# 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<br>8283 | Octal Latch<br>Octal Latch (Inverting) | Bipolar | 20 | Demultiplexes and increases drive of address bus. |
| 8284 | Clock Generator and Driver | Bipolar | 18 | Provides time base. |
| 8286<br>8287 | Octal Bus Transceiver<br>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.

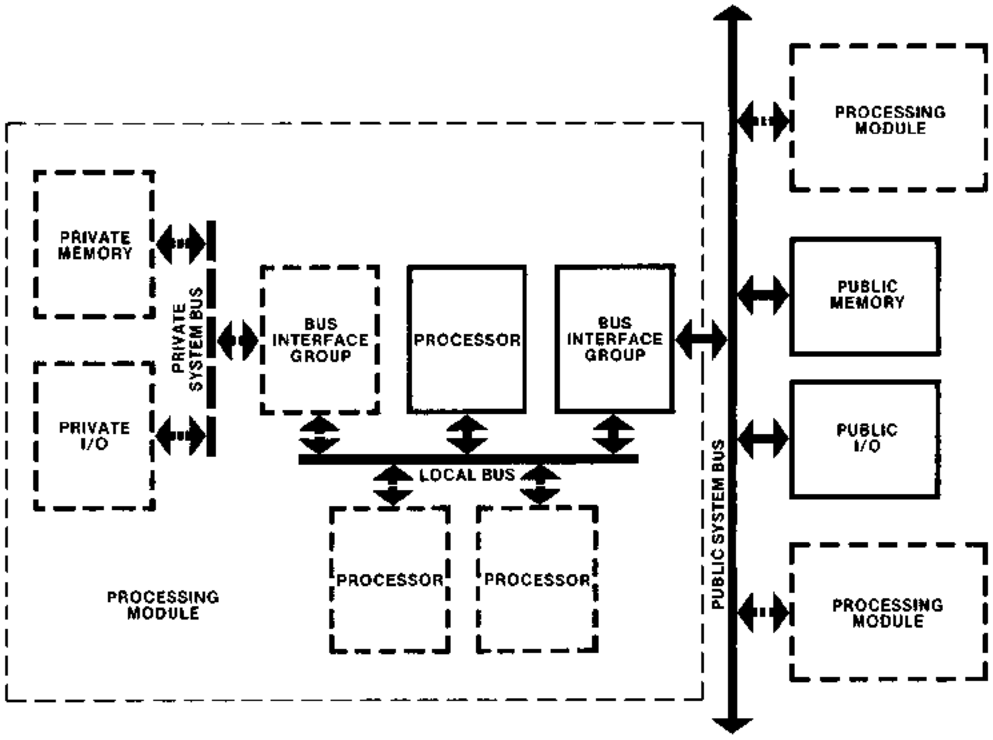


Figure 1-1. Generalized 8086 Family Bus Structure

## 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.

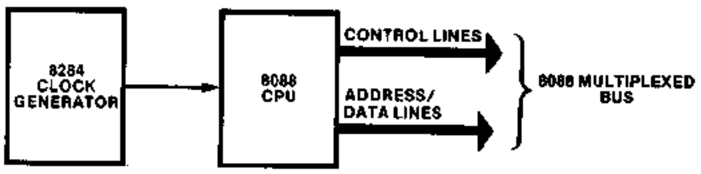


Figure 1-2. 8088 Multiplexed Bus

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.

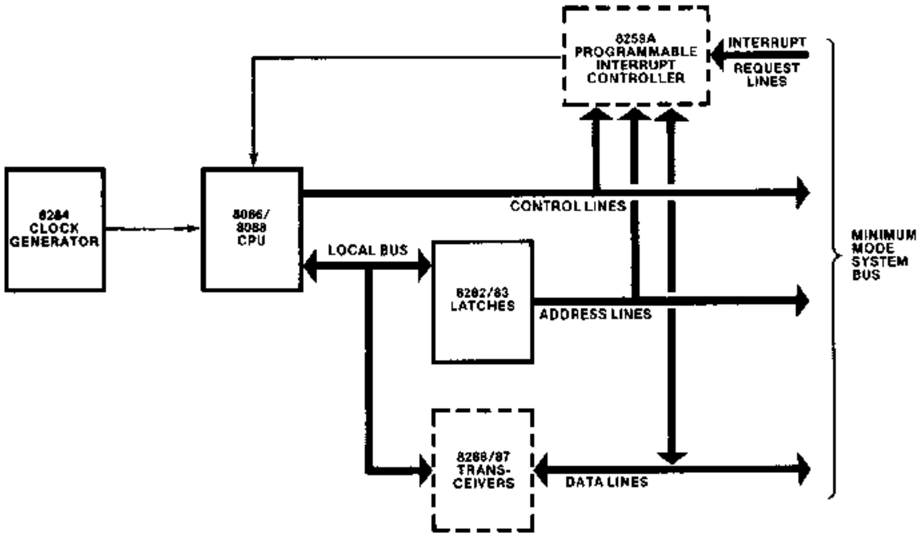


Figure 1-3. Minimum Mode System Bus

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

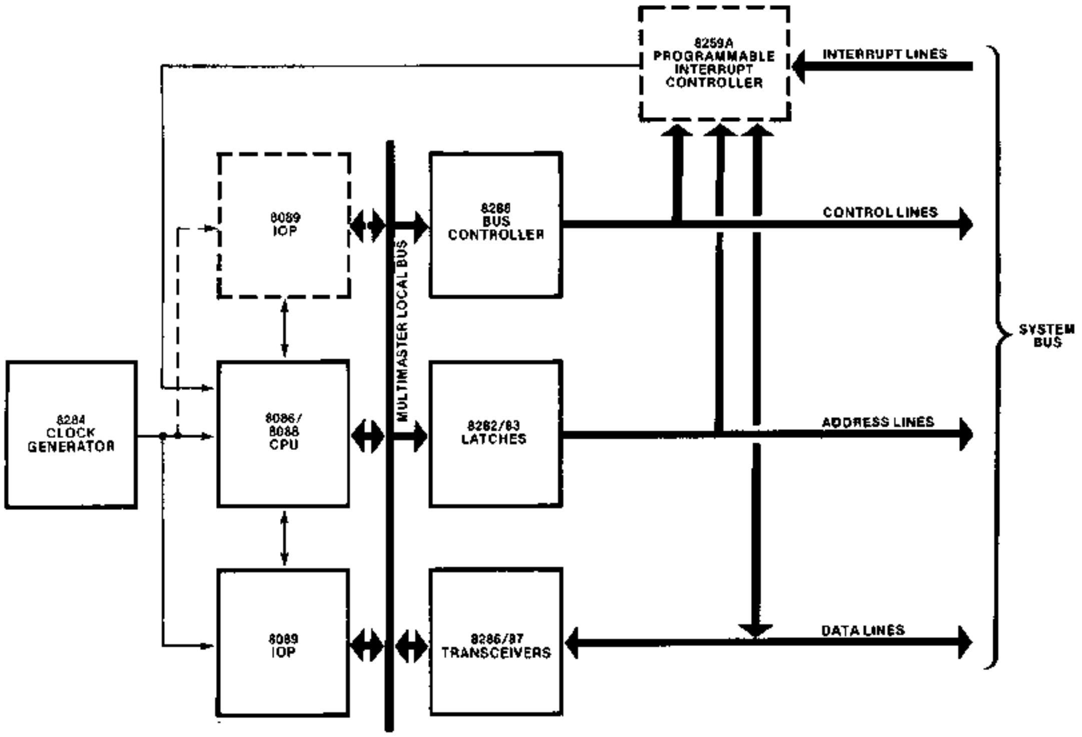


Figure 1-4. Multimaster Local Bus

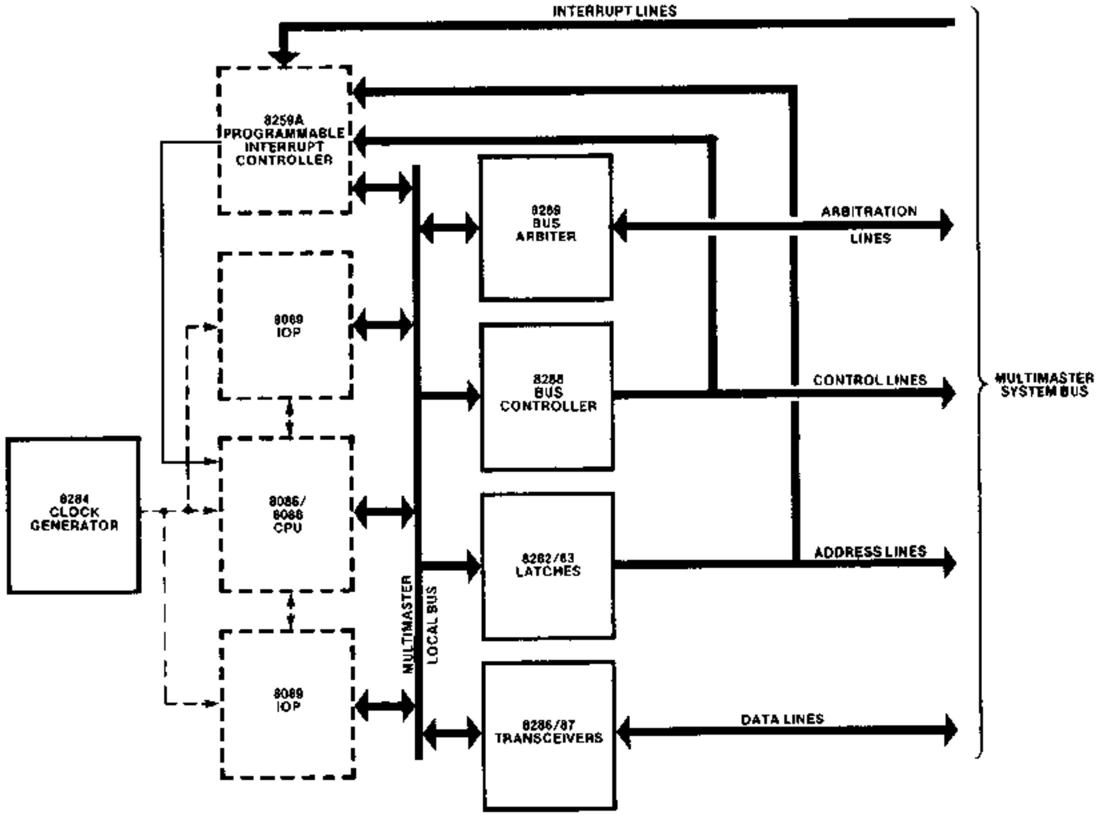


**Figure 1-5. Basic Multimaster Processing Module**

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.

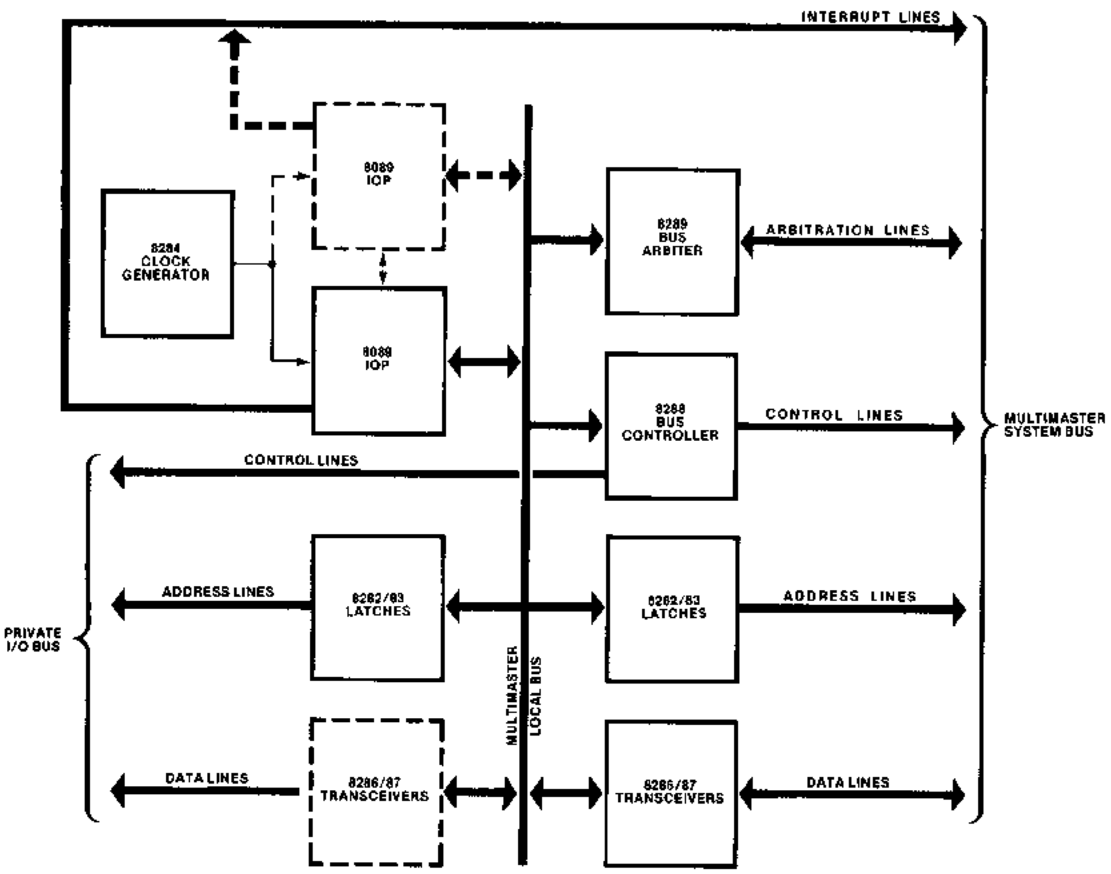


**Figure 1-6. Private I/O Bus**

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.

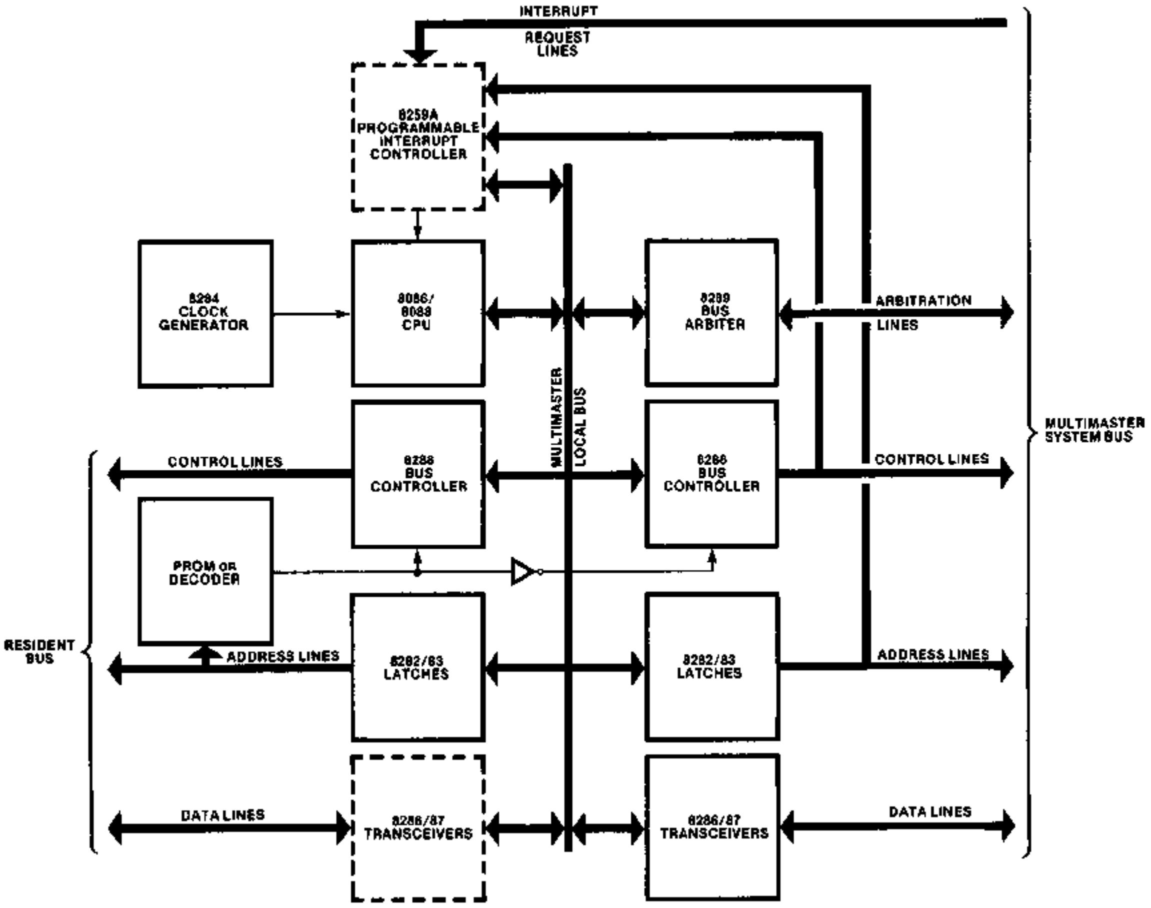


Figure 1-7. Resident Bus

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.

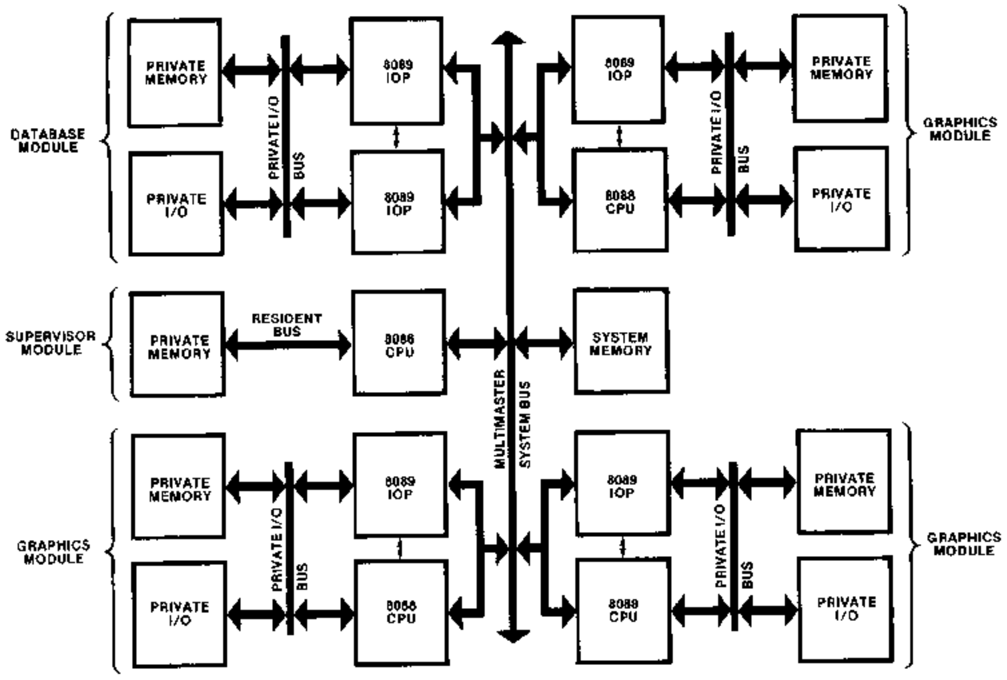


Figure 1-8. Multimaster Design Example

## 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.

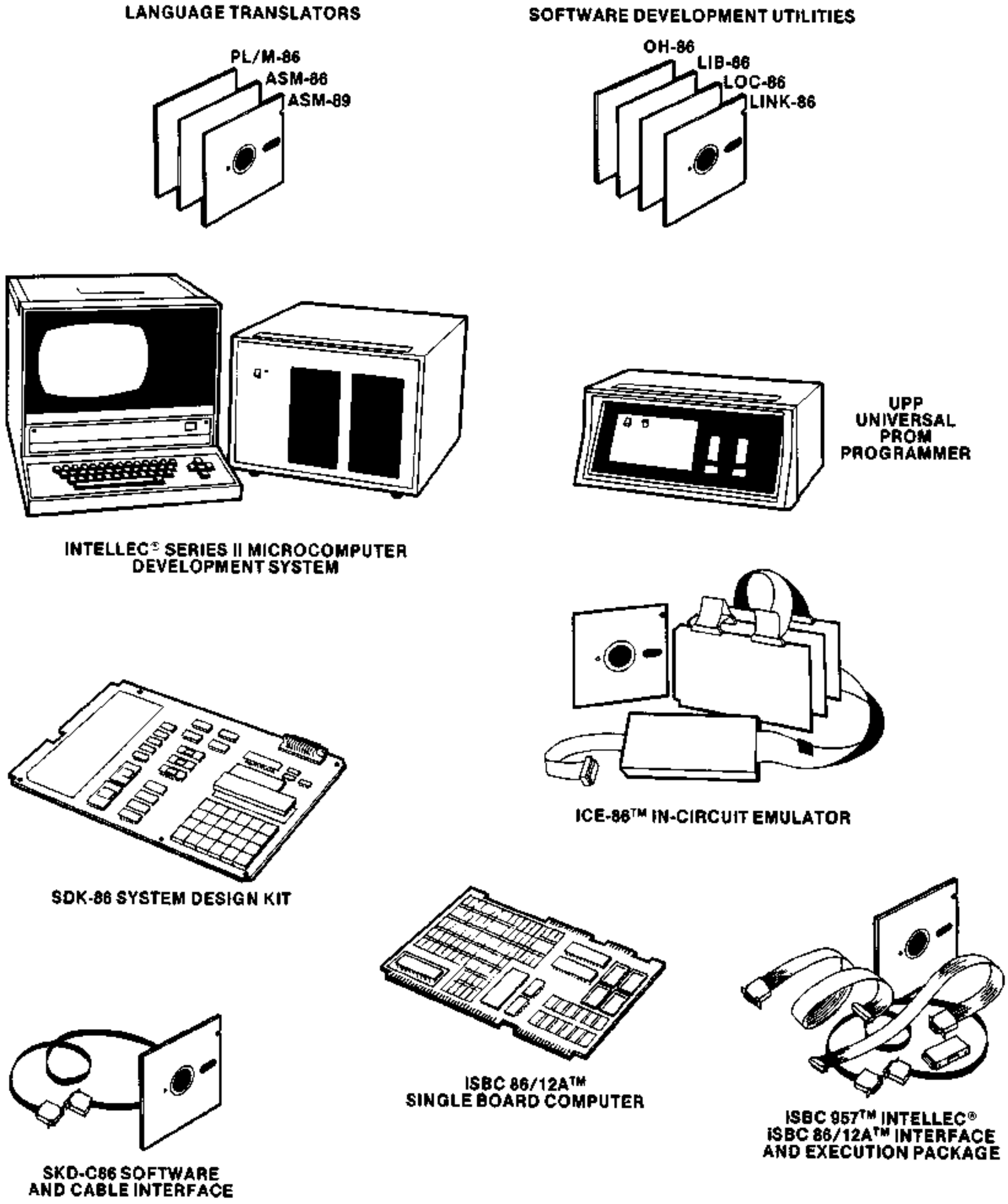
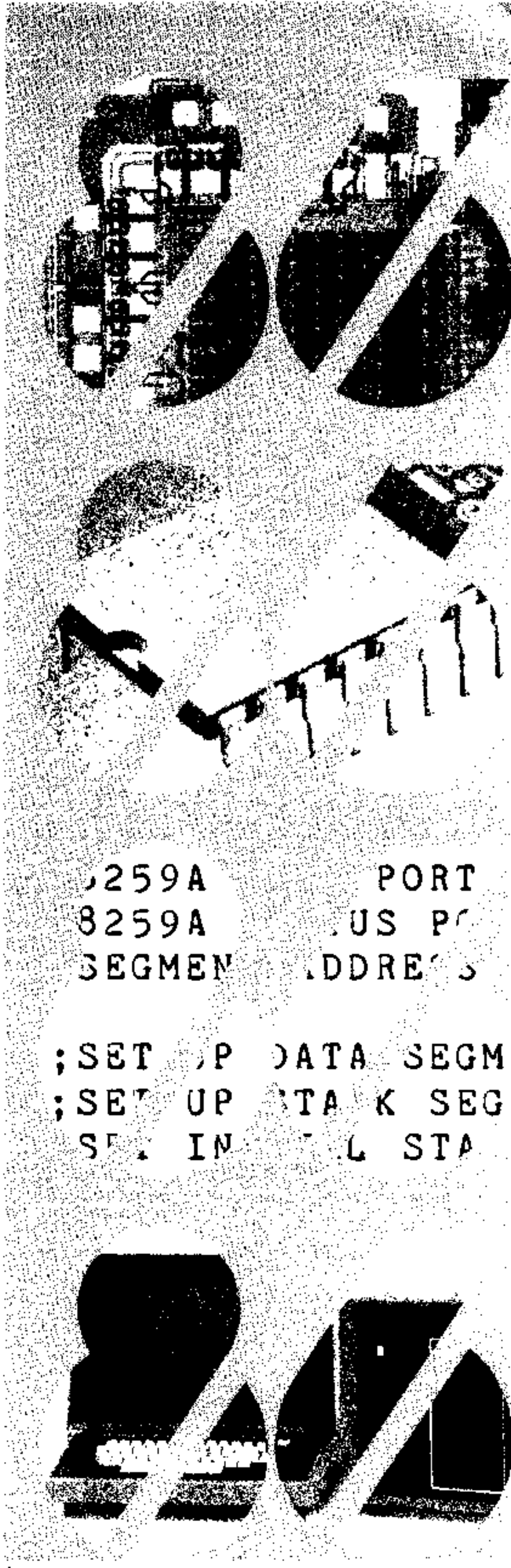

LANGUAGE TRANSLATORS
PL/M-86
ASM-86
ASM-89
SOFTWARE DEVELOPMENT UTILITIES
OH-86
LIB-86
LOC-86
LINK-86
UPP UNIVERSAL PROM PROGRAMMER
INTELLEC® SERIES II MICROCOMPUTER DEVELOPMENT SYSTEM
ICE-86™ IN-CIRCUIT EMULATOR
SDK-86 SYSTEM DESIGN KIT
ISBC 86/12A™ SINGLE BOARD COMPUTER
ISBC 957™ INTELLEC® iSBC 86/12A™ INTERFACE AND EXECUTION PACKAGE
SKD-C86 SOFTWARE AND CABLE INTERFACE

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).

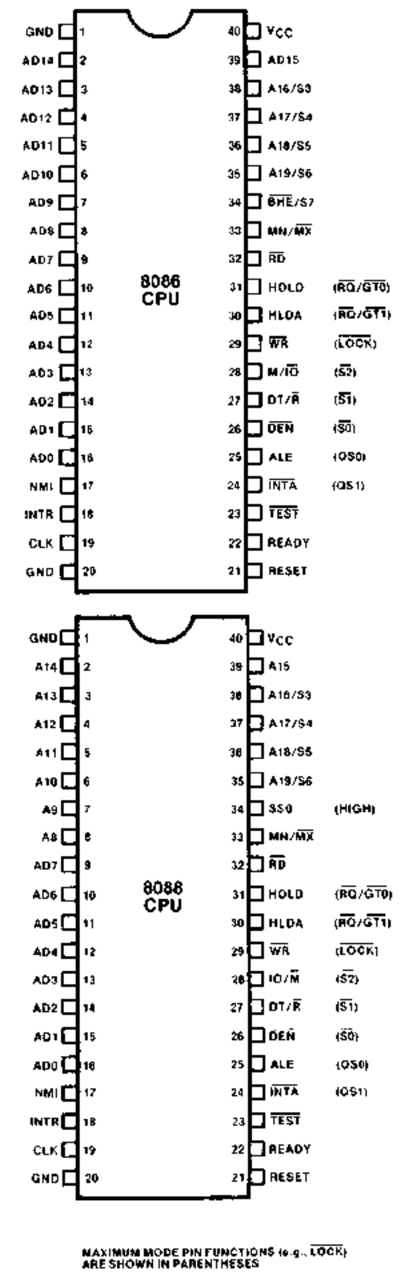

8086 CPU
GND 1, AD14 2, AD13 3, AD12 4, AD11 5, AD10 6, AD9 7, AD8 8, AD7 9, AD6 10, AD5 11, AD4 12, AD3 13, AD2 14, AD1 15, AD0 16, NMI 17, INTR 18, CLK 19, GND 20
40 VCC, 39 AD15, 38 A16/S3, 37 A17/S4, 36 A18/S5, 35 A19/S6, 34 BHE/S7, 33 MN/MX, 32 RD, 31 HOLD (RQ/GT0), 30 HLDA (RQ/GT1), 29 WR (LOCK), 28 M/IO (S2), 27 DT/R (S1), 26 DEN (S0), 25 ALE (QS0), 24 INTA (QS1), 23 TEST, 22 READY, 21 RESET
8088 CPU
GND 1, A14 2, A13 3, A12 4, A11 5, A10 6, A9 7, A8 8, AD7 9, AD6 10, AD5 11, AD4 12, AD3 13, AD2 14, AD1 15, AD0 16, NMI 17, INTR 18, CLK 19, GND 20
40 VCC, 39 A15, 38 A16/S3, 37 A17/S4, 36 A18/S5, 35 A19/S6, 34 SS0 (HIGH), 33 MN/MX, 32 RD, 31 HOLD (RQ/GT0), 30 HLDA (RQ/GT1), 29 WR (LOCK), 28 IO/M (S2), 27 DT/R (S1), 26 DEN (S0), 25 ALE (QS0), 24 INTA (QS1), 23 TEST, 22 READY, 21 RESET
MAXIMUM MODE PIN FUNCTIONS (e.g., LOCK) ARE SHOWN IN PARENTHESES

**Figure 2-1. 8086 and 8088 Central Processing Units**

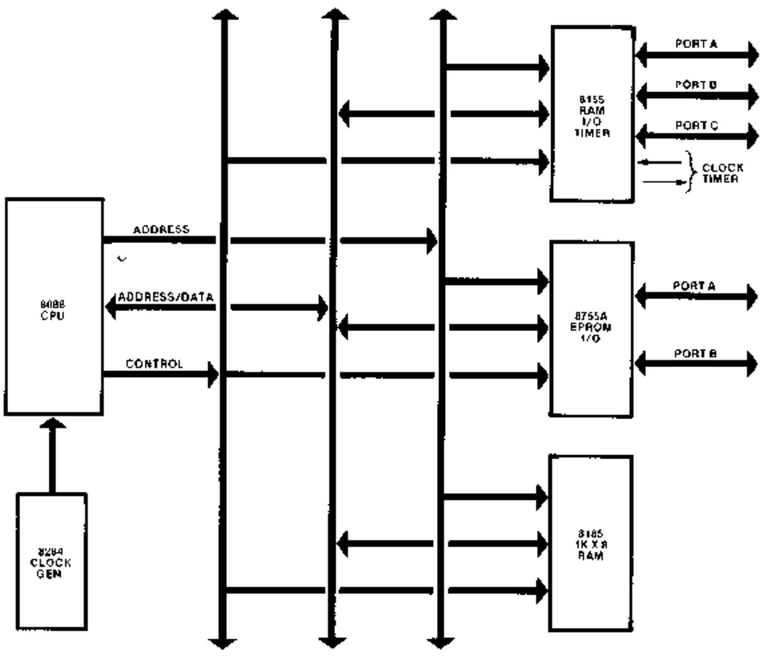


Figure 2-2. Small 8088-Based System

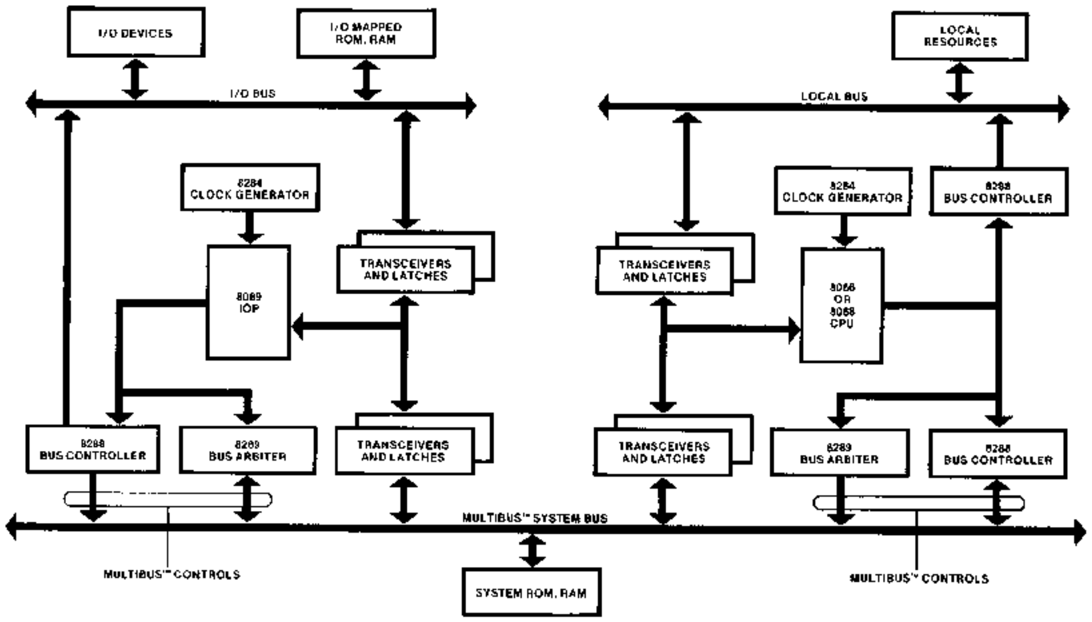


Figure 2-3. 8086/8088/8089 Multiprocessing System

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$^{TM}$ 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).

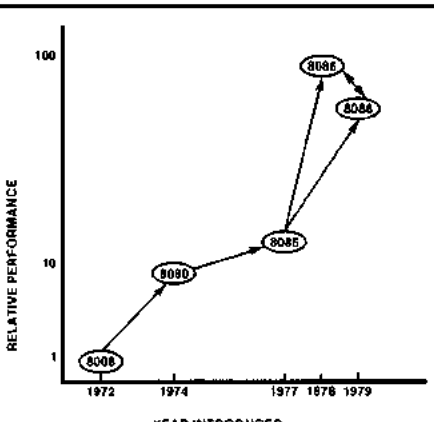


**Figure 2-4. Relative Performance of the 8086 and 8088**

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.

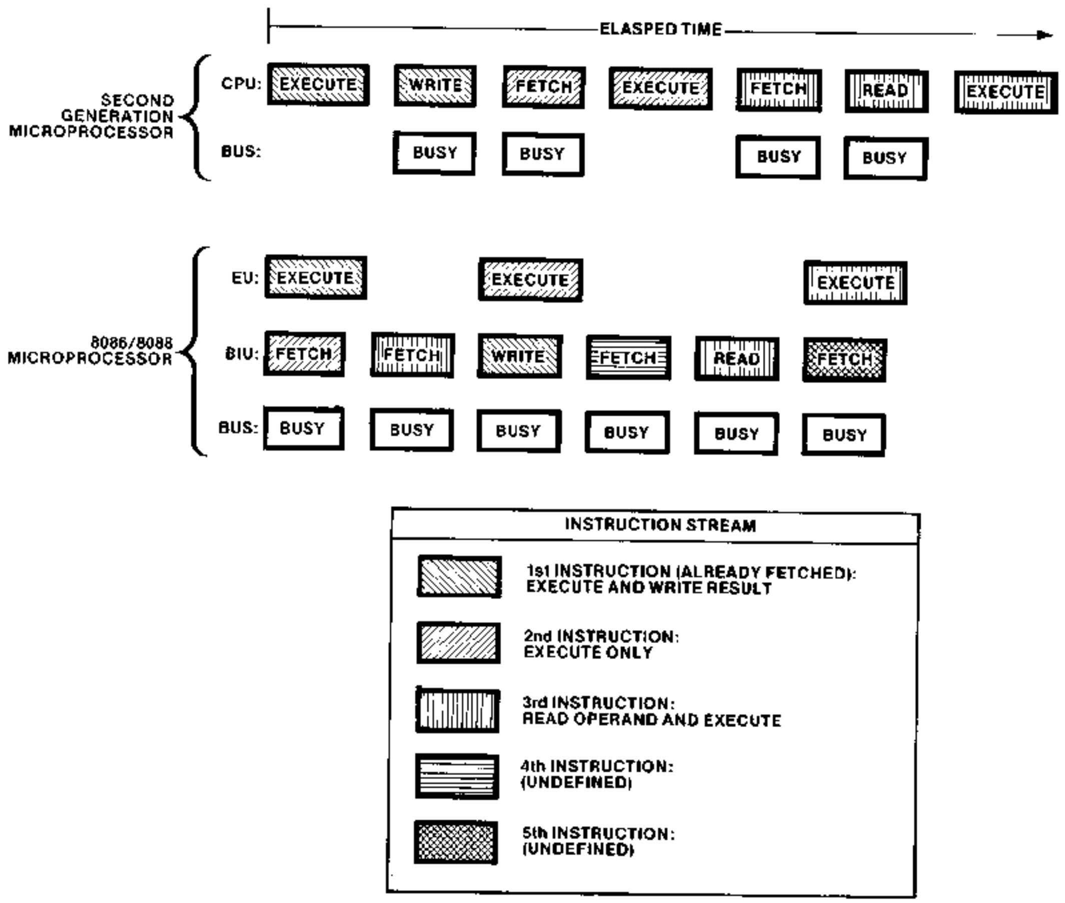

ELASPED TIME
SECOND GENERATION MICROPROCESSOR
CPU:
EXECUTE
WRITE
FETCH
EXECUTE
FETCH
READ
EXECUTE
BUS:
BUSY
BUSY
BUSY
BUSY
8086/8088 MICROPROCESSOR
EU:
EXECUTE
EXECUTE
EXECUTE
BIU:
FETCH
FETCH
WRITE
FETCH
READ
FETCH
BUS:
BUSY
BUSY
BUSY
BUSY
BUSY
BUSY
INSTRUCTION STREAM
1st INSTRUCTION (ALREADY FETCHED): EXECUTE AND WRITE RESULT
2nd INSTRUCTION: EXECUTE ONLY
3rd INSTRUCTION: READ OPERAND AND EXECUTE
4th INSTRUCTION: (UNDEFINED)
5th INSTRUCTION: (UNDEFINED)

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

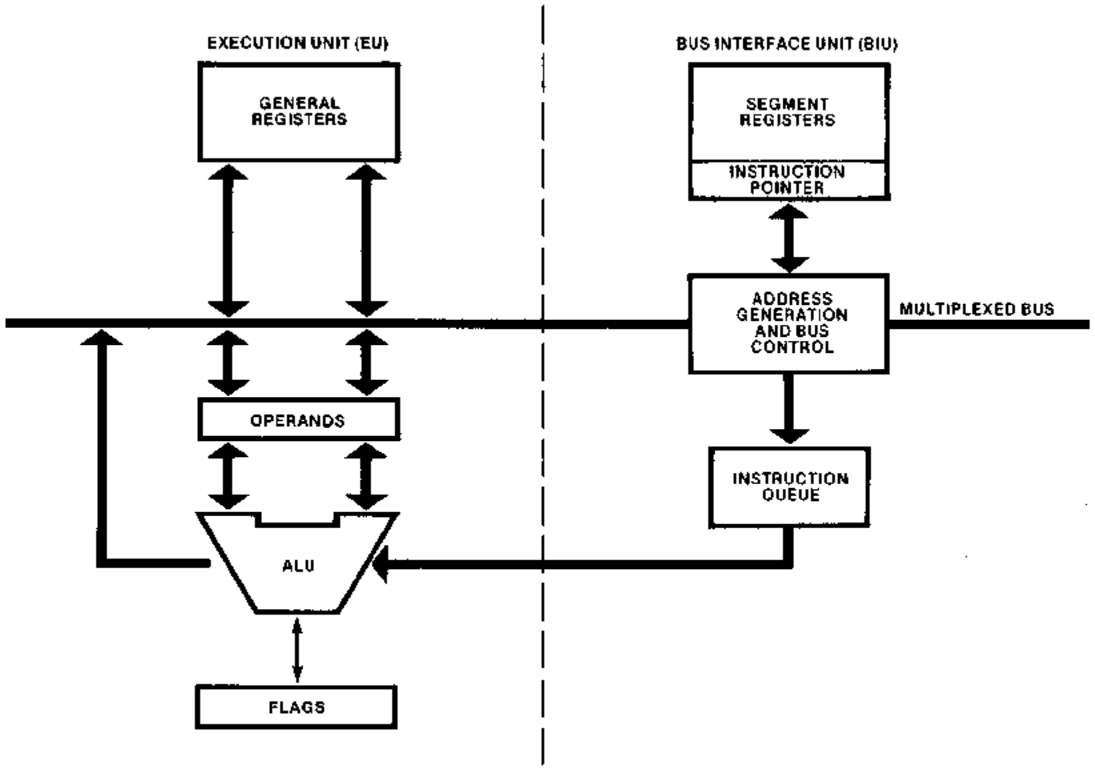


Figure 2-6. Execution and Bus Interface Units (EU and BIU)

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, some instructions use certain registers implicitly (see table 2-1) thus allowing compact yet powerful encoding.

