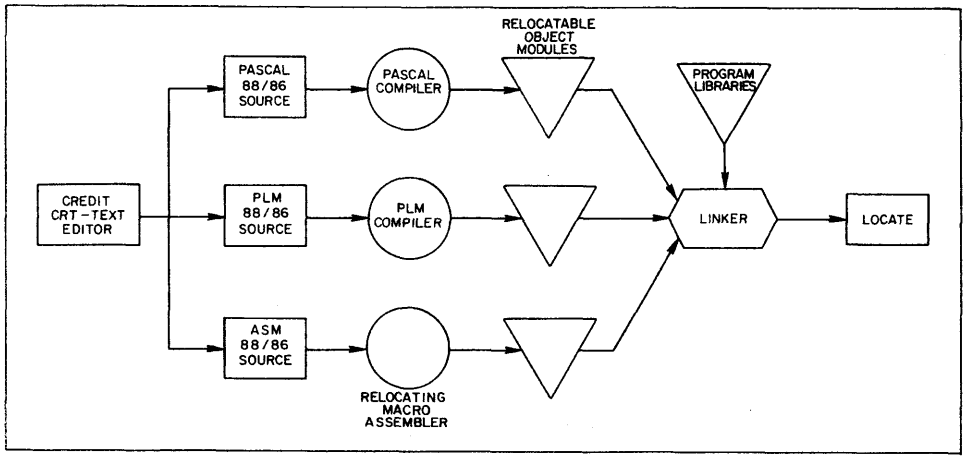
All Intel programming languages output relocatable object models that easily link up with other object modules written in assembly language, PL/M, Fortran, or Pascal. The LINK88/86 utility combines object modules and libraries of modules into a single output module, resolving all external references. Thus, the user can program each task in the language most appropriate to it, rather than write an entire application in one language. Each task can be further broken down into modules that are manageable in size and that are individually coded, translated, and debugged. LINK88/86 also permits incremental linking, by which the output of one link becomes the input to another link. In this way, large programs can be separated into smaller, fully-linked, and executable programs, which can be individually debugged before they are recombined in a final integration. These modular programming techniques pay off with easier debugging and simpler maintenance.

Once the component modules are developed, debugged, and linked together, the LOC88/86 locates the complete program within memory. This utility assigns absolute addresses to the relocatable object modules. The object code of application programs developed on the Series III can be relocated to suit the application. Thus, the entire edit, compile, link, and locate process is made more efficient and cost effective (Fig. 2).

The productivity of the Series III programmer is further enhanced by the library manager, LIB88/86, which creates libraries of object modules that are, in turn, linked to other modules with LINK88/86. Since it links only those modules which are required to resolve external references, LIB88/86 saves space in the application program. LIB88/86 provides the framework for a program library that facilitates program design and speeds development time.

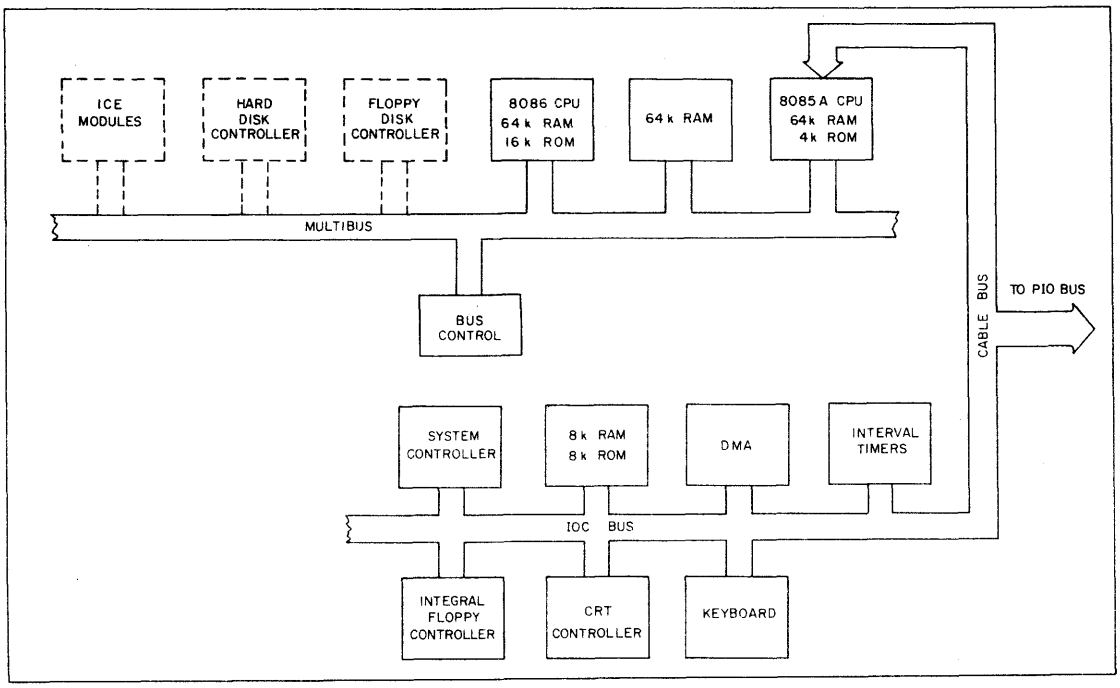
To address the problems of faster compilation and 8086/8088 code execution in the development system itself, the Series III has two host CPUs, an 8085 and an 8086 (Fig. 3). All software previously written on an Intellec Series II will run, without modification, on the Series III. Expandable to 1 Mbyte of system memory, the Series III creates more space for symbol tables and user programs and lets high-level compilers run faster.

The 8-bit CPU board is based on the high performance 8085A-2 microprocessor. It contains its own memory, I/O, interrupt, and bus-interface circuitry, all implemented with Intel's high-technology components. The 8085A-2, which runs at 4 MHz, has access to 62 kbytes of RAM and 4 kbytes of ROM on the 8-bit CPU board. Because ROM is prepro-



2. Because of the diversity of languages that come with the development system, any project can be partitioned into program modules developed in the language best suited to the application. When the system must be integrated, these modules are automatically linked by the Series III LINKER routine.

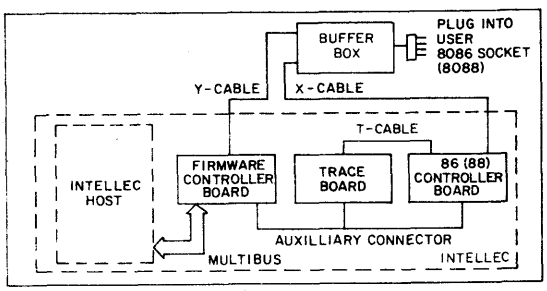
Microcomputer development system



3. Because the Intellec Series III has two CPUs, 8-bit applications and translations are executed on an 8-bit processor (the 8085), and 16-bit applications and translations are executed on a 16-bit processor (the iAPX 86). The latter can also execute 16-bit program code in the development system; previously, this task has to be done on the ICE module.

grammed with the system bootstrap, diagnostic, and monitor programs, the system is ready for operation at the touch of a button.

The second CPU board is based on the powerful 8086, Intel's 16-bit HMOS microprocessor. The 8086, which runs at 5 MHz, has access to 128 kbytes of RAM with onboard refresh. The iAPX 86 CPU board, when combined with the 8085 CPU board, enhances the execution of 16-bit resident application programs. The 8085 CPU board offloads all I/O, operating as an intelligent I/O controller to double-buffer data to and from the iAPX 86.



4. The ICE emulators, optional on the Series III, make it possible to emulate, not only the 16-bit 8086, but also the 8-bit 8085. Full symbolic debugging is an important integral feature of the emulator module.

Just as the microprocessor and microcomputer have revolutionized the electronics industry, design aids (particularly in-circuit emulators) have revolutionized the design, testing, and integration of microcomputers. Design aids offered in conjunction with the Series III include the in-circuit emulators (ICE) for the iAPX 86 and iAPX 88, the application debugger, the real-time breakpoint facility for the 8089 input-output processor, and emulators for all Intel 8-bit and single-chip processors.

The ICE-86 and ICE-88 emulators allow hardware and software development to proceed interactively, rather than through traditional system integration. With the emulators, prototype hardware can be added to the system as it is designed; software and hardware can be tested while the product is being developed.

The emulators support multiprocessor systems by helping to debug the software that controls the interaction between processors and the system bus, as well as the individual software for each processor. Most important, the emulators have such software-oriented debug features as English-like commands, conditional commands, and macro commands.

The application debugger, DEBUG 86, is PROM resident and an integral part of the development system. The debugger controls execution of iAPX 86

or iAPX 88 user programs in the Series III and provides a subset of ICE emulator commands and capabilities so that iAPX-88/86 application software can be developed, tested, and debugged with a standard Series III system.

Comprising three circuit boards that reside in the development system mainframe, the ICE-86 or 88 module also has a buffer box and cable that connect the Series III to the user system by replacing the iAPX 86/88 of the target system (see Fig. 4). The designer can execute prototype software in the continuous or single-step mode and can substitute blocks of development system memory for equivalent memory in the target system.

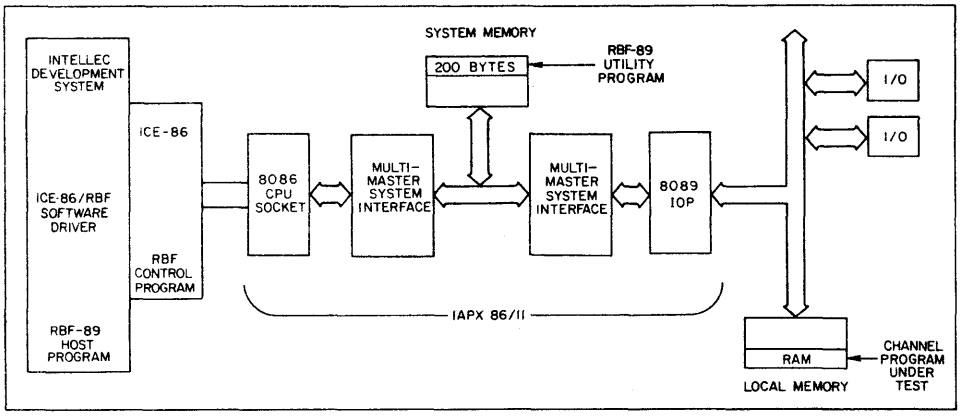
Breakpoints allow the designer to stop emulation on specified conditions, and the emulator's trace capability provides a detailed history of the program execution prior to the breakpoint. All accesses by the designer to the prototype system software can be done symbolically, by referring to the source program variables and labels for all languages.

Finally, RBF-89 (Fig. 5) is a real-time breakpoint facility for the 8089 input/output processor (IOP). The RBF-89 is a software package that tests systems built around the Intel iAPX 86/11 microprocessor by interrogating IOP registers, setting breakpoints in IOP programs, and performing other functions. Special control blocks are created in application system memory, input/output channel-attention commands are issued to the IOP, and the IOP then performs the specified function.

The RBF-89 software package includes a host program residing in the development system RAM and a control program in the ICE-86 emulator memory. The former serves as an extension of the ICE-86 emulator software driver. The latter monitors IOP operations. The RBF package also has a utility program, located in prototype memory, that sets and removes the specified breakpoints.

To complement the hardware of Series III, a user may choose from several peripherals that will increase programmer productivity and system performance. For an additional 2.5 Mbytes of on-line storage, the user may add dual double-density flexible-disk systems. The increased storage capacity gives access to larger on-line programs and data files, and the additional drives simplify diskette backup operations. Large storage requirements are met by the Intellec Model 740 hard-disk system, which provides 7.3 Mbytes of formatted on-line storage. The hard disk consists of two platters, one fixed and one removable, housed in a free-standing pedestal.

Series III can be connected to Intel's new multiuser network, the Intellec Disk-File Sharing System, and be viewed as an extension. DFSS connects up to eight Intellec development systems to form a network manager that distributes the resources of the hard disk. Including Series III in this network produces a 16-bit distributed development environment, in which the Series III can be utilized to the fullest. In a sample scenario, editing and debugging would take place at all work stations while the Series III work station acts as a high-performance compiler, with all files shared on the hard disk at the network manager. Because DFSS distributes the development project workload in a multiuser environment, performance and productivity in project development increase while the costs decrease.□



5. With the real-time breakpoint facility, the designer can test and troubleshoot designs that employ the input/output processor (IOP) chip produced by Intel. The designer can interrogate registers, set breakpoints, and perform other functions necessary for effective debugging.

The Disk File Sharing System interconnects as many as eight Intellec development-system workstations to improve programmer productivity and coordination of effort.

Disk-file sharing simplifies big software projects

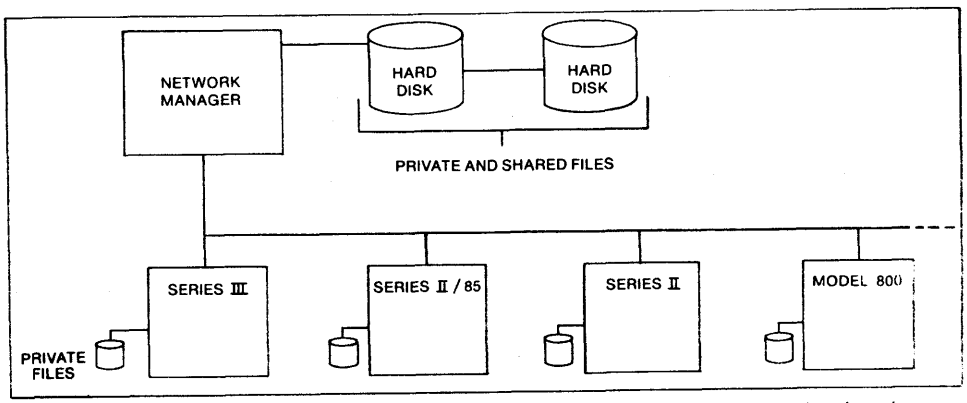
With microcomputer applications becoming increasingly software-intensive, development support must become more sophisticated to keep up. The Disk File Sharing System, which allows up to eight Intellec development systems to be interconnected as workstations in a network, addresses the problems of large software projects. It improves programmer coordination and productivity not only by allowing file sharing with appropriate controls but also by providing project managers with needed management.