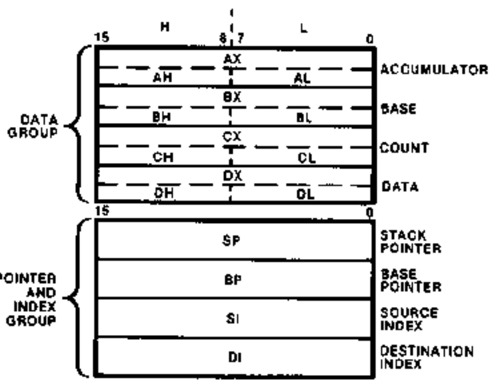


Figure 2-7. General Registers

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.

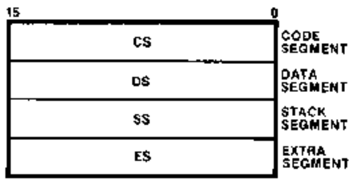


Figure 2-8. Segment Registers

## 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

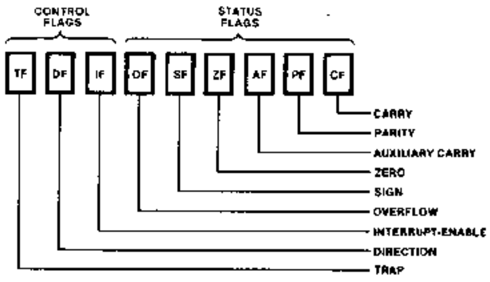


Figure 2-9. Flags

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/$\overline{\text{MX}}$) that defines the function of eight CPU pins in the 8086 and nine pins in the 8088. Connecting MN/$\overline{\text{MX}}$ 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/$\overline{\text{MX}}$ 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,

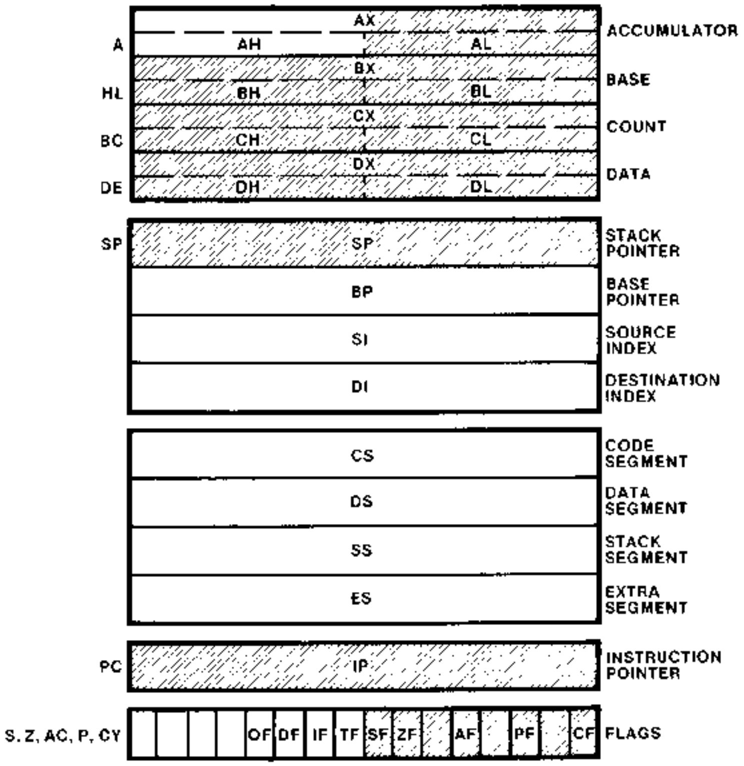


Figure 2-10. 8080/8085 Register Subset (Shaded)

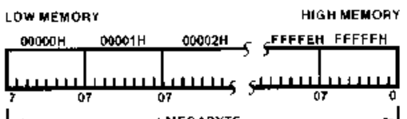


Figure 2-11. Storage Organization

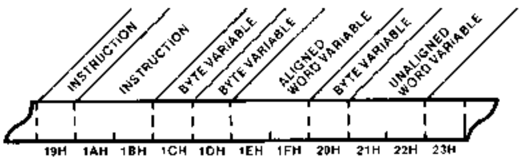


Figure 2-12. Instruction and Variable Storage

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).

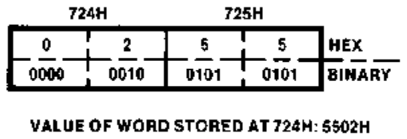


Figure 2-13. Storage of Word Variables

## 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.

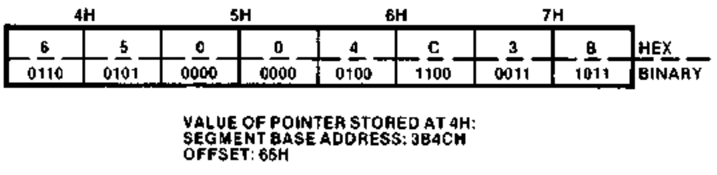


Figure 2-14. Storage of Pointer Variables

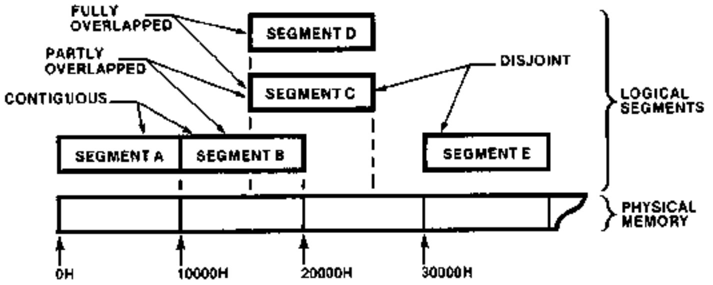


Figure 2-15. Segment Locations in Physical Memory

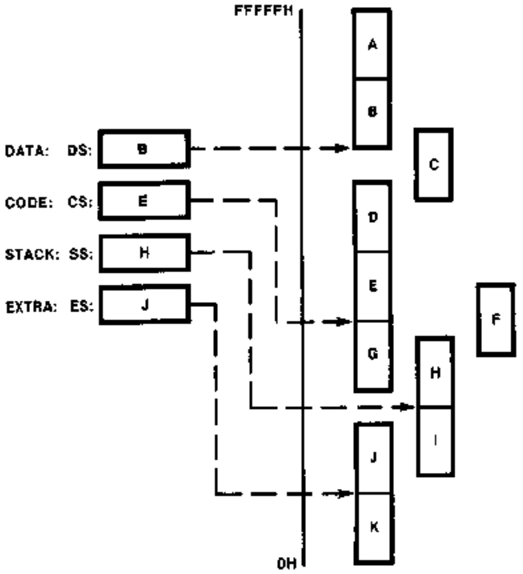


Figure 2-16. Currently Addressable Segments

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

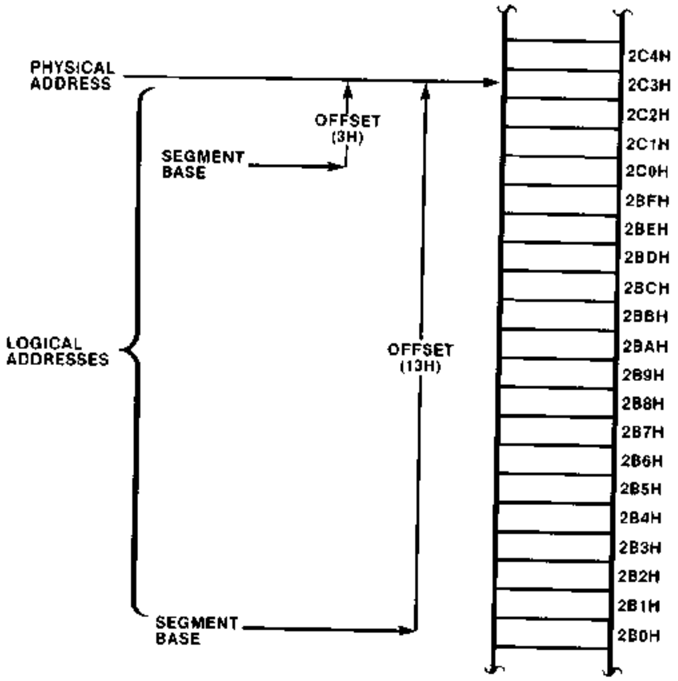


Figure 2-17. Logical and Physical Addresses

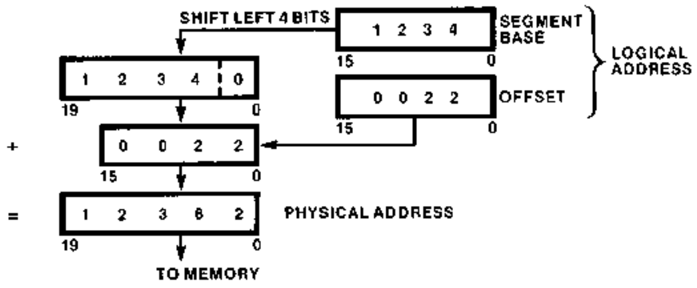


Figure 2-18. Physical Address Generation

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

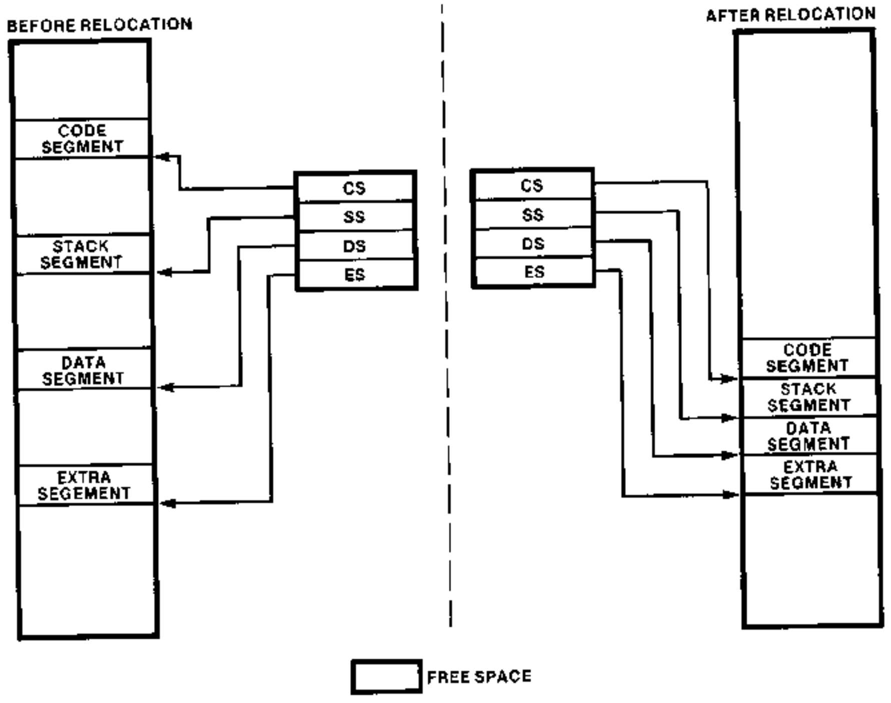


Figure 2-19. Dynamic Code Relocation

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

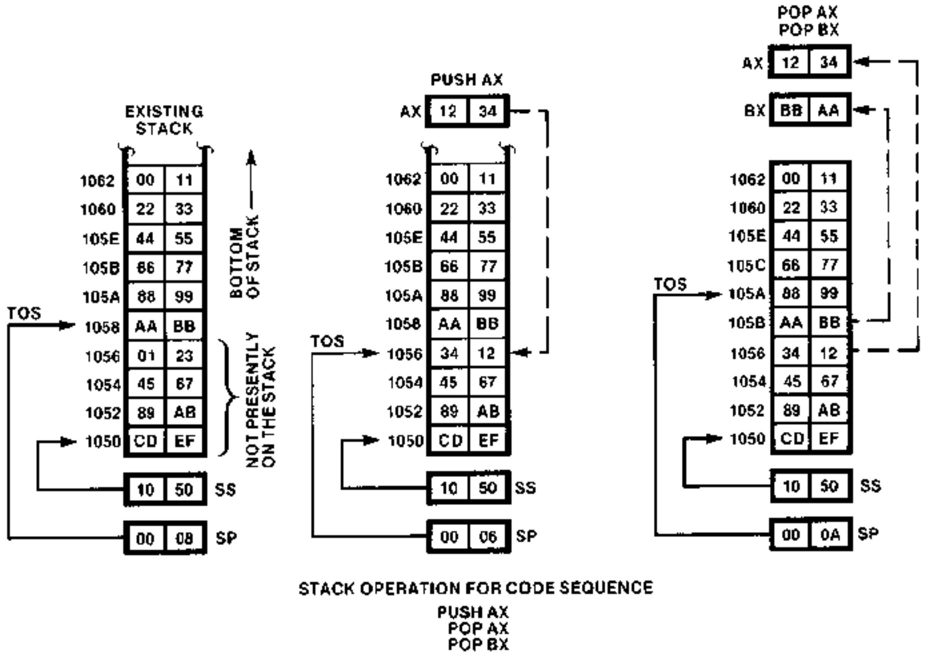


Figure 2-20. Stack Operation

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

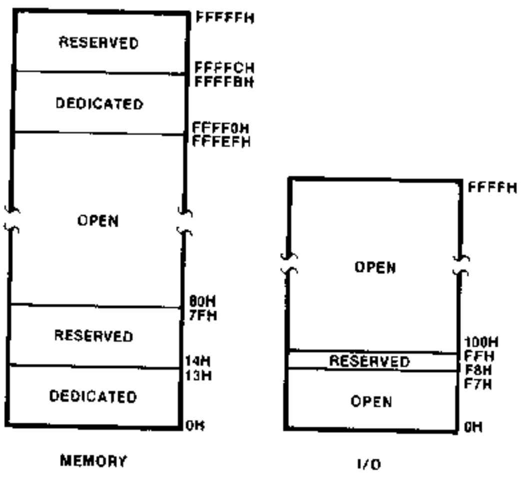


Figure 2-21. Reserved and Dedicated Memory and I/O Locations

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

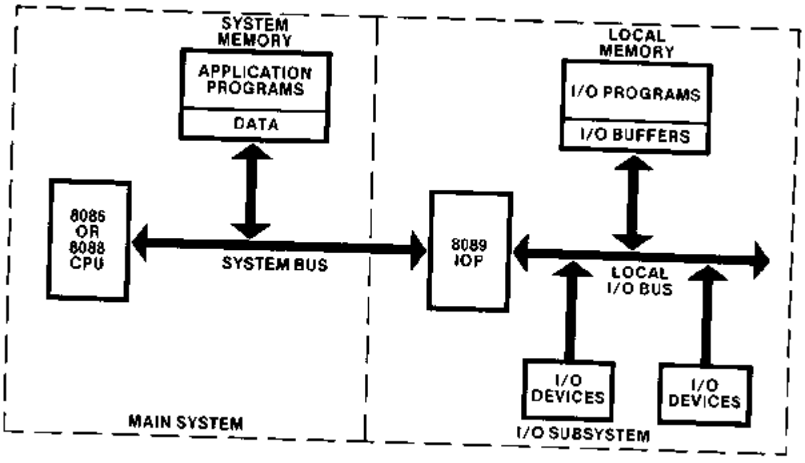


Figure 2-22. Multiprocessing System

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

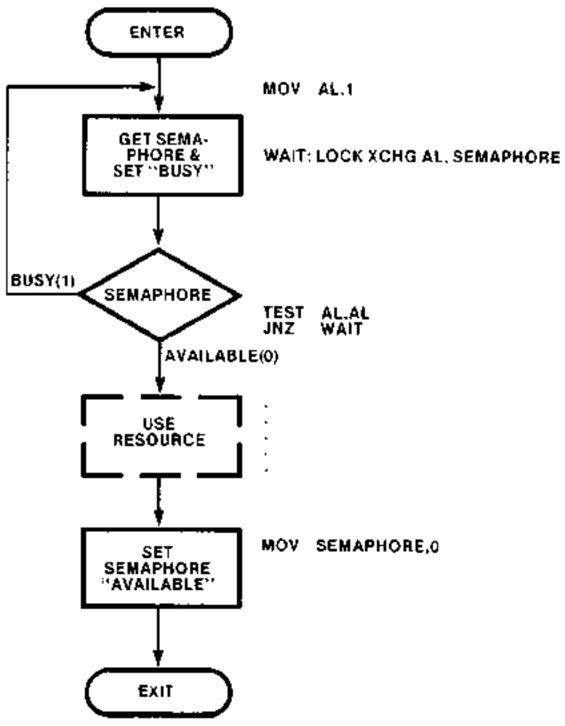


Figure 2-25. Using XCHG and LOCK

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

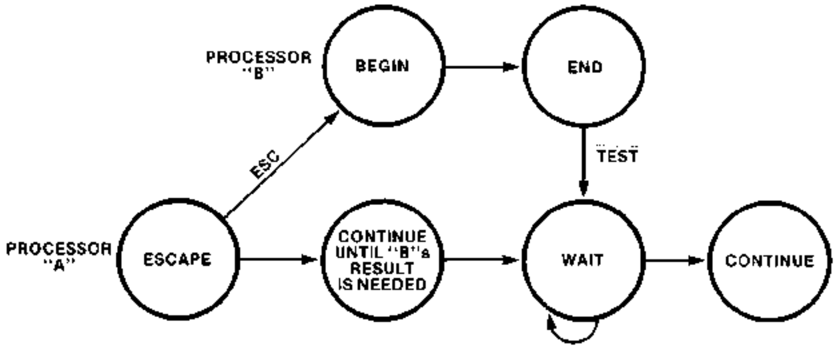


Figure 2-26. Using ESC with WAIT and TEST

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-

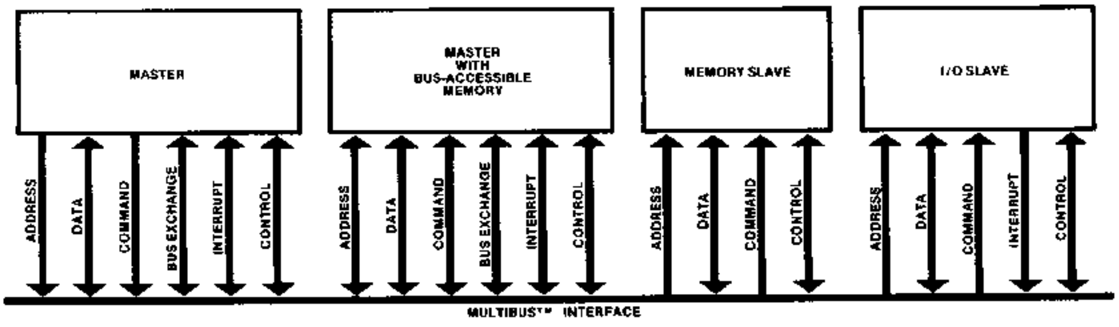


Figure 2-27. Multibus™-Based System

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

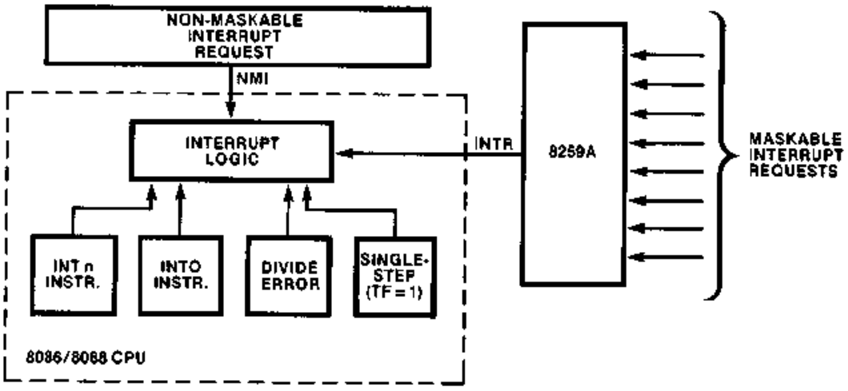


Figure 2-28. Interrupt Sources

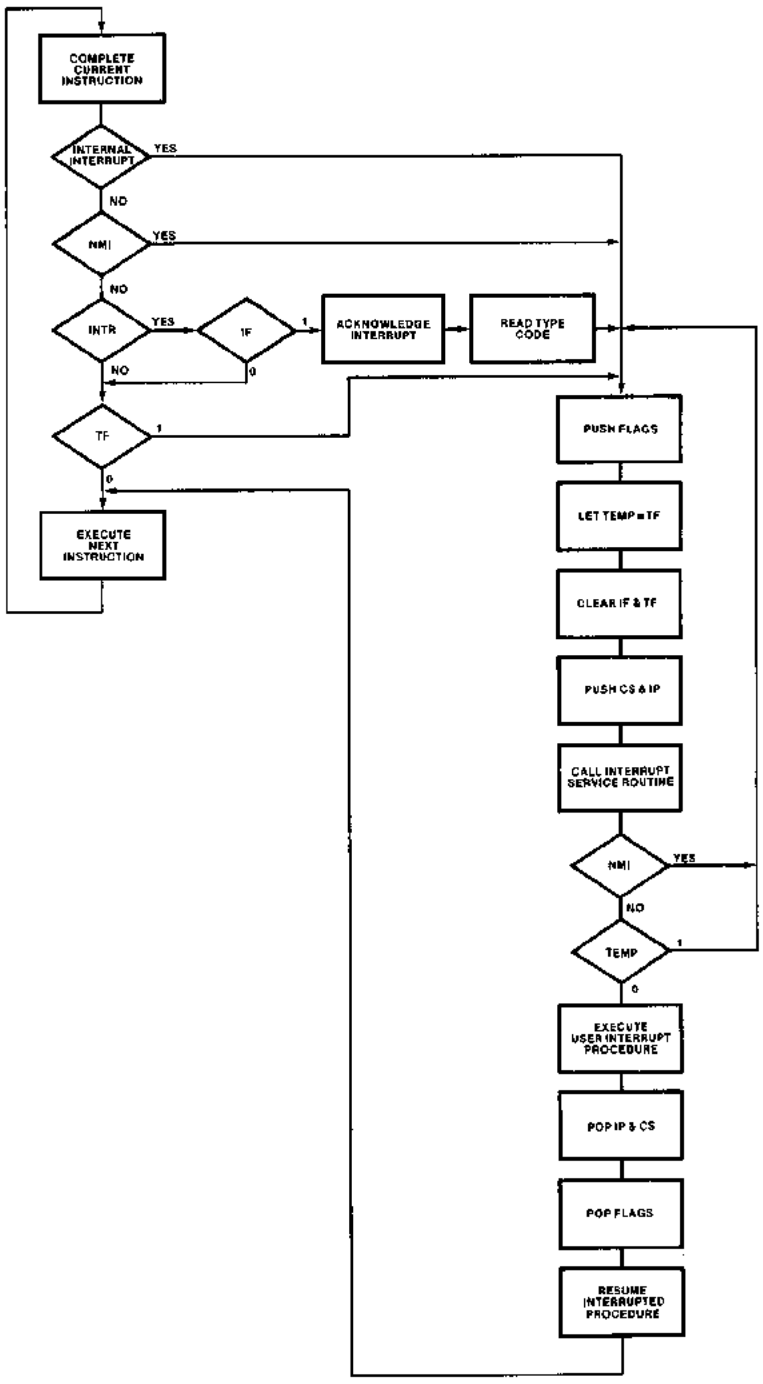


Figure 2-29. Interrupt Processing Sequence

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.

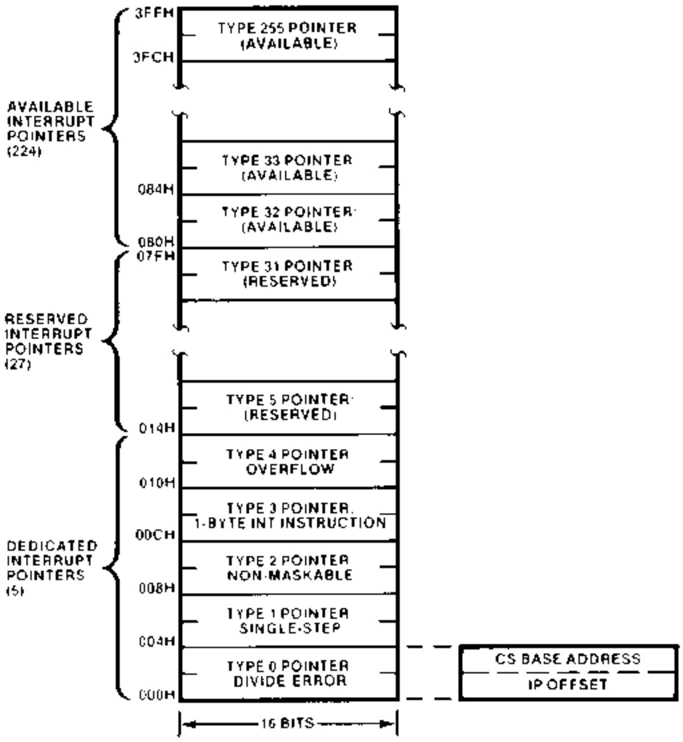
If the trap flag (TF) is set, the CPU automatically generates a type 1 interrupt following *every* instruction. This is called single-step execution and is a powerful debugging tool that is discussed in more detail shortly.

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



AVAILABLE INTERRUPT POINTERS (224)
RESERVED INTERRUPT POINTERS (27)
DEDICATED INTERRUPT POINTERS (5)
3FFH
3FCH
084H
080H
07FH
014H
010H
00CH
008H
004H
000H
TYPE 255 POINTER (AVAILABLE)
TYPE 33 POINTER (AVAILABLE)
TYPE 32 POINTER (AVAILABLE)
TYPE 31 POINTER (RESERVED)
TYPE 5 POINTER (RESERVED)
TYPE 4 POINTER OVERFLOW
TYPE 3 POINTER 1-BYTE INT INSTRUCTION
TYPE 2 POINTER NON-MASKABLE
TYPE 1 POINTER SINGLE-STEP
TYPE 0 POINTER DIVIDE ERROR
CS BASE ADDRESS
IP OFFSET
16 BITS

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.

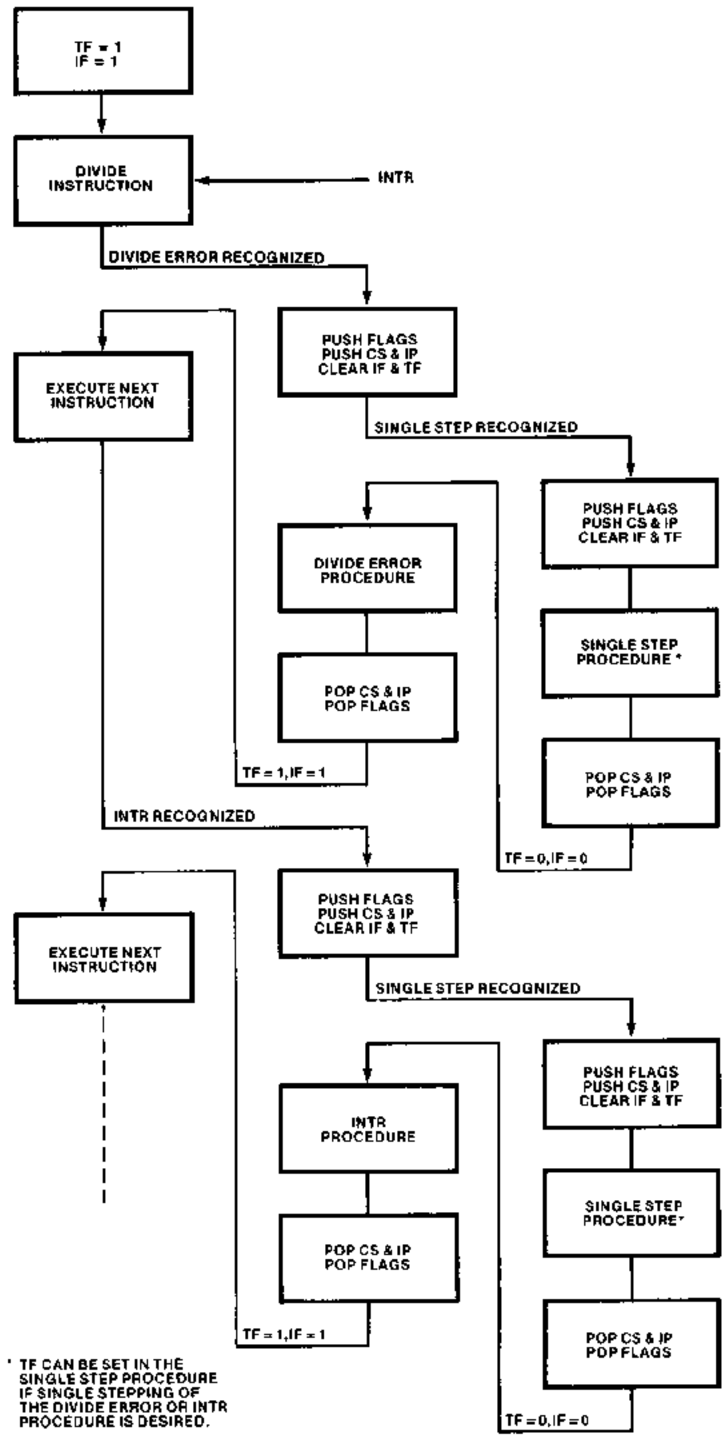


Figure 2-31. Processing Simultaneous Interrupts

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.

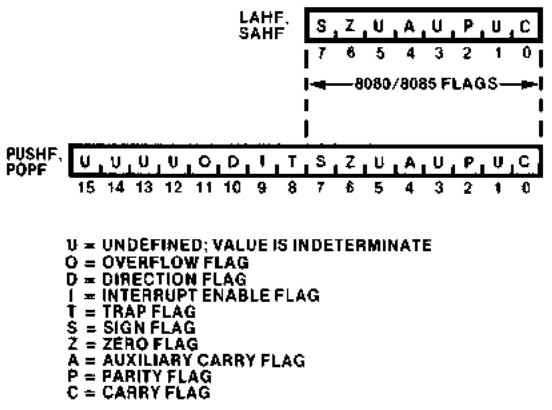


Figure 2-32. Flag Storage Formats

## 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, FFFFFH 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 subsequent DIV to produce the correct result. The quotient is returned in AL, and the remainder is returned in AH; both high-order half-bytes are zeroed. 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.

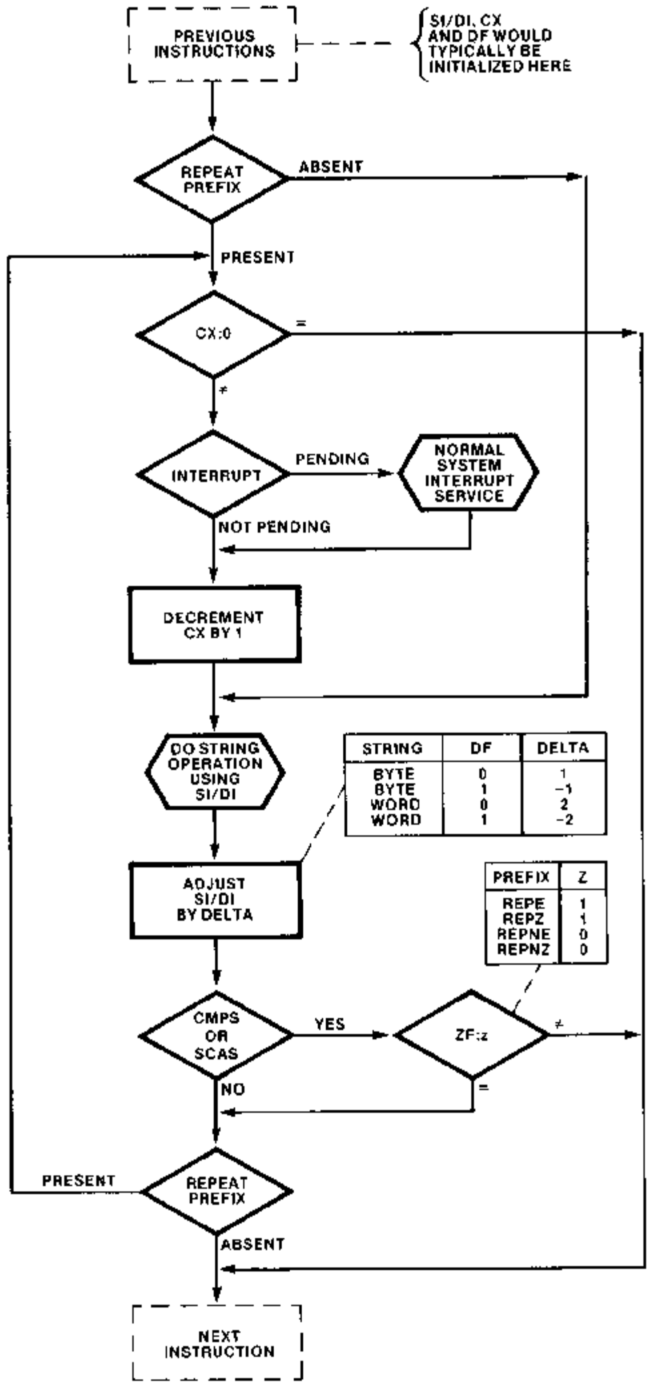
The string instructions operate quite similarly in many respects; the common characteristics are covered here and in table 2-13 and figure 2-33 rather than in the descriptions of the individual instructions. A string instruction may have a source operand, a destination operand, or both. The hardware 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<br>Destination for LODS<br>Source for STOS |
| DF | 0 = auto-increment SI, DI<br>1 = auto-decrement SI, DI |
| ZF | Scan/compare terminator |



| STRING | DF | DELTA |
|---|---|---|
| BYTE | 0 | 1 |
| BYTE | 1 | −1 |
| WORD | 0 | 2 |
| WORD | 1 | −2 |

| PREFIX | Z |
|---|---|
| REPE | 1 |
| REPZ | 1 |
| REPNE | 0 |
| REPNZ | 0 |

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 if 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.

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 |

### 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.

### 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<br>BX + SI + DISP | 11 |
| | BP + SI + DISP<br>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.)

| | | | O | D | I | T | S | Z | A | P | C |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **ADC** | **ADC** destination,source<br>Add with carry | **Flags** | 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 |

| | | | O | D | I | T | S | Z | A | P | C |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **ADD** | **ADD** destination,source<br>Addition | **Flags** | 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 |

| | | | O | D | I | T | S | Z | A | P | C |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **AND** | **AND** destination,source<br>Logical and | **Flags** | 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 |

| | | | O | D | I | T | S | Z | A | P | C |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **CALL** | **CALL** target<br>Call a procedure | **Flags** | | | | | | | | | |

| 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] |

| | | | O | D | I | T | S | Z | A | P | C |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **CBW** | **CBW** (no operands)<br>Convert byte to word | **Flags** | | | | | | | | | |

| 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) | 52 | 5 | 1 | INT 3 |
| immed8 (type ≠ 3) | 51 | 5 | 2 | 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 | 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<br>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<br>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<br>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<br>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<br>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<br>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<br>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<br>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<br>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<br>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<br>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<br>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<br>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<br>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<br>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<br>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<br>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)<br>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<br>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<br>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<br>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)<br>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<br>Load string | | | Flags O D I T S Z A P C |
|---|---|---|---|---|
| **Operands** | **Clocks** | **Transfers*** | **Bytes** | **Coding Example** |
| source-string<br>(repeat) source-string | 12<br>9+13/rep | 1<br>1/rep | 1<br>1 | LODS CUSTOMER_NAME<br>REP LODS NAME |

| **LOOP** | **LOOP** short-label<br>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<br>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<br>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)<br>Interrupt if NMI = 1 | | | Flags O S I T S Z A P C<br>0 0 (I, T) |
|---|---|---|---|---|
| **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<br>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<br>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)<br>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<br>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)<br>+EA | 1 | 2-4 | MUL MONTH [SI] |
| mem16 | (124-139)<br>+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 | 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<br>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<br>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<br>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<br>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<br>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<br>Subtraction | | | Flags O D I T S Z A P C<br>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<br>Test or non-destructive logical and | | | Flags O D I T S Z A P C<br>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)<br>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<br>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<br>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<br>Logical exclusive or | | | **Flags** O D I T S Z A P C<br>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.

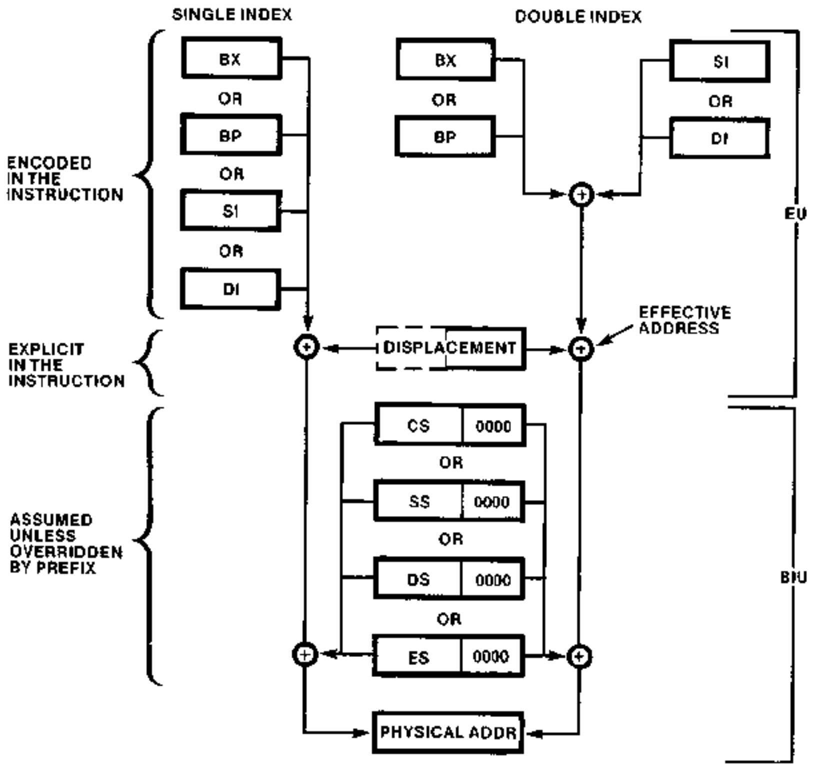


Figure 2-34. Memory Address Computation

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.

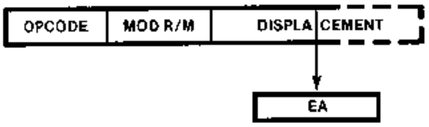


Figure 2-35. Direct Addressing

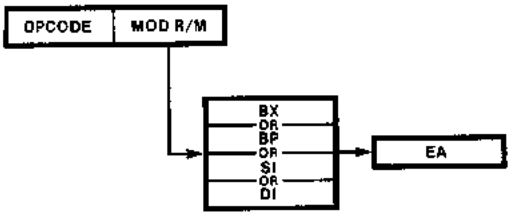


Figure 2-36. Register Indirect Addressing

## 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).

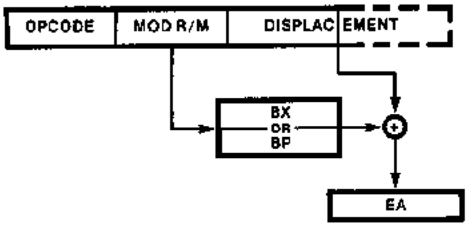


Figure 2-37. Based Addressing

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.

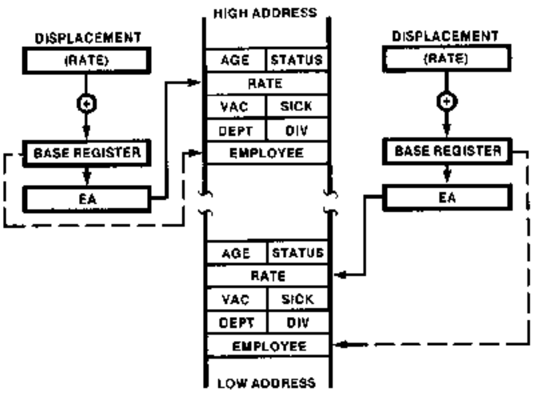


Figure 2-38. Accessing a Structure With Based Addressing

## 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

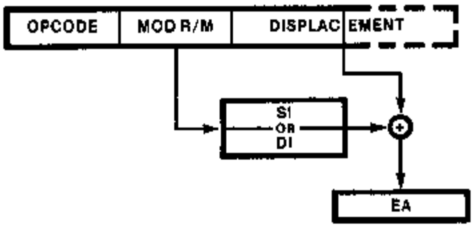


Figure 2-39. Indexed Addressing

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.

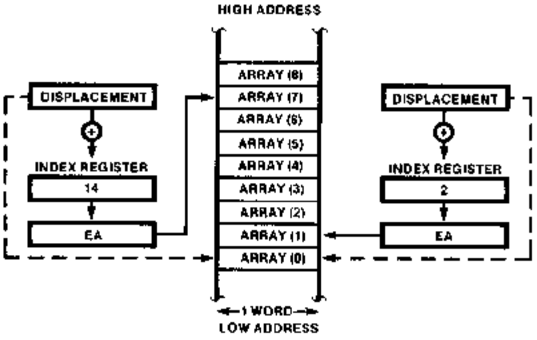


Figure 2-40. Accessing an Array With Indexed Addressing

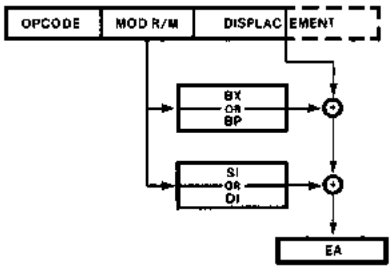


Figure 2-41. Based Indexed Addressing

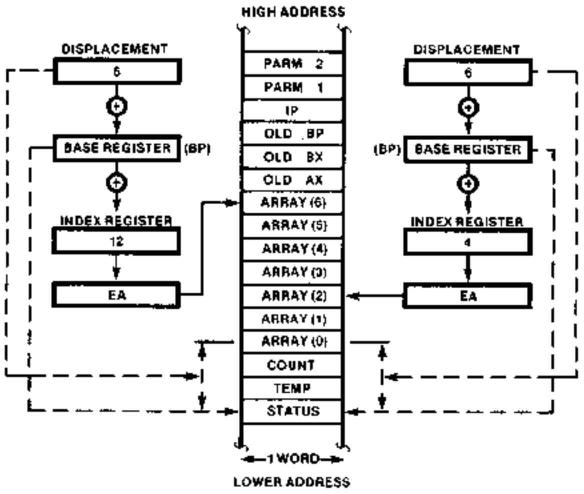


Figure 2-42. Accessing a Stack Array With Based Indexed Addressing

### 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.

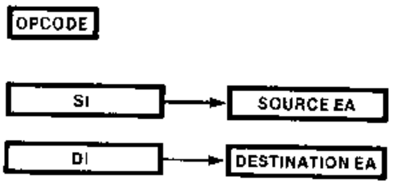


Figure 2-43. String Operand Addressing

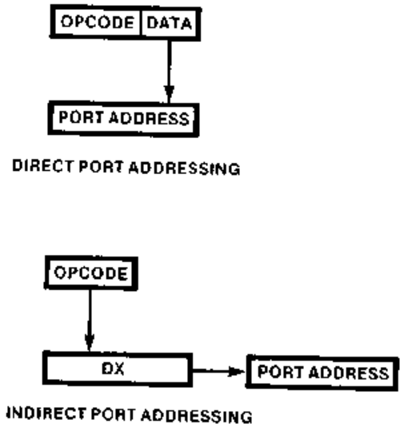


Figure 2-44. I/O Port Addressing

## 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.

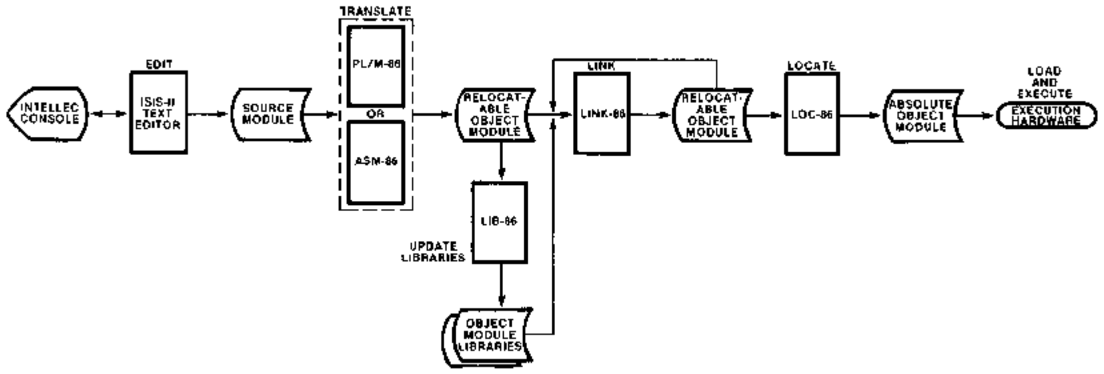


Figure 2-45. Software Development Process

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 | $1 \times 10^{-38}$ 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<br>"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

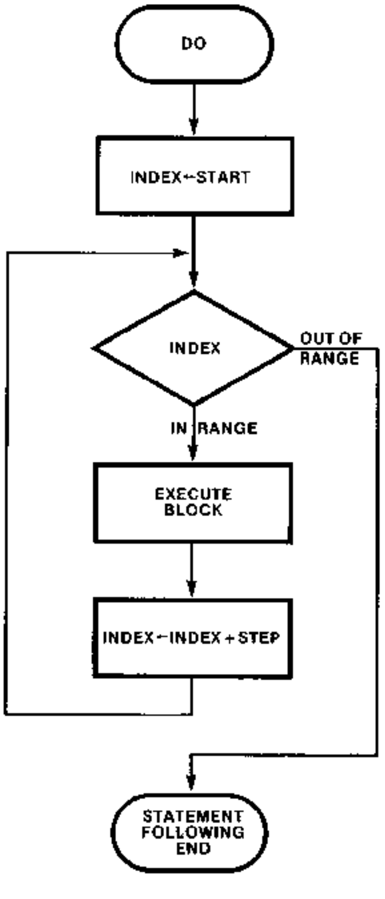


Figure 2-53. PL/M-86 Iterative DO Flowchart

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);