While newer 8 and 16-bit μPs have relieved the memory-space constraints of earlier μP architectures and have provided more functions, programs have correspondingly ballooned in both size and complexity. Today's μP software often requires program memory in the hundreds of kilobytes, and program-development costs have risen proportionately. In 1980, a debugged application program written in 3000 lines of high-level-language code can cost more than $100,000 and take more than one and a half man-years to produce. The same application written in assembly code would cost $450,000 and take more than six and a half man-years to produce.

In response, Intel recently introduced the dual-processor-based, high-performance Series III development system (see ELECTRONIC DESIGN, Aug. 16, 1980 p. 97), which supports three resident iAPX86 high-level languages: PL/M, Pascal, and Fortran. High-level-language support, however, is only a partial solution for large development projects. As program size increases, the number of programmers assigned to a project also increases. After each programmer develops a program module, coordinating and integrating the program modules can become more difficult than developing the individual modules using stand-alone development systems.

The DFSS is an extension of the stand-alone development-system concept. A typical project involves program-module editing and debugging at the Intellec Series II or Model 800 workstations, while

James Schwabe, Product Marketing Manager
Intel Corp.
3065 Bowers Ave., Santa Clara, CA 95051



1. The Disk File Sharing system allows programmers working at various Intellec development systems to access the central hard-disk files. This not only improves coordination but also smoothes the program-module integration. Note that the system supports all past generations of Intellec systems.

the Series III iAPX 86 host-execution environment is used to compile completed programs (Fig. 1).

Workstations abound

All Intellec development systems may be upgraded and integrated as DFSS workstations. A workstation-upgrade package consisting of an interconnect board, software, and cable may serve to interconnect up to eight development systems as workstations using any combination of Intellec Model 800, Series II, Series II/85, or Series III development systems.

A network manager in the DFSS coordinates workstation activities and manages file access to the shared disk (Fig. 2). One or two Intellec hard-disk subsystems, associated with the network manager, provide enough capacity to support large program development. Each hard-disk subsystem consists of two platters, one fixed and one removable, to allow simple disk backup.

Files on the hard disk may be created or accessed by any of the Intellec workstations. A special intercommunications board is installed in the network manager and in each workstation to form a high-speed, parallel-byte interface for the network. Data are transferred between workstation and network manager at 40,000 bytes/s, resulting in minimal performance degradation compared to a dedicated hard-disk configuration.

Workstations can be located up to 20 ft apart and up to 120 ft from the network manager. Since workstations share the disk memory, the cost and performance of a hard disk can be distributed across all of the workstations. Thus, for a fraction of the cost of a stand-alone hard disk, each workstation gains an average of 50% performance improvement over a floppy-based system.

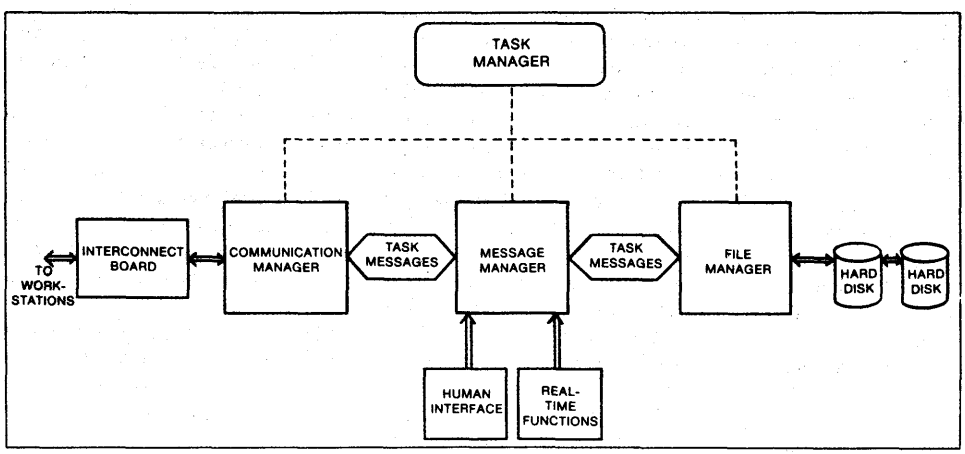
The network manager is implemented with a new multitasking disk operating system that allows communications, disk I/O, and file management to occur concurrently. From a workstation's perspective, the network manager functions as a remote file system. Each workstation functions as a normal, stand-alone development system for all tasks not requiring access to the network manager's resident files.

When access to these files is required, the workstation's resident operating system (ISIS) formats the appropriate file request and forwards the request to the network manager. The network manager processes the request and responds with a block data transfer of the requested file. Data-block sizes are adjusted by the system to minimize latencies due to contention.

Concurrent activity

Since the network manager's operating system is multitasking, a workstation user may access a file on the hard disk while other workstations are concurrently accessing the hard disk. This interleaving of disk accesses enables each workstation to share equally the high performance of the hard disk regardless of the size of the file being accessed. In addition, a large file being accessed by one user does not prevent other workstations from accessing the shared disk.

From the network manager's perspective, communications are based on a one-on-one, master-slave relationship with each workstation. The network manager continually polls each workstation for re-



2. The network manager handles file access through a task manager, which controls the communications, message, and file managers. The communications manager monitors up to eight workstations, transferring file-action requests to the message manager and routing responses to the appropriate workstation. The message manager maintains a table of logged-on users, validates file-action requests, and passes requests and responses to and from the file manager. Finally, the file manager creates, reads, writes, and deletes files on the shared hard disk.

Public and private files

The DFSS does public and private file control through the creation of user IDs. Before users can initially gain access to the shared disk from the workstation, a list of user names and their associated identifiers must be updated by entering a Validate command at the network-manager console. The console then displays a table of valid user IDs and user names. Following the table display, the network manager requests the operator to add or delete user IDs, assigning a unique identifier to each user.

After turning on the workstation, the user may choose either to log onto the DFSS or to operate locally, without any physical change to the network. When a user is logged onto the network, any file created becomes a private file of that user and is inaccessible to anyone else. To create a private file, the user must first create a private file and then use the public function to make the file public.

The network manager restricts creation of public and private files to the point that all public names must be unique. When a private file is created, the directory is searched to ensure that the file name is unique to the user's set of private files and to the set of public files. Public files should be considered an extension of a user's set of private files.

quests to access the shared disk and immediately processes each request. As the master, the network manager eliminates contention among workstations.

Since large software projects are often broken down into smaller tasks, efficient file sharing becomes important to project coordination. DFSS not only allows users to share files at the network manager, it also allows a user's files to be declared private.

Before accessing the shared disk, each user must be identified to the network manager through a log-on procedure. This procedure establishes a unique user ID that is subsequently used to control access to all files created by that user. A user that has logged onto the network has exclusive access to all of his private files as well as access to all public files, but is denied access to another user's private files. In this way, a user's private files are protected against modification by any other user.

Files declared public are write-protected to ensure file integrity; they can be concurrently accessed by any user logged onto the network. Private files can also be maintained on a user's individual workstation floppy disk. Utilities are available to the user to convert private files to public files and vice versa.

The DFSS improves project productivity by allowing complete modules that have been tested and debugged in a user's private file space to be converted to public files and then integrated and tested with other independently developed software modules. Modules declared as public files are guaranteed to be the most current version of the debugged module, and a common data base of completed modules is ensured. Each user can develop portions of the program in a private workspace with guaranteed file protection and can use the public files for integration and testing of the module under development.

The DFSS also increases coordination of the project team. For example, a design project to produce a new terminal built around the iAPX 86 may involve ten engineers, each developing different portions of hardware and software. Two engineers could be working in parallel on the keyboard centered around an 8041, one developing and debugging the keyboard software in his private workspace on DFSS, the other developing the keyboard hardware.

Public and private filing

Private files can be used for individual workspace and public files to store commonly used data and software. As a result, both engineers can proceed without fear of inadvertent modification of their private files either by each other or by any other users of the system.

After the software designer has completed the keyboard program for the 8041, the file can be declared public, which allows the keyboard-hardware designer to access it. At this point, the hardware engineer can integrate and test the software combined with keyboard hardware and ICE-41 at his workstation. (ICE-41 is an in-circuit emulation tool that enables emulation of a partial design.) Even with just the new keyboard and software, the ICE-41 shares the resources of the development system, allowing emulation of all keyboard functions, and gathers trace data to help with the final debugging of the combined keyboard hardware and software.

All the ICE data files can be kept in the user's private workspace, invisible to and protected from other DFSS users. The final version of the 8041 keyboard software can replace the first version as a public file, ready for use as needed by other members of the project team. In addition, the public files can be broken into sectors by disk platter so that one group of programmers can be designated by the project manager to exclusively share a group of public files.

While all this is going on, several other engineers may be working on the software to run on the main processors in the new terminal. Programmer A, who has already completed several modules that will run on the 8086, declares them public files. Programmer B, who is developing software for the 8087 high-speed math processor, then can easily include some of A's

files that interface the 8087 with his 8087 modules to facilitate debugging. The completed modules of programmer A are protected as public files against accidental modification by programmer B. This protection gives programmer A security for all his software and programmer B the facility to borrow completed pieces of the software jigsaw puzzle to help complete his own portion.

Another programmer working on the 8086 software, programmer C, may discover in the public files that programmer A has completed a module similar to one about to be written. The DFSS allows programmer C to transfer a copy of the similar program into his private workspace, edit and debug the required modifications to suit his particular application, and add this new and complete module to the set of public files.

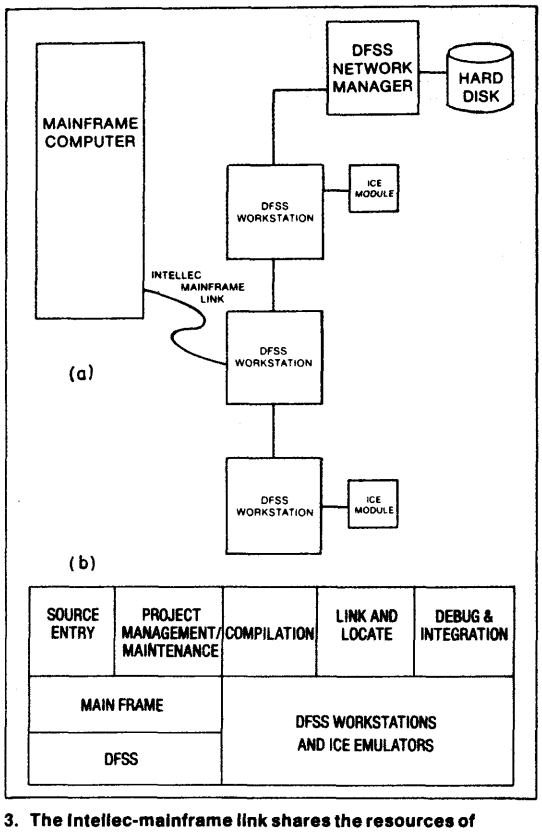
Although it is difficult to quantify, there is no doubt that the file-protection capability and the synergy developed from sharing other programmers files must increase overall project productivity as well as the quality of software.

In addition to being beneficial to the development of one large project, public-private file control facilitates software development of smaller, unrelated projects. For individual projects, programs can be developed and debugged in the user's private workspace. Commonly used utilities and compilers can be accessible as public files to eliminate the necessity of redundant files at each workstation.

Further extending the flexibility of DFSS, a user may connect a workstation to a large mainframe or minicomputer with the Intellec-mainframe link. The Intellec mainframe-link option is designed to integrate the power of microcomputer development systems with mainframes for extra-large development projects. Project-management and source-entry needs can be tailored to meet project needs by the use of large mainframes.

The Intellec mainframe link option functions by using IBM's 2780/3780 Bisync communications protocol for file transfers between the DFSS workstation and the mainframe. The high-speed protocol includes error checking for fast, reliable transfer of both source and object files. The 2780/3780 RJE is available on most mainframes and large minis at least as an option (Fig. 3a).

The development of high-performance, cost-effective software can be diversified on the mainframe and the DFSS. The important project-management facilities are generally found on mainframes, including file sharing and documentation aids. In a sample configuration, both the mainframe and the DFSS are used for file sharing and maintenance. Later, as software begins to take shape, the DFSS is used to compile, link and locate, and debug the programs.



3. The Intellec-mainframe link shares the resources of microcomputer development systems on the DFSS and a mainframe for large project development. The DFSS and the mainframe are used for their most efficient functions: mainframe for source entry end project management, and DFSS workstations for compilation, link and locate, debugging, and system integration.