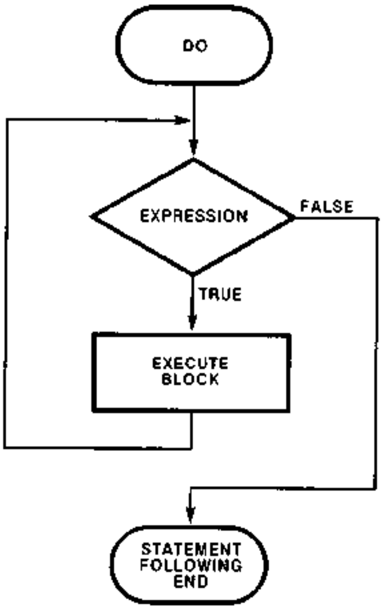


Figure 2-54. PL/M-86 DO WHILE Flowchart

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

Mnemonics © Intel, 1978

```
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.

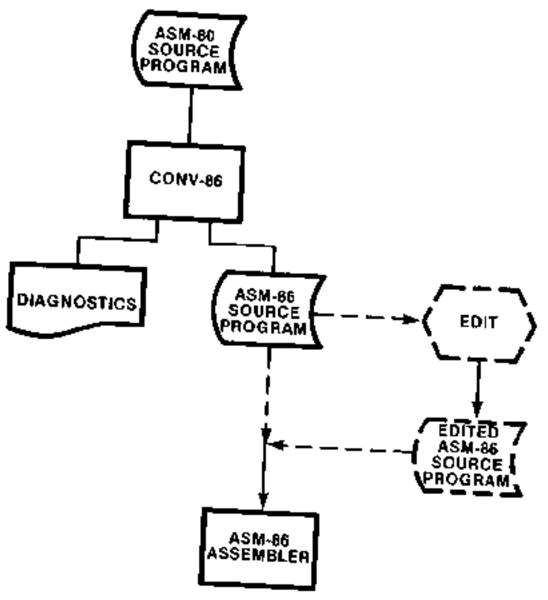


Figure 2-65. ASM-80/ASM-86 Conversion

## 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

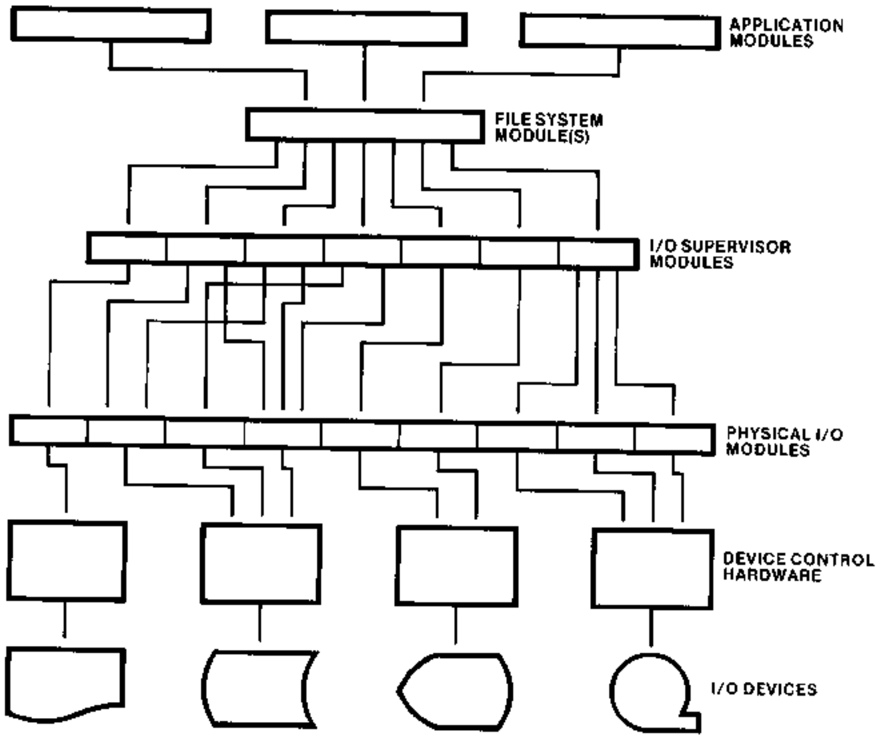


Figure 2-68. I/O System Hierarchy Concept

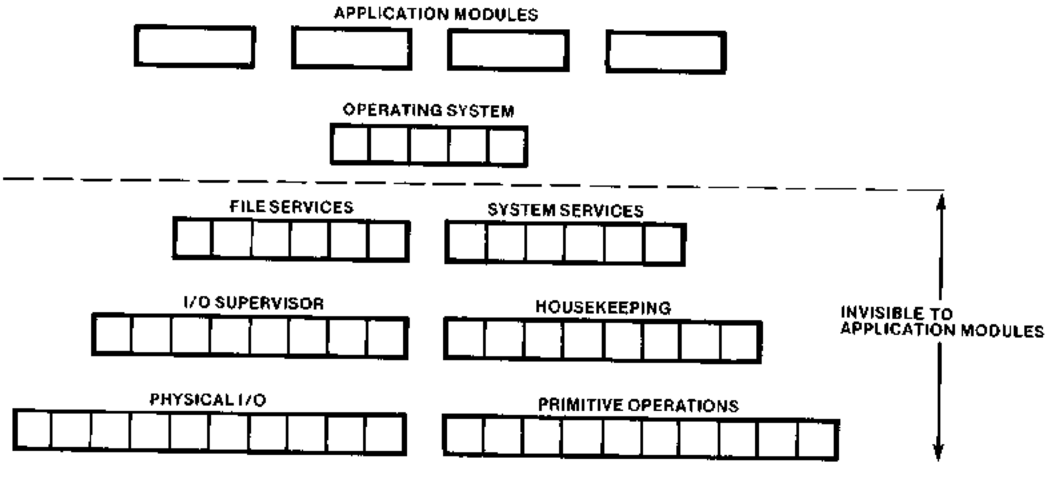


Figure 2-69. Operating System Hierarchy

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.

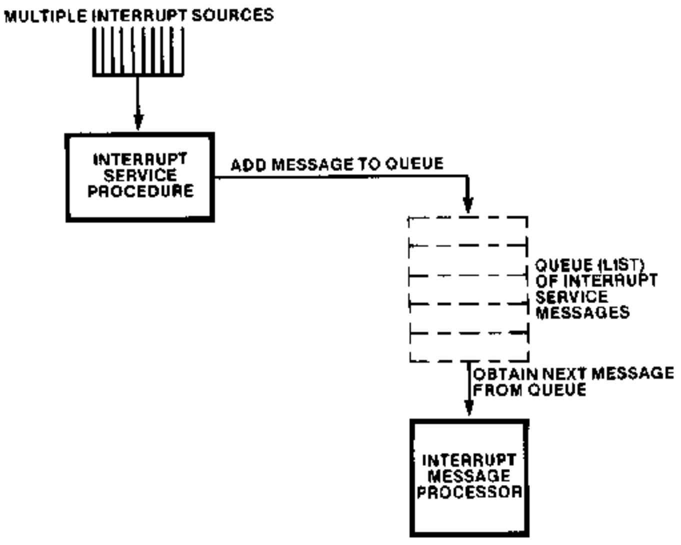

MULTIPLE INTERRUPT SOURCES
INTERRUPT SERVICE PROCEDURE
ADD MESSAGE TO QUEUE
QUEUE (LIST) OF INTERRUPT SERVICE MESSAGES
OBTAIN NEXT MESSAGE FROM QUEUE
INTERRUPT MESSAGE PROCESSOR

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 intrasegment 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.

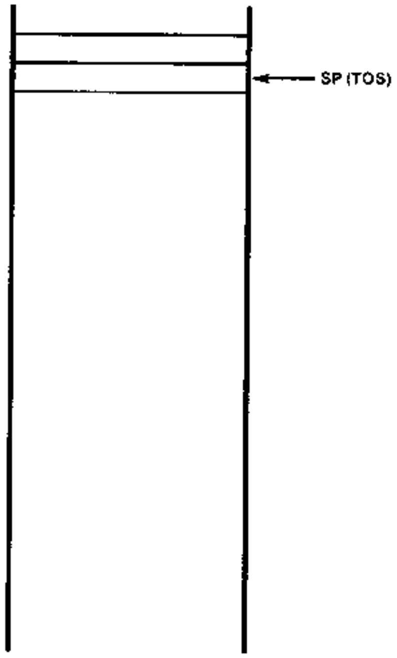


Figure 2-72. Stack Before Pushing Parameters

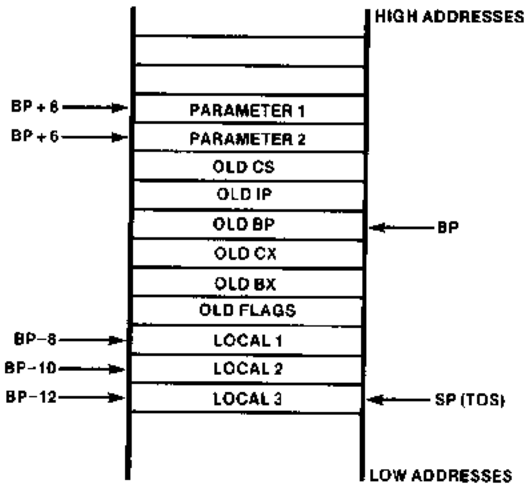


Figure 2-74. Stack Following Procedure Prolog

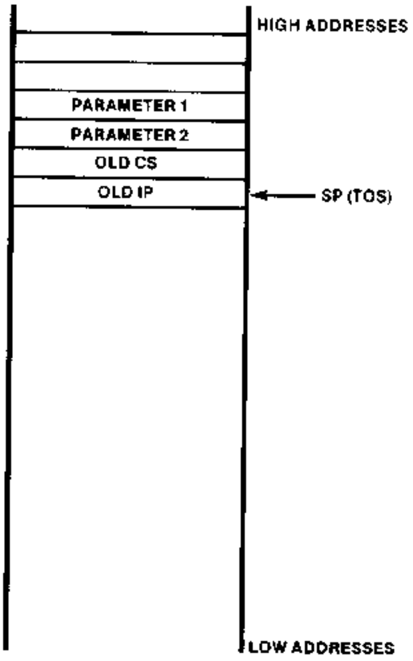


Figure 2-73. Stack at Procedure Entry

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).

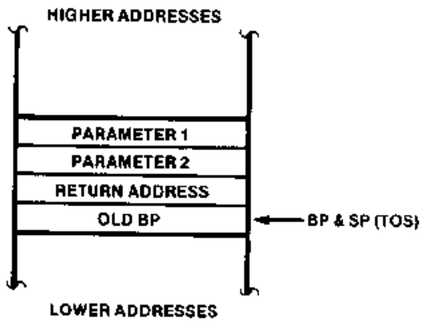


Figure 2-75. Stack Following Procedure Epilog

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.

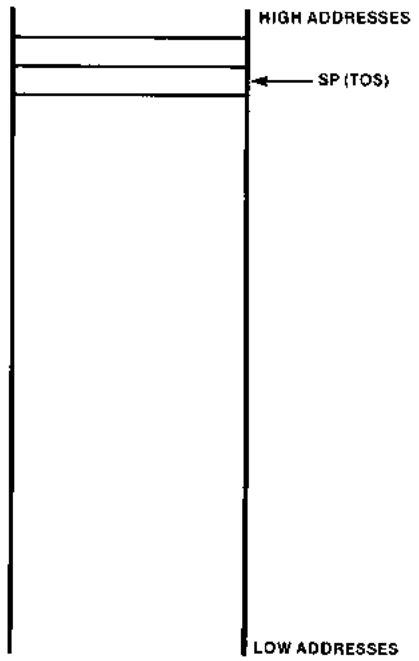


**Figure 2-76. Stack Following Procedure Return**

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

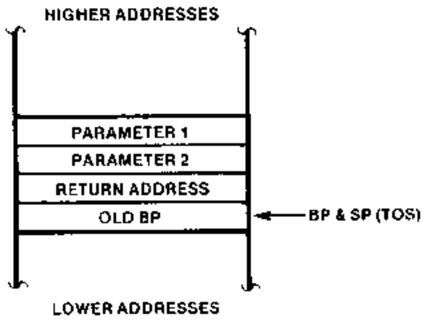


Figure 2-78. Procedure Example 2 Stack Layout

### 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.)**

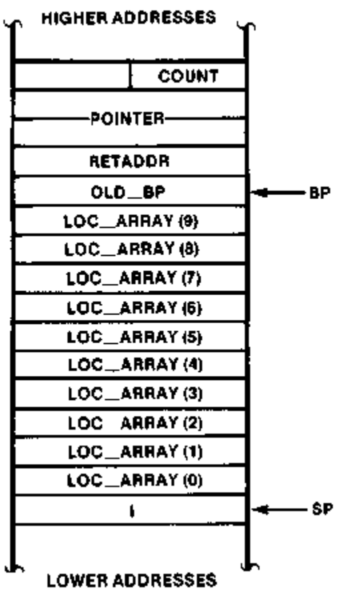


**Figure 2-80. Procedure Example 3 Stack Layout**

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.

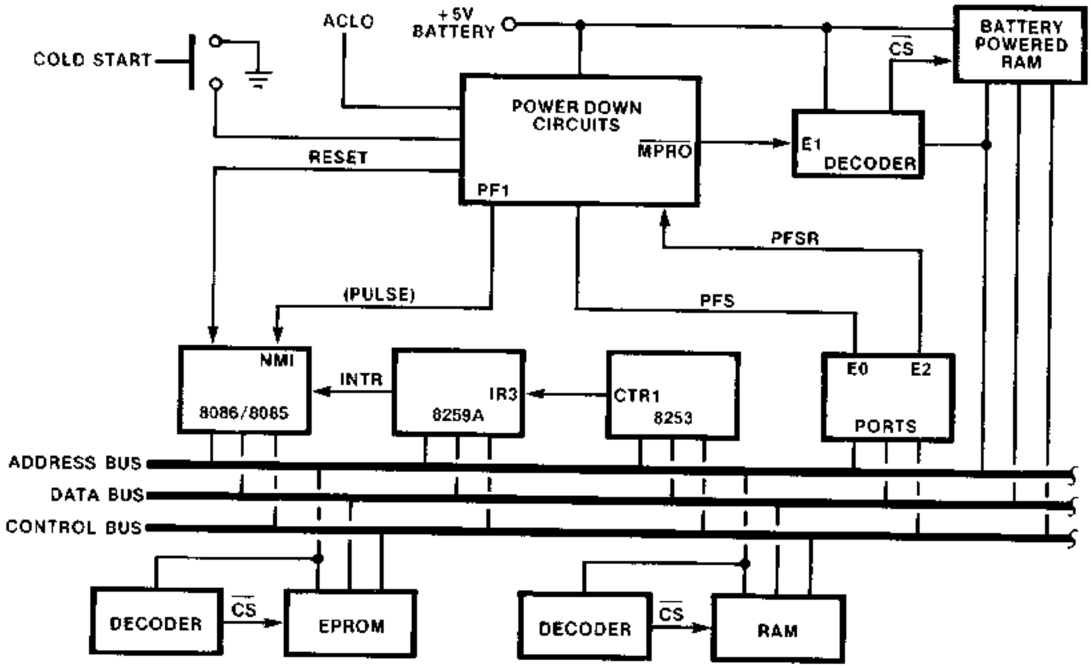


Figure 2-87. Interrupt Example Block Diagram

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,OCW1         ; 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 4
# HARDWARE REFERENCE INFORMATION

## 4.1 Introduction

This chapter presents specific hardware information regarding the operation and functions of the 8086 family processors: the 8086 and 8088 Central Processing Units (CPUs) and the 8089 I/O Processor (IOP). Abbreviated descriptions of the 8086 family support circuits and their circuit functions appear where appropriate within the processor descriptions. For more specific information on any of the 8086 family support circuits, refer to the corresponding data sheets in Appendix B.

## 4.2 8086 and 8088 CPUs

The 8086 and 8088 CPUs are characterized by a 20-bit (1 megabyte) address bus and an identical instruction/function format, and differ essentially from one another by their respective data bus widths (the 8086 uses a 16-bit data bus, and the 8088 uses an 8-bit data bus). Except where expressly noted, the ensuing descriptions are applicable to both CPUs.

Both the 8086 and 8088 feature a combined or "time-multiplexed" address and data bus that permits a number of the pins to serve dual functions and consequently allows the complete CPU to be incorporated into a single, 40-pin package. As explained later in this chapter, a number of the CPU's control pins are defined according to the strapping of a single input pin (the MN/$\overline{\text{MX}}$ pin). In the "minimum mode," the CPU is configured for small, single-processor systems, and the CPU itself provides all control signals. In the "maximum mode," an Intel® 8288 Bus Controller, rather than the CPU, provides the control signal outputs and allows a number of the pins previously delegated to these control functions to be redefined in order to support multiprocessing applications. Figures 4-1 and 4-2 describe the pin assignments and signal definitions for the 8086 and 8088, respectively.

### CPU Architecture

As shown in figures 4-3 and 4-4, both CPUs incorporate two separate processing units: the Execution Unit or "EU" and the Bus Interface Unit or "BIU." The EU for each processor is identical. The BIU for the 8086 incorporates a 16-bit data bus and a 6-byte instruction queue whereas the 8088 incorporates an 8-bit data bus and a 4-byte instruction queue.

The EU is responsible for the execution of all instructions, for providing data and addresses to the BIU, and for manipulating the general registers and the flag register. Except for a few control pins, the EU is completely isolated from the "outside world." The BIU is responsible for executing all external bus cycles and consists of the segment and communications registers, the instruction pointer and the instruction object code queue. The BIU combines segment and offset values in its dedicated adder to derive 20-bit addresses, transfers data to and from the EU on the ALU data bus and loads or "prefetches" instructions into the queue from which they are fetched by the EU.

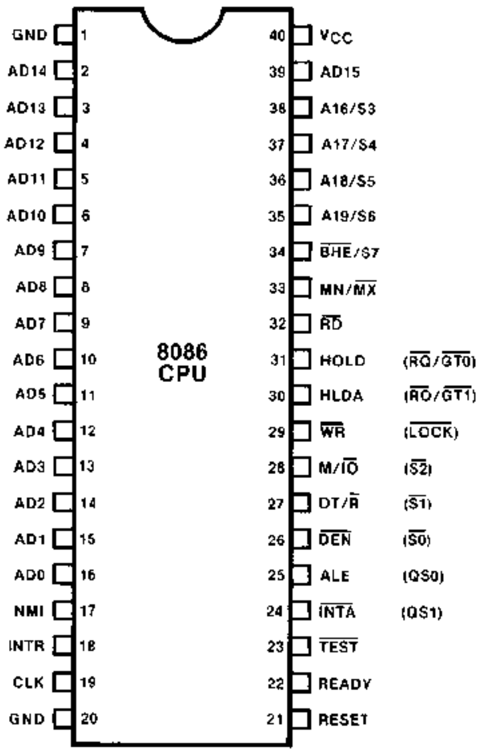
The EU, when it is ready to execute an instruction, fetches the instruction object code byte from the BIU's instruction queue and then executes the instruction. If the queue is empty when the EU is ready to fetch an instruction byte, the EU waits for the instruction byte to be fetched. In the course of instruction execution, if a memory location or I/O port must be accessed, the EU requests the BIU to perform the required bus cycle.

The two processing sections of the CPU operate independently. In the 8086 CPU, when two or more bytes of the 6-byte instruction queue are empty and the EU does not require the BIU to perform a bus cycle, the BIU executes instruction fetch cycles to refill the queue. In the 8088 CPU, when one byte of the 4-byte instruction queue is empty, the BIU executes an instruction fetch cycle. Note that the 8086 CPU, since it has a 16-bit data bus, can access two instruction object code bytes in a single bus cycle, while the 8088 CPU, since it has an 8-bit data bus, accesses one instruction object code byte per bus cycle. If the EU issues a request for bus access while the BIU is in the process of an instruction fetch bus cycle, the BIU completes the cycle before honoring the EU's request.

| Common Signals | | |
|---|---|---|
| **Name** | **Function** | **Type** |
| AD15-AD0 | Address/Data Bus | Bidirectional, 3-State |
| A19/S6-A16/S3 | Address/Status | Output, 3-State |
| $\overline{\text{BHE}}$/S7 | Bus High Enable/ Status | Output, 3-State |
| MN/$\overline{\text{MX}}$ | Minimum/Maximum Mode Control | Input |
| $\overline{\text{RD}}$ | Read Control | Output, 3-State |
| $\overline{\text{TEST}}$ | Wait On Test Control | Input |
| READY | Wait State Control | Input |
| RESET | System Reset | Input |
| NMI | Non-Maskable Interrupt Request | Input |
| INTR | Interrupt Request | Input |
| CLK | System Clock | Input |
| $V_{CC}$ | +5V | Input |
| GND | Ground | |

| Minimum Mode Signals (MN/MX = $V_{CC}$) | | |
|---|---|---|
| **Name** | **Function** | **Type** |
| HOLD | Hold Request | Input |
| HLDA | Hold Acknowledge | Output |
| $\overline{\text{WR}}$ | Write Control | Output, 3-State |
| M/$\overline{\text{IO}}$ | Memory/IO Control | Output, 3-State |
| DT/$\overline{\text{R}}$ | Data Transmit/ Receive | Output, 3-State |
| $\overline{\text{DEN}}$ | Data Enable | Output, 3-State |
| ALE | Address Latch Enable | Output |
| $\overline{\text{INTA}}$ | Interrupt Acknowledge | Output |

| Maximum Mode Signals (MN/MX = GND) | | |
|---|---|---|
| **Name** | **Function** | **Type** |
| $\overline{\text{RQ}}$/$\overline{\text{GT1, 0}}$ | Request/Grant Bus Access Control | Bidirectional |
| $\overline{\text{LOCK}}$ | Bus Priority Lock Control | Output, 3-State |
| $\overline{\text{S2}}$-$\overline{\text{S0}}$ | Bus Cycle Status | Output, 3-State |
| QS1, QS0 | Instruction Queue Status | Output |



MAXIMUM MODE PIN FUNCTIONS (e.g., $\overline{\text{LOCK}}$) ARE SHOWN IN PARENTHESES

Figure 4-1. 8086 Pin Definitions

**Common Signals**

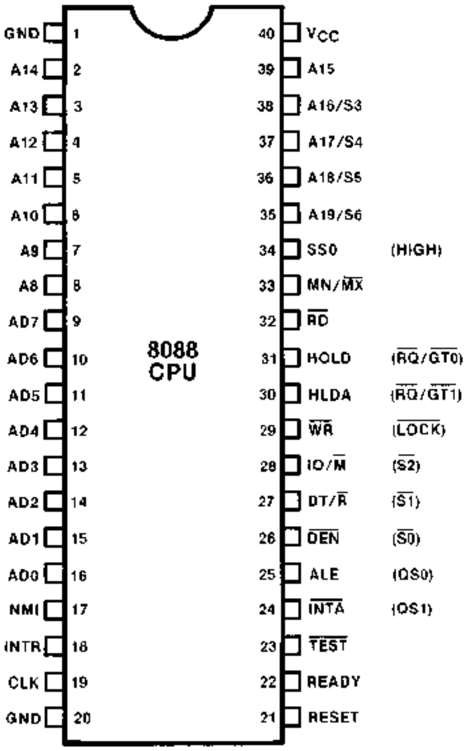
| Name | Function | Type |
|---|---|---|
| AD7-AD0 | Address/Data Bus | Bidirectional, 3-State |
| A15-A8 | Address Bus | Output, 3-State |
| A19/S6-A16/S3 | Address/Status | Output, 3-State |
| MN/$\overline{\text{MX}}$ | Minimum/Maximum Mode Control | Input |
| $\overline{\text{RD}}$ | Read Control | Output, 3-State |
| $\overline{\text{TEST}}$ | Wait On Test Control | Input |
| READY | Wait State Control | Input |
| RESET | System Reset | Input |
| NMI | Non-Maskable Interrupt Request | Input |
| INTR | Interrupt Request | Input |
| CLK | System Clock | Input |
| $V_{CC}$ | +5V | Input |
| GND | Ground | |

**Minimum Mode Signals (MN/$\overline{\text{MX}}$ = $V_{CC}$)**

| Name | Function | Type |
|---|---|---|
| HOLD | Hold Request | Input |
| HLDA | Hold Acknowledge | Output |
| $\overline{\text{WR}}$ | Write Control | Output, 3-State |
| IO/$\overline{\text{M}}$ | IO/Memory Control | Output, 3-State |
| DT/$\overline{\text{R}}$ | Data Transmit/ Receive | Output, 3-State |
| $\overline{\text{DEN}}$ | Data Enable | Output, 3-State |
| ALE | Address Latch Enable | Output |
| $\overline{\text{INTA}}$ | Interrupt Acknowledge | Output |
| SSO | S0 Status | Output, 3-State |

**Maximum Mode Signals (MN/$\overline{\text{MX}}$ = GND)**

| Name | Function | Type |
|---|---|---|
| $\overline{\text{RQ}}/\overline{\text{GT1}}$, 0 | Request/Grant Bus Access Control | Bidirectional |
| $\overline{\text{LOCK}}$ | Bus Priority Lock Control | Output, 3-State |
| $\overline{\text{S2}}$-$\overline{\text{S0}}$ | Bus Cycle Status | Output, 3-State |
| QS1, QS0 | Instruction Queue Status | Output |



8088 CPU
GND 1
A14 2
A13 3
A12 4
A11 5
A10 6
A9 7
A8 8
AD7 9
AD6 10
AD5 11
AD4 12
AD3 13
AD2 14
AD1 15
AD0 16
NMI 17
INTR 18
CLK 19
GND 20
40 Vcc
39 A15
38 A16/S3
37 A17/S4
36 A18/S5
35 A19/S6
34 SSO (HIGH)
33 MN/$\overline{\text{MX}}$
32 $\overline{\text{RD}}$
31 HOLD ($\overline{\text{RQ}}/\overline{\text{GT0}}$)
30 HLDA ($\overline{\text{RQ}}/\overline{\text{GT1}}$)
29 $\overline{\text{WR}}$ ($\overline{\text{LOCK}}$)
28 IO/$\overline{\text{M}}$ ($\overline{\text{S2}}$)
27 DT/$\overline{\text{R}}$ ($\overline{\text{S1}}$)
26 $\overline{\text{DEN}}$ ($\overline{\text{S0}}$)
25 ALE (QS0)
24 $\overline{\text{INTA}}$ (QS1)
23 $\overline{\text{TEST}}$
22 READY
21 RESET

MAXIMUM MODE PIN FUNCTIONS (e.g.,$\overline{\text{LOCK}}$) ARE SHOWN IN PARENTHESES

Figure 4-2. 8088 Pin Definitions

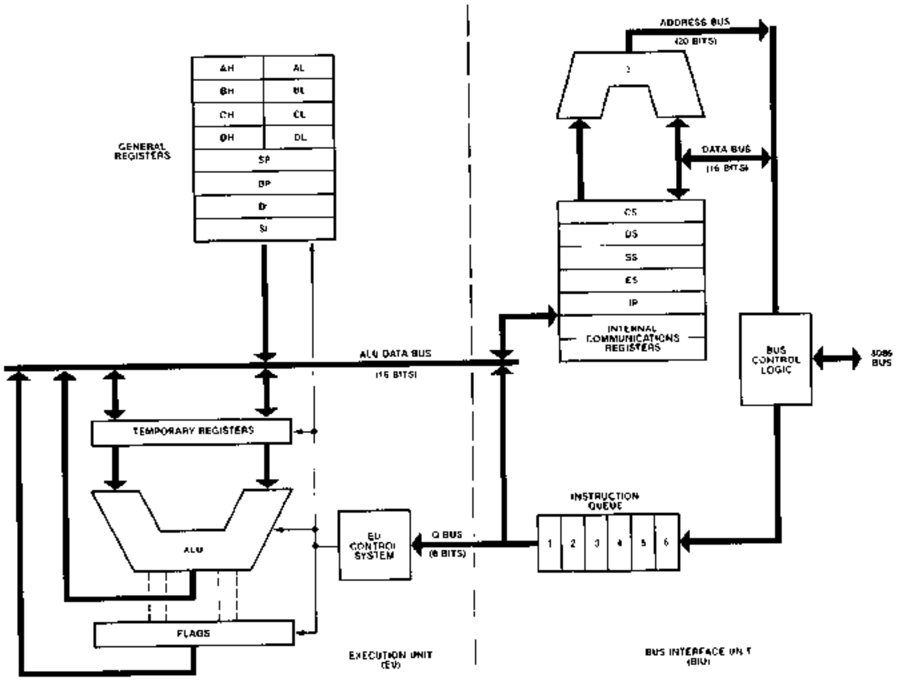


Figure 4-3. 8086 Elementary Block Diagram

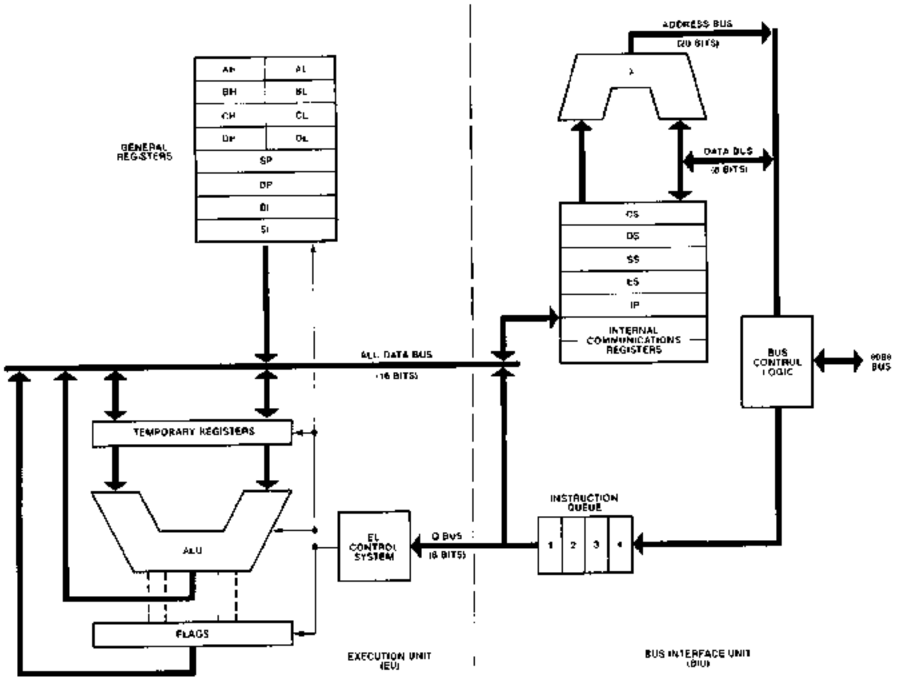


Figure 4-4. 8088 Elementary Block Diagram

## Bus Operation

To explain the operation of the time-multiplexed bus, the BIU's bus cycle must be examined. Essentially, a bus cycle is an asynchronous event in which the address of an I/O peripheral or memory location is presented, followed by either a read control signal (to capture or "read" the data from the addressed device) or a write control signal and the associated data (to transmit or "write" the data to the addressed device). The selected device (memory or I/O peripheral) accepts the data on the bus during a write cycle or places the requested data on the bus during a read cycle. On termination of the cycle, the device latches the data written or removes the data read.

As shown in figure 4-5, all bus cycles consist of a minimum of four clock cycles or "T-states" identified as $T_1$, $T_2$, $T_3$ and $T_4$. The CPU places the address of the memory location or I/O device on the bus during state $T_1$. During a write bus cycle, the CPU places the data on the bus from state $T_2$ until state $T_4$. During a read bus cycle, the CPU accepts the data present on the bus in states $T_3$ and $T_4$, and the multiplexed address/data bus is floated in state $T_2$ to allow the CPU to change from the write mode (output address) to the read mode (input data).

It is important to note that the BIU executes a bus cycle only when a bus cycle is requested by the EU as part of instruction execution or when it must fill the instruction queue. Consequently, clock periods in which there is no BIU activity can occur between bus cycles. These inactive clock periods are referred to as idle states ($T_I$). While idle clock states result from several conditions (e.g., bus access granted to a coprocessor), as an example, consider the case of the execution of a "long" instruction. In the following example, an 8-bit register multiply (MUL) instruction (which requires between 70 and 77 clock cycles) is executed by the 8086. Assuming that the multiplication routine is entered as a result of a program jump (which causes the instruction queue to be reinitialized when the jump is executed) and, as will be explained later in this chapter, that the object code bytes are aligned on even-byte boundaries, the BIU's bus cycle sequence would appear as shown in figure 4-6.

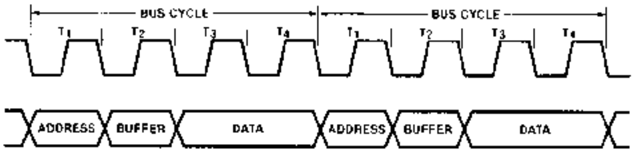
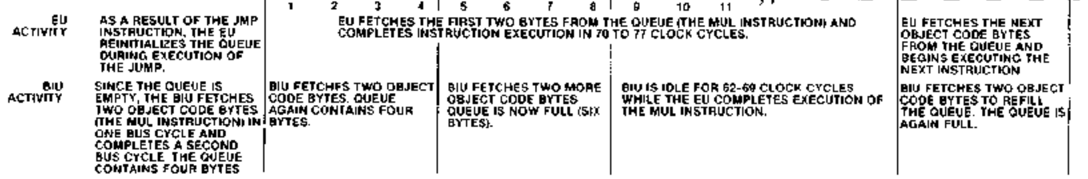


Figure 4-5. Typical BIU Bus Cycles



EU ACTIVITY: AS A RESULT OF THE JMP INSTRUCTION, THE EU REINITIALIZES THE QUEUE DURING EXECUTION OF THE JUMP. | EU FETCHES THE FIRST TWO BYTES FROM THE QUEUE (THE MUL INSTRUCTION) AND COMPLETES INSTRUCTION EXECUTION IN 70 TO 77 CLOCK CYCLES. | EU FETCHES THE NEXT OBJECT CODE BYTES FROM THE QUEUE AND BEGINS EXECUTING THE NEXT INSTRUCTION

BIU ACTIVITY: SINCE THE QUEUE IS EMPTY, THE BIU FETCHES TWO OBJECT CODE BYTES (THE MUL INSTRUCTION) IN ONE BUS CYCLE AND COMPLETES A SECOND BUS CYCLE. THE QUEUE CONTAINS FOUR BYTES | BIU FETCHES TWO OBJECT CODE BYTES. QUEUE AGAIN CONTAINS FOUR BYTES. | BIU FETCHES TWO MORE OBJECT CODE BYTES QUEUE IS NOW FULL (SIX BYTES). | BIU IS IDLE FOR 62-69 CLOCK CYCLES WHILE THE EU COMPLETES EXECUTION OF THE MUL INSTRUCTION. | BIU FETCHES TWO OBJECT CODE BYTES TO REFILL THE QUEUE. THE QUEUE IS AGAIN FULL.

Figure 4-6. BIU Idle States

In addition to the idle state previously described, both the 8086 and 8088 CPUs include a mechanism for inserting additional T-states in the bus cycle to compensate for devices (memory or I/O) that cannot transfer data at the maximum rate. These extra T-states are called wait states ($T_W$) and, when required, are inserted between states $T_3$ and $T_4$. During a wait state, the data on the bus remains unchanged. When the device can complete the transfer (present or accept the data), it signals the CPU to exit the wait state and to enter state $T_4$.

As shown in the following timing diagrams, the actual bus cycle timing differs between a read and a write bus cycle and varies between the two CPUs. Note that the timing diagrams illustrated are for the minimum mode. (Maximum mode timing is described later in this chapter.)

Referring to figures 4-7 and 4-8, the 8086 CPU places a 20-bit address on the multiplexed address/data bus during state $T_1$. During state $T_2$, the CPU removes the address from the bus and either three-states (floats) the lower 16 address/data lines in preparation for a read cycle (figure 4-7) or places write data on these lines (figure 4-8). At this time, bus cycle status is available on the address/status lines. During state $T_3$, bus cycle status is maintained on the address/status lines and either the write data is maintained or read data is sampled on the lower 16 address/data lines. The bus cycle is terminated in state $T_4$ (control lines are disabled and the addressed device deselects from the bus).

The 8088 CPU, like the 8086, places a 20-bit address on the multiplexed address/data bus during state $T_1$ as shown in figures 4-9 and 4-10. Unlike the 8086, the 8088 maintains the address on the address lines ($A_{15}$-$A_8$) for the entire bus cycle. During state $T_2$, the CPU removes the address on the address/data lines ($AD_7$-$AD_0$) and either floats these lines in preparation for a read cycle (figure 4-9) or places write data on these lines (figure 4-10). At this time, bus cycle status is available on the address/status lines. During state $T_3$, bus cycle status is maintained on the address/status lines and either write data is maintained or read data is sampled on the address/data lines. The bus cycle is terminated in state $T_4$ (control lines are disabled and the addressed device deselects from the bus).

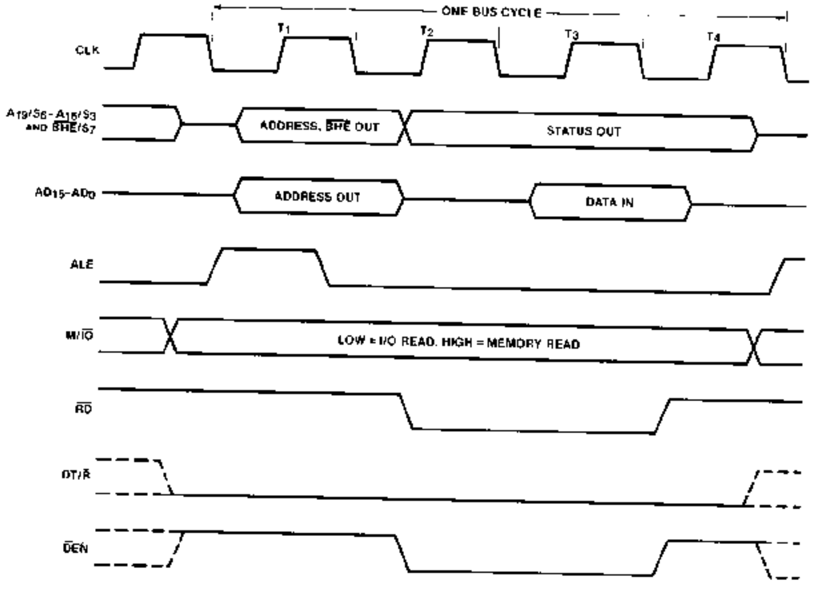


Figure 4-7. 8086 Read Bus Cycle

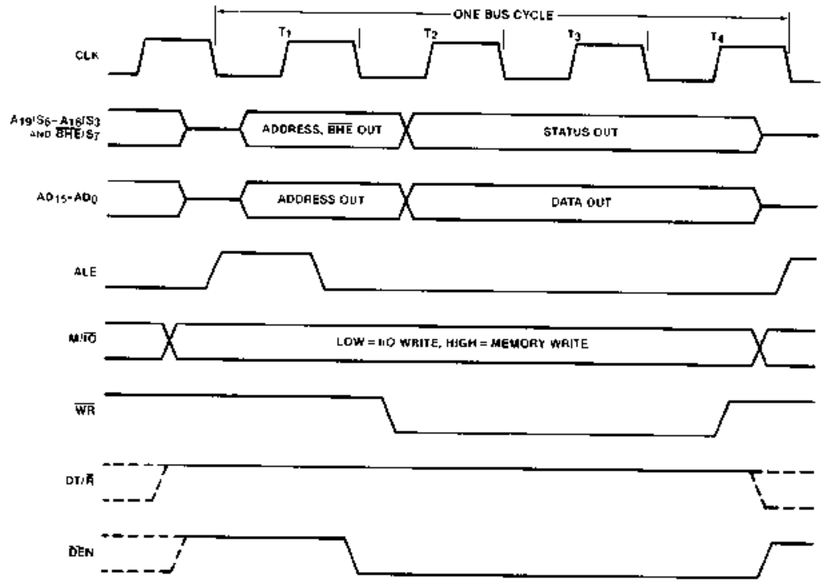


Figure 4-8. 8086 Write Bus Cycle

A majority of system memories and peripherals require a stable address for the duration of the bus cycle (certain MCS-85™ components can operate with a multiplexed address/data bus). During state $T_1$ of every bus cycle, the ALE (Address Latch Enable) control signal is output (either directly from the microprocessor in the minimum mode or indirectly through an 8288 Bus Controller in the maximum mode) to permit the address to be latched (the address is valid on the trailing-edge of ALE). This "demultiplexing" of the address/data bus can be done remotely at each device in the system or locally at the CPU and distributed throughout the system as a separate address bus. For optimum system performance and for compatibility with multiprocessor systems or with the Intel Multibus architecture, the locally-demultiplexed address bus is recommended. To latch the address, Intel® 8282 (non-inverting) or 8283 (inverting) Octal Latches are offered as part of the 8086 product family and are implemented as shown in figure 4-11. These circuits, in addition to providing the desired latch function, provide increased current drive capability and capacitive load immunity.

The data bus cannot be demultiplexed due to the timing differences between read and write cycles and the various read response times among peripherals and memories. Consequently, the multiplexed data bus either can be buffered or used directly. When memory and I/O peripherals are connected directly to an unbuffered bus, it is essential that during a read cycle, a device is prevented from corrupting the address present on the bus during state $T_1$. To ensure that the address is not corrupted, a device's output drivers should be enabled by an output enable function (rather than the device's chip select function) controlled by the CPU's read signal. (The MCS-86 family processors guarantee that the read signal will not be valid until after the address has been latched by ALE.) Many Intel peripheral, ROM/EPROM, and RAM circuits provide an output enable function to allow interface to an unbuffered multiplexed address/data bus. The alternative of using a buffered data bus should be considered since it simplifies the interfacing requirements and offers both increased drive current capability and capacitive load immunity. The Intel® 8286 (non-inverting) and 8287 (inverting)

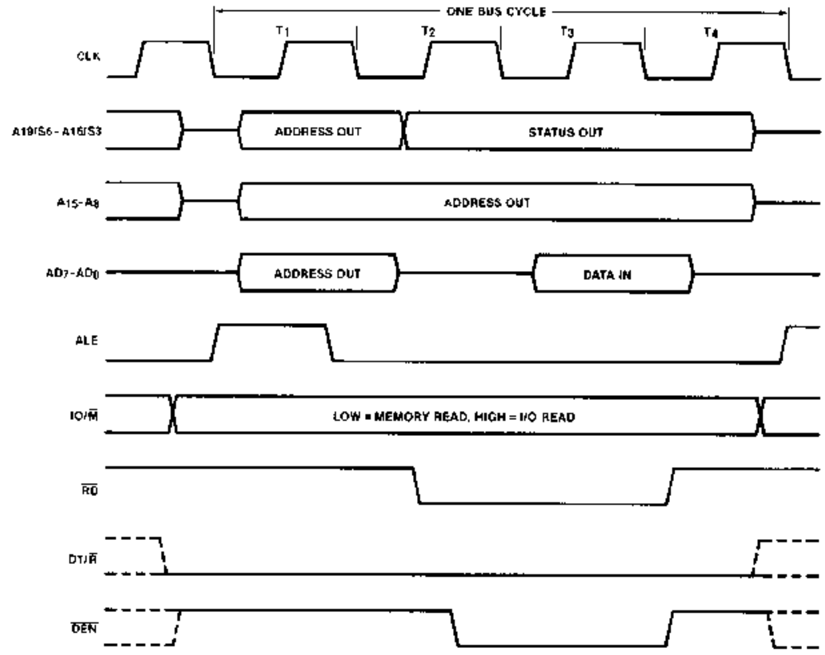


Figure 4-9. 8088 Read Bus Cycle

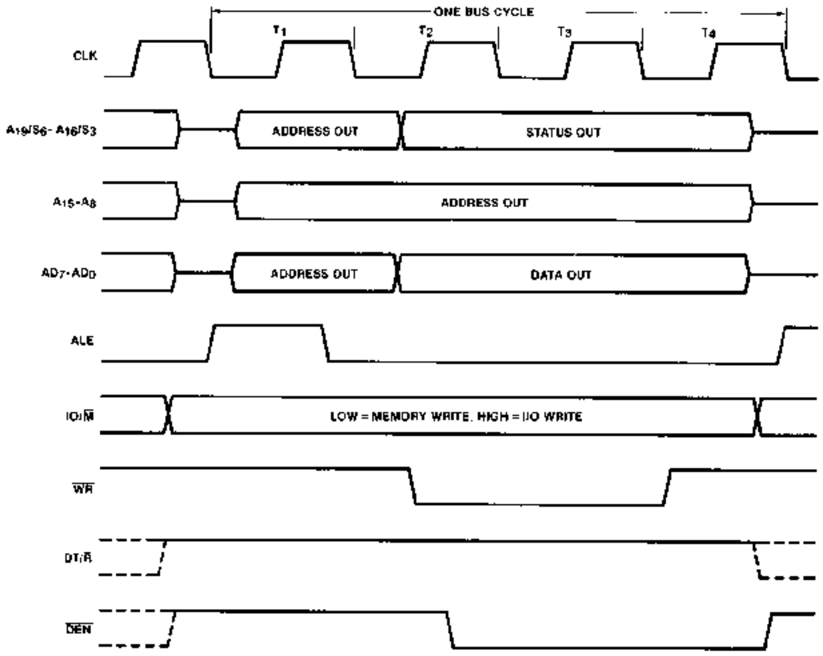


Figure 4-10. 8088 Write Bus Cycle

Octal Bus Transceivers, shown in figure 4-12, are expressly designed to buffer the data bus. These transceivers use the CPU's $\overline{\text{DEN}}$ (Data Enable) and DT/$\overline{\text{R}}$ (Data Transmit/Receive) control signals to enable and control the direction of data on the bus. These signals provide the proper timing relationship to guarantee isolation of the address that is present on the multiplexed bus during state $T_1$.

Except where noted, all subsequent discussions and examples in this chapter assume a locally demultiplexed address bus and a buffered data bus. The resultant address and data buses from the address latches and data transceivers to the memory and I/O devices will be referred to collectively as the "system" bus.

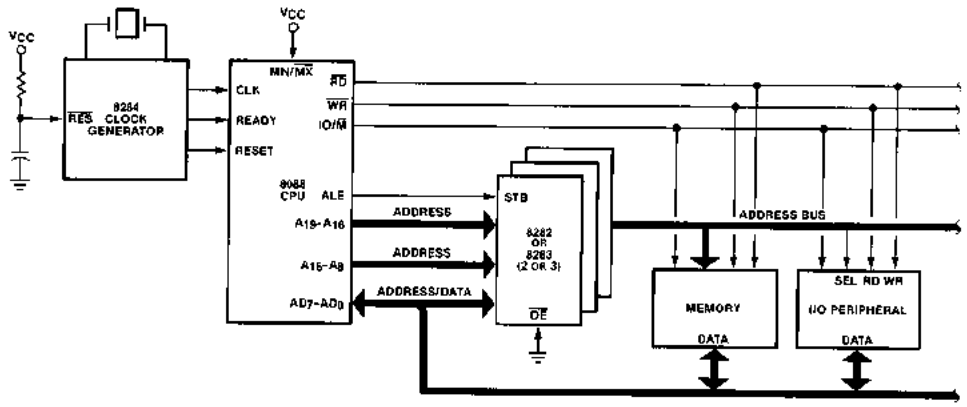


Figure 4-11. Minimum Mode 8088 Demultiplexed Address Bus

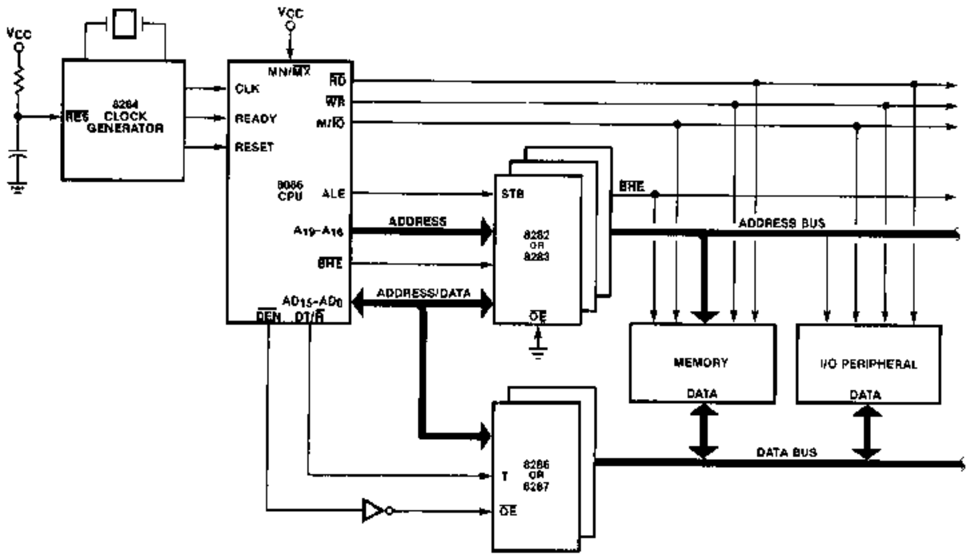


Figure 4-12. Minimum Mode 8086 Buffered Data Bus

## Clock Circuit

To establish the bus cycle time, the CPU requires an external clock signal. As an integral part of the 8086 family, Intel offers the 8284 Clock Generator/Driver for this purpose. In addition to providing the primary (system) clock signal, this device provides both the hardware reset interface and the mechanism for the insertion of wait states in the bus cycle.

The clock generator/driver requires an external series-resonant crystal input (or external frequency source) at three times the required system clock frequency (i.e., to operate the CPU at 5 MHz, a 15 MHz fundamental frequency source is required). The divided-by-three output (CLK) from the 8284 is routed directly to the CPU's CLK input. The clock generator/driver provides a second clock output called PCLK (Peripheral Clock) at one half the frequency of the CLK output and a buffered TTL level OSC (oscillator) output at the applied crystal input frequency. These outputs are available for use by system devices.

The 8284's hardware reset function is accomplished with an internal Schmitt trigger circuit that is activated by the $\overline{\text{RES}}$ (Reset) input. When this input is pulled low (i.e., a contact closure to ground), the RESET output is activated synchronously with the CLK signal. This signal must be active for four clock cycles and causes the CPU to fetch and execute the instruction at location FFFF0H. An external RC circuit is connected to the RES input to provide the power-on reset function (on power-on, the $\overline{\text{RES}}$ input must be active for 50 microseconds). The RESET output is coupled directly to the RESET input of the CPU as well as being available to system peripherals as the system reset signal.

The insertion of wait states in the CPU's bus cycle is accomplished by deactivating one of the 8284's RDY inputs (RDY1 or RDY2). Either of these inputs, when enabled by its corresponding $\overline{\text{AEN1}}$ or $\overline{\text{AEN2}}$ input, can be deactivated directly by a peripheral device when it must extend the CPU's bus cycle (when it is not ready to present or accept data) or by a "wait state generator" circuit (a logic circuit that holds the RDY input inactive for a given number of clock cycles).

The READY output, which is synchronized to the CLK signal is coupled directly to the CPU's READY input. As shown in figure 4-13, when the addressed device needs to insert one or more wait states in a bus cycle, it deactivates the 8284's RDY input prior to the end of state $T_2$ which causes the READY output to be deactivated at the end of state $T_2$. The resultant wait state ($T_W$) is inserted between states $T_3$ and $T_4$. To exit the wait state, the device activates the 8284's RDY input which causes the READY input to the CPU to go active at the end of the current wait state and allows the CPU to enter state $T_4$.

## Minimum/Maximum Mode

A unique feature of the 8086 and 8088 CPUs is the ability of a user to define a subset of the CPU's control signal outputs in order to tailor the CPU to its intended system environment. This "system tailoring" is accomplished by the strapping of the CPU's MN/$\overline{\text{MX}}$ (minimum/maximum) input pin. Table 4-1 defines the 8086 and 8088 pin assignments in both the minimum and maximum modes.

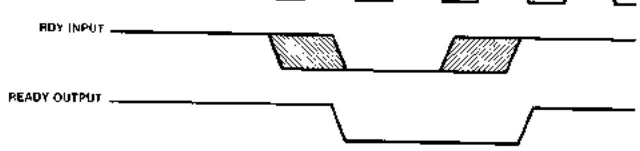


Figure 4-13. Wait State Timing

Table 4-1. Minimum/Maximum Mode Pin Assignments

| 8086 | | | 8088 | | |
|---|---|---|---|---|---|
| Pin | Mode | | Pin | Mode | |
| | Minimum | Maximum | | Minimum | Maximum |
| 31 | HOLD | $\overline{RQ}/\overline{GT0}$ | 31 | HOLD | $\overline{RQ}/\overline{GT0}$ |
| 30 | HLDA | $\overline{RQ}/\overline{GT1}$ | 30 | HLDA | $\overline{RQ}/\overline{GT1}$ |
| 29 | $\overline{WR}$ | $\overline{LOCK}$ | 29 | $\overline{WR}$ | $\overline{LOCK}$ |
| 28 | M/$\overline{IO}$ | $\overline{S2}$ | 28 | IO/$\overline{M}$ | $\overline{S2}$ |
| 27 | DT/$\overline{R}$ | $\overline{S1}$ | 27 | DT/$\overline{R}$ | $\overline{S1}$ |
| 26 | $\overline{DEN}$ | $\overline{S0}$ | 26 | $\overline{DEN}$ | $\overline{S0}$ |
| 25 | ALE | QS0 | 25 | ALE | QS0 |
| 24 | $\overline{INTA}$ | QS1 | 24 | $\overline{INTA}$ | QS1 |
| | | | 34 | SS0 | High State |

**Minimum Mode**

In the minimum mode (MN/$\overline{MX}$ pin strapped to +5V), the CPU supports small, single-processor systems that consist of a few devices and that use the system bus rather than support the Multibus™ architecture. In the minimum mode, the CPU itself generates all bus control signals (DT/$\overline{R}$, $\overline{DEN}$, ALE and either M/$\overline{IO}$ or IO/$\overline{M}$) and the command output signal ($\overline{RD}$, $\overline{WR}$ or $\overline{INTA}$), and provides a mechanism for requesting bus access (HOLD/HLDA) that is compatible with bus master type controllers (e.g., the Intel® 8237 and 8257 DMA Controllers).

In the minimum mode, when a bus master requires bus access, it activates the HOLD input to the CPU (through its request logic). The CPU, in response to the "hold" request, activates HLDA as an acknowledgement to the bus master requesting the bus and simultaneously floats the system bus and control lines. Since a bus request is asynchronous, the CPU samples the HOLD input on the positive transition of each CLK signal and, as shown in figure 4-14, activates HLDA at the end of either the current bus cycle (if a bus cycle is in progress) or idle clock period. The hold state is maintained until the bus master inactivates the HOLD input at which time the CPU regains control of the system bus. Note that during a "hold" state, the CPU will continue to execute instructions until a bus cycle is required.

Note that in the minimum mode, the I/O-memory control line for the 8088 CPU is the converse of the corresponding control line for the 8086 CPU (M/$\overline{IO}$ on the 8086 and IO/$\overline{M}$ on the 8088). This was done to provide the 8088 CPU, since it is an 8-bit device, compatibility with existing MCS-85™ systems and specific MCS-85™ family devices (e.g., the Intel® 8155/56).

**Maximum Mode**

In the maximum mode (MN/$\overline{MX}$ pin strapped to ground), an Intel® 8288 Bus Controller is added to provide a sophisticated bus control function and compatibility with the Multibus architecture (combining an Intel® 8289 Arbiter with the 8288 permits the CPU to support multiple processors on the system bus). As shown in figure 4-15, the bus controller, rather than the CPU, provides all bus control and command outputs, and allows the pins previously delegated to these functions to be redefined to support multiprocessing functions.

**$\overline{S2}$, $\overline{S1}$ and $\overline{S0}$**

Referring to figure 4-15, the 8288 Bus Controller uses the $\overline{S2}$, $\overline{S1}$ and $\overline{S0}$ status bit outputs from the CPU (and the 8089 IOP) to generate all bus control and command output signals required for a bus cycle. The status bit outputs are decoded as outlined in table 4-2. (For a detailed description of the operation of the 8288 Bus Controller, refer to the associated data sheet in Appendix B.)

The 8088 CPU, in the minimum mode, provides an SS0 status output. This output is equivalent to S0 in the maximum mode and can be decoded with DT/$\overline{R}$ and IO/$\overline{M}$ (inverted), which are equivalent to $\overline{S1}$ and $\overline{S2}$ respectively, to provide the same CPU cycle status information defined in table 4-2. This type of decoding could be used in a minimum mode 8088-based system to allow dynamic RAM refresh during passive CPU cycles.

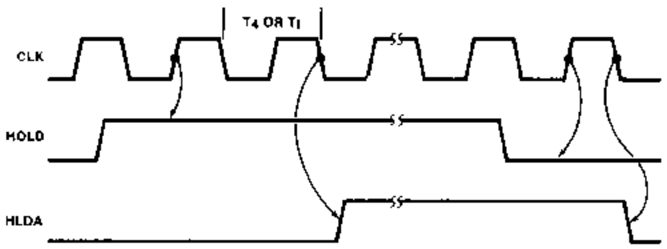


Figure 4-14. HOLD/HLDA Timing

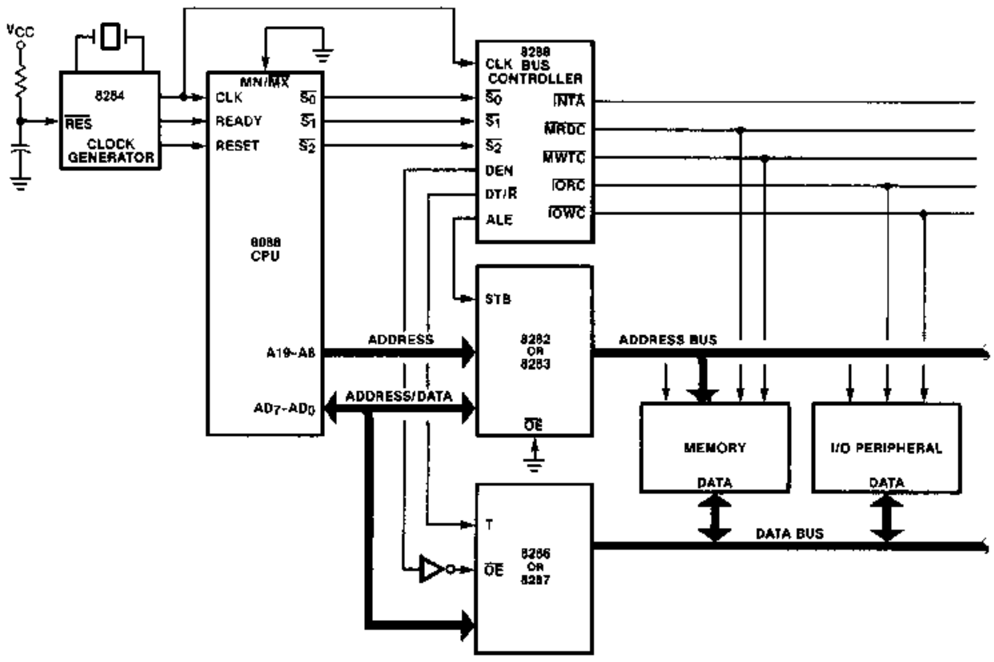


Figure 4-15. Elementary Maximum Mode System

Table 4-2. Status Bit Decoding

| Status Inputs | | | CPU Cycle | 8288 Command |
|---|---|---|---|---|
| $\overline{S2}$ | $\overline{S1}$ | $\overline{S0}$ | | |
| 0 | 0 | 0 | Interrupt Acknowledge | $\overline{INTA}$ |
| 0 | 0 | 1 | Read I/O Port | $\overline{IORC}$ |
| 0 | 1 | 0 | Write I/O Port | $\overline{IOWC}$, $\overline{AIOWC}$ |
| 0 | 1 | 1 | Halt | None |
| 1 | 0 | 0 | Instruction Fetch | $\overline{MRDC}$ |
| 1 | 0 | 1 | Read Memory | $\overline{MRDC}$ |
| 1 | 1 | 0 | Write Memory | $\overline{MWTC}$, $\overline{AMWC}$ |
| 1 | 1 | 1 | Passive | None |

## $\overline{\text{RQ}}/\overline{\text{GT1}}$, $\overline{\text{RQ}}/\overline{\text{GT0}}$

The Request/Grant signal lines ($\overline{\text{RQ}}/\overline{\text{GT0}}$ and $\overline{\text{RQ}}/\overline{\text{GT1}}$) provide the CPU's bus access mechanism in the maximum mode (replacing the HOLD/HLDA function available in the minimum mode) and are designed expressly for multiprocessor applications using the 8089 I/O Processor in its local mode or other processors that can support this function. These lines are unique in that the request/grant function is accomplished over a single line ($\overline{\text{RQ}}/\overline{\text{GT0}}$ or $\overline{\text{RQ}}/\overline{\text{GT1}}$) rather than the two-line HOLD/HLDA function.

As shown in figure 4-16, the request/grant sequence is a three-phase cycle: request, grant and release. The sequence is initiated by another processor on the system bus when it outputs a pulse on one of the $\overline{\text{RQ}}/\overline{\text{GT}}$ lines to request bus access (request phase). In response, the CPU outputs a pulse (on the same line) at the end of either the current bus cycle (if a bus cycle is in progress) or idle clock period to indicate to the requesting processor that it has floated the system bus and that it will logically disconnect from the bus controller on the next clock cycle (grant phase) and enter a "hold" state. Note that the CPU's execution unit (EU) continues to execute the instructions in the queue until an instruction requiring bus access is encountered or until the queue is empty. In the third (release) phase, the requesting processor again outputs a pulse on the $\overline{\text{RQ}}/\overline{\text{GT}}$ line. This pulse alerts the CPU that the processor is ready to release the bus. The CPU regains bus access on its next clock cycle. Note that the exchange of pulses is synchronized and, accordingly, both the CPU and requesting processor must be referenced to the same clock signal.

The request/grant lines are prioritized with $\overline{\text{RQ}}/\overline{\text{GT0}}$ taking precedence over $\overline{\text{RQ}}/\overline{\text{GT1}}$. If a request arrives on both lines simultaneously, the processor on $\overline{\text{RQ}}/\overline{\text{GT0}}$ is granted the bus (the request on $\overline{\text{RQ}}/\overline{\text{GT1}}$ is granted when the bus is released by the first processor following a one or two clock channel transfer delay). Both $\overline{\text{RQ}}/\overline{\text{GT}}$ lines (and the HOLD line in minimum mode) have a higher priority than a pending interrupt.

Request/grant latency (the time interval between the receipt of a request pulse and the return of a grant pulse) for several conditions is given in table 4-3.

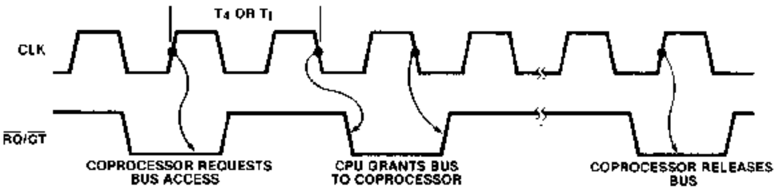

$T_4$ OR $T_I$
CLK
$\overline{\text{RQ}}/\overline{\text{GT}}$
COPROCESSOR REQUESTS BUS ACCESS
CPU GRANTS BUS TO COPROCESSOR
COPROCESSOR RELEASES BUS

Figure 4-16. Request/Grant Timing

Table 4-3. Request/Grant Latency

| Operating Condition | Request/Grant Delay | |
|---|---|---|
| | 8086 | 8088 |
| Normal Instruction Processing—$\overline{\text{LOCK}}$ inactive | 3-6 (10*) clocks | 3-10 clocks |
| $\overline{\text{INTA}}$ Cycle Executing—$\overline{\text{LOCK}}$ active | 15 clocks | 15 clocks |
| Locked XCHG Instruction Processing—$\overline{\text{LOCK}}$ active | 24-31 (39*) clocks | 24-39 clocks |

*The number of clocks in parentheses applies when the instruction being executed references a word operand at an odd address boundary.

Latency during normal instruction processing (LOCK inactive) can be as short as three clock cycles (e.g., during execution of an instruction that does not reference memory) and no more than ten clock cycles. Whenever the LOCK output is active (LOCK is activated during an interrupt acknowledge cycle or during execution of an instruction with a Lock prefix), latency is increased. In the case of the execution of a locked XCHG instruction (used during semaphore examination), maximum latency is limited to 39 clock cycles. Greater latencies occur when a "long" instruction is locked. This, however, is neither necessary nor recommended.

At the end of processor activity, the 8086 or 8088 will not redirve its control and data buses until two clock cycles following receipt of the release pulse (or two clock cycles after HOLD goes inactive in the minimum mode).

A Hold request is honored immediately following CPU reset if the HOLD line is active when the RESET line goes inactive. This action facilitates the downloading of programs and, more specifically, the setting of memory location FFFF0H prior to CPU activation. Note that the same result can be effected in the maximum mode through the RQ/GT line by generating the request pulse in the first or second clock cycle after RESET goes inactive.

## LOCK

The LOCK output is used in conjunction with an Intel 8289® Bus Arbiter to guarantee exclusive access of a shared system bus for the duration of an instruction. This output is software controlled and is effected by preceding the instruction requiring exclusive access with a one byte "lock" prefix (see instruction set description in Chapter 2).

When the lock prefix is decoded by the EU, the EU informs the BIU to activate the LOCK output during the next clock cycle. This signal remains active until one clock cycle after the execution of the associated instruction is concluded.

### QS1, QS0

The QS1 and QS0 (Queue Status) outputs permit external monitoring of the CPU's internal instruction queue to allow instruction set extension processing by a coprocessor. (The corresponding Intel ICE modules use these status bits during "trace" operations.) The encoding of the QS1 and QS0 bits is shown in table 4-4.

Table 4-4. Queue Status Bit Decoding

| QS1 | QS0 | Queue Status |
|---|---|---|
| 0 (low) | 0 | No Operation. During the last clock cycle, nothing was taken from the queue. |
| 0 | 1 | First Byte. The byte taken from the queue was the first byte of the instruction. |
| 1 (high) | 0 | Queue Empty. The queue has been reinitialized as a result of the execution of a transfer instruction. |
| 1 | 1 | Subsequent Byte. The byte taken from the queue was a subsequent byte of the instruction. |

The queue status is valid during the clock cycle after the indicated activity has occurred.

## External Memory Addressing

The 8086 and 8088 CPUs have a 20-bit address bus and are capable of accessing one megabyte of memory address space.

The 8086 memory address space consists of a sequence of up to one million individual bytes in which any two consecutive bytes can be accessed as a 16-bit data word. As shown in figure 4-17, the memory address space is physically divided into two banks of up to 512k bytes each.

One bank is associated with the lower half of the CPU's 16-bit data bus (data bits D7-D0), and the other bank is associated with the upper half of the data bus (data bits D15-D8). Address bits A19 through A1 are used to simultaneously address a specific byte location in both the upper and lower banks, and the A0 address bit is *not* used in memory addressing. Instead, A0 is used in memory bank selection. The lower bank, which

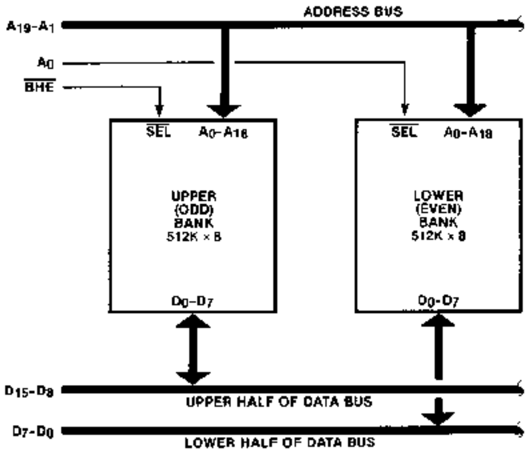


Figure 4-17. 8086 Memory Interface

contains even-address bytes, is selected when A0=0. The upper bank, containing odd address bytes (A0=1), is selected by a separate signal, Bus High Enable ($\overline{BHE}$). Table 4-5 defines the $\overline{BHE}$-A0 bank selection mechanism.

Table 4-5. Memory Bank Selection

| $\overline{BHE}$ | A0 | Byte Transferred |
|---|---|---|
| 0 (low) | 0 | Both bytes |
| 0 | 1 | Upper byte to/from odd address |
| 1 (high) | 0 | Lower byte to/from even address |
| 1 | 1 | None |

When accessing a data byte at an even address, the byte is transferred to or from the lower bank on the lower half of the data bus (D7-D0). In this case, the inactive level of the A0 address bit enables the addressed byte in the lower bank, and the inactive level of the $\overline{BHE}$ signal disables the addressed byte in the upper bank. Conversely, when performing a byte access at an odd address, the data byte is transferred to or from the upper bank on the upper half of the data bus (D15-D8). The active level of the $\overline{BHE}$ signal enables the upper bank, and the active level of the A0 address bit disables the lower bank.

As indicated in table 4-5, the 8086 can access a byte in both the upper and lower banks simultaneously as a 16-bit word. When the low-order byte of the word to be accessed is on an even address boundary (that is, when the low-order byte is in the lower bank), the word is said to be "aligned" and can be accessed in a single operation (a single bus cycle). As with the byte transfers previously described, address bits A19 through A1 address both banks, except that now $\overline{BHE}$ is active (selecting the upper bank) and A0 is inactive (selecting the lower bank) to access both bytes.

When the low-order byte of the word to be accessed is on an odd address boundary (when the low-order byte is in the upper bank), the word is "not aligned" and must be accessed in two bus cycles. During the first cycle, the low-order byte of the word is transferred to or from the upper bank as described for a byte access at an odd address (A0 and $\overline{BHE}$ active). The memory address is then incremented, which causes A0 to shift to an inactive level (selecting the lower bank), and a byte access at an even address is performed during the next bus cycle to transfer the word's high-order byte to or from the lower bank. The above sequence is initiated automatically by the 8086 whenever a word access at an odd address is performed. Also, the directing of the high- and low-order bytes of the 8086's internal word registers to the appropriate halves of the data bus is performed automatically and, except for the additional four clock cycles required to execute the second bus cycle, the entire operation is transparent to the program.

The 8088 memory address space is logically organized as a linear array of up to one million bytes. Since the 8088 uses an 8-bit-wide data bus, memory consists of a single bank. Address bit A0 is used to address memory, and a $\overline{BHE}$ signal is not provided.

Word (16-bit) operands can be located at odd- or even-address boundaries. The low-order byte of the word is stored in the lower-valued address location, and the high-order byte is stored in the next, higher-valued address location. The 8088 automatically executes two bus cycles when accessing word operands.

## I/O Interfacing

The 8086 and 8088 CPUs support both I/O mapped I/O and memory mapped I/O. I/O mapped I/O permits an I/O device to reside in a separate address space (first 64k of address space), and the standard I/O instruction set is available for device communications. Memory mapped I/O permits an I/O device to reside anywhere in memory and allows the complete CPU instruction set to be used for I/O operations.

The 8086 supports both 8-bit and 16-bit I/O devices. An 8-bit I/O device may be associated with either the upper or lower half of the data bus. (Assigning an equal number of devices to each half of the data bus distributes bus loading.) When an I/O device is assigned to the lower half of the bus (D7-D0), all I/O addresses must be even (A0 equal "0"), and when an I/O device is assigned to the upper half of the bus, all I/O addresses must be odd (A0 equal "1"). Note that since A0 always will be either a "1" or a "0" for a specific device, it cannot be used as an address input to select registers within the I/O device. When an I/O device on the upper half of the bus and an I/O device on the lower half of the bus are assigned addresses that differ only by the state of A0 (adjacent odd and even addresses), A0 and $\overline{BHE}$ both must be conditions of device selection to prevent a write operation to one device from overwriting data in the other device.

To permit data transfers to 16-bit I/O devices to be performed in a single bus cycle, the device is assigned an even address. To ensure that the I/O device is selected only for word transfers, A0 and $\overline{BHE}$ both must be conditions of device selection.

The 8088, since its data bus is eight bits wide, is designed to support 8-bit I/O devices and places no restrictions on odd or even addresses.

When the 8086 or the 8088 is operated in the minimum mode, the CPU's read and write commands ($\overline{RD}$ and $\overline{WR}$) are common for memory and I/O devices. If the memory and I/O address spaces overlap, device selection must be qualified by M/$\overline{IO}$ (8086) or IO/$\overline{M}$ (8088) to determine if the device is memory or I/O. This restriction does not apply to systems in which I/O and memory addresses do not overlap or to systems that use memory-mapped I/O exclusively. In the maximum mode, the CPU generates (through the bus controller) separate memory read/write and I/O read/write commands in place of the M/$\overline{IO}$ or IO/$\overline{M}$ signal. In a maximum mode system, an I/O device is assigned to an I/O address or to a memory address (memory mapped I/O) by connecting either the memory or I/O read/write command lines to the device's command inputs.

When the I/O and memory address spaces overlap, device selection is determined by the appropriate read/write command set.

## Interrupts

CPU interrupts can be software or hardware initiated. Software interrupts originate directly from program execution (i.e., execution of a breakpointed instruction) or indirectly through program logic (i.e., attempting to divide by zero). Hardware interrupts originate from external logic and are classified as either non-maskable or maskable. All interrupts, whether software or hardware initiated, result in the transfer of control to a new program location. A 256-entry vector table, which contains address pointers to the interrupt routines, resides in absolute locations 0 through 3FFH. Each entry in this table consists of two 16-bit address values (four bytes) that are loaded into the code segment (CS) and the instruction pointer (IP) registers as the interrupt routine address when an interrupt is accepted. Figure 4-18 illustrates the organization of the 256-entry vector table.

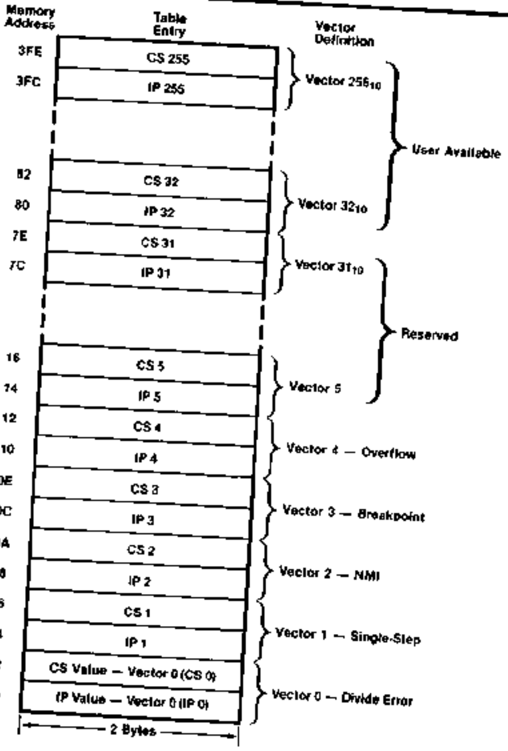


Figure 4-18. Interrupt Vector Table

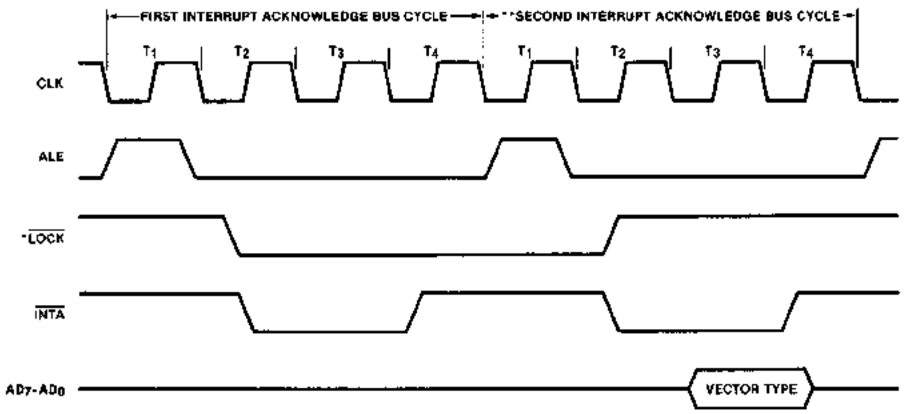
As shown in figure 4-18, the first five interrupt vectors are associated with the software-initiated interrupts and the hardware non-maskable interrupt (NMI). The next 27 interrupt vectors are reserved by Intel and should not be used if compatibility with future Intel products is to be maintained. The remaining interrupt vectors (vectors 32 thorugh 255) are available for user interrupt routines.

The non-maskable interrupt (NMI) occurs as a result of a positive transition at the CPU's NMI input pin. This input is asynchronous and, in order to ensure that it is recognized, is required to have a minimum duration of two clock cycles. NMI is typically used with power fail circuitry, error correcting memory or bus parity detection logic to allow fast response to these fault conditions. When NMI is activated, control is transferred to the interrupt service routine pointed to by vector 2 following execution of the current instruction. When a non-maskable interrupt is acknowledged, the current contents of the flags register are pushed onto the stack (the stack pointer is decremented by two), the interrupt enable and trap bits in the flags register are cleared (disabling maskable and single-step interrupts), and the vector 2 CS and IP address pointers are loaded into the CS and IP registers as the interrupt service routine address.

The CPU provides a single interrupt request input (INTR) that can be software masked by clearing the interrupt enable bit in the flags register through the execution of a CLI instruction. The INTR input is level triggered and is synchronized internally to the positive transition of the CLK signal. In order to be accepted before the next instruction, INTR must be active during the clock period preceding the end of the current instruction (and the interrupt enable bit must be set).

As shown in figure 4-19, when a maskable interrupt is acknowledged, the CPU executes two interrupt acknowledge bus cycles.

During the first bus cycle, the CPU floats the address/data bus and activates the $\overline{\text{INTA}}$ (Interrupt Acknowledge) command output during states $T_2$ through $T_4$. In the minimum mode, the CPU will not recognize a hold request from another bus master until the full interrupt acknowledge sequence is completed. In the maximum mode, the CPU activates the $\overline{\text{LOCK}}$ output from state $T_2$ of the first bus cycle until state $T_2$ of the second bus cycle to signal all 8289 Bus Arbiters in the system that the bus should not be accessed by any other processor. During the second bus cycle, the CPU again activates its $\overline{\text{INTA}}$ command output. In response to the


FIRST INTERRUPT ACKNOWLEDGE BUS CYCLE
**SECOND INTERRUPT ACKNOWLEDGE BUS CYCLE
T1
T2
T3
T4
CLK
ALE
*LOCK
INTA
AD7-AD0
VECTOR TYPE

*MAXIMUM MODE ONLY
**SEVERAL (3 TYPICAL) IDLE CLOCK STATES OCCUR BETWEEN THE FIRST AND SECOND INTERRUPT ACKNOWLEDGE BUS CYCLES IN THE 8086 CPU (DURING THIS INTERVAL THE BUS IS DRIVEN). INTERRUPT ACKNOWLEDGE BUS CYCLES OCCUR BACK-TO-BACK IN THE 8088 CPU.

Figure 4-19. Interrupt Acknowledge Sequence

second $\overline{\text{INTA}}$, the external interrupt system (e.g., an Intel® 8259A Programmable Interrupt Controller) places a byte on the data bus that identifies the source of the interrupt (the vector number or vector "type"). This byte is read by the CPU and then multiplied by four with the resultant value used as a pointer into the interrupt vector table. Before calling the corresponding interrupt routine, the CPU saves the machine status by pushing the current contents of the flags register onto the stack. The CPU then clears the interrupt enable and trap bits in the flags register to prevent subsequent maskable and single-step interrupts, and establishes the interrupt routine return linkage by pushing the current CS and IP register contents onto the stack before loading the new CS and IP register values from the vector table.

The four classes of interrupts are prioritized with software-initiated interrupts having the highest priority and with maskable and single-step interrupts sharing the lowest priority (see section 2.6). Since the CPU disables maskable and single-step interrupts when acknowledging any interrupt, if recognition of maskable interrupts or single-step operation is required as part of the interrupt routine, the routine first must set these bits.

The processing times for the various classes of interrupts are given in table 4-6. (These times also are included with the 8086/8088 instruction times cited in section 2.7.)

**Table 4-6. Interrupt Processing Time**

| Interrupt Class | Processing Time |
|---|---|
| External Maskable Interrupt (INTR) | 61 clocks |
| Non-Maskable Interrupt (NMI) | 50 clocks |
| INT (with vector) | 51 clocks |
| INT Type 3 | 52 clocks |
| INTO | 53 clocks |
| Single Step | 50 clocks |

Note that the times shown in table 4-6 represent only the time required to process the interrupt request after it has been recognized. To determine interrupt latency (the time interval between the posting of the interrupt request and the execution of "useful" instructions within the interrupt routine), additional time must be included for the completion on an instruction being executed when the interrupt is posted (interrupts are generally processed only at instruction boundaries), for saving the contents of any additional registers prior to interrupt processing (interrupts automatically save only CS, IP and Flags) and for any wait states that may be incurred during interrupt processing.

## Machine Instruction Encoding and Decoding

Writing a MOV instruction in ASM-86 in the form:

```
MOV destination,source
```

will cause the assembler to generate 1 of 28 possible forms of the MOV machine instruction. A programmer rarely needs to know the details of machine instruction formats or encoding. An exception may occur during debugging when it may be necessary to monitor instructions fetched on the bus, read unformatted memory dumps, etc. This section provides the information necessary to translate or decode an 8086 or 8088 machine instruction.

To pack instructions into memory as densely as possible, the 8086 and 8088 CPUs utilize an efficient coding technique. Machine instructions vary from one to six bytes in length. One-byte instructions, which generally operate on single registers or flags, are simple to identify. The keys to decoding longer instructions are in the first two bytes. The format of these bytes can vary, but most instructions follow the format shown in figure 4-20.

The first six bits of a multibyte instruction generally contain an opcode that identifies the basic instruction type: ADD, XOR, etc. The following bit, called the D field, generally specifies the "direction" of the operation: 1 = the REG field in the second byte identifies the destination operand, 0 = the REG field identifies the source operand. The W field distinguishes between byte and word operations: 0 = byte, 1 = word.

One of three additional single-bit fields, S, V or Z, appears in some instruction formats. S is used in conjunction with W to indicate sign extension of immediate fields in arithmetic instructions. V distinguishes between single- and variable-bit shifts and rotates. Z is used as a compare bit with the zero flag in conditional repeat and loop instructions. All single-bit field settings are summarized in table 4-7.

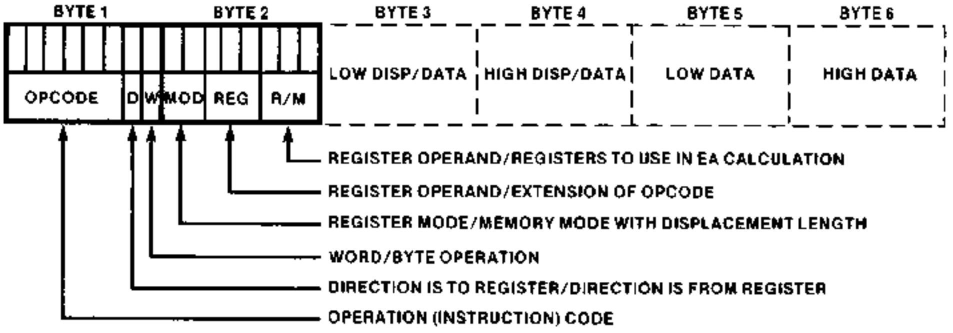

BYTE 1
BYTE 2
BYTE 3
BYTE 4
BYTE 5
BYTE 6
OPCODE
D
W
MOD
REG
R/M
LOW DISP/DATA
HIGH DISP/DATA
LOW DATA
HIGH DATA
REGISTER OPERAND/REGISTERS TO USE IN EA CALCULATION
REGISTER OPERAND/EXTENSION OF OPCODE
REGISTER MODE/MEMORY MODE WITH DISPLACEMENT LENGTH
WORD/BYTE OPERATION
DIRECTION IS TO REGISTER/DIRECTION IS FROM REGISTER
OPERATION (INSTRUCTION) CODE

Figure 4-20. Typical 8086/8088 Machine Instruction Format

Table 4-7. Single-Bit Field Encoding

| Field | Value | Function |
|---|---|---|
| S | 0 | No sign extension |
|  | 1 | Sign extend 8-bit immediate data to 16 bits if W=1 |
| W | 0 | Instruction operates on byte data |
|  | 1 | Instruction operates on word data |
| D | 0 | Instruction source is specified in REG field |
|  | 1 | Instruction destination is specified in REG field |
| V | 0 | Shift/rotate count is one |
|  | 1 | Shift/rotate count is specified in CL register |
| Z | 0 | Repeat/loop while zero flag is clear |
|  | 1 | Repeat/loop while zero flag is set |

The second byte of the instruction usually identifies the instruction's operands. The MOD (mode) field indicates whether one of the operands is in memory or whether both operands are registers (see table 4-8). The REG (register) field identifies a register that is one of the instruction operands (see table 4-9). In a number of instructions, chiefly the immediate-to-memory variety, REG is used as an extension of the opcode to identify the type of operation. The encoding of the R/M (register/memory) field (see table 4-10) depends on how the mode field is set. If MOD = 11 (register-to-register mode), then R/M identifies the second register operand. If MOD selects memory mode, then R/M indicates how the effective address of the memory operand is to be calculated. Effective address calculation is covered in detail in section 2.8.

Bytes 3 through 6 of an instruction are optional fields that usually contain the displacement value of a memory operand and/or the actual value of an immediate constant operand.

Table 4-8. MOD (Mode) Field Encoding

| CODE | EXPLANATION |
|---|---|
| 00 | Memory Mode, no displacement follows* |
| 01 | Memory Mode, 8-bit displacement follows |
| 10 | Memory Mode, 16-bit displacement follows |
| 11 | Register Mode (no displacement) |

*Except when R/M = 110, then 16-bit displacement follows

Table 4-9. REG (Register) Field Encoding

| REG | W=0 | W=1 |
|---|---|---|
| 000 | AL | AX |
| 001 | CL | CX |
| 010 | DL | DX |
| 011 | BL | BX |
| 100 | AH | SP |
| 101 | CH | BP |
| 110 | DH | SI |
| 111 | BH | DI |

There may be one or two displacement bytes; the language translators generate one byte whenever possible. The MOD field indicates how many displacement bytes are present. Following Intel convention, if the displacement is two bytes, the most-significant byte is stored second in the instruction. If the displacement is only a single byte, the 8086 or 8088 automatically sign-extends this quantity to 16-bits before using the information in further address calculations. Immediate values always follow any displacement values that may be present. The second byte of a two-byte immediate value is the most significant.

Table 4-12 lists the instruction encodings for all 8086/8088 instructions. This table can be used to predict the machine encoding of any ASM-86 instruction. Table 4-13 lists the 8086/8088 machine instructions in order by the binary value of their first byte. This table can be used to decode any machine instruction from its binary representation. Table 4-11 is a key to the abbreviations used in tables 4-12 and 4-13. Table 4-14 is a more compact instruction decoding guide.

Table 4-10. R/M (Register/Memory) Field Encoding

| MOD=11 | | | EFFECTIVE ADDRESS CALCULATION | | | |
|---|---|---|---|---|---|---|
| R/M | W=0 | W=1 | R/M | MOD=00 | MOD=01 | MOD=10 |
| 000 | AL | AX | 000 | (BX)+(SI) | (BX)+(SI)+D8 | (BX)+(SI)+D16 |
| 001 | CL | CX | 001 | (BX)+(DI) | (BX)+(DI)+D8 | (BX)+(DI)+D16 |
| 010 | DL | DX | 010 | (BP)+(SI) | (BP)+(SI)+D8 | (BP)+(SI)+D16 |
| 011 | BL | BX | 011 | (BP)+(DI) | (BP)+(DI)+D8 | (BP)+(DI)+D16 |
| 100 | AH | SP | 100 | (SI) | (SI)+D8 | (SI)+D16 |
| 101 | CH | BP | 101 | (DI) | (DI)+D8 | (DI)+D16 |
| 110 | DH | SI | 110 | DIRECT ADDRESS | (BP)+D8 | (BP)+D16 |
| 111 | BH | DI | 111 | (BX) | (BX)+D8 | (BX)+D16 |

Table 4-11. Key to Machine Instruction Encoding and Decoding

| IDENTIFIER | EXPLANATION |
|---|---|
| MOD | Mode field; described in this chapter. |
| REG | Register field; described in this chapter. |
| R/M | Register/Memory field; described in this chapter. |
| SR | Segment register code: 00=ES, 01=CS, 10=SS, 11=DS. |
| W, S, D, V, Z | Single-bit instruction fields; described in this chapter. |
| DATA-8 | 8-bit immediate constant. |
| DATA-SX | 8-bit immediate value that is automatically sign-extended to 16-bits before use. |
| DATA-LO | Low-order byte of 16-bit immediate constant. |
| DATA-HI | High-order byte of 16-bit immediate constant. |
| (DISP-LO) | Low-order byte of optional 8- or 16-bit unsigned displacement; MOD indicates if present. |
| (DISP-HI) | High-order byte of optional 16-bit unsigned displacement; MOD indicates if present. |
| IP-LO | Low-order byte of new IP value. |
| IP-HI | High-order byte of new IP value |
| CS-LO | Low-order byte of new CS value. |
| CS-HI | High-order byte of new CS value. |
| IP-INC8 | 8-bit signed increment to instruction pointer. |
| IP-INC-LO | Low-order byte of signed 16-bit instruction pointer increment. |
| IP-INC-HI | High-order byte of signed 16-bit instruction pointer increment. |
| ADDR-LO | Low-order byte of direct address (offset) of memory operand; EA not calculated. |
| ADDR-HI | High-order byte of direct address (offset) of memory operand; EA not calculated. |
| —— | Bits may contain any value. |
| XXX | First 3 bits of ESC opcode. |
| YYY | Second 3 bits of ESC opcode. |
| REG8 | 8-bit general register operand. |
| REG16 | 16-bit general register operand. |
| MEM8 | 8-bit memory operand (any addressing mode). |
| MEM16 | 16-bit memory operand (any addressing mode). |
| IMMED8 | 8-bit immediate operand. |
| IMMED16 | 16-bit immediate operand. |
| SEGREG | Segment register operand. |
| DEST-STR8 | Byte string addressed by DI. |

Table 4-11. Key to Machine Instruction Encoding and Decoding (Cont'd.)

| IDENTIFIER | EXPLANATION |
|---|---|
| SRC-STR8 | Byte string addressed by SI. |
| DEST-STR16 | Word string addressed by DI. |
| SRC-STR16 | Word string addressed by SI. |
| SHORT-LABEL | Label within ±127 bytes of instruction. |
| NEAR-PROC | Procedure in current code segment. |
| FAR-PROC | Procedure in another code segment. |
| NEAR-LABEL | Label in current code segment but farther than −128 to +127 bytes from instruction. |
| FAR-LABEL | Label in another code segment. |
| SOURCE-TABLE | XLAT translation table addressed by BX. |
| OPCODE | ESC opcode operand. |
| SOURCE | ESC register or memory operand. |

Table 4-12. 8086 Instruction Encoding

**DATA TRANSFER**

**MOV = Move:**

|  | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 |
|---|---|---|---|---|---|---|
| Register/memory to/from register | 1 0 0 0 1 0 d w | mod reg r/m | (DISP-LO) | (DISP-HI) |  |  |
| Immediate to register/memory | 1 1 0 0 0 1 1 w | mod 0 0 0 r/m | (DISP-LO) | (DISP-HI) | data | data if w = 1 |
| Immediate to register | 1 0 1 1 w reg | data | data if w = 1 |  |  |  |
| Memory to accumulator | 1 0 1 0 0 0 0 w | addr-lo | addr-hi |  |  |  |
| Accumulator to memory | 1 0 1 0 0 0 1 w | addr-lo | addr-hi |  |  |  |
| Register/memory to segment register | 1 0 0 0 1 1 1 0 | mod 0 SR r/m | (DISP-LO) | (DISP-HI) |  |  |
| Segment register to register/memory | 1 0 0 0 1 1 0 0 | mod 0 SR r/m | (DISP-LO) | (DISP-HI) |  |  |

**PUSH = Push:**

|  |  |  |  |  |
|---|---|---|---|---|
| Register/memory | 1 1 1 1 1 1 1 1 | mod 1 1 0 r/m | (DISP-LO) | (DISP-HI) |
| Register | 0 1 0 1 0 reg |  |  |  |
| Segment register | 0 0 0 reg 1 1 0 |  |  |  |

**POP = Pop:**

|  |  |  |  |  |
|---|---|---|---|---|
| Register/memory | 1 0 0 0 1 1 1 1 | mod 0 0 0 r/m | (DISP-LO) | (DISP-HI) |
| Register | 0 1 0 1 1 reg |  |  |  |
| Segment register | 0 0 0 reg 1 1 1 |  |  |  |

## Table 4-12. 8086 Instruction Encoding (Cont'd.)

**DATA TRANSFER (Cont'd.)**

| **XCHG = Exchange:** | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 |
|---|---|---|---|---|---|---|
| Register/memory with register | 1 0 0 0 0 1 1 w | mod reg r/m | (DISP-LO) | (DISP-HI) | | |
| Register with accumulator | 1 0 0 1 0 reg | | | | | |
| **IN = Input from:** | | | | | | |
| Fixed port | 1 1 1 0 0 1 0 w | DATA-8 | | | | |
| Variable port | 1 1 1 0 1 1 0 w | | | | | |
| **OUT = Output to:** | | | | | | |
| Fixed port | 1 1 1 0 0 1 1 w | DATA-8 | | | | |
| Variable port | 1 1 1 0 1 1 1 w | | | | | |
| **XLAT** = Translate byte to AL | 1 1 0 1 0 1 1 1 | | | | | |
| **LEA** = Load EA to register | 1 0 0 0 1 1 0 1 | mod reg r/m | (DISP-LO) | (DISP-HI) | | |
| **LDS** = Load pointer to DS | 1 1 0 0 0 1 0 1 | mod reg r/m | (DISP-LO) | (DISP-HI) | | |
| **LES** = Load pointer to ES | 1 1 0 0 0 1 0 0 | mod reg r/m | (DISP-LO) | (DISP-HI) | | |
| **LAHF** = Load AH with flags | 1 0 0 1 1 1 1 1 | | | | | |
| **SAHF** = Store AH into flags | 1 0 0 1 1 1 1 0 | | | | | |
| **PUSHF** = Push flags | 1 0 0 1 1 1 0 0 | | | | | |
| **POPF** = Pop flags | 1 0 0 1 1 1 0 1 | | | | | |

**ARITHMETIC**

| **ADD = Add:** | | | | | | |
|---|---|---|---|---|---|---|
| Reg/memory with register to either | 0 0 0 0 0 0 d w | mod reg r/m | (DISP-LO) | (DISP-HI) | | |
| Immediate to register/memory | 1 0 0 0 0 0 s w | mod 0 0 0 r/m | (DISP-LO) | (DISP-HI) | data | data if s: w=01 |
| Immediate to accumulator | 0 0 0 0 0 1 0 w | data | data if w=1 | | | |
| **ADC = Add with carry:** | | | | | | |
| Reg/memory with register to either | 0 0 0 1 0 0 d w | mod reg r/m | (DISP-LO) | (DISP-HI) | | |
| Immediate to register/memory | 1 0 0 0 0 0 s w | mod 0 1 0 r/m | (DISP-LO) | (DISP-HI) | data | data if s: w=01 |
| Immediate to accumulator | 0 0 0 1 0 1 0 w | data | data if w=1 | | | |
| **INC = Increment:** | | | | | | |
| Register/memory | 1 1 1 1 1 1 1 w | mod 0 0 0 r/m | (DISP-LO) | (DISP-HI) | | |
| Register | 0 1 0 0 0 reg | | | | | |
| **AAA** = ASCII adjust for add | 0 0 1 1 0 1 1 1 | | | | | |
| **DAA** = Decimal adjust for add | 0 0 1 0 0 1 1 1 | | | | | |

## Table 4-12. 8086 Instruction Encoding (Cont'd.)

**ARITHMETIC (Cont'd.)**

| **SUB** = Subtract: | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 |
|---|---|---|---|---|---|---|
| Reg/memory and register to either | 0 0 1 0 1 0 d w | mod reg r/m | (DISP-LO) | (DISP-HI) | | |
| Immediate from register/memory | 1 0 0 0 0 0 s w | mod 1 0 1 r/m | (DISP-LO) | (DISP-HI) | data | data if s: w=01 |
| Immediate from accumulator | 0 0 1 0 1 1 0 w | data | data if w=1 | | | |
| **SBB** = Subtract with borrow: | | | | | | |
| Reg/memory and register to either | 0 0 0 1 1 0 d w | mod reg r/m | (DISP-LO) | (DISP-HI) | | |
| Immediate from register/memory | 1 0 0 0 0 0 s w | mod 0 1 1 r/m | (DISP-LO) | (DISP-HI) | data | data if s: w=01 |
| Immediate from accumulator | 0 0 0 1 1 1 0 w | data | data if w=1 | | | |
| **DEC** Decrement: | | | | | | |
| Register/memory | 1 1 1 1 1 1 1 w | mod 0 0 1 r/m | (DISP-LO) | (DISP-HI) | | |
| Register | 0 1 0 0 1 reg | | | | | |
| **NEG** Change sign | 1 1 1 1 0 1 1 w | mod 0 1 1 r/m | (DISP-LO) | (DISP-HI) | | |
| **CMP** = Compare: | | | | | | |
| Register/memory and register | 0 0 1 1 1 0 d w | mod reg r/m | (DISP-LO) | (DISP-HI) | | |
| Immediate with register/memory | 1 0 0 0 0 0 s w | mod 1 1 1 r/m | (DISP-LO) | (DISP-HI) | data | data if s. w=1 |
| Immediate with accumulator | 0 0 1 1 1 1 0 w | data | | | | |
| **AAS** ASCII adjust for subtract | 0 0 1 1 1 1 1 1 | | | | | |
| **DAS** Decimal adjust for subtract | 0 0 1 0 1 1 1 1 | | | | | |
| **MUL** Multiply (unsigned) | 1 1 1 1 0 1 1 w | mod 1 0 0 r/m | (DISP-LO) | (DISP-HI) | | |
| **IMUL** Integer multiply (signed) | 1 1 1 1 0 1 1 w | mod 1 0 1 r/m | (DISP-LO) | (DISP-HI) | | |
| **AAM** ASCII adjust for multiply | 1 1 0 1 0 1 0 0 | 0 0 0 0 1 0 1 0 | (DISP-LO) | (DISP-HI) | | |
| **DIV** Divide (unsigned) | 1 1 1 1 0 1 1 w | mod 1 1 0 r/m | (DISP-LO) | (DISP-HI) | | |
| **IDIV** Integer divide (signed) | 1 1 1 1 0 1 1 w | mod 1 1 1 r/m | (DISP-LO) | (DISP-HI) | | |
| **AAD** ASCII adjust for divide | 1 1 0 1 0 1 0 1 | 0 0 0 0 1 0 1 0 | (DISP-LO) | (DISP-HI) | | |
| **CBW** Convert byte to word | 1 0 0 1 1 0 0 0 | | | | | |
| **CWD** Convert word to double word | 1 0 0 1 1 0 0 1 | | | | | |

LOGIC

| | | | | |
|---|---|---|---|---|
| **NOT** Invert | 1 1 1 1 0 1 1 w | mod 0 1 0 r/m | (DISP-LO) | (DISP-HI) |
| **SHL/SAL** Shift logical/arithmetic left | 1 1 0 1 0 0 v w | mod 1 0 0 r/m | (DISP-LO) | (DISP-HI) |
| **SHR** Shift logical right | 1 1 0 1 0 0 v w | mod 1 0 1 r/m | (DISP-LO) | (DISP-HI) |
| **SAR** Shift arithmetic right | 1 1 0 1 0 0 v w | mod 1 1 1 r/m | (DISP-LO) | (DISP-HI) |
| **ROL** Rotate left | 1 1 0 1 0 0 v w | mod 0 0 0 r/m | (DISP-LO) | (DISP-HI) |

## Table 4-12. 8086 Instruction Encoding (Cont'd.)

| LOGIC (Cont'd.) | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 |
|---|---|---|---|---|---|---|
| **ROR** Rotate right | 1 1 0 1 0 0 v w | mod 0 0 1 r/m | (DISP-LO) | (DISP-HI) | | |
| **RCL** Rotate through carry flag left | 1 1 0 1 0 0 v w | mod 0 1 0 r/m | (DISP-LO) | (DISP-HI) | | |
| **RCR** Rotate through carry right | 1 1 0 1 0 0 v w | mod 0 1 1 r/m | (DISP-LO) | (DISP-HI) | | |
| **AND = And:** | | | | | | |
| Reg/memory with register to either | 0 0 1 0 0 0 d w | mod reg r/m | (DISP-LO) | (DISP-HI) | | |
| Immediate to register/memory | 1 0 0 0 0 0 0 w | mod 1 0 0 r/m | (DISP-LO) | (DISP-HI) | data | data if w=1 |
| Immediate to accumulator | 0 0 1 0 0 1 0 w | data | data if w=1 | | | |
| **TEST = And function to flags no result:** | | | | | | |
| Register/memory and register | 0 0 0 1 0 0 d w | mod reg r/m | (DISP-LO) | (DISP-HI) | | |
| Immediate data and register/memory | 1 1 1 1 0 1 1 w | mod 0 0 0 r/m | (DISP-LO) | (DISP-HI) | data | data if w=1 |
| Immediate data and accumulator | 1 0 1 0 1 0 0 w | data | | | | |
| **OR = Or:** | | | | | | |
| Reg/memory and register to either | 0 0 0 0 1 0 d w | mod reg r/m | (DISP-LO) | (DISP-HI) | | |
| Immediate to register/memory | 1 0 0 0 0 0 0 w | mod 0 0 1 r/m | (DISP-LO) | (DISP-HI) | data | data if w=1 |
| Immediate to accumulator | 0 0 0 0 1 1 0 w | data | data if w=1 | | | |
| **XOR = Exclusive or:** | | | | | | |
| Reg/memory and register to either | 0 0 1 1 0 0 d w | mod reg r/m | (DISP-LO) | (DISP-HI) | | |
| Immediate to register/memory | 0 0 1 1 0 1 0 w | data | (DISP-LO) | (DISP-HI) | data | data if w=1 |
| Immediate to accumulator | 0 0 1 1 0 1 0 w | data | data if w=1 | | | |
| **STRING MANIPULATION** | | | | | | |
| **REP** = Repeat | 1 1 1 1 0 0 1 z | | | | | |
| **MOVS** = Move byte/word | 1 0 1 0 0 1 0 w | | | | | |
| **CMPS** = Compare byte/word | 1 0 1 0 0 1 1 w | | | | | |
| **SCAS** = Scan byte/word | 1 0 1 0 1 1 1 w | | | | | |
| **LODS** = Load byte/wd to AL/AX | 1 0 1 0 1 1 0 w | | | | | |
| **STDS** = Stor byte/wd from AL/A | 1 0 1 0 1 0 1 w | | | | | |

## Table 4-12. 8086 Instruction Encoding (Cont'd.)

**CONTROL TRANSFER**

| **CALL = Call:** | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 |
|---|---|---|---|---|---|---|
| Direct within segment | 1 1 1 0 1 0 0 0 | IP-INC-LO | IP-INC-HI | | | |
| Indirect within segment | 1 1 1 1 1 1 1 1 | mod 0 1 0 r/m | (DISP-LO) | (DISP-HI) | | |
| Direct intersegment | 1 0 0 1 1 0 1 0 | IP-lo | IP-hi | | | |
| | | CS-lo | CS-hi | | | |
| Indirect intersegment | 1 1 1 1 1 1 1 1 | mod 0 1 1 r/m | (DISP-LO) | (DISP-HI) | | |

| **JMP = Unconditional Jump:** | | | | |
|---|---|---|---|---|
| Direct within segment | 1 1 1 0 1 0 0 1 | IP-INC-LO | IP-INC-HI | |
| Direct within segment-short | 1 1 1 0 1 0 1 1 | IP-INC8 | | |
| Indirect within segment | 1 1 1 1 1 1 1 1 | mod 1 0 0 r/m | (DISP-LO) | (DISP-HI) |
| Direct intersegment | 1 1 1 0 1 0 1 0 | IP-lo | IP-hi | |
| | | CS-lo | CS-hi | |
| Indirect intersegment | 1 1 1 1 1 1 1 1 | mod 1 0 1 r/m | (DISP-LO) | (DISP-HI) |

| **RET = Return from CALL:** | | | |
|---|---|---|---|
| Within segment | 1 1 0 0 0 0 1 1 | | |
| Within seg adding immed to SP | 1 1 0 0 0 0 1 0 | data-lo | data-hi |
| Intersegment | 1 1 0 0 1 0 1 1 | | |
| Intersegment adding immediate to SP | 1 1 0 0 1 0 1 0 | data-lo | data-hi |
| **JE/JZ** = Jump on equal/zero | 0 1 1 1 0 1 0 0 | IP-INC8 | |
| **JL/JNGE** = Jump on less/not greater or equal | 0 1 1 1 1 1 0 0 | IP-INC8 | |
| **JLE/JNG** = Jump on less or equal/not greater | 0 1 1 1 1 1 1 0 | IP-INC8 | |
| **JB/JNAE** = Jump on below/not above or equal | 0 1 1 1 0 0 1 0 | IP-INC8 | |
| **JBE/JNA** = Jump on below or equal/not above | 0 1 1 1 0 1 1 0 | IP-INC8 | |
| **JP/JPE** = Jump on parity/parity even | 0 1 1 1 1 0 1 0 | IP-INC8 | |
| **JO** = Jump on overflow | 0 1 1 1 0 0 0 0 | IP-INC8 | |
| **JS** = Jump on sign | 0 1 1 1 1 0 0 0 | IP-INC8 | |
| **JNE/JNZ** = Jump on not equal/not zero | 0 1 1 1 0 1 0 1 | IP-INC8 | |
| **JNL/JGE** = Jump on not less/greater or equal | 0 1 1 1 1 1 0 1 | IP-INC8 | |
| **JNLE/JG** = Jump on not less or equal/greater | 0 1 1 1 1 1 1 1 | IP-INC8 | |
| **JNB/JAE** = Jump on not below/above or equal | 0 1 1 1 0 0 1 1 | IP-INC8 | |
| **JNBE/JA** = Jump on not below or equal/above | 0 1 1 1 0 1 1 1 | IP-INC8 | |
| **JNP/JPO** = Jump on not par/par odd | 0 1 1 1 1 0 1 1 | IP-INC8 | |
| **JNO** = Jump on not overflow | 0 1 1 1 0 0 0 1 | IP-INC8 | |

Table 4-12. 8086 Instruction Encoding (Cont'd.)

**CONTROL TRANSFER (Cont'd.)**

| **RET = Return from CALL.** | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 |
|---|---|---|---|---|---|---|
| **JNS** = Jump on not sign | 0 1 1 1 1 0 0 1 | IP-INC8 | | | | |
| **LOOP** = Loop CX times | 1 1 1 0 0 0 1 0 | IP-INC8 | | | | |
| **LOOPZ/LOOPE** = Loop while zero/equal | 1 1 1 0 0 0 0 1 | IP-INC8 | | | | |
| **LOOPNZ/LOOPNE** = Loop while not zero/equal | 1 1 1 0 0 0 0 0 | IP-INC8 | | | | |
| **JCXZ** = Jump on CX zero | 1 1 1 0 0 0 1 1 | IP-INC8 | | | | |

**INT = Interrupt:**

| | | |
|---|---|---|
| Type specified | 1 1 0 0 1 1 0 1 | DATA-8 |
| Type 3 | 1 1 0 0 1 1 0 0 | |
| **INTO** = Interrupt on overflow | 1 1 0 0 1 1 1 0 | |
| **IRET** = Interrupt return | 1 1 0 0 1 1 1 1 | |

**PROCESSOR CONTROL**

| | | | | |
|---|---|---|---|---|
| **CLC** = Clear carry | 1 1 1 1 1 0 0 0 | | | |
| **CMC** = Complement carry | 1 1 1 1 0 1 0 1 | | | |
| **STC** = Set carry | 1 1 1 1 1 0 0 1 | | | |
| **CLD** = Clear direction | 1 1 1 1 1 1 0 0 | | | |
| **STD** = Set direction | 1 1 1 1 1 1 0 1 | | | |
| **CLI** = Clear interrupt | 1 1 1 1 1 0 1 0 | | | |
| **STI** = Set interrupt | 1 1 1 1 1 0 1 1 | | | |
| **HLT** = Halt | 1 1 1 1 0 1 0 0 | | | |
| **WAIT** = Wait | 1 0 0 1 1 0 1 1 | | | |
| **ESC** = Escape (to external device) | 1 1 0 1 1 x x x | m o d y y y r / m | (DISP-LO) | (DISP-HI) |
| **LOCK** = Bus lock prefix | 1 1 1 1 0 0 0 0 | | | |
| **SEGMENT** = Override prefix | 0 0 1 reg 1 1 0 | | | |

Table 4-13. Machine Instruction Decoding Guide

| 1ST BYTE | | 2ND BYTE | BYTES 3, 4, 5, 6 | ASM-86 INSTRUCTION FORMAT | |
|---|---|---|---|---|---|
| HEX | BINARY | | | | |
| 00 | 0000 0000 | MOD REG R/M | (DISP-LO),(DISP-HI) | ADD | REG8/MEM8,REG8 |
| 01 | 0000 0001 | MOD REG R/M | (DISP-LO),(DISP-HI) | ADD | REG16/MEM16,REG16 |
| 02 | 0000 0010 | MOD REG R/M | (DISP-LO),(DISP-HI) | ADD | REG8,REG8/MEM8 |
| 03 | 0000 0011 | MOD REG R/M | (DISP-LO),(DISP-HI) | ADD | REG16,REG16/MEM16 |
| 04 | 0000 0100 | DATA-8 | | ADD | AL,IMMED8 |
| 05 | 0000 0101 | DATA-LO | DATA-HI | ADD | AX,IMMED16 |
| 06 | 0000 0110 | | | PUSH | ES |
| 07 | 0000 0111 | | | POP | ES |

Table 4-13. Machine Instruction Decoding Guide (Cont'd.)

| 1ST BYTE HEX | 1ST BYTE BINARY | 2ND BYTE | BYTES 3,4,5,6 | ASM-86 INSTRUCTION FORMAT | |
|---|---|---|---|---|---|
| 08 | 0000 1000 | MOD REG R/M | (DISP-LO),(DISP-HI) | OR | REG8/MEM8,REG8 |
| 09 | 0000 1001 | MOD REG R/M | (DISP-LO),(DISP-HI) | OR | REG16/MEM16,REG16 |
| 0A | 0000 1010 | MOD REG R/M | (DISP-LO),(DISP-HI) | OR | REG8,REG8/MEM8 |
| 0B | 0000 1011 | MOD REG R/M | (DISP-LO),(DISP-HI) | OR | REG16,REG16/MEM16 |
| 0C | 0000 1100 | DATA-8 | | OR | AL,IMMED8 |
| 0D | 0000 1101 | DATA-LO | DATA-HI | OR | AX,IMMED16 |
| 0E | 0000 1110 | | | PUSH | CS |
| 0F | 0000 1111 | | | (not used) | |
| 10 | 0001 0000 | MOD REG R/M | (DISP-LO),(DISP-HI) | ADC | REG8/MEM8,REG8 |
| 11 | 0001 0001 | MOD REG R/M | (DISP-LO),(DISP-HI) | ADC | REG16/MEM16,REG16 |
| 12 | 0001 0010 | MOD REG R/M | (DISP-LO),(DISP-HI) | ADC | REG8,REG8/MEM8 |
| 13 | 0001 0011 | MOD REG R/M | (DISP-LO),(DISP-HI) | ADC | REG16,REG16/MEM16 |
| 14 | 0001 0100 | DATA-8 | | ADC | AL,IMMED8 |
| 15 | 0001 0101 | DATA-LO | DATA-HI | ADC | AX,IMMED16 |
| 16 | 0001 0110 | | | PUSH | SS |
| 17 | 0001 0111 | | | POP | SS |
| 18 | 0001 1000 | MOD REG R/M | (DISP-LO),(DISP-HI) | SBB | REG8/MEM8,REG8 |
| 19 | 0001 1001 | MOD REG R/M | (DISP-LO),(DISP-HI) | SBB | REG16/MEM16,REG16 |
| 1A | 0001 1010 | MOD REG R/M | (DISP-LO),(DISP-HI) | SBB | REG8,REG8/MEM8 |
| 1B | 0001 1011 | MOD REG R/M | (DISP-LO),(DISP-HI) | SBB | REG16,REG16/MEM16 |
| 1C | 0001 1100 | DATA-8 | | SBB | AL,IMMED8 |
| 1D | 0001 1101 | DATA-LO | DATA-HI | SBB | AX,IMMED16 |
| 1E | 0001 1110 | | | PUSH | DS |
| 1F | 0001 1111 | | | POP | DS |
| 20 | 0010 0000 | MOD REG R/M | (DISP-LO),(DISP-HI) | AND | REG8/MEM8,REG8 |
| 21 | 0010 0001 | MOD REG R/M | (DISP-LO),(DISP-HI) | AND | REG16/MEM16,REG16 |
| 22 | 0010 0010 | MOD REG R/M | (DISP-LO),(DISP-HI) | AND | REG8,REG8/MEM8 |
| 23 | 0010 0011 | MOD REG R/M | (DISP-LO),(DISP-HI) | AND | REG16,REG16/MEM16 |
| 24 | 0010 0100 | DATA-8 | | AND | AL,IMMED8 |
| 25 | 0010 0101 | DATA-LO | DATA-HI | AND | AX,IMMED16 |
| 26 | 0010 0110 | | | ES: | (segment override prefix) |
| 27 | 0010 0111 | | | DAA | |
| 28 | 0010 1000 | MOD REG R/M | (DISP-LO),(DISP-HI) | SUB | REG8/MEM8,REG8 |
| 29 | 0010 1001 | MOD REG R/M | (DISP-LO),(DISP-HI) | SUB | REG16/MEM16,REG16 |
| 2A | 0010 1010 | MOD REG R/M | (DISP-LO),(DISP-HI) | SUB | REG8,REG8/MEM8 |
| 2B | 0010 1011 | MOD REG R/M | (DISP-LO,(DISP-HI) | SUB | REG16,REG16/MEM16 |
| 2C | 0010 1100 | DATA-8 | | SUB | AL,IMMED8 |
| 2D | 0010 1101 | DATA-LO | DATA-HI | SUB | AX,IMMED16 |
| 2E | 0010 1110 | | | CS: | (segment override prefix) |
| 2F | 0010 1111 | | | DAS | |
| 30 | 0011 0000 | MOD REG R/M | (DISP-LO),(DISP-HI) | XOR | REG8/MEM8,REG8 |
| 31 | 0011 0001 | MOD REG R/M | (DISP-LO),(DISP-HI) | XOR | REG16/MEM16,REG16 |
| 32 | 0011 0010 | MOD REG R/M | (DISP-LO),(DISP-HI) | XOR | REG8,REG8/MEM8 |
| 33 | 0011 0011 | MOD REG R/M | (DISP-LO),(DISP-HI) | XOR | REG16,REG16/MEM16 |
| 34 | 0011 0100 | DATA-8 | | XOR | AL,IMMED8 |
| 35 | 0011 0101 | DATA-LO | DATA-HI | XOR | AX,IMMED16 |
| 36 | 0011 0110 | | | SS: | (segment override prefix) |

Table 4-13. Machine Instruction Decoding Guide (Cont'd.)

| 1ST BYTE | | 2ND BYTE | BYTES 3,4,5,6 | ASM-86 INSTRUCTION FORMAT | |
|---|---|---|---|---|---|
| HEX | BINARY | | | | |
| 37 | 0011 0110 | | | AAA | |
| 38 | 0011 1000 | MOD REG R/M | (DISP-LO),(DISP-HI) | CMP | REG8/MEM8,REG8 |
| 39 | 0011 1001 | MOD REG R/M | (DISP-LO),(DISP-HI) | CMP | REG16/MEM16,REG16 |
| 3A | 0011 1010 | MOD REG R/M | (DISP-LO),(DISP-HI) | CMP | REG8,REG8/MEM8 |
| 3B | 0011 1011 | MOD REG R/M | (DISP-LO),(DISP-HI) | CMP | REG16,REG16/MEM16 |
| 3C | 0011 1100 | DATA-8 | | CMP | AL,IMMED8 |
| 3D | 0011 1101 | DATA-LO | DATA-HI | CMP | AX,IMMED16 |
| 3E | 0011 1110 | | | DS: | (segment override prefix) |
| 3F | 0011 1111 | | | AAS | |
| 40 | 0100 0000 | | | INC | AX |
| 41 | 0100 0001 | | | INC | CX |
| 42 | 0100 0010 | | | INC | DX |
| 43 | 0100 0011 | | | INC | BX |
| 44 | 0100 0100 | | | INC | SP |
| 45 | 0100 0101 | | | INC | BP |
| 46 | 0100 0110 | | | INC | SI |
| 47 | 0100 0111 | | | INC | DI |
| 48 | 0100 1000 | | | DEC | AX |
| 49 | 0100 1001 | | | DEC | CX |
| 4A | 0100 1010 | | | DEC | DX |
| 4B | 0100 1011 | | | DEC | BX |
| 4C | 0100 1100 | | | DEC | SP |
| 4D | 0100 1101 | | | DEC | BP |
| 4E | 0100 1110 | | | DEC | SI |
| 4F | 0100 1111 | | | DEC | DI |
| 50 | 0101 0000 | | | PUSH | AX |
| 51 | 0101 0001 | | | PUSH | CX |
| 52 | 0101 0010 | | | PUSH | DX |
| 53 | 0101 0011 | | | PUSH | BX |
| 54 | 0101 0100 | | | PUSH | SP |
| 55 | 0101 0101 | | | PUSH | BP |
| 56 | 0101 0110 | | | PUSH | SI |
| 57 | 0101 0111 | | | PUSH | DI |
| 58 | 0101 1000 | | | POP | AX |
| 59 | 0101 1001 | | | POP | CX |
| 5A | 0101 1010 | | | POP | DX |
| 5B | 0101 1011 | | | POP | BX |
| 5C | 0101 1100 | | | POP | SP |
| 5D | 0101 1101 | | | POP | BP |
| 5E | 0101 1110 | | | POP | SI |
| 5F | 0101 1111 | | | POP | DI |
| 60 | 0110 0000 | | | (not used) | |
| 61 | 0110 0001 | | | (not used) | |
| 62 | 0110 0010 | | | (not used) | |
| 63 | 0110 0011 | | | (not used) | |
| 64 | 0110 0100 | | | (not used) | |
| 65 | 0110 0101 | | | (not used) | |
| 66 | 0110 0110 | | | (not used) | |
| 67 | 0110 0111 | | | (not used) | |

Table 4-13. Machine Instruction Decoding Guide (Cont'd.)

| 1ST BYTE | | 2ND BYTE | BYTES 3,4,5,6 | ASM-86 INSTRUCTION FORMAT | |
|---|---|---|---|---|---|
| HEX | BINARY | | | | |
| 68 | 0110 1000 | | | (not used) | |
| 69 | 0110 1001 | | | (not used) | |
| 6A | 0110 1010 | | | (not used) | |
| 6B | 0110 1011 | | | (not used) | |
| 6C | 0110 1100 | | | (not used) | |
| 6D | 0110 1101 | | | (not used) | |
| 6E | 0110 1110 | | | (not used) | |
| 6F | 0110 1111 | | | (not used) | |
| 70 | 0111 0000 | IP-INC8 | | JO | SHORT-LABEL |
| 71 | 0111 0001 | IP-INC8 | | JNO | SHORT-LABEL |
| 72 | 0111 0010 | IP-INC8 | | JB/JNAE/ JC | SHORT-LABEL |
| 73 | 0111 0011 | IP-INC8 | | JNB/JAE/ JNC | SHORT-LABEL |
| 74 | 0111 0100 | IP-INC8 | | JE/JZ | SHORT-LABEL |
| 75 | 0111 0101 | IP-INC8 | | JNE/JNZ | SHORT-LABEL |
| 76 | 0111 0110 | IP-INC8 | | JBE/JNA | SHORT-LABEL |
| 77 | 0111 0111 | IP-INC8 | | JNBE/JA | SHORT-LABEL |
| 78 | 0111 1000 | IP-INC8 | | JS | SHORT-LABEL |
| 79 | 0111 1001 | IP-INC8 | | JNS | SHORT-LABEL |
| 7A | 0111 1010 | IP-INC8 | | JP/JPE | SHORT-LABEL |
| 7B | 0111 1011 | IP-INC8 | | JNP/JPO | SHORT-LABEL |
| 7C | 0111 1100 | IP-INC8 | | JL/JNGE | SHORT-LABEL |
| 7D | 0111 1101 | IP-INC8 | | JNL/JGE | SHORT-LABEL |
| 7E | 0111 1110 | IP-INC8 | | JLE/JNG | SHORT-LABEL |
| 7F | 0111 1111 | IP-INC8 | | JNLE/JG | SHORT-LABEL |
| 80 | 1000 0000 | MOD 000 R/M | (DISP-LO),(DISP-HI), DATA-8 | ADD | REG8/MEM8,IMMED8 |
| 80 | 1000 0000 | MOD 001 R/M | (DISP-LO),(DISP-HI), DATA-8 | OR | REG8/MEM8,IMMED8 |
| 80 | 1000 0000 | MOD 010 R/M | (DISP-LO),(DISP-HI), DATA-8 | ADC | REG8/MEM8,IMMED8 |
| 80 | 1000 0000 | MOD 011 R/M | (DISP-LO),(DISP-HI), DATA-8 | SBB | REG8/MEM8,IMMED8 |
| 80 | 1000 0000 | MOD 100 R/M | (DISP-LO),(DISP-HI), DATA-8 | AND | REG8/MEM8,IMMED8 |
| 80 | 1000 0000 | MOD 101 R/M | (DISP-LO),(DISP-HI), DATA-8 | SUB | REG8/MEM8,IMMED8 |
| 80 | 1000 0000 | MOD 110 R/M | (DISP-LO),(DISP-HI), DATA-8 | XOR | REG8/MEM8,IMMED8 |
| 80 | 1000 0000 | MOD 111 R/M | (DISP-LO),(DISP-HI), DATA-8 | CMP | REG8/MEM8,IMMED8 |
| 81 | 1000 0001 | MOD 000 R/M | (DISP-LO),(DISP-HI), DATA-LO,DATA-HI | ADD | REG16/MEM16,IMMED16 |
| 81 | 1000 0001 | MOD 001 R/M | (DISP-LO),(DISP-HI), DATA-LO,DATA-HI | OR | REG16/MEM16,IMMED16 |
| 81 | 1000 0001 | MOD 010 R/M | (DISP-LO),(DISP-HI), DATA-LO,DATA-HI | ADC | REG16/MEM16,IMMED16 |
| 81 | 1000 0001 | MOD 011 R/M | (DISP-LO),(DISP-HI), DATA-LO,DATA-HI | SBB | REG16/MEM16,IMMED16 |

Table 4-13. Machine Instruction Decoding Guide (Cont'd.)

| 1ST BYTE | | 2ND BYTE | BYTES 3,4,5,6 | ASM-86 INSTRUCTION FORMAT | |
|---|---|---|---|---|---|
| HEX | BINARY | | | | |
| 81 | 1000 0001 | MOD 100 R/M | (DISP-LO),(DISP-HI), DATA-LO,DATA-HI | AND | REG16/MEM16,IMMED16 |
| 81 | 1000 0001 | MOD 101 R/M | (DISP-LO),(DISP-HI), DATA-LO,DATA-HI | SUB | REG16/MEM16,IMMED16 |
| 81 | 1000 0001 | MOD 110 R/M | (DISP-LO),(DISP-HI), DATA-LO,DATA-HI | XOR | REG16/MEM16,IMMED16 |
| 81 | 1000 0001 | MOD 111 R/M | (DISP-LO),(DISP-HI), DATA-LO,DATA-HI | CMP | REG16/MEM16,IMMED16 |
| 82 | 1000 0010 | MOD 000 R/M | (DISP-LO),(DISP-HI), DATA-8 | ADD | REG8/MEM8,IMMED8 |
| 82 | 1000 0010 | MOD 001 R/M | | (not used) | |
| 82 | 1000 0010 | MOD 010 R/M | (DISP-LO),(DISP-HI), DATA-8 | ADC | REG8/MEM8,IMMED8 |
| 82 | 1000 0010 | MOD 011 R/M | (DISP-LO),(DISP-HI), DATA-8 | SBB | REG8/MEM8,IMMED8 |
| 82 | 1000 0010 | MOD 100 R/M | | (not used) | |
| 82 | 1000 0010 | MOD 101 R/M | (DISP-LO),(DISP-HI), DATA-8 | SUB | REG8/MEM8,IMMED8 |
| 82 | 1000 0010 | MOD 110 R/M | | (not used) | |
| 82 | 1000 0010 | MOD 111 R/M | (DISP-LO),(DISP-HI), DATA-8 | CMP | REG8/MEM8,IMMED8 |
| 83 | 1000 0011 | MOD 000 R/M | (DISP-LO),(DISP-HI), DATA-SX | ADD | REG16/MEM16, IMMED8 |
| 83 | 1000 0011 | MOD 001 R/M | | (not used) | |
| 83 | 1000 0011 | MOD 010 R/M | (DISP-LO), (DISP-HI), DATA-SX | ADC | REG16/MEM16,IMMED8 |
| 83 | 1000 0011 | MOD 011 R/M | (DISP-LO),(DISP-HI), DATA-SX | SBB | REG16/MEM16,IMMED8 |
| 83 | 1000 0011 | MOD 100 R/M | | (not used) | |
| 83 | 1000 0011 | MOD 101 R/M | (DISP-LO),(DISP-HI), DATA-SX | SUB | REG16/MEM16,IMMED8 |
| 83 | 1000 0011 | MOD 110 R/M | | (not used) | |
| 83 | 1000 0011 | MOD 111 R/M | (DISP-LO),(DISP-HI), DATA-SX | CMP | REG16/MEM16,IMMED8 |
| 84 | 1000 0100 | MOD REG R/M | (DISP-LO),(DISP-HI) | TEST | REG8/MEM8,REG8 |
| 85 | 1000 0101 | MOD REG R/M | (DISP-LO),(DISP-HI) | TEST | REG16/MEM16,REG16 |
| 86 | 1000 0110 | MOD REG R/M | (DISP-LO),(DISP-HI) | XCHG | REG8,REG8/MEM8 |
| 87 | 1000 0111 | MOD REG R/M | (DISP-LO),(DISP-HI) | XCHG | REG16,REG16/MEM16 |
| 88 | 1000 1000 | MOD REG R/M | (DISP-LO),(DISP-HI) | MOV | REG8/MEM8,REG8 |
| 89 | 1000 1001 | MOD REG R/M | (DISP-LO),(DISP-HI) | MOV | REG16/MEM16/REG16 |
| 8A | 1000 1010 | MOD REG R/M | (DISP-LO),(DISP-HI) | MOV | REG8,REG8/MEM8 |
| 8B | 1000 1011 | MOD REG R/M | (DISP-LO),(DISP-HI) | MOV | REG16,REG16/MEM16 |
| 8C | 1000 1100 | MOD 0SR R/M | (DISP-LO),(DISP-HI) | MOV | REG16/MEM16,SEGREG |
| 8C | 1000 1100 | MOD 1-- R/M | | (not used) | |
| 8D | 1000 1101 | MOD REG R/M | (DISP-LO),(DISP-HI) | LEA | REG16,MEM16 |
| 8E | 1000 1110 | MOD 0SR R/M | (DISP-LO),(DISP-HI) | MOV | SEGREG,REG16/MEM16 |
| 8E | 1000 1110 | MOD 1-- R/M | | (not used) | |
| 8F | 1000 1111 | MOD 000 R/M | (DISP-LO),(DISP-HI) | POP | REG16/MEM16 |
| 8F | 1000 1111 | MOD 001 R/M | | (not used) | |
| 8F | 1000 1111 | MOD 010 R/M | | (not used) | |

Table 4-13. Machine Instruction Decoding Guide (Cont'd.)

| 1ST BYTE | | 2ND BYTE | BYTES 3,4,5,6 | ASM-86 INSTRUCTION FORMAT | |
|---|---|---|---|---|---|
| HEX | BINARY | | | | |
| 8F | 1000 1111 | MOD 011 R/M | | (not used) | |
| 8F | 1000 1111 | MOD 100 R/M | | (not used) | |
| 8F | 1000 1111 | MOD 101 R/M | | (not used) | |
| 8F | 1000 1111 | MOD 110 R/M | | (not used) | |
| 8F | 1000 1111 | MOD 111 R/M | | (not used) | |
| 90 | 1001 0000 | | | NOP | (exchange AX,AX) |
| 91 | 1001 0001 | | | XCHG | AX,CX |
| 92 | 1001 0010 | | | XCHG | AX,DX |
| 93 | 1001 0011 | | | XCHG | AX,BX |
| 94 | 1001 0100 | | | XCHG | AX,SP |
| 95 | 1001 0101 | | | XCHG | AX,BP |
| 96 | 1001 0110 | | | XCHG | AX,SI |
| 97 | 1001 0111 | | | XCHG | AX,DI |
| 98 | 1001 1000 | | | CBW | |
| 99 | 1001 1001 | | | CWD | |
| 9A | 1001 1010 | DISP-LO | DISP-HI,SEG-LO, SEG-HI | CALL | FAR_PROC |
| 9B | 1001 1011 | | | WAIT | |
| 9C | 1001 1100 | | | PUSHF | |
| 9D | 1001 1101 | | | POPF | |
| 9E | 1001 1110 | | | SAHF | |
| 9F | 1001 1111 | | | LAHF | |
| A0 | 1010 0000 | ADDR-LO | ADDR-HI | MOV | AL,MEM8 |
| A1 | 1010 0001 | ADDR-LO | ADDR-HI | MOV | AX,MEM16 |
| A2 | 1010 0010 | ADDR-LO | ADDR-HI | MOV | MEM8,AL |
| A3 | 1010 0011 | ADDR-LO | ADDR-HI | MOV | MEM16,AL |
| A4 | 1010 0100 | | | MOVS | DEST-STR8,SRC-STR8 |
| A5 | 1010 0101 | | | MOVS | DEST-STR16,SRC-STR16 |
| A6 | 1010 0110 | | | CMPS | DEST-STR8,SRC-STR8 |
| A7 | 1010 0111 | | | CMPS | DEST-STR16,SRC-STR16 |
| A8 | 1010 1000 | DATA-8 | | TEST | AL,IMMED8 |
| A9 | 1010 1001 | DATA-LO | DATA-HI | TEST | AX,IMMED16 |
| AA | 1010 1010 | | | STOS | DEST-STR8 |
| AB | 1010 1011 | | | STOS | DEST-STR16 |
| AC | 1010 1100 | | | LODS | SRC-STR8 |
| AD | 1010 1101 | | | LODS | SRC-STR16 |
| AE | 1010 1110 | | | SCAS | DEST-STR8 |
| AF | 1010 1111 | | | SCAS | DEST-STR16 |
| B0 | 1011 0000 | DATA-8 | | MOV | AL,IMMED8 |
| B1 | 1011 0001 | DATA-8 | | MOV | CL,IMMED8 |
| B2 | 1011 0010 | DATA-8 | | MOV | DL,IMMED8 |
| B3 | 1011 1011 | DATA-8 | | MOV | BL,IMMED8 |
| B4 | 1011 0100 | DATA-8 | | MOV | AH,IMMED8 |
| B5 | 1011 0101 | DATA-8 | | MOV | CH,IMMED8 |
| B6 | 1011 0110 | DATA-8 | | MOV | DH,IMMED8 |
| B7 | 1011 0111 | DATA-8 | | MOV | BH,IMMED8 |
| B8 | 1011 1000 | DATA-LO | DATA-HI | MOV | AX,IMMED16 |
| B9 | 1011 1001 | DATA-LO | DATA-HI | MOV | CX,IMMED16 |
| BA | 1011 1010 | DATA-LO | DATA-HI | MOV | DX,IMMED16 |
| BB | 1011 1011 | DATA-LO | DATA-HI | MOV | BX,IMMED16 |

Table 4-13. Machine Instruction Decoding Guide (Cont'd.)

| 1ST BYTE | | 2ND BYTE | BYTES 3,4,5,6 | ASM-86 INSTRUCTION FORMAT | |
|---|---|---|---|---|---|
| HEX | BINARY | | | | |
| BC | 1011 1100 | DATA-LO | DATA-HI | MOV | SP,IMMED16 |
| BD | 1011 1101 | DATA-LO | DATA-HI | MOV | BP,IMMED16 |
| BE | 1011 1110 | DATA-LO | DATA-HI | MOV | SI,IMMED16 |
| BF | 1011 1111 | DATA-LO | DATA-HI | MOV | DI,IMMED16 |
| C0 | 1100 0000 | | | (not used) | |
| C1 | 1100 0001 | | | (not used) | |
| C2 | 1100 0010 | DATA-LO | DATA-HI | RET | IMMED16 (intraseg) |
| C3 | 1100 0011 | | | RET | (intrasegment) |
| C4 | 1100 0100 | MOD REG R/M | (DISP-LO),(DISP-HI) | LES | REG16,MEM16 |
| C5 | 1100 0101 | MOD REG R/M | (DISP-LO),(DISP-HI) | LDS | REG16,MEM16 |
| C6 | 1100 0110 | MOD 000 R/M | (DISP-LO),(DISP-HI), DATA-8 | MOV | MEM8,IMMED8 |
| C6 | 1100 0110 | MOD 001 R/M | | (not used) | |
| C6 | 1100 0110 | MOD 010 R/M | | (not used) | |
| C6 | 1100 0110 | MOD 011 R/M | | (not used) | |
| C6 | 1100 0110 | MOD 100 R/M | | (not used) | |
| C6 | 1100 0110 | MOD 101 R/M | | (not used) | |
| C6 | 1100 0110 | MOD 110 R/M | | (not used) | |
| C6 | 1100 0110 | MOD 111 R/M | | (not used) | |
| C7 | 1100 0111 | MOD 000 R/M | (DISP-LO),(DISP-HI), DATA-LO,DATA-HI | MOV | MEM16,IMMED16 |
| C7 | 1100 0111 | MOD 001 R/M | | (not used) | |
| C7 | 1100 0111 | MOD 010 R/M | | (not used) | |
| C7 | 1100 0111 | MOD 011 R/M | | (not used) | |
| C7 | 1100 0111 | MOD 100 R/M | | (not used) | |
| C7 | 1100 0111 | MOD 101 R/M | | (not used) | |
| C7 | 1100 0111 | MOD 110 R/M | | (not used) | |
| C7 | 1100 0111 | MOD 111 R/M | | (not used | |
| C8 | 1100 1000 | | | (not used) | |
| C9 | 1100 1001 | | | (not used) | |
| CA | 1100 1010 | DATA-LO | DATA-HI | RET | IMMED16 (intersegment) |
| CB | 1100 1011 | | | RET | (intersegment) |
| CC | 1100 1100 | | | INT | 3 |
| CD | 1100 1101 | DATA-8 | | INT | IMMED8 |
| CE | 1100 1110 | | | INTO | |
| CF | 1100 1111 | | | IRET | |
| D0 | 1101 0000 | MOD 000 R/M | (DISP-LO),(DISP-HI) | ROL | REG8/MEM8,1 |
| D0 | 1101 0000 | MOD 001 R/M | (DISP-LO),(DISP-HI) | ROR | REG8/MEM8,1 |
| D0 | 1101 0000 | MOD 010 R/M | (DISP-LO),(DISP-HI) | RCL | REG8/MEM8,1 |
| D0 | 1101 0000 | MOD 011 R/M | (DISP-LO),(DISP-HI) | RCR | REG8/MEM8,1 |
| D0 | 1101 0000 | MOD 100 R/M | (DISP-LO),(DISP-HI) | SAL/SHL | REG8/MEM8,1 |
| D0 | 1101 0000 | MOD 101 R/M | (DISP-LO),(DISP-HI) | SHR | REG8/MEM8,1 |
| D0 | 1101 0000 | MOD 110 R/M | | (not used) | |
| D0 | 1101 0000 | MOD 111 R/M | (DISP-LO),(DISP-HI) | SAR | REG8/MEM8,1 |
| D1 | 1101 0001 | MOD 000 R/M | (DISP-LO),(DISP-HI) | ROL | REG16/MEM16,1 |
| D1 | 1101 0001 | MOD 001 R/M | (DISP-LO),(DISP-HI) | ROR | REG16/MEM16,1 |
| D1 | 1101 0001 | MOD 010 R/M | (DISP-LO),(DISP-HI) | RCL | REG16/MEM16,1 |
| D1 | 1101 0001 | MOD 011 R/M | (DISP-LO),(DISP-HI) | RCR | REG16/MEM16,1 |
| D1 | 1101 0001 | MOD 100 R/M | (DISP-LO),(DISP-HI) | SAL/SHL | REG16/MEM16,1 |

Table 4-13. Machine Instruction Decoding Guide (Cont'd.)

| 1ST BYTE | | 2ND BYTE | BYTES 3,4,5,6 | ASM-86 INSTRUCTION FORMAT | |
|---|---|---|---|---|---|
| HEX | BINARY | | | | |
| D1 | 1101 0001 | MOD 101 R/M | (DISP-LO),(DISP-HI) | SHR | REG16/MEM16,1 |
| D1 | 1101 0001 | MOD 110 R/M | | (not used) | |
| D1 | 1101 0001 | MOD 111 R/M | (DISP-LO),(DISP-HI) | SAR | REG16/MEM16,1 |
| D2 | 1101 0010 | MOD 000 R/M | (DISP-LO),(DISP-HI) | ROL | REG8/MEM8,CL |
| D2 | 1101 0010 | MOD 001 R/M | (DISP-LO),(DISP-HI) | ROR | REG8/MEM8,CL |
| D2 | 1101 0010 | MOD 010 R/M | (DISP-LO),(DISP-HI) | RCL | REG8/MEM8,CL |
| D2 | 1101 0010 | MOD 011 R/M | (DISP-LO),(DISP-HI) | RCR | REG8/MEM8,CL |
| D2 | 1101 0010 | MOD 100 R/M | (DISP-LO),(DISP-HI) | SAL/SHL | REG8/MEM8,CL |
| D2 | 1101 0010 | MOD 101 R/M | (DISP-LO),(DISP-HI) | SHR | REG8/MEM8,CL |
| D2 | 1101 0010 | MOD 110 R/M | | (not used) | |
| D2 | 1101 0010 | MOD 111 R/M | (DISP-LO),(DISP-HI) | SAR | REG8/MEM8,CL |
| D3 | 1101 0011 | MOD 000 R/M | (DISP-LO),(DISP-HI) | ROL | REG16/MEM16,CL |
| D3 | 1101 0011 | MOD 001 R/M | (DISP-LO),(DISP-HI) | ROR | REG16/MEM16,CL |
| D3 | 1101 0011 | MOD 010 R/M | (DISP-LO),(DISP-HI) | RCL | REG16/MEM16,CL |
| D3 | 1101 0011 | MOD 011 R/M | (DISP-LO),(DISP-HI) | RCR | REG16/MEM16,CL |
| D3 | 1101 0011 | MOD 100 R/M | (DISP-LO),(DISP-HI) | SAL/SHL | REG16/MEM16,CL |
| D3 | 1101 0011 | MOD 101 R/M | (DISP-LO),(DISP-HI) | SHR | REG16/MEM16,CL |
| D3 | 1101 0011 | MOD 110 R/M | | (not used) | |
| D3 | 1101 0011 | MOD 111 R/M | (DISP-LO),(DISP-HI) | SAR | REG16/MEM16,CL |
| D4 | 1101 0100 | 00001010 | | AAM | |
| D5 | 1101 0101 | 00001010 | | AAD | |
| D6 | 1101 0110 | | | (not used) | |
| D7 | 1101 0111 | | | XLAT | SOURCE-TABLE |
| D8 | 1101 1000 | MOD 000 R/M | | | |
| | 1XXX | MOD YYY R/M | (DISP-LO), (DISP-HI) | ESC | OPCODE,SOURCE |
| DF | 1101 1111 | MOD 111 R/M | | | |
| E0 | 1110 0000 | IP-INC-8 | | LOOPNE/ LOOPNZ | SHORT-LABEL |
| E1 | 1110 0001 | IP-INC-8 | | LOOPE/ LOOPZ | SHORT-LABEL |
| E2 | 1110 0010 | IP-INC-8 | | LOOP | SHORT-LABEL |
| E3 | 1110 0011 | IP-INC-8 | | JCXZ | SHORT-LABEL |
| E4 | 1110 0100 | DATA-8 | | IN | AL,IMMED8 |
| E5 | 1110 0101 | DATA-8 | | IN | AX,IMMED8 |
| E6 | 1110 0110 | DATA-8 | | OUT | AL,IMMED8 |