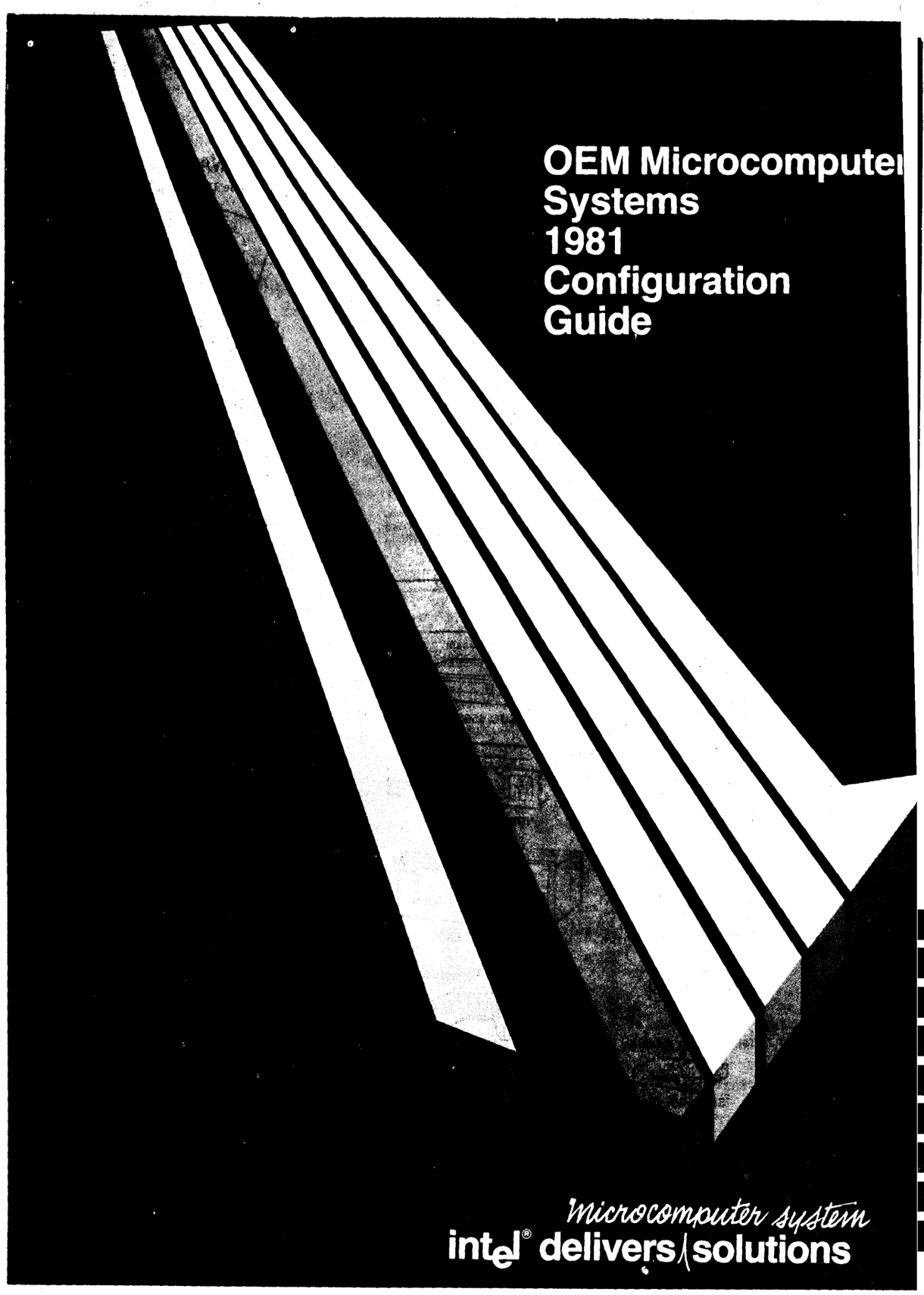
While this happens, the mainframe serves to carry on software editing and documentation.

Finally, system integration of the hardware and software takes place on DFSS workstations with Intel's full range of in-circuit emulators (Fig. 3b). The Intellec mainframe link, combined with the DFSS, allows cost-effective support facilities to be tailored to a particular project's needs and resources. It also allows mainframe users to design with state-of-the-art processors where software and hardware support are immediately available on Intellec development systems. □

INTEL CORPORATION, 3065 Bowers Ave., Santa Clara, California 95051

Printed in U.S.A./H-291/20K/0980/HP

BOARD COMPUTERS



Intel delivers solutions . . . including the industry-standard MULTIBUS™ system bus.

Intel created the MULTIBUS™ architecture which has become a de facto industry standard while standing as the foundation upon which the iSBC™ single board computer family of OEM microcomputer systems is built. Now, an IEEE committee is preparing certification for the universal MULTIBUS bus structure.

The bus structure is one of the most important elements in a computer system, as it contains all the necessary signals to allow the various system components to interact with each other. For example, it allows memory and I/O data transfers, direct memory access (DMA), generation of interrupts, and much more. In 1976, Intel introduced the MULTIBUS system bus as the flexible bus structure used to interface the family of 8-bit (iSBC 80) products. In 1978, when the 16-bit iSBC 86 products were developed by Intel, they were also MULTIBUS-compatible. The current 8-bit and 16-bit families now include single board computers, memory expansion boards, digital and analog I/O boards and peripheral controllers. The MULTIBUS system bus supports direct addressability up to one megabyte through 20-bit addresses and 8-bit and 16-bit data transfers.

Since its announcement, the MULTIBUS architecture has been accepted as a standard throughout the microcomputer system industry. Intel has constantly reinforced its corporate commitment to the MULTIBUS system bus with introduction after introduction of newer, more sophisticated MULTIBUS-compatible components and OEM-oriented board-level products. Currently, the world's largest engineering association — the Institute of Electrical and Electronics Engineers (IEEE) — is preparing to approve the MULTIBUS system bus as a recognizable, definable, certified standard (number 796) for computer architecture.

The Intel® MULTIBUS™ architecture provides support for high-performance system configurations such as Multi-master, master and intelligent slave, and master and slave.

The bus structure is built upon the master-slave concept in that the master device in the system takes control of the MULTIBUS interface and the slave device — upon decoding its address — acts upon the command provided by the master. This handshake between master and slave devices allows modules of different speeds to use the MULTIBUS interface and allows data rates of up to 5-million transfers per second.

Another important MULTIBUS interface attribute is the ability to connect multiple master modules for multiprocessing configurations. The MULTIBUS interface provides control signals for connecting multiple masters — either in a daisy-chain priority fashion or in parallel. With the parallel arrangement, up to 16 masters may share MULTIBUS bus resources in a multiprocessing environment.

Because of the power and flexibility of the MULTIBUS system bus, it provides the framework for a wide spectrum of applications ranging from simple relay logic replacement to sophisticated multiprocessing systems supporting high-speed peripherals.

The MULTIBUS™ system bus below graphically illustrates the Intel® iSBC™ single board computer family.

The Table of Contents is contained in the MULTIBUS symbol below. The following pages will provide descriptions and clarification on the breadth and capabilities of all the products which work together on the universal MULTIBUS system bus.

Single Board Computers Pages 8 and 9

iSBX™ Bus and MULTIMODULE™ Boards Pages 10 and 11

Memory Expansion Pages 12 and 13

Communications and Arithmetic Processing Pages 18 and 19

Peripheral Controllers Pages 20 and 21

Run-Time and Operating Systems Software Pages 24, 25, 26 and 27

Development Software and Hardware Pages 28 and 29

iSBX™ Bus Multimodule™ Boards

Parallel I/O — iSBX 350

Serial I/O — iSBX 351

Floating Point Math — iSBX 332

Fixed/Floating Math — iSBX 331

Diskette Controller — *iSBX 218

Analog Input — *iSBX 311

Analog Output — *iSBX 328

iSBX™ Bus

iSBX™ Bus

iSBC Extension Multimodule™ Boards

32K RAM — iSBC 300

Math Co-Processor — *iSBC 337

16K EPROM — iSBC 340

4K RAM — iSBC 301

16K EPROM — *iSBC 341

Math Co-Processor — *iSBC 337

4K RAM — iSBC 301

Single Board Computers

iSBC 80/04

iSBC 80/05

iSBC 80/10B

iSBC 80/20-4

iSBC 80/24

iSBC 80/30

iSBC 86/12A

*iSBC 88/40

MULTIBUS™ System Bus

System Expansion Modules

RAM Memory
- iSBC 064
- iSBC 048
- iSBC 032
- iSBC 016

CMOS RAM Memory with Battery Back-up
- iSBC 094

iSBC 090 (Storage System) Interface
- Memory Storage Unit

EPROM/ROM Boards
- iSBC 464
- iSBC 416

Bubble Memory
- iSBC 250

Combination Memory and I/O
- iSBC 116A
- iSBC 108A

High-Speed DMA
- iSBC 501

High-Speed Math
- iSBC 310

Serial I/O
- iSBC 544 Intelligent
- iSBC 534

Digital I/O
- iSBC 569 Intelligent
- iSBC 556
- iSBC 519
- iSBC 517
- iSBC 508

Analog I/O
- iSBC 732
- iSBC 724
- iSBC 711

Flexible Disk Controllers
- iSBC 204
- iSBC 202
- *iSBC 208

Disk Controller
- iSBC 206

Industrial Chassis
- iCS 80

Analog Signal Conditioning
- iCS 910

Digital Signal Conditioning
- iCS 920
- iCS 930

Digital I/O Processor

iSBC 941

Teletypwriter Adapter

iSBC 530

System Chassis

iSBC 655

*iSBC 665/666

iSBC 660

Cardcage/Backplanes

iSBC 604/614

Power Supplies

iSBC 635/640

*See Configuration Guide Addendum

SOFTWARE

Run Time Operating Systems

*iRMX 88 Real-Time Multi-Tasking Executive for iSBC 86/12A, iSBC 88/40, iAPX 86, and iAPX 88 products.

iRMX 86 Real-Time Multi-Tasking Executive for iSBC 86/12A, iSBC 88/40, iAPX 86, and iAPX 88 products.

iRMX 80 Real-Time Multi-Tasking Executive for iSBC 80/10B, 80/20-4, 80/24, 80/30

Run-Time Language Support

iSBC 801 FORTRAN Run-Time Package for iRMX 80

iSBC 802 BASIC Configurable Interpreter for iRMX 80

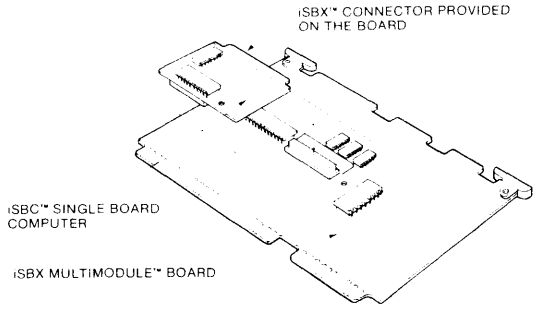
Development System Support

FORTRAN-80 ANS 77 Compiler

PL/M-80 High-Level Programming Language Compiler

PASCAL-80 Compiler

Intel delivers solutions . . . a new family of Multimodule™ boards and a new Intel bus standard, the iSBX™ bus, provide incremental, low-cost, on-board expansion for iSBC™ single board computers



The new iSBX™ Bus Multimodule™ boards connect directly to Intel single board computers by means of a special iSBX Bus connector.

A new generation of single board computers will be designed around yet another Intel standard, the iSBX™ bus.

Now, users of Intel's single board computers can incrementally expand system resources by adding small (2.85" × 3.7") iSBX Multimodule boards which plug directly into iSBC boards. Currently, the iSBX boards allow users to add capability to a single board computer in the areas of parallel I/O, serial I/O and advanced mathematics functions...without going to the expense of adding another full MULTIBUS expansion board.

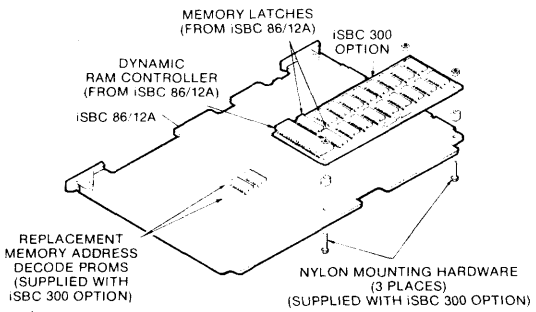
Customers can choose iSBX Multimodule boards to precisely configure single board computers for their individual applications at a lower cost than was previously possible. The iSBX boards enable users to buy exactly the capabilities they require for their iSBC-based systems, which keeps both system size and system cost at a minimum.

The iSBX bus facilitates the interface between iSBX boards and iSBC-based products. This new bus will definitely become an Intel standard similar to the MULTIBUS system architecture. The iSBX Bus allows system expansion through iSBX boards with minimum demand on the iSBC system's Multibus interface. As a result, the system design achieves maximum on-board performance while freeing up the Multibus interface for other system activities. Plus, the user can easily build a "custom-tailored" iSBX board with the use of a high-reliability, 36-pin iSBX male connector, which mates directly to the female iSBX bus connector on the single board computer.

The iSBX™ bus and Multimodule™ boards will be compatible with future 8-bit and 16-bit single board computers from Intel.

New iSBX bus-compatible Multimodule boards will be introduced on a regular schedule, along with new iSBC single board computers hosting the iSBX bus and connectors. Multimodule boards and the new iSBX bus represent a major commitment by Intel for the future and offer system designers new options to complement single board computers and Multibus expansion.

Intel has further extended the concept of Multimodule™ boards to a new family of iSBC memory add-ons.



Capacity of the already-powerful 16-bit iSBC 86/12A™ single board computer is increased with the simple attachment of an iSBC Multimodule™ board, such as the 32K byte expansion the iSBC 300™ RAM Multimodule board, shown here.

A second new family of Multimodules — designated iSBC Multimodule boards — is to be used for the lowest possible cost for expansion of memory and other specialized on-board functions. Each iSBC Multimodule board has a unique interface to its host iSBC board. The user merely unplugs one (or two) chips on his iSBC single board computer, and plugs the iSBC Multimodule into the now vacant socket(s); the connection is thus established between the iSBC Multimodule board and the host iSBC single board computer.

Use of the new iSBC memory Multimodules doubles on-board memory capacity of iSBC single board computers.