| E7 | 1110 0111 | DATA-8 | | OUT | AX,IMMED8 |
| E8 | 1110 1000 | IP-INC-LO | IP-INC-HI | CALL | NEAR-PROC |
| E9 | 1110 1001 | IP-INC-LO | IP-INC-HI | JMP | NEAR-LABEL |
| EA | 1110 1010 | IP-LO | IP-HI,CS-LO,CS-HI | JMP | FAR-LABEL |
| EB | 1110 1011 | IP-INC8 | | JMP | SHORT-LABEL |
| EC | 1110 1100 | | | IN | AL,DX |
| ED | 1110 1101 | | | IN | AX,DX |
| EE | 1110 1110 | | | OUT | AL,DX |
| EF | 1110 1111 | | | OUT | AX,DX |
| F0 | 1111 0000 | | | LOCK | (prefix) |
| F1 | 1111 0001 | | | (not used) | |
| F2 | 1111 0010 | | | REPNE/REPNZ | |
| F3 | 1111 0011 | | | REP/REPE/REPZ | |
| F4 | 1111 0100 | | | HLT | |
| F5 | 1111 0101 | | | CMC | |

Table 4-13. Machine Instruction Decoding Guide (Cont'd.)

| 1ST BYTE | | 2ND BYTE | BYTES 3,4,5,6 | ASM-86 INSTRUCTION FORMAT | |
|---|---|---|---|---|---|
| HEX | BINARY | | | | |
| F6 | 1111 0110 | MOD 000 R/M | (DISP-LO),(DISP-HI), DATA-8 | TEST | REG8/MEM8,IMMED8 |
| F6 | 1111 0110 | MOD 001 R/M | | (not used) | |
| F6 | 1111 0110 | MOD 010 R/M | (DISP-LO),(DISP-HI) | NOT | REG8/MEM8 |
| F6 | 1111 0110 | MOD 011 R/M | (DISP-LO),(DISP-HI) | NEG | REG8/MEM8 |
| F6 | 1111 0110 | MOD 100 R/M | (DISP-LO),(DISP-HI) | MUL | REG8/MEM8 |
| F6 | 1111 0110 | MOD 101 R/M | (DISP-LO),(DISP-HI) | IMUL | REG8/MEM8 |
| F6 | 1111 0110 | MOD 110 R/M | (DISP-LO),(DISP-HI) | DIV | REG8/MEM8 |
| F6 | 1111 0110 | MOD 111 R/M | (DISP-LO),(DISP-HI) | IDIV | REG8/MEM8 |
| F7 | 1111 0111 | MOD 000 R/M | (DISP-LO),(DISP-HI), DATA-LO,DATA-HI | TEST | REG16/MEM16,IMMED16 |
| F7 | 1111 0111 | MOD 001 R/M | | (not used) | |
| F7 | 1111 0111 | MOD 010 R/M | (DISP-LO),(DISP-HI) | NOT | REG16/MEM16 |
| F7 | 1111 0111 | MOD 011 R/M | (DISP-LO),(DISP-HI) | NEG | REG16/MEM16 |
| F7 | 1111 0111 | MOD 100 R/M | (DISP-LO),(DISP-HI) | MUL | REG16/MEM16 |
| F7 | 1111 0111 | MOD 101 R/M | (DISP-LO),(DISP-HI) | IMUL | REG16/MEM16 |
| F7 | 1111 0111 | MOD 110 R/M | (DISP-LO),(DISP-HI) | DIV | REG16/MEM16 |
| F7 | 1111 0111 | MOD 111 R/M | (DISP-LO),(DISP-HI) | IDIV | REG16/MEM16 |
| F8 | 1111 1000 | | | CLC | |
| F9 | 1111 1001 | | | STC | |
| FA | 1111 1010 | | | CLI | |
| FB | 1111 1011 | | | STI | |
| FC | 1111 1100 | | | CLD | |
| FD | 1111 1101 | | | STD | |
| FE | 1111 1110 | MOD 000 R/M | (DISP-LO),(DISP-HI) | INC | REG8/MEM8 |
| FE | 1111 1110 | MOD 001 R/M | (DISP-LO),(DISP-HI) | DEC | REG8/MEM8 |
| FE | 1111 1110 | MOD 010 R/M | | (not used) | |
| FE | 1111 1110 | MOD 011 R/M | | (not used) | |
| FE | 1111 1110 | MOD 100 R/M | | (not used) | |
| FE | 1111 1110 | MOD 101 R/M | | (not used) | |
| FE | 1111 1110 | MOD 110 R/M | | (not used) | |
| FE | 1111 1110 | MOD 111 R/M | | (not used) | |
| FF | 1111 1111 | MOD 000 R/M | (DISP-LO),(DISP-HI) | INC | MEM16 |
| FF | 1111 1111 | MOD 001 R/M | (DISP-LO),(DISP-HI) | DEC | MEM16 |
| FF | 1111 1111 | MOD 010 R/M | (DISP-LO),(DISP-HI) | CALL | REG16/MEM16 (intra) |
| FF | 1111 1111 | MOD 011 R/M | (DISP-LO),(DISP-HI) | CALL | MEM16 (intersegment) |
| FF | 1111 1111 | MOD 100 R/M | (DISP-LO),(DISP-HI) | JMP | REG16/MEM16 (intra) |
| FF | 1111 1111 | MOD 101 R/M | (DISP-LO),(DISP-HI) | JMP | MEM16 (intersegment) |
| FF | 1111 1111 | MOD 110 R/M | (DISP-LO),(DISP-HI) | PUSH | MEM16 |
| FF | 1111 1111 | MOD 111 R/M | | (not used) | |

Table 4-14. Machine Instruction Encoding Matrix

| Hi \ Lo | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | ADD b.f.r/m | ADD w.f.r/m | ADD b.t.r/m | ADD w.t.r/m | ADD b.ia | ADD w.ia | PUSH ES | POP ES | OR b.f.r/m | OR w.f.r/m | OR b.t.r/m | OR w.t.r/m | OR b.i | OR w.i | PUSH CS | |
| 1 | ADC b.f.r/m | ADC w.f.r/m | ADC b.t.r/m | ADC w.t.r/m | ADC b.i | ADC w.i | PUSH SS | POP SS | SBB b.f.r/m | SBB w.f.r/m | SBB b.t.r/m | SBB w.t.r/m | SBB b.i | SBB w.i | PUSH DS | POP DS |
| 2 | AND b.f.r/m | AND w.f.r/m | AND b.t.r/m | AND w.t.r/m | AND b.i | AND w.i | SEG =ES | DAA | SUB b.f.r/m | SUB w.f.r/m | SUB b.t.r/m | SUB w.t.r/m | SUB b.i | SUB w.i | SEG =CS | DAS |
| 3 | XOR b.f.r/m | XOR w.f.r/m | XOR b.t.r/m | XOR w.t.r/m | XOR b.i | XOR w.i | SEG =SS | AAA | CMP b.f.r/m | CMP w.f.r/m | CMP b.t.r/m | CMP w.t.r/m | CMP b.i | CMP w.i | SEG =DS | AAS |
| 4 | INC AX | INC CX | INC DX | INC BX | INC SP | INC BP | INC SI | INC DI | DEC AX | DEC CX | DEC DX | DEC BX | DEC SP | DEC BP | DEC SI | DEC DI |
| 5 | PUSH AX | PUSH CX | PUSH DX | PUSH BX | PUSH SP | PUSH BP | PUSH SI | PUSH DI | POP AX | POP CX | POP DX | POP BX | POP SP | POP BP | POP SI | POP DI |
| 6 | | | | | | | | | | | | | | | | |
| 7 | JO | JNO | JB/ JNAE | JNB/ JAE | JE/ JZ | JNE/ JNZ | JBE/ JNA | JNBE/ JA | JS | JNS | JP/ JPE | JNP/ JPO | JL/ JNGE | JNL/ JGE | JLE/ JNG | JNLE/ JG |
| 8 | Immed b.r/m | Immed w.r/m | Immed b.r/m | Immed is.r/m | TEST b.r/m | TEST w.r/m | XCHG b.r/m | XCHG w.r/m | MOV b.f.r/m | MOV w.f.r/m | MOV b.t.r/m | MOV w.t.r/m | MOV sr.f.r/m | LEA | MOV sr.t.r/m | POP r/m |
| 9 | XCHG AX | XCHG CX | XCHG DX | XCHG BX | XCHG SP | XCHG BP | XCHG SI | XCHG DI | CBW | CWD | CALL l.d | WAIT | PUSHF | POPF | SAHF | LAHF |
| A | MOV m → AL | MOV m → AX | MOV AL → m | MOV AX → m | MOVS | MOVS | CMPS | CMPS | TEST b.i.a | TEST w.i.a | STOS | STOS | LODS | LODS | SCAS | SCAS |
| B | MOV i → AL | MOV i → CL | MOV i → DL | MOV i → BL | MOV i → AH | MOV i → CH | MOV i → DH | MOV i → BH | MOV i → AX | MOV i → CX | MOV i → DX | MOV i → BX | MOV i → SP | MOV i → BP | MOV i → SI | MOV i → DI |
| C | | | RET, (i+SP) | RET | LES | LDS | MOV b.i.r/m | MOV w.i.r/m | | | RET, l.(i+SP) | RET l | INT Type 3 | INT (Any) | INTO | IRET |
| D | Shift b | Shift w | Shift b.v | Shift w.v | AAM | AAD | | XLAT | ESC 0 | ESC 1 | ESC 2 | ESC 3 | ESC 4 | ESC 5 | ESC 6 | ESC 7 |
| E | LOOPNZ/ LOOPNE | LOOPZ/ LOOPE | LOOP | JCXZ | IN b | IN w | OUT b | OUT w | CALL d | JMP d | JMP l.d | JMP si.d | IN v.b | IN v.w | OUT v.b | OUT v.w |
| F | LOCK | | REP | REP z | HLT | CMC | Grp 1 b.r/m | Grp 1 w.r/m | CLC | STC | CLI | STI | CLD | STD | Grp 2 b.r/m | Grp 2 w.r/m |

where

| mod ☐ r/m | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|
| Immed | ADD | OR | ADC | SBB | AND | SUB | XOR | CMP |
| Shift | ROL | ROR | RCL | RCR | SHL/SAL | SHR | — | SAR |
| Grp 1 | TEST | — | NOT | NEG | MUL | IMUL | DIV | IDIV |
| Grp 2 | INC | DEC | CALL id | CALL l.id | JMP id | JMP l.id | PUSH | — |

| | |
|---|---|
| b = byte operation | m = memory |
| d = direct | r/m = EA is second byte |
| f = from CPU reg | si = short intrasegment |
| i = immediate | sr = segment register |
| ia = immed. to accum. | t = to CPU reg |
| id = indirect | v = variable |
| is = immed. byte, sign ext. | w = word operation |
| l = long ie. intersegment | z = zero |

## 8086 Instruction Sequence

Figure 4-22 illustrates the internal operation and bus activity that occur as an 8086 CPU executes a sequence of instructions. This figure presents the signals and timing relationships that are important in understanding 8086 operation. The following discussion is intended to help in the interpretation of the figure.

Figure 4-22 shows the repeated execution of an instruction loop. This loop is defined in both machine code and assembly language by figure 4-21. A loop was chosen both to demonstrate the effects of a program jump on the queue and to make the instruction sequence easy to follow. The program sequence shown was selected for several reasons. First, consisting of seven instructions and 16 bytes, the sequence is typical of the tight loops found in many application programs. Second, this particular sequence contains several short, fast-executing instructions that demonstrate both the effect of the queue on CPU performance and the interaction between the execution unit (EU) fetching code from the queue and the bus interface unit (BIU) filling the queue and performing the requested bus cycles. Last, for the purpose of this discussion, code, stack, and memory data references were arranged to be aligned on even word boundaries.

| ASSEMBLY LANGUAGE | MACHINE CODE |
|---|---|
| MOV AX, 0F802H | B802F8 |
| PUSH AX | 50 |
| MOV CX, BX | 8BCB |
| MOV DX, CX | 8BD1 |
| ADD AX, [SI] | 0304 |
| ADD SI, 8086H | 81C68680 |
| JMP $ −14 | EBF0 |

**Figure 4-21. Instruction Loop Sequence**

Figure 4-22 can be more easily interpreted by keeping the following guidelines in mind.

- The queue status lines (QS0, QS1) are the key indicators of EU activity.
- Status lines $\overline{S2}$ through $\overline{S0}$ are the main indicators of 8086/8088 bus activity.
- Interaction of the BIU and EU is via the queue for prefetched opcodes and via the EU for requested bus cycles for data operands.

Keeping these guidelines in mind, the instruction sequence depicted in figure 4-22 can be described as follows. Starting the loop arbitrarily in clock cycle 1 with the queue reinitialization that occurs as part of the JMP instruction, JMP instruction execution is completed by the EU, while the BIU performs an opcode fetch to begin refilling the queue. (Note that a shorthand notation has been used in the figure to represent the two queue status lines and the three status lines—active periods on any of these lines are noted and the binary value of the lines is indicated above each active region.)

In clock cycle 8, the queue status lines indicate that the first byte of the MOV immediate instruction has been removed from the queue (one clock cycle after it was placed there by the BIU fetch) and that execution of this instruction has begun. The second byte of this instruction is taken from the queue in clock cycle 10 and then, in clock cycle 12, the EU pauses to wait one clock cycle for the BIU's second opcode fetch to be completed and for the third byte of the MOV immediate instruction to be available for execution (remember the queue status lines indicate queue activity that has occurred in the previous clock cycle).

Clock cycle 13 begins the execution of the PUSH AX instruction, and in clock cycle 15, the BIU begins the fourth opcode fetch. The BIU finishes the fourth fetch in clock cycle 18 and prepares for another fetch when it receives a request from the EU for a memory write (the stack push). Instead of completing the opcode fetch and forcing the EU to wait four additional clock cycles, the BIU immediately aborts the fetch cycle (resulting in two idle clock cycles ($T_I$) in clock cycles 19 and 20) and performs the required memory write. This interaction between the EU and BIU results in a single clock extension to the execution time of the PUSH AX instruction, the maximum delay that can occur in response to an EU bus cycle request.

Execution continues in clock cycle 24 with the execution of back-to-back, register-to-register MOV instructions. The first of these instructions takes full advantage of the prefetched opcode to complete this operation in two clock cycles. The second MOV instruction, however, depletes the queue and requires two additional clock cycles (clock cycles 28 and 29).

## Table 4-14. Machine Instruction Encoding Matrix

| Hi \ Lo | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | ADD b,f,r/m | ADD w,f,r/m | ADD b,t,r/m | ADD w,t,r/m | ADD b,ia | ADD w,ia | PUSH ES | POP ES | OR b,f,r/m | OR w,f,r/m | OR b,t,r/m | OR w,t,r/m | OR b,i | OR w,i | PUSH CS | |
| 1 | ADC b,f,r/m | ADC w,f,r/m | ADC b,t,r/m | ADC w,t,r/m | ADC b,i | ADC w,i | PUSH SS | POP SS | SBB b,f,r/m | SBB w,f,r/m | SBB b,t,r/m | SBB w,t,r/m | SBB b,i | SBB w,i | PUSH DS | POP DS |
| 2 | AND b,f,r/m | AND w,f,r/m | AND b,t,r/m | AND w,t,r/m | AND b,i | AND w,i | SEG -ES | DAA | SUB b,f,r/m | SUB w,f,r/m | SUB b,t,r/m | SUB w,t,r/m | SUB b,i | SUB w,i | SEG -CS | DAS |
| 3 | XOR b,f,r/m | XOR w,f,r/m | XOR b,t,r/m | XOR w,t,r/m | XOR b,i | XOR w,i | SEG -SS | AAA | CMP b,f,r/m | CMP w,f,r/m | CMP b,t,r/m | CMP w,t,r/m | CMP b,i | CMP w,i | SEG -DS | AAS |
| 4 | INC AX | INC CX | INC DX | INC BX | INC SP | INC BP | INC SI | INC DI | DEC AX | DEC CX | DEC DX | DEC BX | DEC SP | DEC BP | DEC SI | DEC DI |
| 5 | PUSH AX | PUSH CX | PUSH DX | PUSH BX | PUSH SP | PUSH BP | PUSH SI | PUSH DI | POP AX | POP CX | POP DX | POP BX | POP SP | POP BP | POP SI | POP DI |
| 6 | | | | | | | | | | | | | | | | |
| 7 | JO | JNO | JB/ JNAE | JNB/ JAE | JE/ JZ | JNE/ JNZ | JBE/ JNA | JNBE/ JA | JS | JNS | JP/ JPE | JNP/ JPO | JL/ JNGE | JNL/ JGE | JLE/ JNG | JNLE/ JG |
| 8 | Immed b,r/m | Immed w,r/m | Immed b,r/m | Immed is,r/m | TEST b,r/m | TEST w,r/m | XCHG b,r/m | XCHG w,r/m | MOV b,f,r/m | MOV w,f,r/m | MOV b,t,r/m | MOV w,t,r/m | MOV sr,f,r/m | LEA | MOV sr,t,r/m | POP r/m |
| 9 | XCHG AX | XCHG CX | XCHG DX | XCHG BX | XCHG SP | XCHG BP | XCHG SI | XCHG DI | CBW | CWD | CALL l,d | WAIT | PUSHF | POPF | SAHF | LAHF |
| A | MOV m → AL | MOV m → AX | MOV AL → m | MOV AX → m | MOVS | MOVS | CMPS | CMPS | TEST b,i,a | TEST w,i,a | STOS | STOS | LODS | LODS | SCAS | SCAS |
| B | MOV i → AL | MOV i → CL | MOV i → DL | MOV i → BL | MOV i → AH | MOV i → CH | MOV i → DH | MOV i → BH | MOV i → AX | MOV i → CX | MOV i → DX | MOV i → BX | MOV i → SP | MOV i → BP | MOV i → SI | MOV i → DI |
| C | | | RET (i+SP) | RET | LES | LDS | MOV b,i,r/m | MOV w,i,r/m | | | RET l,(i+SP) | RET l | INT Type 3 | INT (Any) | INTO | IRET |
| D | Shift b | Shift w | Shift b,v | Shift w,v | AAM | AAD | | XLAT | ESC 0 | ESC 1 | ESC 2 | ESC 3 | ESC 4 | ESC 5 | ESC 6 | ESC 7 |
| E | LOOPNZ/ LOOPNE | LOOPZ/ LOOPE | LOOP | JCXZ | IN b | IN w | OUT b | OUT w | CALL d | JMP d | JMP l,d | JMP si,d | IN v,b | IN v,w | OUT v,b | OUT v,w |
| F | LOCK | | REP | REP z | HLT | CMC | Grp 1 b,r/m | Grp 1 w,r/m | CLC | STC | CLI | STI | CLD | STD | Grp 2 b,r/m | Grp 2 w,r/m |

where

| mod ☐ r/m | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|
| Immed | ADD | OR | ADC | SBB | AND | SUB | XOR | CMP |
| Shift | ROL | ROR | RCL | RCR | SHL/SAL | SHR | — | SAR |
| Grp 1 | TEST | — | NOT | NEG | MUL | IMUL | DIV | IDIV |
| Grp 2 | INC | DEC | CALL id | CALL l,id | JMP id | JMP l,id | PUSH | — |

| | |
|---|---|
| b = byte operation | m = memory |
| d = direct | r/m = EA is second byte |
| f = from CPU reg | si = short intrasegment |
| i = immediate | sr = segment register |
| ia = immed. to accum. | t = to CPU reg |
| id = indirect | v = variable |
| is = immed. byte, sign ext | w = word operation |
| l = long ie. intersegment | z = zero |

## 8086 Instruction Sequence

Figure 4-22 illustrates the internal operation and bus activity that occur as an 8086 CPU executes a sequence of instructions. This figure presents the signals and timing relationships that are important in understanding 8086 operation. The following discussion is intended to help in the interpretation of the figure.

Figure 4-22 shows the repeated execution of an instruction loop. This loop is defined in both machine code and assembly language by figure 4-21. A loop was chosen both to demonstrate the effects of a program jump on the queue and to make the instruction sequence easy to follow. The program sequence shown was selected for several reasons. First, consisting of seven instructions and 16 bytes, the sequence is typical of the tight loops found in many application programs. Second, this particular sequence contains several short, fast-executing instructions that demonstrate both the effect of the queue on CPU performance and the interaction between the execution unit (EU) fetching code from the queue and the bus interface unit (BIU) filling the queue and performing the requested bus cycles. Last, for the purpose of this discussion, code, stack, and memory data references were arranged to be aligned on even word boundaries.

| ASSEMBLY LANGUAGE | MACHINE CODE |
|---|---|
| MOV AX, 0F802H | B802F8 |
| PUSH AX | 50 |
| MOV CX, BX | 8BCB |
| MOV DX, CX | 8BD1 |
| ADD AX, [SI] | 0304 |
| ADD SI, 8086H | 81C68680 |
| JMP $ –14 | EBF0 |

**Figure 4-21. Instruction Loop Sequence**

Figure 4-22 can be more easily interpreted by keeping the following guidelines in mind.

- The queue status lines (QS0, QS1) are the key indicators of EU activity.
- Status lines $\overline{S2}$ through $\overline{S0}$ are the main indicators of 8086/8088 bus activity.
- Interaction of the BIU and EU is via the queue for prefetched opcodes and via the EU for requested bus cycles for data operands.

Keeping these guidelines in mind, the instruction sequence depicted in figure 4-22 can be described as follows. Starting the loop arbitrarily in clock cycle 1 with the queue reinitialization that occurs as part of the JMP instruction, JMP instruction execution is completed by the EU, while the BIU performs an opcode fetch to begin refilling the queue. (Note that a shorthand notation has been used in the figure to represent the two queue status lines and the three status lines—active periods on any of these lines are noted and the binary value of the lines is indicated above each active region.)

In clock cycle 8, the queue status lines indicate that the first byte of the MOV immediate instruction has been removed from the queue (one clock cycle after it was placed there by the BIU fetch) and that execution of this instruction has begun. The second byte of this instruction is taken from the queue in clock cycle 10 and then, in clock cycle 12, the EU pauses to wait one clock cycle for the BIU's second opcode fetch to be completed and for the third byte of the MOV immediate instruction to be available for execution (remember the queue status lines indicate queue activity that has occurred in the previous clock cycle).

Clock cycle 13 begins the execution of the PUSH AX instruction, and in clock cycle 15, the BIU begins the fourth opcode fetch. The BIU finishes the fourth fetch in clock cycle 18 and prepares for another fetch when it receives a request from the EU for a memory write (the stack push). Instead of completing the opcode fetch and forcing the EU to wait four additional clock cycles, the BIU immediately aborts the fetch cycle (resulting in two idle clock cycles ($T_I$) in clock cycles 19 and 20) and performs the required memory write. This interaction between the EU and BIU results in a single clock extension to the execution time of the PUSH AX instruction, the maximum delay that can occur in response to an EU bus cycle request.

Execution continues in clock cycle 24 with the execution of back-to-back, register-to-register MOV instructions. The first of these instructions takes full advantage of the prefetched opcode to complete this operation in two clock cycles. The second MOV instruction, however, depletes the queue and requires two additional clock cycles (clock cycles 28 and 29).

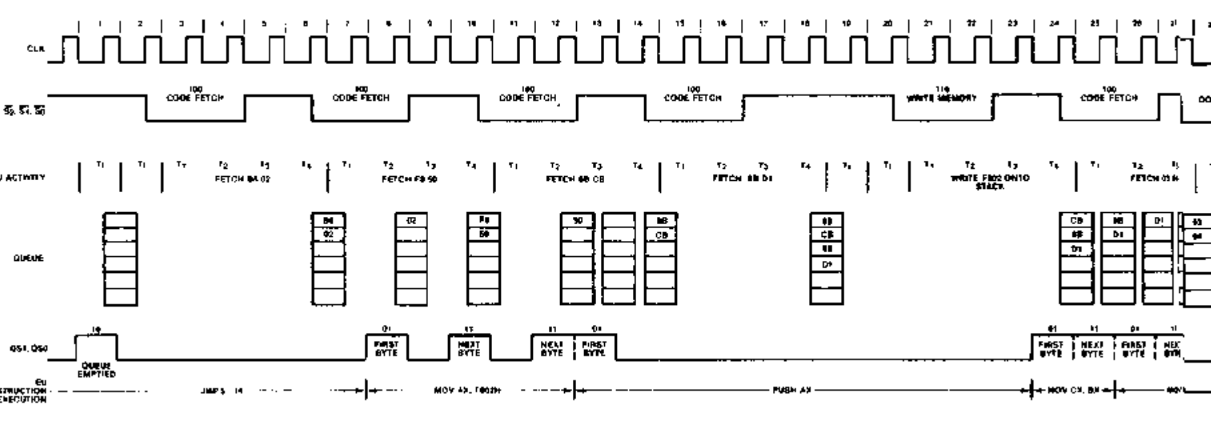


**Figure 4-22. Sample Instruction Sequence Execution**

In clock cycle 30, the ADD memory indirect to AX instruction begins. In the time required to execute this instruction, the BIU completes two opcode fetch cycles and a memory read and begins a fourth opcode fetch cycle. Note that in the case of the memory read, the EU's request for a bus cycle occurs at a point in the BIU fetch cycle where it can be incorporated directly (idle states are not required and no EU delay is imposed).

In clock cycle 44, the EU begins the ADD immediate instruction, taking four bytes from the queue and completing instruction execution in four clock cycles. Also during this time, the BIU senses a full queue in clock cycle 45 and enters a series of bus idle states (five or six bytes constitute a full queue in the 8086; the BIU waits until it can fetch a full word of opcode before accessing the bus).

At clock cycle 47, the BIU again begins a bus cycle sequence, one that is destined to be an "overfetch" since the EU is executing a JMP instruction. As part of the JMP instruction, the queue reinitialization (which began the instruction sequence) occurs.

The entire sequence of instructions has taken 55 clock cycles. Eighteen opcode bytes were fetched, one word memory read occurred, and one word stack write was performed.

This example was, by design, partially bus limited and indicates the types of EU and BIU interaction that can occur in this situation. Most application code sequences, however, use a higher proportion of more complex, longer-executing instructions and addressing modes, and therefore tend to be execution limited. In this case, less BIU-EU interaction is required, the queue more often is full, and more idle states occur on the bus.

The previous example sequence can be easily extended to incorporate wait states in the bus access cycles. In the case of a single wait state, each bus cycle would be lengthened to five clock cycles with a wait state ($T_W$) inserted between every $T_3$ and $T_4$ state of the bus cycle. As a first approximation, the instruction sequence exection time would appear to be lengthened by 10 clock cycles, one cycle for each useful read or write bus cycle that occurs. Actually, this approximation for the number of wait states inserted is incorrect since the queue can compensate for wait states by making use of previously idle bus time. For the example sequence, this compensation reduced the actual execution time by one wait state, and the sequence was completed in 64 clock cycles, one less than the approximated 65 clock cycles.

## 4.3 8089 I/O Processor

The Intel® 8089 I/O Processor (IOP) combines the functions of a DMA controller with the processing capabilities of a microprocessor. In addition to the normal DMA function of transferring data, the 8089 is capable of dynamically translating and comparing the data as it is

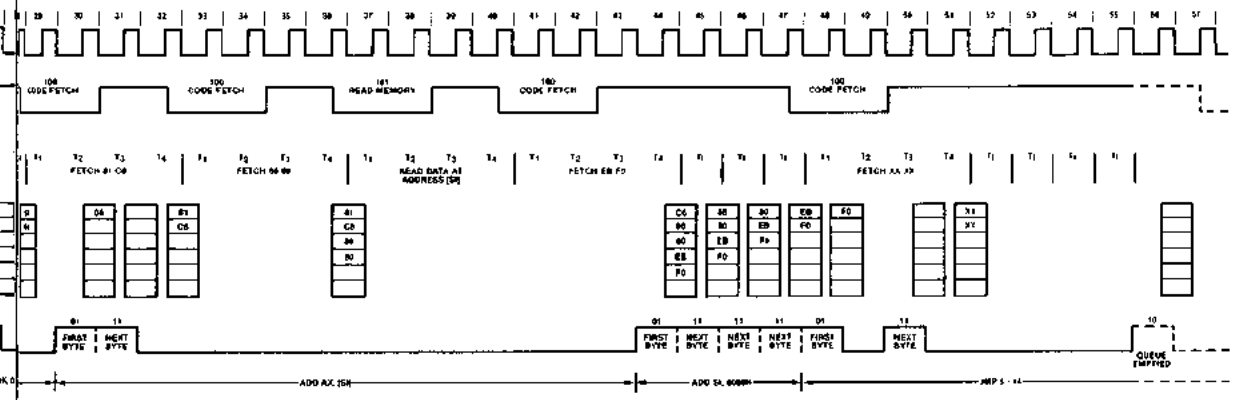


**Figure 4-22. Sample Instruction Sequence Execution**

transferred and of supporting a number of terminate conditions including byte count expired, data compare or miscompare and the occurrence of an external event. The 8089 contains two separate DMA channels, each with its own register set. Depending on the established priorities (both inherent and program determined), the two channels can alternate (interleave) their respective operations.

Designed expressly to relieve the 8086 or 8088 CPU of the overhead associated with I/O operations, the 8089, when configured in the remote mode, can perform a complete I/O task while the CPU is performing data processing tasks. The 8089, when it has completed its I/O task, can then interrupt the CPU.

Transfer flexibility is an integral part of the 8089's design. In addition to routine transfers between an I/O peripheral and memory, transfers can be performed between two I/O devices or between two areas of memory. Transfers between dissimilar bus widths are automatically handled by the 8089. When data is transferred from an 8-bit peripheral bus to a 16-bit memory bus, the 8089 reads two bytes from the peripheral, assembles the bytes into a 16-bit word and then writes the single word to the addressed memory location. Also, both 8- and 16-bit peripherals can reside on the same (16-bit) bus; byte transfers are performed with the 8-bit peripheral, and word transfers are performed with the 16-bit peripheral.

## System Configuration

The 8089 can be implemented in one of two system configurations: a "local" mode in which the 8089 shares the system bus with an 8086 or 8088 CPU and a "remote" mode in which the 8089 has exclusive access to its own dedicated bus as well as access to the system bus. Note that in either the local or remote mode, the 8089 can address a full megabyte of system memory and 64k bytes of I/O space.

### Local Mode

In the local mode, the 8089 acts as a slave to an 8086 or 8088 CPU that is operating in the maximum mode. In this configuration, the 8089 shares the system address latches, data transceivers and bus controller with the CPU as shown in figure 4-23.

Since the IOP and CPU share the system bus, either the IOP or the CPU will have access to the bus at any one time. When one processor is using the bus, the other processor floats its address/data and control lines. Bus access between the IOP and CPU is determined through the request/grant function. Recalling the CPU's request/grant sequence, the IOP requests the bus from the CPU, the CPU grants the bus to the IOP, and the IOP relinquishes the bus to the CPU when its operation is complete. Remember that the CPU cannot request the bus from the IOP (the CPU is only capable of granting the bus and

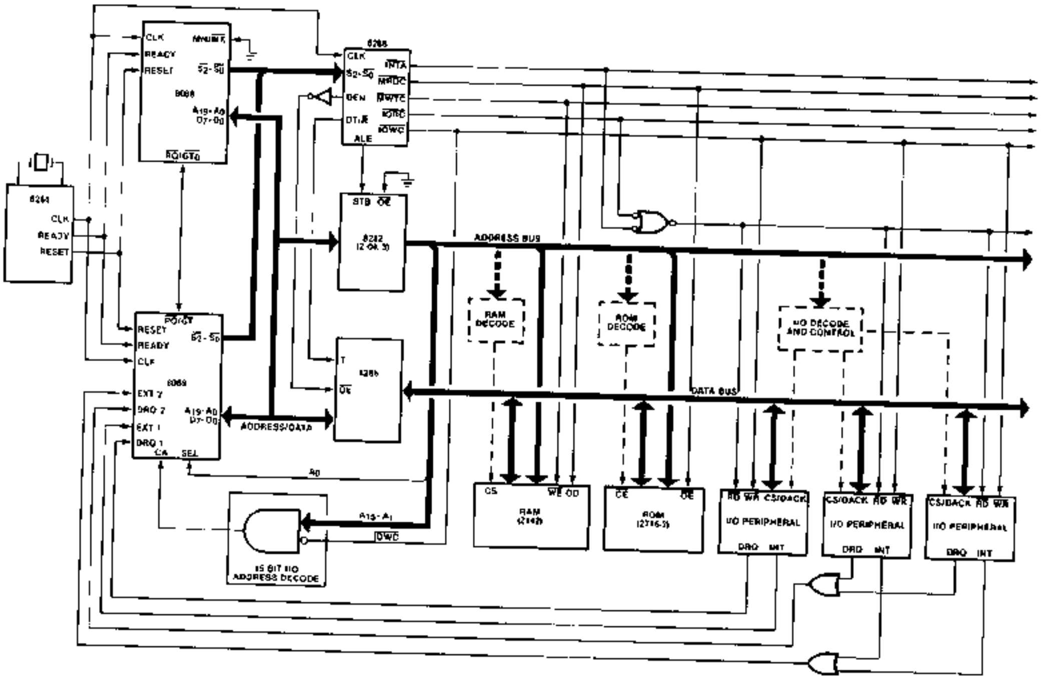


Figure 4-23. Typical 8088/8089 Local Mode Configuration

must wait for the IOP to release the bus). Also, since the request/grant pulse exchange must be synchronized, both the CPU and IOP must be referenced to the same clock signal.

The 8089 IOP, when used in the local mode, can be added to an 8086 or 8088 maximum mode configuration with little affect on component count (channel attention decoding logic as required) and offers the benefits of intelligent DMA (scan/match, translate, variable termination conditions), modular programming in a full megabyte of memory address space and a set of optimized I/O instructions that are unavailable to the 8086 and 8088 CPUs. The major disadvantage to the local configuration is that since the system bus is shared, bus contention always exists between the CPU and IOP. The use of the bus load limit field in the channel control word can help reduce IOP bus access during task block program execution (bus load limiting has no affect on DMA transfers) although, for I/O intensive systems, the remote mode should be considered.

## Remote Mode

The 8089, when used in the remote mode, provides a multiprocessor system with true parallel processing. In this mode, the 8089 has a separate (local) bus and memory for I/O peripheral communications, and the system bus is completely isolated from the I/O peripheral(s). Accordingly, I/O transfers between an I/O peripheral and the IOP's local memory can occur simultaneously with CPU operations on the system bus.

As shown in figure 4-24, to interface the 8089 to the system bus, data transceivers and address latches are used to separate the IOP's local bus from the system bus, an 8288 Bus Controller is used to generate the bus control signals for both the local and system buses as well as to govern the operation of the transceivers/latches, and an 8289 Bus Arbiter is used to control access to the system bus (each processor in the system would have an associated 8289 Bus Arbiter). To interface the 8089 to its local bus, another set of address

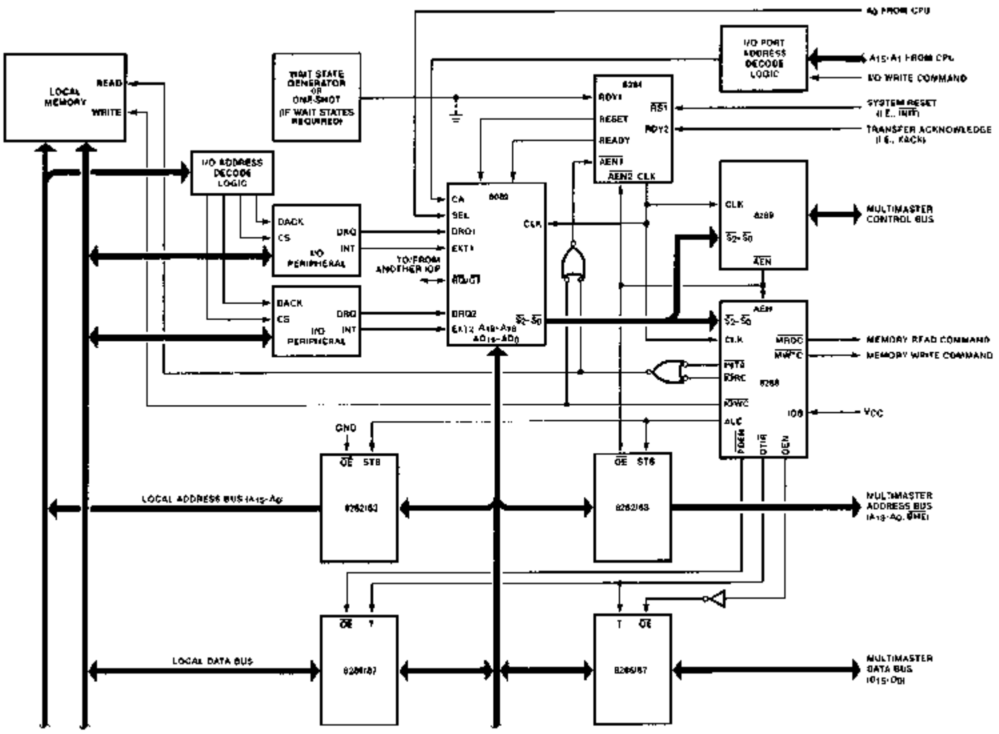


Figure 4-24. Typical 8089 Remote Mode Configuration

latches is required (unless MCS-85™ multiplexed address components are exclusively interfaced) and, depending on the bus loading demands, one (8-bit bus) or two (16-bit bus) data transceivers would be used.

In the remote mode, the IOP's local bus is treated as I/O space (up to 64k bytes), and the system bus is treated as memory space (1 megabyte). The 8288 Bus Controller's I/O command outputs control the local (I/O) bus, and its memory command outputs control the system (memory) bus. The 8289 Bus Arbiter, which is operated in its IOB (I/O peripheral bus) mode, also decodes the IOP's $\overline{S2}$ through $\overline{S0}$ status outputs. In this mode, the 8289 will not request the multimaster system bus when the IOP indicates an operation on its local bus. If the IOP's bus arbiter currently has access to the system bus, the CPU's arbiter (or any other arbiter in the system) can acquire use of the system bus at this time (a bus arbiter maintains bus access until another arbiter requests the bus).

## Bus Operation

The 8089 utilizes the same bus structure as an 8086 or 8088 CPU that is configured in the maximum mode and performs a bus cycle only on demand (e.g., to fetch an instruction during task block execution or to perform a data transfer). The bus cycle itself is identical to an 8086 or 8088 CPU's bus cycle in that all cycles consist of four T-states and use the same time-multiplexing technique of the addressdata lines. As shown in the following timing diagrams, the address (and ALE signal) is output during state $T_1$ for either a read or write cycle. Depending on the type of cycle indicated, the address/data lines are floated during state $T_2$ for a read cycle (figure 4-25) or data is output on these lines during a write cycle (figure 4-26). During state $T_3$, write data is maintained or read data is sampled, and the busy cycle is concluded in state $T_4$.

Since the 8089 is capable of transferring data to or from both 8-bit and 16-bit buses, when an 8-bit physical bus is specified (bus width is specified

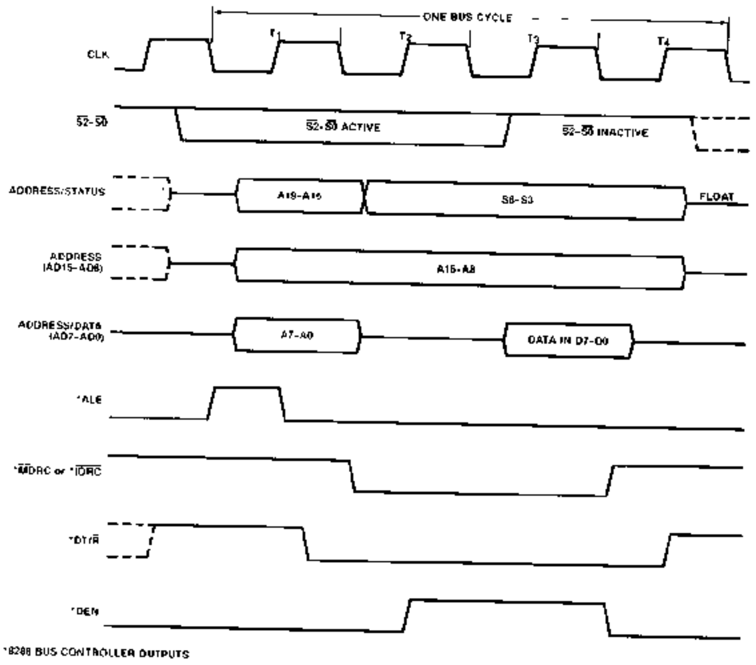

ONE BUS CYCLE
T1
T2
T3
T4
CLK
$\overline{S2}$-$\overline{S0}$
$\overline{S2}$-$\overline{S0}$ ACTIVE
$\overline{S2}$-$\overline{S0}$ INACTIVE
ADDRESS/STATUS
A19-A16
S6-S3
FLOAT
ADDRESS (AD15-AD8)
A15-A8
ADDRESS/DATA (AD7-AD0)
A7-A0
DATA IN D7-D0
*ALE
*$\overline{MRDC}$ or *$\overline{IORC}$
*DT/$\overline{R}$
*DEN
*8288 BUS CONTROLLER OUTPUTS

Figure 4-25. Read Bus Cycle (8-Bit Bus)

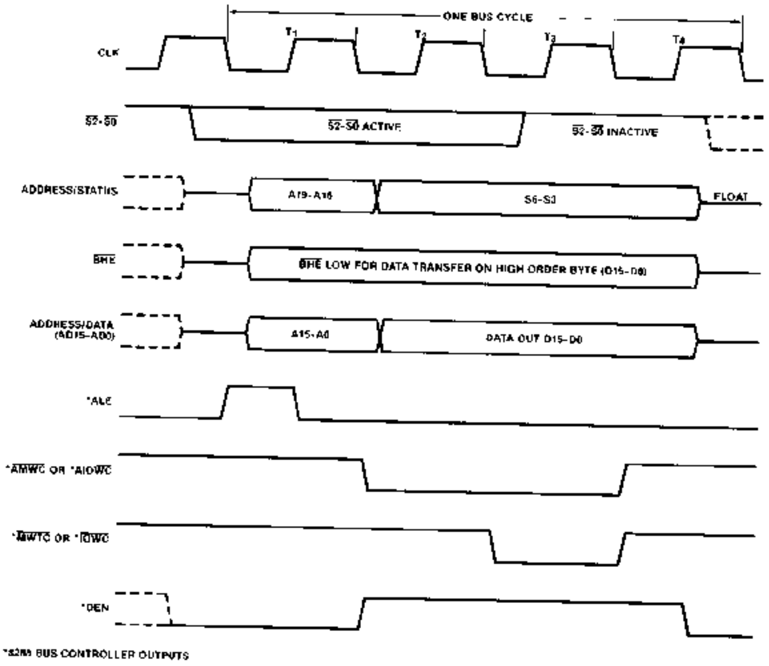

ONE BUS CYCLE
T1
T2
T3
T4
CLK
$\overline{S2}$-$\overline{S0}$
$\overline{S2}$-$\overline{S0}$ ACTIVE
$\overline{S2}$-$\overline{S0}$ INACTIVE
ADDRESS/STATUS
A19-A16
S6-S3
FLOAT
$\overline{BHE}$
$\overline{BHE}$ LOW FOR DATA TRANSFER ON HIGH ORDER BYTE (D15-D8)
ADDRESS/DATA (AD15-AD0)
A15-A0
DATA OUT D15-D0
*ALE
*$\overline{AMWC}$ OR *$\overline{AIOWC}$
*$\overline{MWTC}$ OR *$\overline{IOWC}$
*DEN
*8288 BUS CONTROLLER OUTPUTS

Figure 4-26. Write Bus Cycle (16-Bit Bus)

during the initialization sequence), the address present on the AD15 through AD8 address/data lines is maintained for the entire bus cycle as shown in figure 4-25 and, unless added drive capability is required, the associated address latch can be eliminated. An 8-bit data bus is compatible with the 8088 CPU and with the MCS-85™ multiplexed address peripherals (8155, 8185, etc.).

The 8089 operates identically to the 8086 CPU with respect to the use of the low- and high-order halves of the data bus. Table 4-14 defines the data bus use for the various combinations of bus width and address boundary.

The $\overline{S2}$ through $\overline{S0}$ status lines define the bus cycle to be performed. These lines are used by an 8288 Bus Controller to generate all memory and I/O command and control signals, and are decoded according to table 4-15.

Table 4-14. Data Bus Usage

| Logical Bus Width[1] | Address Boundary | Physical Bus Width[2] | | |
|---|---|---|---|---|
| | | 8 | 16 | |
| | | | Byte Transfer | Word Transfer |
| 8 | Even | AD7-AD0 = DATA ($\overline{BHE}$ not used) | AD7-AD0 = DATA ($\overline{BHE}$ high) | N/A |
| | Odd | AD7-AD0 = DATA ($\overline{BHE}$ not used) | AD15-AD8 = DATA ($\overline{BHE}$ low) | N/A |
| 16 | Even | Illegal | AD7-AD0 = DATA ($\overline{BHE}$ high) | AD15-AD0 = DATA ($\overline{BHE}$ low) |
| | Odd | Illegal | AD15-AD8 = DATA ($\overline{BHE}$ low) | N/A[3] |

**Notes:**

1. Logical bus width is specified by the WID instruction prior to the DMA transfer.
2. Physical bus width is specified when the 8089 is initialized.
3. A word transfer to or from an odd boundary is performed as two byte transfers. The first byte transferred is the low-order byte on the high-order data bus (AD15-AD8), and the second byte is the high-order byte on the low-order data bus (AD7-AD0). The 8089 automatically assembles the two bytes in their proper order.

Table 4-15. Bus Cycle Decoding

| Status Output | | | Bus Cycle Indicated | Bus Controller Command Output |
|---|---|---|---|---|
| $\overline{S2}$ | $\overline{S1}$ | $\overline{S0}$ | | |
| 0 | 0 | 0 | Instruction fetch from I/O space | $\overline{INTA}$ |
| 0 | 0 | 1 | Data read from I/O space | $\overline{IORC}$ |
| 0 | 1 | 0 | Data write to I/O space | $\overline{IOWC}$, $\overline{AIOWC}$ |
| 0 | 1 | 1 | Not used | None |
| 1 | 0 | 0 | Instruction fetch from system memory | $\overline{MRDC}$ |
| 1 | 0 | 1 | Data read from system memory | $\overline{MRDC}$ |
| 1 | 1 | 0 | Data write to system memory | $\overline{MWTC}$, $\overline{AMWC}$ |
| 1 | 1 | 1 | Passive | None |

Note that the 8089 indicates an instruction fetch from I/O space as a status of zero ($\overline{S2}$, $\overline{S1}$ and $\overline{S0}$ equal 0). Since the 8288 Bus Controller decodes an input status value of zero as an interrupt acknowledge bus cycle, the bus controller's $\overline{INTA}$ output must be OR'ed with its $\overline{IORC}$ output to permit fetching of task block instructions from local 8089 memory (remote configuration) or system I/O space (local and remote configurations).

The $\overline{S2}$ through $\overline{S0}$ status lines become active in state $T_4$ if a subsequent bus cycle is to be performed. These lines are set to the passive state (all "ones") in the state immediately prior to state $T_4$ of the current bus cycle (state $T_3$ or $T_W$) and are floated when the 8089 does not have access to the bus.

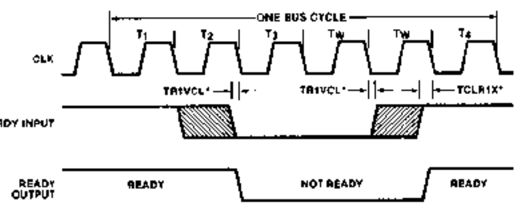
The S6 through S3 status lines are multiplexed with the high-order address bits (A19-A16) and, accordingly, become valid in state $T_2$ of the bus cycle. The S4 and S3 status lines reflect the type of bus cycle being performed on the corresponding channel as indicated in table 4-16.

Table 4-16. Type of Cycle Decoding

| Status Output | | Type of Cycle |
|---|---|---|
| S4 | S3 | |
| 0 | 0 | DMA on Channel 1 |
| 0 | 1 | DMA on Channel 2 |
| 1 | 0 | Non-DMA on Channel 1 |
| 1 | 1 | Non-DMA on Channel 2 |

The S6 and S5 status lines are always "1" on the 8089. Since these lines are not both "1" on the other processors in the 8086 family (S6 is always "0" on the 8086 and 8088 CPUs), these status lines can be used as a "signature" in a multiprocessor environment to identify the type of processor performing the bus cycle.

The 8089 includes the same provision as do the 8086 and 8088 CPUs for the insertion of wait states ($T_W$) in a bus cycle when the associated memory or I/O device cannot respond within the alloted time interval or when, in the remote mode, the 8089 must wait for access to the system bus. An 8284 Clock Generator/Driver is used to control the insertion of wait states which, when required, are inserted between states $T_3$ and $T_4$. The actual insertion of wait states is accomplished by deactivating one of the 8284's RDY inputs (RDY1 or RDY2). Either of these inputs, when enabled by its corresponding $\overline{AEN1}$ or $\overline{AEN2}$ input, can be deactivated directly by the memory or I/O device when it must extend the 8089's bus cycle (when the addressed device is not ready to present or accept data). The 8284's READY output, which is synchronized to the CLK signal, is directly connected to the 8089's READY input. As shown in figure 4-27, when the addressed device requires one or more wait states to be inserted into a bus cycle, it deactivates the 8284's RDY input prior to the end of state $T_2$. The READY output from the 8284 is subsequently deactivated at the end of state $T_2$ which causes the 8089 to insert wait states following state $T_3$. To exit the wait state, the device activates the 8284's RDY input which causes the READY input to the 8089 to go active on the next clock cycle and allows the 8089 to enter state $T_4$.



*REFER TO THE 8284 CLOCK GENERATOR/DRIVER DATA SHEET IN APPENDIX B FOR TIMING INFORMATION

Figure 4-27. Wait State Timing

Periods of inactivity can occur between bus cycles. These inactive periods are referred to as idle states ($T_I$) and, as with the 8086 and 8088 CPUs, can result from the execution of a "long" instruction or the loss of the bus to another processor during task block instruction execution. Additionally, the 8089 can experience idle states when it is in the DMA mode and it is waiting for a DMA request from the addressed I/O device or when the bus load limit (BLL) function is enabled for a channel performing task block instruction execution and the other channel is idle.

## Initialization

Initialization of the IOP is generally the responsibility of the host processor which, as stated in Chapter 3, prepares the communications data structure in shared memory. Initialization of the IOP itself begins with the activation of its RESET input. This input (originating typically from an

8284 Clock Generator/Driver) must be held active for at least five clock cycles to allow the 8089's internal reset sequence to be completed. Note that like the 8086 and 8088 CPUs, the RESET input must be held active for at least 50 microseconds when power is first applied. Following the reset interval, the host processor signals the IOP to begin its initialization sequence by activating the 8089's CA (Channel Attention) input. The 8089 will not recognize a pulse at its CA input until one clock cycle after the RESET input returns to an inactive level. Note that the minimum width for a CA pulse is one clock cycle and that this pulse may go active prior to RESET returning to an inactive level provided that the negative-going, trailing-edge of the CA pulse does not occur prior to one clock cycle after RESET goes inactive. Figure 4-28 illustrates the timing for this portion of the initialization sequence.

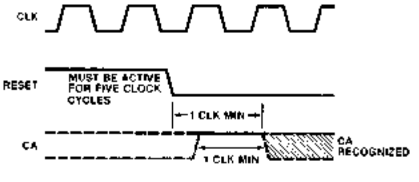


Figure 4-28. RESET-CA Initialization Timing

Coincident with the trailing edge of the first CA pulse following reset, the 8089 samples its SEL (Select) input from the host processor to determine master/slave status for its request/grant circuity. If the SEL input is low, the 8089 is designated a "master," and if the SEL input is high, the 8089 is designated a "slave." As a master, the 8089 assumes that it has the bus initially, and it will subsequently grant the bus to a requesting slave when the bus becomes available (i.e., the 8089 will respond to a "request" pulse on its $\overline{RQ}/\overline{GT}$ line with a "grant" pulse). A single 8089 in the remote configuration (or one of two 8089s in a remote configuration) would be designated a master. As a slave, the 8089 can only request the bus from a master processor (i.e., the 8089 initiates the request/grant sequence by outputting a "request" pulse on its $\overline{RQ}/\overline{GT}$ line). An 8089 that shares a bus with an 8086 or 8088 (or one of two 8089s in a remote configuration) would be designated a slave. Note that since the 8086 and 8088 CPUs can grant the bus only in response to a request, whenever an 8086 or 8088 and an 8089 share a common bus, the 8089 *must* be designated the slave. Also, when the $\overline{RQ}/\overline{GT}$ line is not used (i.e., a single 8089 in the remote configuration), the 8089 *must* be designated a master.

In addition to determining master/slave status, the CA pulse also causes the 8089 to begin execution of its internal ROM initialization sequence. Note that since the 8089 must have access to the *system* bus in order to perform this sequence, the 8089 immediately initiates a request/grant sequence (if designated a slave) and, if required, then requests the bus through the 8289 Arbiter. (If designated a master, the 8089 requests the bus through the 8289 Arbiter.) In the execution of the initialization sequence, the 8089 first fetches the SYSBUS byte from location FFFF6H. The W bit (bit 0) of this byte specifies the *physical* bus width of the *system* bus. Depending on the bus width specified, the 8089 then fetches the address of the system configuration block (SCB) contained in locations FFFF8H through FFFFBH in either two bus cycles (16-bit bus, W bit equal 1) or four bus cycles (8-bit bus, W bit equal 0). The SCB offset and segment address values fetched are combined into a 20-bit physical address that is stored in an internal register. Using this address, the 8089 next fetches the system operation command (SOC) byte. As explained in Chapter 3, this byte specifies both the request/grant operational mode (R bit) and the *physical* width of the I/O bus (I bit). After reading the SOC byte, the 8089 fetches the channel control block (CB) offset and segment address values. These values are combined into a 20-bit physical address and are stored in another internal register. To inform the host CPU that it has completed the initialization sequence, the 8089 clears the Channel 1 Busy flag in the channel control block by writing an all "zeroes" byte to CB + 1.