The iSBC Multimodules offer a new level of flexibility to system designers in defining and implementing system memory requirements. When the iSBC Multimodule is used (plugged into a host iSBC computer), on-board memory can be accessed as quickly as that of the existing single board computer memory. The need is eliminated for accessing the additional memory via the Multibus system bus, as the iSBC Multimodule has placed it directly on board the computer, in existing memory sockets.

Complete descriptions of both the new iSBX and iSBC Multimodule boards can be found on pages 10 and 11 of this Configuration Guide.

How to use this Configuration Guide

The purpose of this guide is to help you configure the optimum system for your application — whether your system is based on 8-bit Intel single board computers, the 16-bit iSBC 86/12A board, or a combination of the 8-bit and 16-bit products. The following 22 pages of the OEM Microcomputer Systems Configuration Guide are divided into nine sections. As you saw from the multi-colored Table of Contents as you opened the cover of this guide, the sections are: Single Board Computers, pages 8 and 9; the new iSBX™ Bus and MULTIMODULE™ boards, pages 10 and 11; Memory Expansion, pages 12 and 13; Digital I/O, Analog I/O and Signal Conditioning, pages 14 through 17; Communications and Arithmetic Processing, pages 18 and 19; Peripheral Controllers, pages 20 and 21; System Packaging and Power Supplies, pages 22 and 23; Run-Time and Operating Systems Software, pages 24 through 27; and Development Software and Hardware, pages 28 and 29.

You will notice that most sections include product summaries, which are provided in "scorecard" fashion. These "scorecards" are also reproduced in a Summary of Configuration Parameters, which you will find in the accompanying OEM Microcomputer Systems Project Configuration Workbook. The Workbook may be used in conjunction with this guide to allow you to construct the optimal system for your application. Its instructions will make it easy for you to map your system's capacity, performance and power requirements with off-the-shelf Intel product alternatives. The Workbook includes sufficient information and worksheets to enable you to conveniently determine your system's input and output requirements, single board computer and expansion boards to fit your needs, the accessories and packaging appropriate for your system, and even provides System Memory and I/O Maps enabling you to estimate your system's memory requirements. When you have completed the Workbook's Project Configuration Worksheet, you will have successfully configured the optimum system for your application. Your nearby Intel Sales and Applications Engineers will be happy to provide any assistance you require in configuring and pricing your optimum system.

Single Board Computers ...Pages 8 and 9

Many OEM applications can be solved with just one iSBC single board computer. There are six 8-bit iSBC 80 single board computers and one 16-bit iSBC 86/12A single board computer. Each computer board provides full computer capabilities. All the single board computers and expansion modules in the iSBC family (with the exception of the iSBC 80/04 single board computer) are provided in the same form factor on a 6.75" × 12" printed circuit card. Moreover, each may be housed in MULTIBUS-compatible cardcages.

iSBX™ Bus and Multimodule™ Products . . . Pages 10 and 11

The new iSBX Multimodule board products include parallel I/O, programmable serial I/O, a floating point math capability as well as another high-speed arithmetic board for fixed and floating point mathematics, and a 36-pin iSBX male connectors for OEM customizations of iSBX boards. Moreover, three iSBC Multimodule boards are presented, offering 4K byte RAM, 32K byte RAM, and 16K byte EPROM expansion.

Memory Expansion ...Pages 12 and 13

Expansion memory modules include 16K, 32K, 48K and 64K byte dynamic RAM memory boards; an add-on memory system with storage capacity up to 1 Megabyte; a 4K byte CMOS RAM battery back-up board; two EPROM memory boards; two combination RAM, EPROM and I/O expansion boards; and a new bubble memory board.

Digital I/O Expansion and Signal Conditioning...Pages 14 and 15

These expansion boards include a DMA channel controller, a general purpose I/O board, a combination I/O expansion board, a programmable I/O board, an optically isolated I/O board, and an intelligent digital controller. Additionally, two signal conditioning/termination panels are presented (digital signal conditioning and termination and AC signal conditioning and termination), as well as a 40-pin industrial digital I/O processor.

Analog I/O Expansion and Signal Conditioning...Pages 16 and 17

There are three MULTIBUS-compatible analog I/O expansion boards — for analog input, analog output, and a combination of both input and output on one board. An analog signal conditioning/termination panel is also described.

Communications and Arithmetic Processing...Pages 18 and 19

This section of the guide describes an off-the-shelf teletypewriter adapter, a 4-channel programmable communications board, and an intelligent communications controller board. Moreover, page 19 includes a description of a high-speed mathematics board which serves as a slave processor for iSBC MULTIBUS masters.

Peripheral Controllers ...Pages 20 and 21

These products provide a family of interfaces between the MULTIBUS system bus and a wide variety of industry-standard mass storage devices. Included is a 2-board double density diskette controller; a flexible diskette controller; and a 2-board disk controller.

System Packaging and Power Supplies . . . Pages 22 and 23

System packaging products include a 4-slot system chassis, a system chassis with eight MULTIBUS slots, and a special industrial chassis. Plus, there are two modular cardcage/backplane assemblies, two power supplies, and a MULTIBUS-compatible prototyping board.

Operating System Run-Time Software... Pages 24, 25, 26 and 27

On pages 24 and 25, the powerful new iRMX™ 86 real-time operating system is described for 16-bit systems. Also, there is a new interface and execution package which adds a "virtual terminal" capability to Intellec® Microcomputer Development Systems. On pages 26 and 27, systems software for 8-bit systems, spearheaded by the iRMX 80 real-time multi-tasking Executive is described. There is also a FORTRAN run-time package and a BASIC-80 configurable disk-based interpreter.

Development Software and Hardware... Pages 28 and 29

Presented in the final section of the Configuration Guide are the Intel Development System products which provide a range of solutions to minimize development time for single board computer applications — in terms of hardware and software development, debug and integration. The Intellec® Microcomputer Development Systems are described, as well as the powerful ISIS-II Operating System and a universal PROM programmer. A variety of sophisticated development software aids are presented, including FORTRAN-80, ASM 80, FSP, BASIC-80, PASCAL-80, PL/M-80 high-level languages and assemblers, plus run-time packages for iRMX-based systems.

Intel support is readily available.

Microcomputer Training Courses

Intel offers comprehensive, regularly-scheduled training courses providing "hands-on" experience in workshops tailored to a variety of audiences. One of the many 4-day courses which are available is specifically designed for engineers planning to utilize iSBC single board computers in their applications.

INSITE™ Microcomputer User's Library

Through the INSITE User's Library, Intel makes a broad collection of programs, procedures and macros written for the 8080/8085, 8086/8088 and 8048 processor families available. These general purpose programs can substantially cut programming and debugging time for INSITE users. The library has hundreds of programs, games and utilities and is continually updated. The INSITE Library is created by Intel customers, who exchange their own programs or routines for selections from the INSITE Library. Programs from the library are also available at a modest charge.

Worldwide technical support

Intel provides Field Application assistance at most field locations throughout the world. These application engineers provide technical expertise to assist you in your development efforts with Intel single board computers, Intel software, and Intellec systems.

Service Centers

Intel has established a network of service centers throughout the United States. Trained Customer Service Representatives are on call for site service and repair.

Comprehensive Intel documentation

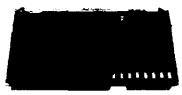
Intel supports all its single board computers with full documentation to assist the OEM in design. This documentation includes technical literature, reference cards, reliability reports, application notes, and both hardware and software manuals for single board computer users.

Single Board Computers

iSBC 86/12A™

iSBC 86/12A™ SINGLE BOARD COMPUTER

- 8086 16-bit HMOS central processing unit; 5MHz clock rate
- 32K bytes of dual-port dynamic RAM expandable to 64K bytes via iSBC 300 MULTIMODULE board
- Sockets for up to 16K bytes of EPROM; expandable to 32K bytes via iSBC 340 MULTIMODULE board
- 24 programmable parallel I/O lines
- Programmable synchronous/asynchronous communications interface
- Two programmable 16-bit BCD or binary interval timers/event counters.



iSBC 80/30™

iSBC 80/30™ SINGLE BOARD COMPUTER

- 8085A central processing unit; 2.76 MHz clock rate
- 16K bytes of dual port dynamic RAM
- Sockets for up to 8K bytes of EPROM
- 24 programmable parallel I/O lines
- Programmable synchronous/asynchronous communications interface
- Two programmable 16-bit BCD or binary interval timers/event counters
- Socket for 8041A/8741A Universal Peripheral Interface

iSBC 80/24™

iSBC 80/24™ SINGLE BOARD COMPUTER

- 8085A-2 central processing unit; 4.84 MHz clock rate
- Two iSBX bus connectors for iSBX MULTIMODULE expansion
- 4K bytes of static RAM; iSBC 301 MULTIMODULE expandable to 8K bytes
- Sockets for up to 32K bytes of EPROM
- 48 programmable parallel I/O lines
- Programmable synchronous/asynchronous serial communications interface
- Two programmable 16-bit BCD or binary interval timers/event counters

iSBC 80/20-4™

iSBC 80/20-4™ SINGLE BOARD COMPUTER

- 8080A central processing unit; 2.15 MHz clock rate
- 4K bytes of static RAM
- Sockets for up to 8K bytes of EPROM
- 48 programmable parallel I/O lines
- Programmable synchronous/asynchronous serial communications interface
- Two programmable 16-bit BCD or binary interval timers/event counters
- Programmable interrupt controller

Single Board Computers

SINGLE BOARD COMPUTERS

Product	CPU	Clock Rate	RAM (bytes)	EPROM (bytes)	Serial I/O Ports	Lines
iSBC 86/12A	8086 (16-bit)	5.00 MHz	32K dual port	16K(2732) 8K(2716) 4K(2758)	1 RS232C	24
iSBC 80/30	8085A (8-bit)	2.76 MHz	16K dual port	8K(2732) 4K(2716) 2K(2708/2758)	1[2] RS232C	24 (42)[2]
iSBC 80/24	8085A-2 (8-bit)	4.84 MHz	4K	32K(2764) 16K(2732) 8K(2716) 4K(2708/2758)	1 RS232C	48
iSBC 80/20-4	8080A (8-bit)	2.15 MHz	4K	8K(2716) 4K(2708/2758)	1 RS232C	48
iSBC 80/10B	8080A (8-bit)	2.05 MHz	1K with sockets to 4K	16K(2732) 8K(2716) 4K(2708/2758)	1 TTY or RS232C	48
iSBC 80/05	8085A (8-bit)	1.97 MHz	512	4K(2716) 2K(2708/2758)	1	22
iSBC 80/04	8085A (8-bit)	1.97 MHz	256	4K(2716) 2K(2708/2758)	1	22

[1] Does not include power for EPROM ROM I/O Line Drivers Terminators and other optional components

[2] The optional 8041 8741A Universal Peripheral Interface (UPI-41 A) provides 18 I/O lines for use as parallel and or serial I/O

iSBC 80/10B™

iSBC 80/10B™ SINGLE BOARD COMPUTER

- 8080A central processing unit; 2.05 MHz clock rate
- One iSBX connector for iSBX MULTIMODULE expansion
- 1K bytes of static RAM with sockets for expansion up to 4K bytes
- Sockets for up to 16K bytes of EPROM
- 48 programmable parallel I/O lines
- Programmable synchronous/asynchronous serial communications interface
- 1.04 millisecond interval timer

iSBC 80/05™

iSBC 80/05™ SINGLE BOARD COMPUTER

- 8085A central processing unit; 1.97 MHz clock rate
- 512 bytes of static RAM
- Sockets for up to 4K bytes of EPROM
- 22 programmable parallel I/O lines
- Two serial I/O lines
- Programmable 14-bit binary interval timer/event counter

iSBC 80/04™

iSBC 80/04™ SINGLE BOARD COMPUTER

- 8085A central processing unit; 1.97 MHz clock rate
- 256 bytes of static RAM
- Sockets for up to 4K bytes of EPROM
- 22 programmable parallel I/O lines
- Two serial I/O lines
- Programmable 14-bit binary interval timer/event counter
- Optimized for standalone applications

Parallel I/O Connectors	Timers	Interrupts	Multibus Expansion	Multimodule Expansion	Software Support	Power Requirements[1]				On-Board CPU Addressing Capability	Multibus Transfer Mode	On-Board RAM Multibus Access	On-Board Dedicated Addresses (max)
						+5V	+12V	-5V	-12V				
1	2	9 Levels expandable to 65 16 Sources	Multimaster	iSBC 300 - 32K RAM iSBC 340 - 16K EPROM	iRMX 86	5.2A	350mA		40mA	0-1M	8 16-bit	0-1M	C0-DEH(I/O) FC000-FFFFFH(ROM) 0-07FFFH(RAM)
2	2	12 Levels 18 Sources	Multimaster	N/A	iRMX 80	3.5A	220mA	2.5mA	50mA	0-64K	8-bit	0-1M	ED8-EFH(I/O) 0-1FFFH(ROM) any 16K boundary(RAM)
2	2	12 Levels 23 Sources	Multimaster	2 iSBX bus iSBC 301 - 4K RAM	iRMX 80	3.3A	40mA		20mA	0-64K	8-bit	NA	E4-EFH(I/O) 0-7FFFH(ROM) any 16K boundary(RAM) C0-CFH(iSBX) F0-FFH(iSBX)
2	2	8 Levels 26 Sources	Multimaster	N/A	iRMX 80	4.0A	90mA	2mA	20mA	0-64K	8-bit	N/A	E4-EFH(I/O) 0-1FFFH(ROM) any 16K boundary(RAM)
2	0	1 Level 11 Sources	Limited master	1 iSBX bus	iRMX 80	2.0A	150mA	2mA	175mA	0-64K	8-bit	N/A	E4-EFH(I/O) 0-3FFFH(ROM) 3C00-3FFFH(RAM)
1	1	4 Levels 12 Sources	Multimaster	N/A	—	1.8A	—	—	—	0-64K	8-bit	N/A	0-7H(I/O) 0-0FFFH(ROM) 3E00-3FFFH(RAM)
1	1	4 Levels 4 Sources	N/A	N/A	—	600mA	—	—	—	N/A	N/A	N/A	0-07H(I/O) 0-0FFFH(ROM) 3F00-3FFFH(RAM)

iSBX™ Bus and Multimodule™ Boards

iSBX 350™

iSBX 350™ PARALLEL I/O MULTIMODULE™ BOARD

- 24 programmable I/O lines using the Intel 8255A-5
- Sockets for interchangeable line drivers/terminators
- Three jumper selectable interrupt request sources to host processor
- Single +5V low power requirement

iSBX 351™

iSBX 351™ PROGRAMMABLE SERIAL I/O MULTIMODULE™ BOARD

- Provides serial communications capability using 8251A USART (universal synchronous/asynchronous receiver/transmitter)
- Serial interface RS232C or RS422 buffered
- Software programmable baud rate generator
- Two programmable 16-bit BCD/binary timers
- 4 jumper selectable interrupt requests to the host board

iSBX 332™

iSBX 332™ FLOATING POINT MATHEMATICS MULTIMODULE™ BOARD

- Uses Intel 8232 Floating Point Processor at 4MHz
- Compatible with the proposed new IEEE floating point format and existing Intel standard single-precision (32-bit) and double precision (64-bit) arithmetic and data manipulation functions
- Add, subtract, multiply and divide
- End-of-operation and error interrupts to host processor
- Software reset control
- High speed math (for example, 32-bit multiply in 50 μsec)

iSBX 331™

iSBX 331™ FIXED/FLOATING POINT MATHEMATICS MULTIMODULE™ BOARD

- Uses Intel 8231 Arithmetic Processing Unit at 4 MHz
- Fixed point single (16-bit) or double (32-bit) precision arithmetic functions
- Floating point double-precision (32-bit) arithmetic functions
- Add, subtract, multiply and divide
- Software reset control
- High speed math (for example: 16-bit fixed point multiply in 24 μsec or 32-bit floating point multiply in 42 μsec)
- Trigonometric and inverse trigonometric functions
- Square root, log and exponential functions
- End of operation interrupt to host processor

iSBX 960-5™ MALE CONNECTORS

- 36-pin male connectors which mate directly to the female MULTIMODULE/iSBX Bus connector on single board computer
- Package of five connectors
- Allows easy implementation of custom MULTIMODULE boards

iSBX™ Bus and Multimodule™ Boards

intel® delivers solutions...low-cost, incremental on-board memory expansion with iSBC™ Multimodule™ boards

iSBC 301™

iSBC 301™ 4K BYTE RAM MULTIMODULE™ BOARD

- Doubles the on-board memory for iSBC 80/24™ board to 8K bytes
- Provides 4K bytes of static RAM that plugs directly on the iSBC 80/24 board
- Uses high-speed 5MHz Intel 8185-2 RAMs
- Single +5V power supply
- 0.5 watts incremental power dissipation
- Measures 3.95" × 1.2" and mounts above the RAM area in sockets on the iSBC 80/24 single board computer

iSBC 300™

iSBC 300™ 32K BYTE RAM MULTIMODULE™ BOARD

- Doubles the on-board RAM memory for the iSBC 86/12A board from 32K to 64K bytes
- Provides 32K bytes of dual port dynamic RAM memory
- 0.3 watts incremental power dissipation
- Measures 7.75" × 2.35" and mounts above the RAM area on iSBC 86/12A board

iSBC 340™

iSBC 340™ 16K BYTE EPROM MULTIMODULE™ BOARD

- Doubles the on-board EPROM memory for the iSBC 86/12A to 32K bytes
- Provides sockets for up to 16K bytes of EPROM memory
- Supports Intel 2732 EPROM as supplied by Intel
- Measures 3.3" × 2.8" and mounts above the EPROM area on iSBC 86/12A board.

Memory Expansion

iSBC 032™

iSBC 016™, iSBC 032™, iSBC 048™ AND iSBC 064™ RAM MEMORY BOARDS

- 16K, 32K, 48K or 64K bytes of dynamic read/write memory with on-board refresh
- Independent address selection for each 16K byte bank

iSBC 094™

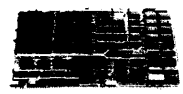
iSBC 094™ 4K BYTE CMOS RAM BATTERY BACK-UP BOARD

- Contains 4K bytes of read/write memory, using CMOS RAMS
- On-board rechargable batteries and charging circuitry insure that data in RAM will be retained for at least 96 hours after power is removed

iSBC 090™ ADD-ON MEMORY SYSTEM

- Storage capacity up to one Megabyte: 128K, 256K, 384K, 512K, 768K and 1024K
- ECC — single-bit error correction, double-bit error detection
- Includes standalone storage system, interface card and interconnecting cable
- Error logging and display
- Field expandable memory capacity
- Storage system dimensions: 5.25" high, 19" wide, 19.5" deep
- Starting/ending addresses switch-selectable to any 4K increment

iSBC 090™



iSBC 464™

iSBC 416™ AND iSBC 464™ EPROM MEMORY BOARDS

- Contains 16 sockets for up to 16K or 64K bytes of EPROM
- EPROM memory can be added in 1K, 2K or 4K byte increments

Memory Expansion

MEMORY EXPANSION

Product	Memory Type	Memory Size (bytes)	Access Time (nsec)
iSBC 016	Dynamic RAM	16K	690
iSBC 032	Dynamic RAM	32K	450
iSBC 048	Dynamic RAM	48K	450
iSBC 064	Dynamic RAM	64K	450
iSBC 090*	Dynamic RAM with ECC	128K-1M	450
iSBC 094	CMOS RAM	4K	750
iSBC 250	Bubble Memory	128K	48Msec (avg.)
iSBC 416	EPROM 2708	0-16K	Selectable
iSBC 464	EPROM 2708/2758 2716/2732	0-64K	Selectable

COMBINATION MEMORY AND I/O EXPANSION

Product	RAM Memory (bytes)	EPROM Memory (bytes)	Serial I/O Ports	Parallel I/O Lines	Parallel I/O Connectors
iSBC 108A	8K	32K (2764) 16K(2732) 8K(2716) 4K(2708/2758)	1 RS232C	48	2
iSBC 116A	16K	32K (2764) 16K(2732) 8K(2716) 4K(2708/2758)	1 RS232C	48	2

Does not include power for EPROM, I/O Line Drivers, Terminators and other optional components.

iSBC 116A™

iSBC 108A™ AND iSBC 116A™ COMBINATION RAM, EPROM AND I/O EXPANSION BOARDS

- Sockets for up to 32K bytes of EPROM
- Available with 8K or 16K bytes of dynamic RAM with on-board refresh
- 48 programmable parallel I/O lines
- Programmable synchronous/asynchronous serial communications interface
- Jumper selected 1 ms interval timer

iSBC 250™

iSBC 250™ BUBBLE MEMORY BOARD

- 128K bytes with on-board controller
- Non-volatile solid-state storage
- Direct Multibus interface
- Automatic error detection and correction
- Power-fail circuit protects bubble memory and stored data
- Average access time 40 milliseconds
- Fully compatible with iSBC 80 and iSBC 86 systems
- Operates using ±12 and +5 volt supplies

		Power Requirements			Software Support	iSBC 86 Compatibility	
	Data Retention	+5V	+12V	-5V		Multibus Transfer Mode	Multibus Address Range
	—	1.5A	1.0A	3.2mA	—	8-bit	0-64K[2]
	—	3.2A	600mA	10mA	—	8/16-bit	0-1M
	—	3.2A	600mA	10mA	—	8/16-bit	0-1M
	—	3.2A	600mA	10mA	—	8/16-bit	0-1M
	—	—	—	3.0A[4]	—	8/16-bit	0-1M
	96 hours	1.7A	—	—	—	8/16-bit	0-64K[2]
	Non-volatile read/write memory	2.7A	0.5A	—	—	8-bit	0-255
	Permanent	750mA[3]	—	—	—	8-bit	0-64K[2]
	Permanent	1.1A[3]	—	—	—	8/16-bit	0-1M

Timers	Interrupts	Power Requirements[1] +5V	+12V	-5V	-12V	Multibus Transfer Mode	Software Support	iSBC 86 Compatibility Multibus Address Range	Assignable I/O Address Range	Number of I/O Addresses Dedicated	Default Starting I/O Address
1	1 Level 9 Sources	2.9A	250mA	—	70mA	8-bit	—	0-1M	0-FFFH[5]	16	ODOH
1	1 Level 9 Sources	2.9A	250mA	—	70mA	8-bit	—	0-1M	0-FFFH[5]	16	ODOH

[2] Device only decodes 16 address lines and therefore may be directly addressed from any 64K memory page.

[3] Device limits system I/O port range to 0-4095 (0-FFFH).

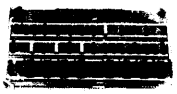
[4] iSBC 090 is an add-on expansion system consisting of a standalone storage system, interface card, and interconnecting cable. Requires 110/220 VAC power in addition to +5V power from Multibus backplane.

Digital I/O Expansion and Signal Conditioning

iSBC 501™

iSBC 501™ DMA CHANNEL CONTROLLER

- High-speed DMA control and interfacing for transfers between memory and up to 16 peripherals
- Block transfers up to 64K bytes long to or from RAM memory at rates up to 1-million bytes per second
- Software maskable interrupts
- 8-level priority interrupt



iSBC 508™

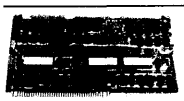
iSBC 508™ GENERAL PURPOSE I/O BOARD

- Four 8-bit terminated input ports which may be latched or unlatched
- Four 8-bit latched output ports which may be strobed

iSBC 517™

iSBC 517™ COMBINATION EXPANSION BOARD

- 48 programmable parallel I/O lines
- Programmable synchronous/asynchronous serial communications interface
- Eight maskable interrupt request lines with a pending interrupt register
- Jumper selectable 1 ms interval timer (real-time clock)



iSBC 519™

iSBC 519™ PROGRAMMABLE I/O BOARD

- 72 programmable parallel I/O lines
- Programmable interrupt controller for vectoring of eight interrupt levels
- Jumper selectable interval timer (real-time clock)

iSBC 556™

iSBC 556™ OPTICALLY ISOLATED I/O BOARD

- 48 optically isolated I/O data lines — 24 fixed input lines, 16 fixed output lines, and eight programmable lines
- Voltage/current levels for inputs are up to 48V, for outputs up to 30V @ 60mA with user-supplied compatible optically isolated receivers, drivers and input terminators
- Up to eight interrupt sources

iSBC 569™

iSBC 569™ INTELLIGENT DIGITAL CONTROLLER

- Standalone digital I/O controller or intelligent slave digital I/O expansion board with sockets for up to four processors on one board
- 8085A master processor for control algorithms and management of three UPI-41A Universal Peripheral Interface processors; 1.30 μs instruction cycle
- UPI-41A processors offload 8085A of common digital tasks
- 54 parallel I/O lines, programmable through three UPI-41A processors, either programmed by user or using preprogrammed Intel UPI's, such as iSBC 941 Industrial Digital I/O Processor or 8278 Printer Controller
- 2K bytes dual port static RAM
- Three independent programmable interval timers/counters
- Sockets for up to 8K bytes EPROM/16K bytes ROM

DMA CONTROLLER

Product	Transfer Rate		Maximum
	Block	Interleaved	Block size
iSBC 501	1MB/S	330 KB/S	65K

PARALLEL AND SERIAL I/O EXPANSION

Product	CPU	RAM (bytes)	EPROM/ ROM (bytes)	Serial I/O Ports	Parallel I/O Lines	
					Programmable	Dedicated In
iSBC 508	—	—	—	0	0	32
iSBC 517	—	—	—	1 RS232C	48	—
iSBC 519	—	—	—	0	72	—
iSBC 556	—	—	—	0	8	24
iSBC 941	8041A	64	1K	0	16	—

INTELLIGENT DIGITAL CONTROLLER

Product	CPU	Clock Rate	RAM (bytes)	EPROM/ROM (bytes)	Serial I/O Ports	Lines
iSBC 569	8085A	3.07 MHz	2K (dual Port)	16K(2364) 8K(2832) 4K(2716,2316E) 2K(2758)	1	54 3 UPI-41As

SIGNAL CONDITIONING (w/ optical isolation) AND SCREW

Product	Parallel Lines[1] In or Out
iCS 920	24 inputs or outputs
iCS 930	24 inputs or outputs

[1]Does not include power for EPROM, ROM, I/O Line Drivers, Terminators, and other optional components.
[2]Device limits system I/O port range to 0-4095 (OFFFH).
[3]The optional 8041-8741A Universal Peripheral Interface (UPI-41A) provides 18 I/O lines for use as parallel and/or serial I/O.

Digital I/O Expansion and Signal Conditioning

iCS 920™ DIGITAL SIGNAL CONDITIONING/ TERMINATION PANEL

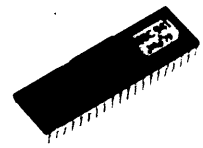
iCS 920™

- Interconnects iSBC board digital I/O ports to field signal/control wiring
- Ribbon cable connection from panel is pin-compatible with iSBC digital and CPU board I/O ports
- Mounting kits for 19" width RETMA rack, NEMA-type backwall and the iCS 80 Industrial Chassis
- Sockets for optically isolated input filters and solid state output switches
- Pad space for transient suppressors, current limiting resistors and voltage dividers

iCS 930™ AC SIGNAL CONDITIONING/ TERMINATION PANEL

iCS 930™

- Interconnects iSBC board digital I/O ports to field signal/control wiring
- Handles higher AC and DC currents and voltages (up to 280V, 3A)
- Ribbon cable connection from panel is pin-compatible with iSBC digital and CPU board I/O ports
- Mounting kits for 19" width RETMA rack, NEMA-type backwall and the iCS 80 Industrial Chassis
- Sockets for optically isolated input filters and solid state output switches plus socketed fuse for overload protection
- Pad space for transient suppressors, current limiting resistors and voltage dividers



iSBC 941™

iSBC 941™ INDUSTRIAL DIGITAL I/O PROCESSOR

- Measurement and control of the most common industrial digital I/O signals such as: change-of-state interrupt (16); pulse counting (8); frequency measurement (8); pulse outputs (8); stepper motor control (8); simplex serial I/O (2)
- 40-pin DIP package compatible with 8041A/8741A sockets
- Up to 18 parallel inputs/outputs; unused pulse, etc. inputs/outputs can be used for status sense or latched digital outputs
- Simple command programming interface to MCS 85/86 processors including compatibility with 8085A on the iSBC 80/30 computer and the iSBC 569 Intelligent Digital Controller

Interrupts	Power Requirements +5V	Transfer Range	MULTIBUS Transfer Mode	Assignable I/O Address Range	No. of I/O Addresses Used
1 level, 3 sources	3.35A	0-64K	8-bit	0-FFH[2]	32

Out	Connectors	Timers	Interrupts	Power Requirements[1] +5V	+12V	-5V	-12V	iSBC 86 Compatibility: Software Support	MULTIBUS Transfer Mode	Assignable I/O Address Range	No. of I/O Addresses Dedicated	Default Starting I/O Address	On-board Dedicated Addresses (max.)
32	1	—	1 level, 8 sources	2.6A	—	—	—	—	8-bit	0-FFH[3]	4	—	—
—	2	—	1 level, 8 sources	2.4A	40mA	—	60mA	—	8-bit	0-FFH[3]	16	00H	—
—	3	—	8 levels, 10 sources	1.5A	—	—	—	—	8-bit	0-FFH[3]	16	00H	—
16	2	—	1 level, 8 sources	1.0A	—	—	—	—	8-bit	0-FFH[3]	8	00H	—
—	UPI-414A sockets	—	—	—	—	—	—	On-Chip Firmware	—	—	—	—	—

Parallel I/O Connectors	Timers	Interrupts	Bus Interface	Power Requirements[1] +5V	+12V	-5V	-12V	On-board CPU Addressing Capability	MULTIBUS Transfer Mode	On-board RAM MULTIBUS Access	On-board Dedicated Addresses (max.)
3	3	12 levels, 22 sources	Single Master	2.6A	—	—	—	0-64K	8-bit	0-1M[2]	E0-EDH (I/O) 0-3 FFFH (ROM) 8000-87FFH (RAM)

TERMINATION PANELS

Parallel Input/Output: Connectors to iSBC	+5V Power Requirements	Compatible Isolators
1	23mA/line (inputs) or 61mA/line (outputs)	See "Optional Components" MCS2, T1L113, T1L117, T175472
1 or 2	12mA/line (inputs) or 61mA/line (outputs)	See "Optional Components" IAC5, IDC5, OAC5, ODC5

[2]0-1M addressability when configured as an intelligent slave. When configured as a single master, Multibus access is not applicable.
[3]Each Parallel Line includes a two-wire termination (plus and minus).

Analog I/O Expansion and Signal Conditioning

iSBC 711™

iSBC 711™ ANALOG INPUT BOARD

- Eight differential or 16 single-ended non-isolated inputs (expandable to 16 differential or 32 single-ended)
- 12-bit Analog-to-Digital (A/D) Converter
- Three modes of operation for acquisition of analog inputs
  - — Repetitive Single Channel Input
  - — Sequential Input Scan
  - — Random Channel Input
- A/D conversion can be initiated by external trigger, pacer clock or programmed I/O
- All input channels are protected up to ±28V via diode clamping together with fusible current limit resistors

iSBC 724™ ANALOG OUTPUT BOARD

- Four independent 12-bit Digital-to-Analog (D/A) converters
- Switch-selectable bipolar or unipolar ranges
- Jumper selectable D/A converter output voltage ranges
- Short circuit protection is standard for voltage inputs

iSBC 732™

iSBC 732™ COMBINATION ANALOG I/O BOARD

- Eight differential or 16 single-ended 12-bit analog inputs (expandable to 16 differential or 32 single-ended)
- Two 12-bit D/A converter output channels
- Provides 12-bit bipolar or unipolar resolution for either analog input or analog output
- Programmable gain amplifier
- A/D conversion can be initiated by external trigger, pacer clock or programmed I/O
- D/A converter providing voltage or current loop output

iCS 910™

iCS 910™ ANALOG SIGNAL CONDITIONING/ TERMINATION PANEL

- Screw terminations for 16 3-wire or 32 singled-ended analog inputs and four analog outputs
- Rack mountable in 19" RETMA Standard rack or NEMA backwall. Ribbon cable extension to iSBC boards.
- Pin-compatible with iSBC 711/724/732 Analog I/O boards
- Engineered signal conditioning component mounting space for analog input filters, voltage dividers, current-to-voltage inputs, etc.
- Plexiglass safety cover and signal labelling strip

Analog I/O Expansion and Signal Conditioning

ANALOG I/O EXPANSION

Product	Input Channels	Input Voltage Ranges	Input Current Ranges	Throughput Rate (Max.)	Programmable Gain
iSBC 711	8-16	±5, ±10 ±5, ±10 at 5mA	0-1mA 0-20mA 0-50mA	28KHz	1, 2, 4, 8
iSBC 724	--		-	—	—
iSBC 732	8-16	±5, ±10 ±5, ±10 at 5mA	0-1mA 0-20mA 0-50mA	28KHz	1, 2, 4, 8

ANALOG SIGNAL CONDITIONING AND SCREW TERMINA

Product	Connectors to iSBC	+5V Power Requirements
iSBC 910	1	—

LEGEND:
DI-Differential Input
SI-Single ended input
IO-Current Output
VO-Voltage Output
LOW-Low Level Differential Input

Output Channels	Output Ranges	Analog Connectors In	Analog Connectors Out	Resolution	Power Requirement +5V	Software Support	iSBC 86 Compatibility Multi-Page Access[2]	iSBC 86 Compatibility MULTIBUS Transfer Mode	Default Starting I/O Address	No. of Bytes Dedicated
—	—	2	—	12-bits	1.7A	iRMX 80 Drivers	Yes	8-bit	F700H	8
4	+5, +10 -5, +10 at 5mA	—	1	12-bits	2.0A	iRMX 80 Drivers	Yes	8-bit	F708H	8
2	+5, +10 -5, ±10 at 5mA 0-20mA	2	1	12-bits	2.3A	iRMX 80 Drivers	Yes	8-bit	F700H	16

TION PANELS

DI	SI	Analog Signals (see Legend) LOW	Analog Input/Output I_O	Analog Input/Output V_O	Connectors IN	Connectors OUT
16	32	—	2	4	1	1
or 3-wire						

Device only decodes 16 address lines and therefore may be directly addressed from any 64K memory page

Communications and Arithmetic Processing

iSBC 530™

iSBC 530™ TELETYPEWRITER ADAPTER

- Compatible with iSBC Single Board Computers and combination boards
- Jumper selectable RS232C data set or data terminal configuration
- Interface opto-isolated for high noise immunity
- Provides general-purpose RS232C to 20mA current loop interface
- Power Requirements: +12V @ 98mA max; -12V @ 98mA max.

iSBC 534™

iSBC 534™ FOUR-CHANNEL COMMUNICATIONS BOARD

- Four fully programmable synchronous and asynchronous serial communications channels (RS232C and 20mA current loop)
- 16-bit parallel I/O interface compatible with the Bell 801 Automatic Calling Unit (ACU)
- Each serial I/O channel has individual software-programmable baud rate generation
- 16 maskable interrupt request lines
- Two independent programmable 16-bit interval timers

iSBC 544™

iSBC 544™ INTELLIGENT COMMUNICATIONS CONTROLLER

- iSBC communications controller acts as a single board communications controller or an intelligent slave for communications expansion
- On-board dedicated 8085A CPU provides communications control and buffer management for four programmable synchronous/asynchronous channels (RS232C)
- 16K bytes of dual port dynamic RAM with on-board refresh
- Sockets for up to 8K bytes of EPROM
- 10 programmable parallel I/O lines compatible with Bell 801 Automatic Calling Unit (ACU)
- Three independent programmable interval timer/counters

Communications and Arithmetic Processing

HIGH-SPEED MATH UNIT

Product	Maximum Execution Time (Fixed Point Integer) Multiply	Divide	Extended Divide	Add	Subtract
iSBC 310	19	38	94	56	56

COMMUNICATION CONTROLLERS

Product	CPU	RAM (bytes)	EPROM/ROM (bytes)	Serial I/O Ports	Parallel I/O Lines	Parallel I/O Connectors
iSBC 534				4 CTS/DSR RS232C	24 RS232C	
iSBC 544	8085A	16K (8K ports / 8K Static	[illegible]	4 RS232C	10 RS232C	

*Does not include power for EPROM/ROM components
Device only decodes for address lines and therefore looks for [illegible] memory page

iSBC 310™

iSBC 310™ HIGH-SPEED MATHEMATICS UNIT

- High-speed arithmetic slave processor for iSBC 80 and iSBC 86 MULTIBUS masters
- Floating point arithmetic functions — add, subtract, multiply, divide, square and square root — in Intel standard 32-bit format
- Fixed-point integer arithmetic functions — multiply, divide and extended divide — in 16-bit and 32-bit functions

(Microseconds) Floating Point					Power Requirement	iSBC 86 Compatibility				No. of Bytes Dedicated
Multiply	Divide	Square	Square Root	Interrupts	+5V	Software Support iSBC 801	MULTIBUS Address Range[2]	MULTIBUS Transfer Mode	Default Starting I/O Address	
91	102	91	199	1-2 levels, 2 sources	6.7A max	FORTRAN (driver) Run-time Support	Yes	8-bit	98H	8

Timers	Interrupts	Power Requirements[1]				iSBC 86 Compatibility					No. of I/O Addresses Dedicated	Default Starting I/O Address	On-board Dedicated Addresses (Max.)
		+5V	+12V	-5V	-12V	Software Support	On-board CPU Address Capability	MULTIBUS Transfer Mode	On-board RAM MULTIBUS Access	Assignable I/O Address Range			
2	16 levels, 16 sources	1.9A	275mA	—	250mA	—	—	8-bit	—	0-FFH	16	00H	—
3	12 levels, 21 sources	3.4A	350mA	5mA	200mA	iRMX 80 (Future Release)	0-64K	8-bit	0-1M	Local Resources Only			D0-EFH (I/O) 0-01FFH (ROM) 7F00-BFFFH (RAM)

[1]Device limits system I/O port range to 0-4095 (0-FFFH).
0-1M addressability when configured as an intelligent slave. When configured as a single master, Multibus access is not applicable.

iSBC 202™

iSBC 202™ DOUBLE DENSITY DISKETTE CONTROLLER

- Controls industry standard double density diskette drives
- Microprogrammed controller independently executes command list set up by system master single board computer — minimal interaction is required between CPU and controller

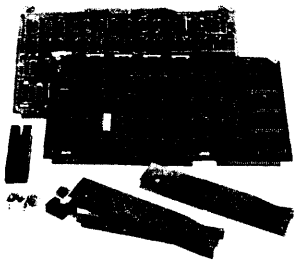
iSBC 204™

iSBC 204™ UNIVERSAL FLEXIBLE DISKETTE CONTROLLER

- Handles single density diskette drives, single or dual sided, in IBM 3740-compatible format and recording techniques
- Non-IBM formats may be used for greater data capacity on the media
- Also controls mini-diskettes

iSBC 206™ DISK CONTROLLER

- Handles industry standard 5440-type disk drives
- Full on-board buffering of each complete sector minimizes controller's real-time sensitivity, eliminates data overruns
- Controller self-diagnostic in on-board ROM pinpoints controller logic problems for rapid servicing
- Supports double-frequency (FM) and NRZ recording techniques
- Microprogrammed controller independently executes command lists generated by the system master single board computer, reducing system overhead



Peripheral Controllers

UNIVERSAL FEATURES

- All controllers generate and check CRC codes to guarantee data integrity
- Disks and diskettes may be user-formatted in sequential or user-specified sector order
- DMA is used for all controller-memory data transfers, freeing the CPU to carry out other tasks during disk and diskette read/write times

SOFTWARE SUPPORT

- All disk and diskette controllers are supported by the disk file system of iRMX 80, Intel's Real-Time Multi-Tasking Executive software system
- The iSBC 204 controller and iSBC 206 controller are supported by the extended I/O System of iRMX 86, Intel's 16-Bit operating system

DISK & DISKETTE CONTROLLERS

Product	Number of Boards	Recording Density	Bytes per Disk or Diskette drive (Formatted)	Number of Drives Supported	Compatible Drives	Software Support	Power Requirements[1]		iSBC 86 Compatibility			
							+5V	-5V	Transfer Range	MULTIBUS Transfer Mode	Assignable I/O Address Range	Number of I/O Addresses Dedicated
iSBC 202	2	Double	512K	4	Shugart 800-1	iRMX 80	7.5A	2A	0-64K	8 bit	0-FFH[3]	8
iSBC 204	1	Single	256K (Std. size single-sided) 512K (Std. size dual-sided) 80K (Mini-size)	4 Single-sided[2] 2 Dual-sided[2]	Shugart SA400, SA800, SA850 Memorex 550,552 CDC 9404 Pertec FD200 GSI 110 Wangco Mod 82, 765	iRMX 80 iRMX 86	2.5A	—	0-1MB	8 bit	0-FFFH	16
iSBC 206	2	Single (200 TPI)	10M (512 bytes/sector 7.5M) (128 bytes/sector) (capacity of one fixed and one removable platter per spindle)	4	CDC 9427H Diablo 44B Pertec D3422 Wangco ST2222 Caelus 306R	iRMX 80 iRMX 86	5.5A (typ) 6.5A (max)	—	0-1MB	8/16 bit	0-FFH[3]	8

[1]Does not include power for EPROM/ROM, I/O Line drivers Terminators and other optional components.
[2]Device limits system I/O port range to 0-4095 (0-FFFH).
[3]Requires addition of optional Intel® 8271 FDC circuit; one dual-sided or two single-sided drives are supported by the iSBC 204 as delivered.

System Packaging and Power Supplies

iSBC 655™

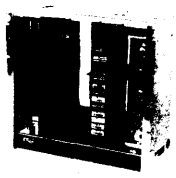
iSBC 655™ MICROCOMPUTER SYSTEM CHASSIS

- Complete microcomputer system chassis for Intel Single Board Computer System
- MULTIBUS-compatible backplane with four slots for standard or custom expansion boards, plus one auxiliary connector and provision for three additional auxiliary connectors
- 3.5-inch high, 19-inch wide rack-mountable chassis
- Front panel with control switches and indicator lights
- Heavy duty power supply
- Dual fans for cardcage and power supply cooling

iSBC 660™

iSBC 660™ MICROCOMPUTER SYSTEM CHASSIS

- Complete microcomputer system chassis for Intel Single Board Computer systems
- MULTIBUS-compatible backplane with eight slots for standard or custom expansion boards and provisions for optional auxiliary connector and provisions for seven additional auxiliary connectors
- 7-inch high, 19-inch wide rack-mountable chassis
- Front panel with control switches and indicator lights
- Heavy duty power supply
- Dual fans for cardcage and power supply cooling



iCS 80™

iCS 80™ INDUSTRIAL CHASSIS

- MULTIBUS system bus standard 4-slot backplane, expandable to 12 slots
- Vertical board orientation for convection cooling
- 19-inch wide RETMA rack mounting or NEMA type backwall mounting brackets
- Front access servicability
  - — iSBC boards
  - — Power supplies
  - — Interrupt and reset buttons
  - — Operation indicators and fuse
- Recessed mounting space for signal conditioning/wire termination panels

iSBC 614™

iSBC 604™ AND iSBC 614™ MODULAR BACKPLANE AND CARDCAGE

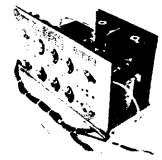
- Interconnects and houses up to four Intel iSBC boards per cardcage
- iSBC 604 cardcage provides terminator backplane
- iSBC 614 cardcage provides expansion backplane
- Compatible with 3.5-inch RETMA rack mount increments

System Packaging and Power Supplies

iSBC 635™

iSBC 635™ POWER SUPPLY

- Provides ±5 and ±12 volt system power
- Sufficient power for a fully loaded Intel Single Board Computer plus residual power for up to three iSBC expansion boards
- Single chassis



iSBC 640™

iSBC 640™ POWER SUPPLY

- Provides ±5 and ±12 volt system power
- Sufficient power for a fully loaded Intel Single Board Computer plus residual power for up to 11 iSBC expansion boards
- Single chassis

iSBC 905™ PROTOTYPING BOARD

- MULTIBUS-compatible form factor; mounts in iSBC 604/614 backplane/ cardcage assemblies
- Can accommodate up to 95 16-pin wire-wrapped sockets or equivalent mix of other standard DIP sockets
- Includes 100-pin I/O edge conector

SYSTEM CHASSIS

Product	Single Board Computer	Chassis Size	Input Power		Power Available*			
					+5V	+12V	-5V	-12V
iSBC 655	All MULTIBUS iSBC's & options	3.5" 4 slots	47-63Hz	110/115/215/230 VAC (±10%)	14A	2.0A	0.9A	0.8A
iSBC 660	All MULTIBUS iSBC's & options	7.0" 8 slots	47-63Hz	110/115/215/230 VAC (±10%)	30A	4.5A	1.75A	1.75A
iCS80 Kit 635	All MULTIBUS iSBC's & options	17.4" 4 slots- 12 slots	47-63Hz	110/115/215/230 VAC (±10%)	14A	2.0A	0.9A	0.8A
iCS80 Kit 640	All MULTIBUS iSBC's & options	17.4" 4 slots- 12 slots	47-63Hz	110/115/215/230 VAC (±10%)	30A	4.5A	1.75A	1.75A

MODULAR CARDCAGE

Product	No. of Slots	Terminators	Slot-to-Slot Spacing	Height
iSBC 604	4	Included	0.6"	3.5"
iSBC 614	4	Not Required	0.6"	3.5"

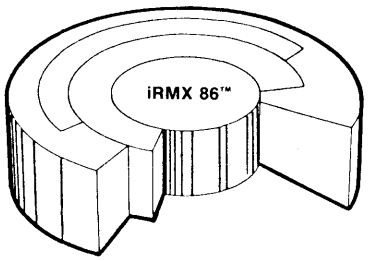
POWER SUPPLIES

Product	Input Power Requirements		Output Power Available			
	Frequency	Voltage	+5V	+12V	-5V	-12V
iSBC 635	47-63Hz	100/115/215/230 VAC	14.0A	2.0A	0.9A	0.8A
iSBC 640	47-63Hz	100/115/215/230 VAC	30A	4.5A	1.75A	1.75A

*Current availability is with no iSBC, EPROM/ROM, UPIA's or other board options installed

Resolving the software crisis

Intel's President Andy Grove has been widely quoted regarding an impending "programmer catastrophe", caused by low-cost microcomputers being designed into applications growing increasingly more complex. A solution is modular, easy-to-use, off-the-shelf software such as the new iRMX 86™ Operating System. In January 1980, Andy Grove signalled what he views as a coming software crisis. As production technologies such as VLSI (Very Large Scale Integration) proliferate — with their greater levels of density — the assignment of the economy-priced microcomputer hardware into the solving of increasingly complex applications has created a crisis in software. Grove has described the need for over 1-million programmers by the year 1990. Intel's President suggested that a partial solution to the programmer shortage would be the creation of "building blocks" for system development.



Intel delivers solutions: Software "building blocks"

The key to solving the software crisis rests on increasing the productivity of the existing system implementers. For years, Intel has provided an extensive range of software tools which center around the view of increasing customer productivity. Real-time executives, multi-programming Operating Systems and complete development software packages are a few of Intel's productivity products.

A major factor in each new application system is the resources which will be spent implementing software ...with indications that over 60% of the cost of each new project will be for software. To break the dependence of customers on a limited population of talented software engineers, Intel provides a series of OEM-oriented software tools.

A large percentage of each applications software investment is typically spent for "system software". System software provides that set of functions required to schedule tasks, control peripherals, and manage data. Intel's system software tools provide comprehensive libraries of frequently-used functions which the OEM may use...rather than having to "re-invent the wheel".

Because a large amount of time is spent in implementing application software, Intel has gone to great lengths to provide efficient tools which assist the OEM in building his software reliably. Intel has historically helped to solve the OEMs' problems with the right complement of products. Now, software represents one more example of Intel's comprehensive set of products designed specifically for OEM customers. Intel's productivity-oriented products take a major stride in eliminating the "software crisis".

The iRMX™ Real-Time Operating System is an easy-to-use, sophisticated software system which operates on user systems based on the 16-bit Intel® 8086 microprocessor and Intel® iSBC 86/12A™ boards. The operating system provides a comprehensive library of multi-programming, multi-tasking facilities for the 8086 and adds extensive file capabilities, including a device-independent I/O interface with hierarchia directories and with automatic file buffering. The iRMX 86 Operating System provides a structured, efficient environment for many applications, including process control, office systems, medical electronics, and data communications. Services provided by the operating system include facilities for executing programs concurrently, sharing resources and information, servicing asynchronous events, and interactively controlling system resources and utilities. The system also provides all major real-time facilities including priority-based system resource allocation; the means to concurrently monitor and control multiple external events; real-time clock control; interrupt management; and task dispatching.

16-Bit Run-Time Systems Software

The iRMX 86 Operating System contains the following modules: a Nucleus; a device-independent Input/Output subsystem; a Human Interface subsystem with command language interpreter and ASCII console interface; a Terminal Handler; and an interactive object-oriented Debugging subsystem. Because the modules and services provided by the operating system are user-selectable, application-specific operating systems can be created by iRMX 86 users. The iRMX 86 Operating System thus eliminates the need for custom operating system design. As a result, development time, cost and risk are reduced.

The iRMX 86 system can be EPROM-resident or loaded from a mass storage device into RAM, depending upon application requirements. Being able to place all of the software in EPROM offers two benefits. First, if the application is in a harsh environment, mass storage devices cannot be used because of the contamination danger. Second, if the application is small, the expense and overhead of mass storage devices is eliminated.

The iRMX 86 object-oriented Nucleus provides a foundation upon which a variety of application systems can be built. Object-oriented architecture provides a symmetric interface for application programs and operating system extensions that are machine independent. Objects provide facilities including multi-programming, multi-tasking, critical section management, extensive task-to-task communication and control.

Real-time priority-oriented scheduling is provided by the iRMX 86 Scheduler, which insures that the highest priority task ready to execute is given system control. The scheduler recognizes 255 software priority levels. Also, the system supports eight hardware priority levels, allowing the application system to be responsive to its external environment.

The iRMX 86 Operating System provides extensive error management and reporting mechanisms. Both excessive system loading and user programming errors can be reported, reducing system debug time. The flexibility of the Error Management subsystem allows errors to be serviced directly by the user task or sent to a specific error handler.

The iRMX 86 system provides interactive software debugging. The object-oriented Debugging subsystem has two capabilities that greatly simplify the process of debugging a multi-tasking system. First, the Debugger allows you to debug several tasks while the balance of the application system continues to run in real-time. Second, the Debugger lets the programmer interactively view and modify objects as well as RAM and 8086 registers.

The device-independent iRMX 86 I/O subsystem provides a standard interface for application programs to communicate with all I/O devices. File management systems provide powerful features, such as a hierarchical directory structure for quick, efficient file access. A standardized device driver interface allows users to easily create custom device drivers. A wide range of standard drivers are available.

The iRMX 86 Human Interface provides a powerful man-machine interface for interactive control of system resources and utilities. iRMX 86 system utilities include display file directories, copy files, rename files, and others. The Intel-supplied command line interpreter allows application-oriented commands to be created.

The iSBC 957A™ is a powerful interface and execution package. The iSBC 957A™ package allows ISIS-II files from an Intellec® Microcomputer Development System to be transferred to or from the iSBC 86/12A™ single board computer. The interface and execution package allows full speed execution of 8086 programs. It adds a "virtual terminal" capability, which permits the Intellec system console to access the iSBC 86/12A system monitor. Also, the package permits powerful console commands to be executed for software debugging. The iSBC 957A interface and execution package includes all necessary cables and software to interface the Intellec system to the 16-bit single board computer.

8-Bit Run-Time Systems Software

For several years, the iRMX 80 Real-Time Multi-Tasking Executive has set the reliability standard for real-time executives in the 8-bit world. It operates on iSBC single board computers and systems and provides a complete set of device drivers for mass storage, analog, an other peripheral controllers. The small, highly functional iRMX 80 system provides the means to concurrently monitor and control multiple external events occurring asynchronously in real-time. The executive contains all major real-time facilities including priority-based system resource allocation, inter-task communication and control, interrupt driver control for standard I/O devices, real-time clock control, interrupt handling, and many optional features. These facilities eliminate the need for users to design and implement application specific executives, greatly simplifying application design and reducing development time and risk.

FEATURES OF THE iRMX 80™ REAL-TIME MULTI-TASKING EXECUTIVE

Structured Environment — Provides a consistent structure from application to application, thus allowing experience gained on one system to be easily transferred to others. Often, entire programs may be used in multiple applications.

Simple Interface — Provides a straightforward program interface for user programs. This interface is consistent throughout the range of facilities offered, reducing the number of concepts which must be learned.

Library Modules — Constructed in a thoroughly modular manner with the full range of facilities being offered in multiple library modules, allowing easy selection of the exact facilities required.

Small Nucleus — A small, efficient foundation upon which application systems may be easily built. A wide range of multi-tasking, real-time facilities such as inter-task communication and control are included.

Priority Oriented Scheduler — Insures that highest priority task which is ready to execute is given system control, allowing the application system to be responsive to its external world.

Comprehensive I/O Support — Libraries support a wide range of I/O boards, simplifying addition of peripherals to an application system. For applications requiring custom boards, the iRMX 80 device handler philosophy allows easy addition of user written handlers.

User Configurable — Applications may be configured from wide range of facilities, selecting only those meeting specific requirements of the application system. Resultant system contains only modules necessary for its use, allowing iRMX 80 to fit a wide range of applications from small special purpose dedicated applications to large general purpose systems.

Broad Technological Support — Provides support for a range of processor technologies from iSBC 80/10A board (8080-based) to iSBC 80/30 board (8085-based). Applications are offered an easy upgrade path with iRMX 80 multi-tasking executive, allowing greater price/performance to be achieved without expensive software modification.

Extensive Debugging Aids — Provides two user oriented, interactive software debugging aids. Debuggers allow memory examination and modification, execution breakpoints, and automatic stack overflow monitoring. These powerful aids allow simplified task debugging and faster application system development.

iRMX 80 Generation — Process allows application programs written in PL/M-80, FORTRAN-80 or 8080/8085 Assembly Language to be merged with the specific iRMX 80 modules desired. System may then be debugged using Intel's sophisticated ICE In-Circuit Emulation products or the iRMX 80 debugger. The final application system is then available for either PROM or disk-based systems.

iSBC 801™ iRMX 80 FORTRAN RUN-TIME PACKAGE

The iSBC 801 FORTRAN Run-Time Package is a library of routines which provide a convenient and effective mechanism for interfacing to iRMX 80 multi-tasking executive services by application tasks written in FORTRAN-80.

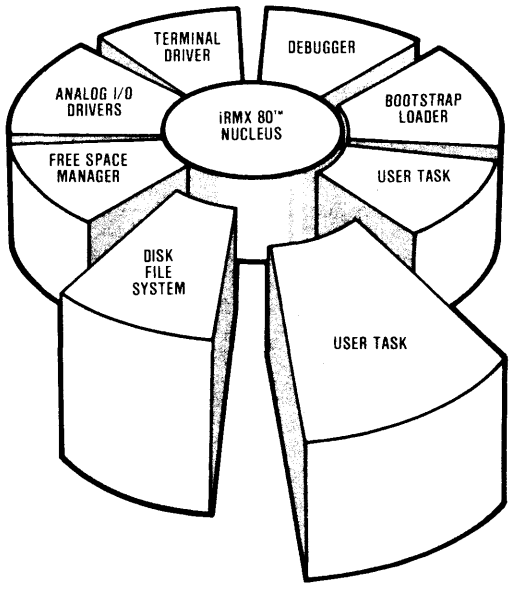
- Full FORTRAN formatted Disk and Terminal I/O capability under iRMX 80
- Supports Intel's floating point standard with either special FORTRAN drivers for the iSBC 310 high-speed math board or with floating point software equivalents
- All modules in the package are completely compatible and linkable with iRMX 80 modules and other application tasks coded in FORTRAN-80, PL/M-80, or 8080A/8085A Macro Assembler

8-Bit Run-Time Systems Software

iSBC 802™
BASIC-80 CONFIGURABLE iRMX 80 DISK-BASED INTERPRETER

The iSBC 802 BASIC-80 Configurable Disk-Based Interpreter provides users with the capability to create BASIC applications on iRMX 80 based iSBC systems.

- The package includes the Interpreter in two forms:
  - — A configurable module which can be combined with iRMX 80 modules, other OEM-written application modules in 8080A/8085A Assembler, PL/M-80 or FORTRAN-80 to create a tailored iSBC software solution
  - — As a predefined version which can be utilized to provide an "instant-on" iRMX BASIC-80 System for the OEM-suppied iSBC 80/30 board, iSBC 204 disk controller, Intellec Microcomputer Development System Diskette Chassis, and a terminal
- Meets and exceeds ANS 78 BASIC Standard
- Full formatted I/O operation to the terminal and to the Disk File System, allowing either random or sequential file access
- Single and double precision floating point arithmetic consistent with the Intel floating point format
- Extensive string handling functions for easy and efficient data manipulation
- Easy-to-use interactive program development and execution allows the OEM to quickly utilize BASIC application program from the large number of public programs available
- Both the Interpreter and the BASIC source can reside in PROM



iRMX 80™ REAL-TIME MULTI-TASKING EXECUTIVE

Memory Requirements (Bytes)								
Module	Nucleus	Full Terminal Handler	Minimal Terminal Handler	Free Space Manager	File System	Disk I/O	Analog I/O	Bootstrap Loader and Initializer
PROM*	2K	3K	600	1K	5.5K	700	800	600
RAM	250	950	120	250	1.6K	100	50	900

Minimum Development System Requirements		
Operating System	Memory Size (Bytes)	Minimum Diskette Drives
ISIS-II	48K RAM	2

iSBC 801™ FORTRAN RUN-TIME PACKAGE

Memory Requirements[10] (Bytes)				
Module	Formatted I/O Modules	Software Floating Point Functions	Floating Point Functions using iSBC 310	Trigonometric and Related Functions
PROM*	20.8K	5.8K	4.3K	4.4K
RAM	1.5K	80	60	80

Minimum Development System Requirements		
Operating System	Memory Size (Bytes)	Minimum Diskette Drives
ISIS-II	64K RAM	2

iSBC 802™ BASIC-80 INTERPRETER

Memory Requirements (bytes)[10] (BASIC-80 only)		
Module	Configurable BASIC-80 Module	Pre-Defined BASIC Software Modules
PROM*	22.5K	4K (bootstrap)
RAM	2.5K	48K

Minimum Development System Requirements	
Configurable BASIC-80 Modules	Predefined BASIC-80 Modules
ISIS-II 64K RAM Dual Diskette Drives	None Pre-defined to run with iSBC 80/30 iSBC 032 iSBC 204

*Indicates amount of code which can be configured in PROM
[10]Does not include iRMX 80 memory requirements.

INTEL DEVELOPMENT SOLUTIONS

Intel Development System products provide you with a range of solutions to minimize development time for your Intel Single Board Computer application. Hardware and software development, debug and integration is simplified.

INTELLEC® MICROCOMPUTER DEVELOPMENT SYSTEMS

- Powerful, easy-to-use microcomputer development systems give you complete centers for development of single board computer-based products
- Intellec Series II Development Systems (225/235/245) are upgradeable to new development system technology as it is introduced

INTELLEC® ISIS-II OPERATING SYSTEM

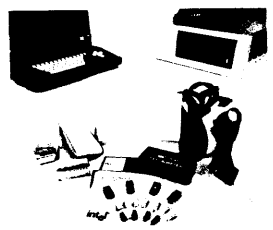
- ISIS-II provides the ability to edit, assemble, compile, relocate, execute and debug programs
- Supports generation of iRMX 80-based software
- Supports all of Intel's assemblers, compilers and debuggers

INTELLEC® UPP UNIVERSAL PROM PROGRAMMER

- Intellec system peripheral for PROM programming and verification
- Personality cards available for programming all the Intel EPROMs

iSBC 957™ INTELLEC-iSBC 86/12A™ INTERFACE AND EXECUTION PACKAGE

- Allows Intellec ISIS-II files to be transferred to or from the iSBC 86/12A.
- Allows full speed execution of MCS-86 programs.
- "Virtual Terminal" capability permits the Intellec console to access the iSBC 86/12A monitor.
- Powerful console commands for software debug.
- Includes all necessary cables and software for interface on Intellec system to iSBC 86/12A board.



DEBUG SUPPORT

- Real-time emulation of the user's iSBC 80, iSBC 86/12A or System 80 products via ICE In-Circuit Emulation
- Allows iSBC 80 or iSBC 86 system to share Intellec system RAM, ROM, EPROM and I/O facilities
- Full symbolic debugging capabilities
- Real-time trace provides address, data status information on previous machine cycles emulated
- Examine and alter CPU registers, memory and flag values
- Breakpoint and single-step capability

iSBC/ICE	ICE-80™	ICE-85™	ICE-86™
iSBC 80/10B	x		
iSBC 80/20-4	x		
iSBC 80/24		x	
iSBC 80/30		x	
iSBC 86/12A			x

SOFTWARE DEVELOPMENT MODULE

The Model 810 Software Development Module provides low-cost, ROM-based development software for 8080/8085-based iSBC Single Board Computers on a standard iSBC 464 memory expansion board.

- Development software fits easily into user-defined iSBC target system
- The system is ideal for editing, assembling and debugging of small program modules

You may configure your target system using the following iSBC chassis and single board computers:

Single Board Computer	iSBC 660™ Chassis	iSBC 655™ Chassis	iCS 80™ Chassis
iSBC 80/10B	x	x	x
iSBC 80/20-4	x	x	x
iSBC 80/24	x	x	x
iSBC 80/30	x	x	x

Also required:

iSBC 016 RAM Expansion Board (not required with iSBC 80/30 Single Board Computer)
iSBC 955 RS232C Serial I/O Cable Set (or equivalent)
iSBC 530 Teletype Adapter (not required for iSBC 80/10B)

Optional Hardware Supported

iSBC 80/10A Single Board Computer — interchangeable with iSBC 80/10B Single Board Computer
iSBC 032 RAM Expansion Board (32K RAM)

Development Software and Hardware

HIGH-LEVEL LANGUAGES AND ASSEMBLERS

When you're doing iSBC Single Board Computer development, Intel provides languages that execute on the Intellec Microcomputer Development System to develop your application (FORTRAN-80, ASM 80, FSP, BASIC-80, PASCAL-80, PL/M-80) and run-time packages that operate with iRMX (Realtime Multi-Tasking Executive) run-time packages (iRMX 80, FORTRAN, BASIC, and PASCAL).

FORTRAN-80 ANS FORTRAN 77 INTELLEC RESIDENT COMPILER

- Meets and exceeds ANS FORTRAN 1977 Subset Language Specification.
- Supports Intel floating point standards.
- Compiler fully supports symbolic debugging with ICE-80 or ICE-85 In-Circuit Emulators.
- Formatted data may be written to or read from iSBC System disk and terminal when supported through iRMX 80 FORTRAN run-time package.
- Produces relocatable and linkable object code compatible with resident PL/M-80 and 8080A/8085A Macro Assembler.

BASIC-80

- Easy to learn and implement. Developed as an instruction language, BASIC is a natural, easy language for a beginning programmer.
- Engineering or math problems can be programmed quickly for immediate results.
- Quick tool for prototyping program logic for testing and debugging.
- Meets ANS 1978 BASIC Language Specification plus has additional features: PEEK; POKE; IN, THEN, ELSE; and calls to user-supplied routines.

FSP (8080/8085 Fundamental Support Package)

- FSP has nine mathematical and technical libraries of commonly used subroutines which can easily be included in your program.
- FSP can eliminate the need for you to code the following functions:
  - —String handling
  - —Binary and decimal integer arithmetic
  - —Floating point arithmetic
  - —Number conversion and numeric I/O
  - —Floating point transcendental functions
  - —Statistical routines
  - —PID process control routines

8086 SOFTWARE DEVELOPMENT PACKAGE

The 8086 software development package provides complete support for developing application software for the iSBC 86/12A Single Board Computer. The package includes PL/M-86, Intel's high-level systems programming language. PL/M-86 provides the ease of use and increased productivity of a high level language while allowing the user to fully ultilize the features of the iSBC 86/12A. The 8086 Macro Assembler is included in the package for implementing those applications which require maximum code efficiency and execution speeds. The complete 8086 software package works together to ease the software development task and increase programmer productivity. The package includes the following tools:

- PL/M-86-high level programming language
- ASM86-"high level" Macro Assembler
- LINK86 and LOC86-linkage and relocation utilities
- CONV86-converts 8080 assembly language source to 8086 assembly language source.
- LIB86-manages 8086 object modules.
- OH86-object to hexadecimal converter.
- UPM-universal PROM programming software.

PASCAL-80

- PASCAL's well-defined structure encourages good programming techniques for improved software reliability.
- PASCAL-80 is a superset of standard PASCAL (as defined in Jensen and Wirth's **PASCAL USER'S MANUAL AND REPORT).**
- Segmented procedure definition allows you to break large programs into small modules.
- Microprocessor extensions to PASCAL include calling and linking of separate PASCAL or assemble language subroutines.

8080/8085 MACRO ASSEMBLER

- Fully symbolic assembler for powerful software development and debugging.
- Advanced MACRO facility provides more structure in coding, and easier program maintenance.
- Assembler generates relocatable object modules for linking with other ASM80, PL/M-80 or FORTRAN modules.
- ASM80 supports conditional assembly of source code.

PL/M-80 HIGH LEVEL PROGRAMMING LANGUAGE COMPILER

- "Block-oriented" language supports modular structured programming — programs are easier to debug and maintain.
- Compiler generates relocatable object code for linkage to other 8080 or 8085 language program modules.
- Compiler options include OPTIMIZATION, and the use of REENTRANT procedures.
- Offered since 1973, PL/M-80 was the first high-level language designed specifically for the microprocessor environment.

Development System	iSBC Processor Supported	Disk Storage	RAM	ISIS Operating System Memory Required	Text Editor	Packaged Software	Host Processor
Model 225	iSBC 80/10B iSBC 80/20-4 iSBC 80/24 iSBC 80/30 iSBC 86/12A	Integral Single Density Disk Drive — 250Kbytes MDS-720* - 1Mbytes MDS-730** - 1Mbytes MDS-740*** - 7.5Mbytes	64K	64K	Credit	8080/8085 Symbolic Assembler	8085A-2
Model 235	iSBC 80/10B iSBC 80/20-4 iSBC 80/24 iSBC 80/30 iSBC 86/12A	Integral Single Density Disk Drive — 250Kbytes MDS-730** - 1Mbytes MDS-740*** - 7.5Mbytes	64K	64K	Credit	8080/8085 Symbolic Assembler	8085A-2
Model 245	iSBC 80/10B iSBC 80/20-4 iSBC 80/24 iSBC 80/30 iSBC 86/12A	Integral Single Density Disk Drive — 250Kbytes MDS-720* - 1Mbytes MDS-730** - 1Mbytes	64K	64K	Credit	8080/8085 Symbolic Assembler	8085A-2
Model 810							

*Double density diskette operating system
**Double density diskette add-on drives
***Hard disk add-on subsystem

Contact your local Intel® Sales Engineer for additional configuration and pricing information.

intel delivers

INTEL CORPORATION, 5200 N.E. Elam Young Parkway, Hillsboro, Oregon 97123, (503) 640-6611, TWX: 910-460-8815