After the IOP has been initialized, the system configuration block may be altered in order to initialize another IOP. Once an IOP has been initialized, its channel control block in system memory cannot be moved since the CB address, which is internally stored by the IOP during the initialization sequence, is automatically accessed on every subsequent CA pulse.

As previously stated, the generation of the CA and SEL inputs to the IOP are the responsibility of the host CPU. Typically, these signals result from the CPU's execution of an I/O write instruction to one of two adjacent I/O ports (I/O port addresses that only differ by A0). Figure 4-29 illustrates a simple decoding circuit that could be used to generate the CA and SEL signals. Note that by qualifying the CA output with $\overline{\text{IOWC}}$, the SEL output, since it is latched for the entire I/O bus cycle, is guaranteed to be stable on the trailing edge of the CA pulse.

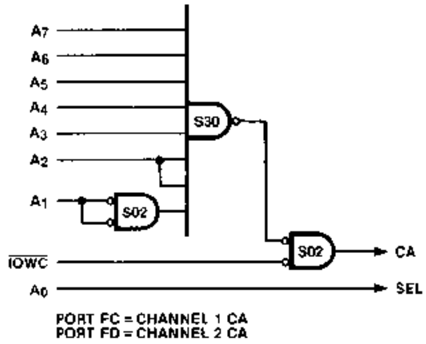


**Figure 4-29. Channel Attention Decoding Circuit**

## I/O Dispatching

During normal operation, the I/O supervisory program running in the host CPU will receive a request to perform a specific I/O operation on one of the 8089's channels. In response to this request, the supervisory program will typically perform the following sequence of operations:

- Check the availability of the specified channel by examining the channel's busy flag in the Channel Control Block. If it is possible for another processor to access the channel, a semaphore operation (implemented by a locked XCHG instruction) is used to check channel availability.
- Load the variable parameters required for the intended operation into the channel's parameter block.
- Load the channel command word (CCW) into the channel control block.
- Establish the necessary linkages by writing the starting address of the channel program (task block) in the first four bytes of the parameter block and writing the address of the parameter block in the channel control block.
- Issue a channel attention (CA) to the specified channel.

In response to the CA, the 8089 interrupts any current activity at its first opportunity (see "Concurrent Channel Operation" in section 3.2) and begins execution of an internal instruction sequence that fetches and decodes the channel command word (CCW) and then performs the operation indicated (i.e., start, halt or continue channel program execution).

If the CCW specifies start channel program (start task block execution), the address of the parameter block is fetched from the channel control block, the address of the first channel program instruction (contained in the first four bytes of the parameter block) is fetched and then loaded into the TP (task pointer) register and, finally, task block execution is initiated from either system or I/O space. Task block execution continues, subject to the activity on the other channel as described in "Concurrent Channel Operation," until a XFER instruction is executed. Following execution of this instruction, the next sequential channel program instruction is executed before the channel enters the DMA transfer mode.

If the CCW specifies halt channel, the current operation on the specified channel is halted. If the channel is performing task block execution (either chained or not chained), channel operation is stopped at an instruction boundary, and if the channel is performing a DMA transfer, channel operation is stopped at a DMA transfer cycle boundary. Note that a channel will not stop a locked DMA transfer until the operation is completed. There are two unique halt channel commands. One command simply halts the channel and clears the busy flag in the channel control block. This command is used when the halted operation is to be discarded. The other command halts the channel, saves the task pointer and program status word (PSW) byte, and clears the busy flag. This command is used when the halted operation is to be resumed. Note that this halt command will not affect the integrity of resumed task block execution or a memory-to-memory DMA transfer, but could affect the integrity of a synchronized DMA transfer (a DMA request occuring while the channel is halted could be missed).

If the CCW specifies continue channel, an operation that has been previously halted is resumed (and the busy flag is set). Since this command restores the task pointer and PSW, it should be used only if the task pointer and PSW have been saved by a previous halt command.

Table 4-17 outlines the various CCW command execution times. Note that the times listed in the table for the halt commands do *not* include the time required to complete any current channel activity when the channel attention is received (completion of the current DMA transfer cycle or task block instruction).

## DMA Transfers

The number of bytes transferred during a single DMA cycle is determined by both the source and destination logical bus widths as well as by the address boundary (odd or even address). The 8089 performs DMA transfers between dissimilar bus widths by assembling bytes or disassembling words in its internal assembly register file. As explained in Chapter 3, the DMA source and destination bus widths are defined by the execution of a WID instruction during task block (channel command) execution. Note that the bus widths specified remain in force until changed by a subsequent WID instruction. Table 4-18 defines the various byte (B) and word (W) source/destination transfer combinations based on address boundary and bus width specified.

The 8089 additionally optimizes bus accesses during transfers between dissimilar bus widths whenever possible. When either the source or destination is a 16-bit memory bus (auto-incrementing) that is initially aligned on an odd

Table 4-17. CCW Command Execution Times

| CCW Command | Minimum Time* | Maximum Time** |
|---|---|---|
| CA NOP | 48 + 2n clocks | 48 + 2n clocks |
| CA Halt (no save) | 48 + 2n clocks | 48 + 2n clocks |
| CA Halt (with save) | 94 + 5n clocks | 100 + 6n clocks |
| CA Start (memory) | 108 + 6n clocks | 124 + 10n clocks |
| CA Start (I/O) | 96 + 5n clocks | 108 + 8n clocks |
| CA Continue | 95 + 5n clocks | 103 + 6n clocks |

Notes:
n is the number of wait states per bus cycle.

* Minimum time occurs when both the channel control block and parameter block addresses are aligned on an even address boundary and a 16-bit bus is used.

** Maximum time occurs when both the channel control block and parameter block addresses are aligned on an odd address boundary on a 16-bit bus or when an 8-bit bus is used.

Table 4-18. DMA Assembly Register Operation

| Address Boundary (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 |

address boundary (causing the first transfer cycle to be byte-to-byte), following the first transfer cycle, the memory address will be aligned on an even address boundary, and word transfers will subsequently occur. For example, when performing a memory-to-port transfer from a 16-bit bus to an 8-bit bus with the source beginning on an odd address boundary, the first transfer cycle will be byte-to-byte (B → B) as indicated in table 4-18, but subsequent transfers will be word-to-byte/byte (W → B/B).

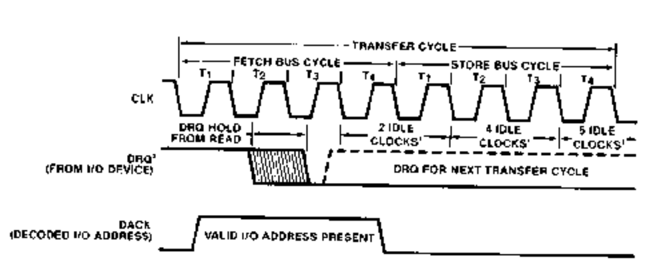
All DMA transfer cycles consist of at least two bus cycles; one bus cycle to fetch (read) the data form the source into the IOP, and one bus cycle to store (write) the data previously fetched from the IOP into the destination. Note that in all transfers, the data passes through the IOP to allow mask/compare and translate operations to be optionally performed during the transfer as well as to allow the data to be assembled or disassembled.

The IOP performs DMA transfers in one of three modes: unsynchronized, source synchronized or destination synchronized (the transfer mode is specified in the channel control register). The unsynchronized mode is used when both the source and destination devices do not provide a data request (DRQ) signal to the IOP as in the case of a memory-to-memory transfer. In the synchronized transfer modes, the source (source synchronized) or destination (destination synchronized) device initiates the transfer cycle by activating the IOP's DRQ1 (channel 1) or DRQ2 (channel 2) input.

The DRQ input is asynchronous and usually originates from an I/O device controller rather than from a memory circuit. This input is latched on the positive transition of the clock (CLK) signal and therefore must remain active for more than one clock period (more than 200 nanoseconds when using a 5 MHz clock) in order to guarantee that it is recognized.

During state $T_1$ of the associated fetch bus cycle (source synchronized) or store bus cycle (destination synchronized), the IOP outputs the address of the I/O device (the port address). This address must be decoded (by external circuitry) to generate the DMA acknowledge (DACK) signal to the I/O controller as the response to the controller's DMA request. An I/O controller will typically use DACK as a conditional input for the removal of DRQ. (After receipt of the DACK signal, most Intel peripheral controllers deactivate DRQ following receipt of the corresponding read or write signal.) Figures 4-30 and 4-31 illustrate the DRQ/DACK timing for both source synchronized (i.e., port-to-memory) and destination synchronized (i.e., memory-to-port) transfers.

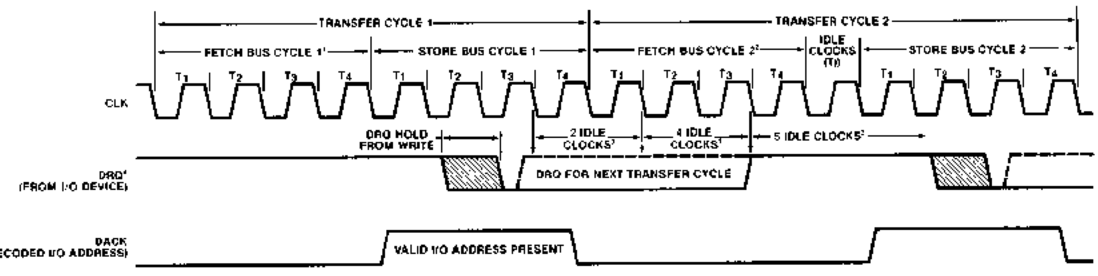
Table 4-19 defines the DMA transfer cycles in terms of the number of bus and clock cycles required. Note that the number of clocks required to complete a transfer cycle does not take into account the effects of possible concurrent operations on the other channel or wait states within any of the bus cycles.



NOTES

1. INDICATES THE NUMBER OF IDLE CLOCK CYCLES INSERTED BEFORE THE NEXT TRANSFER CYCLE BEGINS. IF DRQ IS RECEIVED PRIOR TO STATE $T_4$ OF THE CURRENT FETCH CYCLE, THE NEXT FETCH CYCLE BEGINS IMMEDIATELY FOLLOWING THE CURRENT STORE CYCLE.
2. IF THE 8089 IS IDLE WHEN DRQ IS RECOGNIZED, FIVE IDLE CLOCK CYCLES OCCUR BEFORE THE ASSOCIATED TRANSFER CYCLE IS INITIATED

Figure 4-30. Source Synchronized Transfer Cycle



NOTES:
1. FIRST DMA FETCH CYCLE OCCURS IMMEDIATELY AFTER THE LAST TASK BLOCK INSTRUCTION IS EXECUTED.
2. FETCH BUS CYCLE 2 BEGINS IMMEDIATELY FOLLOWING STORE BUS CYCLE 1.
3. INDICATES THE NUMBER OF IDLE CLOCK CYCLES INSERTED BEFORE STORE BUS CYCLE 2 BEGINS. IF DRQ IS RECEIVED PRIOR TO STATE $T_4$ OF STORE BUS CYCLE 1, STORE BUS CYCLE 2 BEGINS IMMEDIATELY FOLLOWING FETCH BUS CYCLE 2.
4. IF THE 8089 IS IDLE WHEN DRQ IS RECOGNIZED, FIVE IDLE CLOCK CYCLES OCCUR BEFORE THE ASSOCIATED STORE BUS CYCLE IS INITIATED.

Figure 4-31. Destination Synchronized Transfer Cycle

Table 4-19. DMA Transfer Cycles

| Logical Bus Width | | Transfer Mode | | | | | |
|---|---|---|---|---|---|---|---|
| | | Unsynchronized | | Source Synchronized | | Destination Synchronized | |
| Source | Destination | Bus Cycles Required | Total[1] Clocks | Bus Cycles Required | Total[1] Clocks | Bus Cycles Required | Total[1] Clocks |
| 8 | 8 | 2 (1 fetch, 1 store) | 8[2] | 2 (1 fetch, 1 store) | 8[2] | 2 (1 fetch, 1 store) | 8[2] |
| 8 | 16[3] | 3 (2 fetch, 1 store) | 12 | 3 (2 fetch, 1 store) | 16[4] | 3 (2 fetch, 1 store) | 12 |
| 16[3] | 8 | 3 (1 fetch, 2 store) | 12 | 3 (1 fetch, 2 store) | 12 | 3 (1 fetch, 2 store) | 16[4] |
| 16[3] | 16[3] | 2 (1 fetch, 1 store) | 8 | 2 (1 fetch, 1 store) | 8 | 2 (1 fetch, 1 store) | 8 |

Notes:
1. The "Total Clocks Required" does not include wait states. One clock cycle per wait state must be added to each fetch and/or store bus cycle in which a wait state is inserted. When performing a memory-to-memory transfer, three additional clocks must be added to the total clocks required (the first fetch cycle of any memory-to-memory transfer requires seven clock cycles).
2. When performing a translate operation, one additional 7-clock bus cycle must be added to the values specified in the table.
3. Word transfers in the table assume an even address word boundary. Word transfers to or from odd address boundaries are performed as indicated in table 4-18 and are subject to the bus cycle/clock requirements for byte-to-byte transfers.
4. Transfer cycles that include two synchronized bus cycles (i.e., synchronous transfers between dissimilar logical bus widths) insert four idle clock cycles between the two synchronized bus cycles to allow additional time for the synchronzing device to remove its initial DMA request.

DACK latency is defined as the time required for the 8089 to acknowledge, by outputting the device's corresponding port address, a DMA request at its DRQ input. This response latency is dependent on a number of factors including the transfer cycle being performed, activity on the other channel, memory address boundaries, wait states present in either bus cycle and bus arbitration times.

Generally, when the other channel is idle, the maximum DACK latency is five clock cycles (1 microsecond at 5 MHz), excluding wait states and bus arbitration times. An exception occurs when performing a word transfer to or from an odd memory address boundary. This operation, since two store (source synchronized) or two fetch (destination synchronized) bus cycles are required to access memory, has a maximum possible latency of nine clock cycles. When the other channel is performing DMA transfers of equal priority ("P" bits equal), interleaving occurs at bus cycle boundaries, and the maximum latency is either nine clock cycles when the other channel is performing a normal 4-clock fetch or store bus cycle or twelve clock cycles when the other channel is performing the first fetch cycle of a memory-to-memory transfer. If the other channel is performing "chained" task block instruction execution of equal priority, maximum latency can be as high as 12 clock cycles (channel command instruction execution is interrupted at machine cycle boundaries which range from two to eight clock cycles).

## DMA Termination

As stated in Chapter 3, a channel can exit the DMA transfer mode (and return to task block execution) on any of the following terminate conditions:

- Single cycle transfer
- Byte count expired
- Mask/compare match or mismatch
- External event

The terminate conditions are specified by individual fields in the channel control register. More than one terminate condition can be specified for a transfer (e.g., a transfer can be terminated when a specific byte count is reached or on the occurrence of an external event). When more than one terminate condition is possible, displacements (which are added to the task pointer register value) are specified to cause task block execution to resume at a unique entry point for each condition. Three reentry points are available: TP, TP + 4 and TP + 8. The time interval between the occurrence of a terminate condition and the resumption of task block execution is 12 clock cycles for reentry point TP and 15 clock cycles for reentry points TP + 4 and TP + 8.

## Peripheral Interfacing

When interfacing a peripheral to an 8-bit physical data bus, the 8089 uses only the lower half of the address/data lines (AD7-AD0) as the bidirectional data bus, and the upper half of the address/data lines (AD15-AD8) maintain address information for the entire bus cycle. Consequently, with this bus configuration, only one octal latch (e.g., an Intel® 8282/83 Octal Latch) is required since only the lower half of the address/data lines is time-multiplexed (unless the address bus requires the increased current drive capability and capacitive load immunity provided by the latch).

When interfacing a peripheral to a 16-bit data bus, both the lower and upper halves of the address/data lines are time-multipelxed, and two octal latches are required. Note that unlike the 8086 and 8088 CPUs, the 8089 does not time-multiplex $\overline{\text{BHE}}$ (this signal is valid for the entire bus cycle). Both 8- and 16-bit peripherals can be interfaced to a 16-bit bus. An 8-bit peripheral can be connected to either the upper or lower half of the bus. An 8-bit peripheral on the lower half of the bus must use an even source/destination address, and an 8-bit peripheral on the upper half of the bus must use an odd source/destination address. To take advantage of word transfers, a 16-bit peripheral must use an even source/destination address.

To prepare a peripheral device for a DMA transfer, command and parameter data is written to the device's command/status port. This is usually accomplished using pointer register GC. Recalling that the 8089 executes one additional task block instruction following execution of the XFER instruction (the XFER instruction causes the 8089 to enter the DMA mode), this additional instruction is used to access the command port of an I/O device that immediately begins DMA operation on receipt of the last command (the 8271 Floppy Disk Controller begins its DMA transfer on receipt of the last command parameter). Since a translate DMA operation requires the use of all three pointer registers (GA and GB specify the source and destination addresses; GC specifies the base address of the translation table), when it is necessary to use the last task block instruction to start the device, command port access can be accomplished relative to one of the pointer registers or relative to the PP register. If the device's data port address (GA or GB) is below the device's command port address, either an offset or an indexed reference can be used to access the command port.

A peripheral's (or peripheral controller's) DMA communication protocol with the 8089 is as follows:

- The peripheral (when source or destination synchronized) initiates a DMA transfer cycle by activating the 8089's DRQ (DMA request) input.
- The 8089 acknowledges the request by placing the peripheral's assigned data port address on the bus during state $T_1$ of the corresponding fetch (source synchronized) or store (destination synchronized) bus cycle. The peripheral is responsible for decoding this address as the DMA acknowledge (DACK) to its request.
- The data is transferred between the peripheral and the 8089 during the $T_2$ through $T_4$ state interval of the bus cycle. The peripheral must remove its DMA request during this interval.
- The peripheral, when ready, requests another DMA transfer cycle by again activating the DRQ input, and the above sequence is repeated.
- The peripheral can, as an option, end the DMA transfer by activating the 8089's EXT (external terminate) input.

The 8089 can support mulitple peripheral devices on a single channel provided that only one device is in the active transfer mode at any one time. To interface multiple devices, the DMA request (DRQ) lines are OR'ed together as are the external terminate (EXT) lines. Unique port addresses are, however, assigned to each device so that an individual DMA acknowledge (DACK) is returned to only the active device. DACK decoding can be accomplished with an Intel® 8205 Binary Decoder or a ROM circuit. Note that the 8089 can only determine which device has requested service or terminated by the context of the task block program.

Most peripheral devices interfaced to the 8089 will use the decoded DMA acknowledge signal (DACK) as the "chip select" input. Peripheral devices that do not follow this convention must use DACK as a conditional input of chip select.

While most interrupts associated with the 8089 will be DMA requests or external terminates, non-DMA related interrupts can additionally be supported.

One technique that would be used when an 8089 is the local configuration (or when an 8086 or 8088 and an 8089 are locally connected as a remote module) is to allow the CPU to accept the interrupt and then direct the 8089 to the interrupt service routine. Another technique is to allow the 8089 to "poll" the device to determine when an interrupt has occurred (most peripheral controllers have an interrupt pending bit in a status word). The 8089's bit testing instructions are ideally suited for polling.

When the 8089 is in a remote configuration, non-DMA related interrupts can be supported with the addition of an Intel® 8259A Programmable Interrupt Controller. Systems that require this type of interrupt structure would dedicate one of the 8089's channels to interrupt servicing. In implementing this structure, the interrupt output from the 8259A is directly connected to the channel's external terminate (EXT) input, and the channel's DMA request (DRQ) input is not used. A task block program is initially executed to perform a source-synchronized DMA transfer (with an external terminate) on the "interrupt" channel to "arm" the interrupt mechanism. Since the DRQ input is not used, when the channel enters the DMA transfer mode, the channel idles while waiting for the first DMA request (which never occurs). The other channel, since the interrupt channel is idle, operates at maximum throughput. When an interrupt occurs, the "pseudo" DMA transfer is immediately terminated, and task block instruction execution is resumed. The task block program would write a "poll" command to the 8259A's command port and then read the

8259A's data port to acknowledge the interrupt and to determine the device responsible for the interrupt (the device is identified by a 3-bit binary number in the associated data byte). The device number read would be used by the task block program as a vector into a jump table for the device's interrupt service routine. Pertinent interrupt data could be written into the associated parameter block for subsequent examination by the host processor.

The interrupt mechanism previously described, since it uses the 8089's external terminate function, provides an extremely fast interrupt response time.

Note that when using dynamic RAM memory with the 8089, an Intel® 8202 Dynamic RAM Controller can be used to simplify the interface and to perform the RAM refresh cycle. When maximum transfer rates are required, the RAM refresh cycle can be externally initiated by the 8089. By connecting the decoded DACK (DMA acknowledge) signal to the 8202's REFRQ (refresh request) input, the refresh cycle will occur coincident with the I/O device bus cycle and therefore will not impose wait states in the memory bus cycle.

## Instruction Encoding

Most 8089 programming will be performed at the assembly language level using ASM-89, the 8089 assembler. During program debugging, however, it may be necessary to work directly with machine instructions when monitoring the bus, reading unformatted memory dumps, etc. This section contains both a table to encode any ASM-89 instruction into its corresponding machine instruction (table 4-24) and a table to "disassemble" any machine instruction back into its associated assembly language equivalent (table 4-26).

Figure 4-32 shows the format of a typical 8089 machine instruction. Except for the LPDI and memory-to-memory forms of the MOV and MOVB instructions that are six bytes long, all 8089 machine instructions consist of from two to five bytes. The first two bytes are always present and are generally formatted as shown in figure 4-32 (table 4-24 contains the exact encoding of every instuction).

Bits 5 through 7 of the first byte of an instruction comprise the R/B/P field. This field identifies a register, bit select or pointer register operand as outlined in table 4-20.

Table 4-20. R/B/P Field Encoding

| Code | Register | Bit | Pointer |
|---|---|---|---|
| 000 | GA | 0 | GA |
| 001 | GB | 1 | GB |
| 010 | GC | 2 | GC |
| 011 | BC | 3 | N/A |
| 100 | TP | 4 | TP |
| 101 | IX | 5 | N/A |
| 110 | CC | 6 | N/A |
| 111 | MC | 7 | N/A |

The WB field (bits 3 and 4 of the first byte) indicates how many displacement/data bytes are present in the instruction as outlined in table 4-21. The displacement bytes are used in program transfers; one byte is present for short transfers, while long transfers contain a two-byte (word) displacement. As mentioned in Chapter 3, the

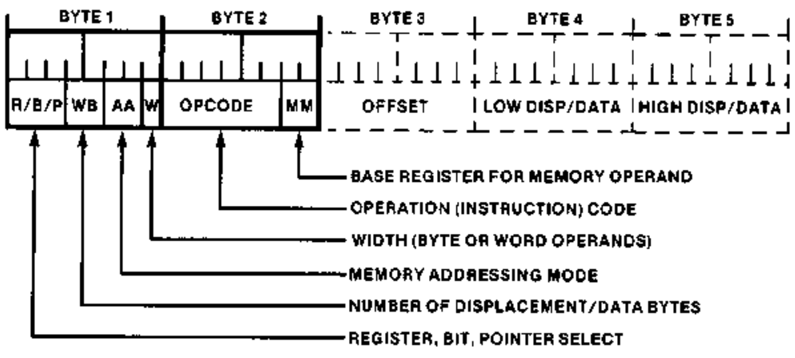

BYTE 1
BYTE 2
BYTE 3
BYTE 4
BYTE 5
R/B/P
WB
AA
W
OPCODE
MM
OFFSET
LOW DISP/DATA
HIGH DISP/DATA
BASE REGISTER FOR MEMORY OPERAND
OPERATION (INSTRUCTION) CODE
WIDTH (BYTE OR WORD OPERANDS)
MEMORY ADDRESSING MODE
NUMBER OF DISPLACEMENT/DATA BYTES
REGISTER, BIT, POINTER SELECT

Figure 4-32. Typical 8089 Machine Instruction Format

displacement is stored in two's complement notation with the high-order bit indicating the sign. Data bytes contain the value of an immediate constant operand. A byte immediate instruction (e.g., MOVBI) will have one data byte, and a word immediate instruction (e.g., ADDI) will have two bytes (a word) of immediate data. An instruction may contain either displacement or data bytes, but not both (the TSL instruction is an exception and contains one byte of displacement and one byte of data). If an offset byte is present, the displacement/data byte(s) always follow the offset byte.

Table 4-21. WB Field Encoding

| Code | Interpretation |
|---|---|
| 00 | No displacement/data bytes |
| 01 | One displacement/data byte |
| 10 | Two displacement/data bytes |
| 11 | TSL instruction only |

The AA field specifies the addressing mode that the processor is to use in order to construct the effective address of a memory operand. Four addressing modes are available as outlined in table 4-22. (Address modes are described in detail in section 3.8.)

Table 4-22. AA Field Encoding

| Code | Interpretation |
|---|---|
| 00 | Base register only |
| 01 | Base register plus offset |
| 10 | Base register plus IX |
| 11 | Base register plus IX, auto-increment |

Bit 0 of the first instruction byte indicates whether the instruction operates on a byte (W=0) or a word (W=1).

Bits 7 through 2 of the second instruction byte specify the instruction opcode. The opcode, in conjunction with the W field of the first byte, identifies the instruction. For example, the opcode "111011" denotes the decrement instruction; if W=0, the assembly language instruction is DECB, while if W=1, the instruction is DEC. Table 4-26 lists, in hexadecimal order, the opcode of every assembly language instruction.

The MM field (bits 0 and 1) indicates which pointer (base) register is to be used to construct the effective address of a memory operand. Table 4-23 defines the MM field encoding. (Memory operand addressing is described in section 3.8.)

Table 4-23. MM Field Encoding

| Code | Base Register |
|---|---|
| 00 | GA |
| 01 | GB |
| 10 | GC |
| 11 | PP |

When the AA field value is "01" (base register + offset addressing), the third byte of the instruction contains the offset value. This unsigned value is added to the content of the base register specified by the MM field to form the effective address of the memory operand.

When the AA field value is "10," the IX register value is added to the content of the base register specified by the MM field to provide a 64k range of effective addresses. (Note that the upper four bits of the IX register are not sign-extended.)

When the AA field value is "11," the IX register value is added to the base register value to form the effective address as described for an AA field value of "10." In this addressing mode, however, the IX register value is incremented by one after every byte accessed.

Table 4-24. 8089 Instruction Encoding

DATA TRANSFER INSTRUCTIONS

| MOV = Move word variable | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 |
|---|---|---|---|---|---|---|
| Memory to register | R R R 0 0 A A 1 | 1 0 0 0 0 0 M M | offset if AA=01 | | | |
| Register to memory | R R R 0 0 A A 1 | 1 0 0 0 0 1 M M | offset if AA=01 | | | |
| Memory to memory | 0 0 0 0 0 A A 1 | 1 0 0 1 0 0 M M | offset if AA=01 | 0 0 0 0 0 A A 1 | 1 1 0 0 1 1 M M | offset if AA=01 |

## Table 4-24. 8089 Instruction Encoding (Cont'd.)

**DATA TRANSFER INSTRUCTIONS (Cont'd.)**

| | 76543210 | 76543210 | 76543210 | 76543210 | 76543210 | 76543210 |
|---|---|---|---|---|---|---|
| **MOVB** = Move byte variable | | | | | | |
| Memory to register | R R R 0 0 A A 0 | 1 0 0 0 0 0 M M | offset if AA=01 | | | |
| Register to memory | R R R 0 0 A A 0 | 1 0 0 0 0 1 M M | offset if AA=01 | | | |
| Memory to memory | 0 0 0 0 0 A A 0 | 1 0 0 1 0 0 M M | offset if AA=01 | 0 0 0 0 0 A A 0 | 1 1 0 0 1 1 M M | offset if AA=01 |
| **MOVBI** = Move byte immediate | | | | | | |
| Immediate to register | R R R 0 1 0 0 0 | 0 0 1 1 0 0 0 0 | data-8 | | | |
| Immediate to memory | 0 0 0 0 1 A A 0 | 0 1 0 0 1 1 M M | offset if AA=01 | data-8 | | |
| **MOVI** = Move word immediate | | | | | | |
| Immediate to register | R R R 1 0 0 0 1 | 0 0 1 1 0 0 0 0 | data-lo | data-hi | | |
| Immediate to memory | 0 0 0 1 0 A A 1 | 0 1 0 0 1 1 M M | offset if AA=01 | data-lo | data-hi | |
| **MOVP** = Move pointer | | | | | | |
| Memory to pointer register | P P P 0 0 A A 1 | 1 0 0 0 1 1 M M | offset if AA=01 | | | |
| Pointer register to memory | P P P 0 0 A A 1 | 1 0 0 1 1 0 M M | offset if AA=01 | | | |
| **LPD** = Load pointer with doubleword variable | P P P 0 0 A A 1 | 1 0 0 0 1 0 M M | offset if AA=01 | | | |
| **LPDI** = Load pointer with doubleword immediate | P P P 1 0 0 0 1 | 0 0 0 0 1 0 0 0 | offset-lo | offset-hi | segment-lo | segment-hi |

**ARITHMETIC INSTRUCTIONS**

| | 76543210 | 76543210 | 76543210 | 76543210 | 76543210 |
|---|---|---|---|---|---|
| **ADD** = Add word variable | | | | | |
| Memory to register | R R R 0 0 A A 1 | 1 0 1 0 0 0 M M | offset if AA=01 | | |
| Register to memory | R R R 0 0 A A 1 | 1 1 0 1 0 0 M M | offset if AA=01 | | |
| **ADDB** = Add byte variable | | | | | |
| Memory to register | R R R 0 0 A A 0 | 1 0 1 0 0 0 M M | offset if AA=01 | | |
| Register to memory | R R R 0 0 A A 0 | 1 1 0 1 0 0 M M | offset if AA=01 | | |
| **ADDI** = Add word immediate | | | | | |
| Immediate to register | R R R 1 0 0 0 1 | 0 0 1 0 0 0 0 0 | data-lo | data-hi | |
| Immediate to memory | 0 0 0 1 0 A A 1 | 1 1 0 0 0 0 M M | offset if AA=01 | data-lo | data-hi |

## Table 4-24. 8089 Instruction Encoding (Cont'd.)

**ARITHMETIC INSTRUCTIONS (Cont'd.)**

| | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 |
|---|---|---|---|---|---|---|
| **ADDBI** = Add byte immediate | | | | | | |
| Immedaite to register | R R R 0 1 0 0 0 | 0 0 1 0 0 0 0 0 | data-8 | | | |
| Immediate to memory | 0 0 0 0 1 A A 0 | 1 1 0 0 0 0 M M | offset if AA=01 | data-8 | | |
| **INC** = Increment word by 1 | | | | | | |
| Register | R R R 0 0 0 0 0 | 0 0 1 1 1 0 0 0 | | | | |
| Memory | 0 0 0 0 0 A A 1 | 1 1 1 0 1 0 M M | offset if AA=01 | | | |
| **INCB** = Increment byte by 1 | 0 0 0 0 0 A A 0 | 1 1 1 0 1 0 M M | offset if AA=01 | | | |
| **DEC** = Decrement word by 1 | | | | | | |
| Register | R R R 0 0 0 0 0 | 0 0 1 1 1 1 0 0 | | | | |
| Memory | 0 0 0 0 0 A A 1 | 1 1 1 0 1 1 M M | offset if AA=01 | | | |
| **DECB** = Decrement byte by 1 | 0 0 0 0 0 A A 0 | 1 1 1 0 1 1 M M | offset if AA=01 | | | |

**LOGICAL AND BIT MANIPULATION INSTRUCTIONS**

| | | | | | |
|---|---|---|---|---|---|
| **AND** = AND word variable | | | | | |
| Memory to register | R R R 0 0 A A 1 | 1 0 1 0 1 0 M M | offset if AA=01 | | |
| Register to memory | R R R 0 0 A A 1 | 1 1 0 1 1 0 M M | offset if AA=01 | | |
| **ANDB** = AND byte variable | | | | | |
| Memory to register | R R R 0 0 A A 0 | 1 0 1 0 1 0 M M | offset if AA=01 | | |
| Register to memory | R R R 0 0 A A 0 | 1 1 0 1 1 0 M M | offset if AA=01 | | |
| **ANDI** = AND word immediate | | | | | |
| Immediate to register | R R R 1 0 0 0 1 | 0 0 1 0 1 0 0 0 | data-lo | data-hi | |
| Immediate to memory | 0 0 0 1 0 A A 1 | 1 1 0 0 1 0 M M | offset if AA=01 | data-lo | data-hi |
| **ANDBI** = AND byte immediate | | | | | |
| Immediate to register | R R R 0 1 0 0 0 | 0 0 1 0 1 0 0 0 | data-8 | | |
| Immediate to memory | 0 0 0 0 1 A A 0 | 1 1 0 0 1 0 M M | offset if AA=01 | data-8 | |
| **OR** = OR word variable | | | | | |
| Memory to register | R R R 0 0 A A 1 | 1 0 1 0 0 1 M M | offset if AA=01 | | |
| Register to memory | R R R 0 0 A A 1 | 1 1 0 1 0 1 M M | offset if AA=01 | | |

## Table 4-24. 8089 Instruction Encoding (Cont'd.)

**LOGICAL AND BIT MANIPULATION INSTRUCTIONS** (Cont'd.)

| | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 |
|---|---|---|---|---|---|---|
| **ORB** = OR byte variable | | | | | | |
| Memory to register | R R R 0 0 A A 0 | 1 0 1 0 0 1 M M | offset if AA=01 | | | |
| Register to memory | R R R 0 0 A A 0 | 1 1 0 1 0 1 M M | offset if AA=01 | | | |
| **ORI** = OR word immediate | | | | | | |
| Immediate to register | R R R 1 0 0 0 1 | 0 0 1 0 0 1 0 0 | data-lo | data-hi | | |
| Immediate to memory | 0 0 0 1 0 A A 1 | 1 1 0 0 0 1 M M | offset if AA=01 | data-lo | data-hi | |
| **ORBI** = OR byte immediate | | | | | | |
| Immediate to register | R R R 0 1 0 0 0 | 0 0 1 0 0 1 0 0 | data-8 | | | |
| Immediate to memory | 0 0 0 0 1 A A 0 | 1 1 0 0 0 1 M M | offset if AA=01 | data-8 | | |
| **NOT** = NOT word variable | | | | | | |
| Register | R R R 0 0 0 0 0 | 0 0 1 0 1 1 0 0 | | | | |
| Memory | 0 0 0 0 0 A A 1 | 1 1 0 1 1 1 M M | offset if AA=01 | | | |
| Memory to register | R R R 0 0 A A 1 | 1 0 1 0 1 1 M M | offset if AA=01 | | | |
| **NOTB** = NOT byte variable | | | | | | |
| Memory | 0 0 0 0 0 A A 0 | 1 1 0 1 1 1 M M | offset if AA=01 | | | |
| Memory to register | R R R 0 0 A A 0 | 1 0 1 0 1 1 M M | offset if AA=01 | | | |
| **SETB** = Set bit to 1 | B B B 0 0 A A 0 | 1 1 1 1 0 1 M M | offset if AA=01 | | | |
| **CLR** = Clear bit to 0 | B B B 0 0 A A 0 | 1 1 1 1 1 0 M M | offset if AA=01 | | | |

**PROGRAM TRANSFER INSTRUCTIONS**

| | | | | | |
|---|---|---|---|---|---|
| ***CALL** = Call | 1 0 0 0 1 A A 1 | 1 0 0 1 1 1 M M | offset if AA=01 | disp-8 | |
| **LCALL** = Long call | 1 0 0 1 0 A A 1 | 1 0 0 1 1 1 M M | offset if AA=01 | disp-lo | disp-hi |
| ***JMP** = Jump unconditional | 1 0 0 0 1 0 0 0 | 0 0 1 0 0 0 0 0 | disp-8 | | |
| **LJMP** = Long jump unconditional | 1 0 0 1 0 0 0 1 | 0 0 1 0 0 0 0 0 | disp-lo | disp-hi | |

*The ASM-89 Assembler will automatically generate the long form of a program transfer instruction when the target is known to be beyond the byte-displacement range.

## Table 4-24. 8089 Instruction Encoding (Cont'd.)

**PROGRAM TRANSFER INSTRUCTIONS (Cont'd.)**

| | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 |
|---|---|---|---|---|---|---|
| ***JZ** = Jump if word is 0 | | | | | | |
| Label to register | R R R 0 1 0 0 0 | 0 1 0 0 0 1 0 0 | disp-8 | | | |
| Label to memory | 0 0 0 0 1 A A 1 | 1 1 1 0 0 1 M M | offset if AA=01 | disp-8 | | |
| **LJZ** = Long jump if word is 0 | | | | | | |
| Label to register | R R R 1 0 0 0 0 | 0 1 0 0 0 1 0 0 | disp-lo | disp-hi | | |
| Label to memory | 0 0 0 1 0 A A 1 | 1 1 1 0 0 1 M M | offset if AA=01 | disp-lo | disp-hi | |
| ***JZB** = Jump if byte is 0 | 0 0 0 0 1 A A 0 | 1 1 1 0 0 1 M M | offset if AA=01 | disp-8 | | |
| **LJZB** = Long jump if byte is 0 | 0 0 0 1 0 A A 0 | 1 1 1 0 0 1 M M | offset if AA=01 | disp-lo | disp-hi | |
| ***JNZ** = Jump if word not 0 | | | | | | |
| Label to register | R R R 0 1 0 0 0 | 0 1 0 0 0 0 0 0 | disp-8 | | | |
| Label to memory | 0 0 0 0 1 A A 1 | 1 1 1 0 0 0 M M | offset if AA=01 | disp-8 | | |
| **LJNZ** = Long jump if word not 0 | | | | | | |
| Label to register | R R R 1 0 0 0 0 | 0 1 0 0 0 0 0 0 | disp-lo | disp-hi | | |
| Label to memory | 0 0 0 1 0 A A 1 | 1 1 1 0 0 0 M M | offset if AA=01 | disp-lo | disp-hi | |
| ***JNZB** = Jump if byte not 0 | 0 0 0 0 1 A A 0 | 1 1 1 0 0 0 M M | offset if AA=01 | disp-8 | | |
| **LJNZB** = Long jump if byte not 0 | 0 0 0 1 0 A A 0 | 1 1 1 0 0 0 M M | offset if AA=01 | disp-lo | disp-hi | |
| ***JMCE** = Jump if masked compare equal | 0 0 0 0 1 A A 0 | 1 0 1 1 0 0 M M | offset if AA=01 | disp-8 | | |
| **LJMCE** = Long jump if masked compare equal | 0 0 0 1 0 A A 0 | 1 0 1 1 0 0 M M | offset if AA=01 | disp-lo | disp-hi | |
| ***JMCNE** = Jump if masked compare not equal | 0 0 0 0 1 A A 0 | 1 0 1 1 0 1 M M | offset if AA=01 | disp-8 | | |
| **LJMCNE** = Long jump if masked compare not equal | 0 0 0 1 0 A A 0 | 1 0 1 1 0 1 M M | offset if AA=01 | disp-lo | disp-hi | |
| ***JBT** = Jump if bit is 1 | B B B 0 1 A A 0 | 1 0 1 1 1 1 M M | offset if AA=01 | disp-8 | | |

*The ASM-89 Assembler will automatically generate the long form of a program transfer instruction when the target is known to be beyond the byte-displacement range.

Table 4-24. 8089 Instruction Encoding (Cont'd.)

**PROGRAM TRANSFER INSTRUCTIONS (Cont'd.)**

| | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 |
|---|---|---|---|---|---|---|
| **LJBT** = Long jump if bit is 1 | B B B 1 0 A A 0 | 1 0 1 1 1 1 M M | offset if AA=01 | disp-lo | disp-hi | |
| ***JNBT** = Jump if bit is not 1 | B B B 0 1 A A 0 | 1 0 1 1 1 0 M M | offset if AA=01 | disp-8 | | |
| **LJNBT** = Long jump if bit is not 1 | B B B 1 0 A A 0 | 1 0 1 1 1 0 M M | offset if AA=01 | disp-lo | disp-hi | |

**PROCESSOR CONTROL INSTRUCTIONS**

| | | | | | |
|---|---|---|---|---|---|
| **TSL** = Test and set while locked | 0 0 0 1 1 A A 0 | 1 0 0 1 0 1 M M | offset if AA=01 | data-8 | disp-8 |
| **WID** = Set logical bus widths | 1 S D* 0 0 0 0 0 | 0 0 0 0 0 0 0 0 | | | |

*S=source width, D=destination width, 0=8 bits, 1=16 bits

| | | |
|---|---|---|
| **XFER** = Enter DMA mode | 0 1 1 0 0 0 0 0 | 0 0 0 0 0 0 0 0 |
| **SINTR** = Set interrupt service bit | 0 1 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 |
| **HLT** = Halt channel program | 0 0 1 0 0 0 0 0 | 0 1 0 0 1 0 0 0 |
| **NOP** = No operation | 0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 |

*The ASM-89 Assembler will automatically generate the long form of a program transfer instruction when the target is known to be beyond the byte-displacement range

Table 4-26 lists all of the 8089 machine instructions in hexadecimal/binary order by their *second* byte. This table may be used to "decode" an assembled machine instruction into its ASM-89 symbolic form. The preceding table (table 4-25) defines the notation used in table 4-26.

Table 4-25. Key to 8089 Machine Instruction Decoding Guide

| Identifier | Explanation |
|---|---|
| S | Logical width of source bus; 0=8, 1=16 |
| D | Logical width of destination bus; 0=8, 1=16 |
| PPP | Pointer register encoded in R/B/P field |
| RRR | Register encoded in R/B/P field |
| AA | AA (addressing mode) field |
| BBB | Bit select encoded in R/B/P field |
| offset-lo | Low-order byte of offset word in doubleword pointer |
| offset-hi | High-order byte of offset word in doubleword pointer |
| segment-lo | Low-order byte of segment word in doubleword pointer |
| segment-hi | High-order byte of segment word in doubleword pointer |
| data-8 | 8-bit immediate constant |
| data-lo | Low-order byte of 16-bit immediate constant |
| data-hi | High-order byte of 16-bit immediate constant |
| disp-8 | 8-bit signed displacement |
| disp-lo | Low-order byte of 16-bit signed displacement |
| disp-hi | High-order byte of 16-bit signed displacement |
| (offset) | Optional 8-bit offset used in offset addressing |

Table 4-26. 8089 Machine Instruction Decoding Guide

| Byte 1 | Byte 2 | | Bytes 3, 4, 5, 6 | ASM89 Instruction Format |
|---|---|---|---|---|
| | Hex | Binary | | |
| 00000000 | 00 | 00000000 | | NOP |
| 01000000 | 00 | 00000000 | | SINTR |
| 1SD00000 | 00 | 00000000 | | WID source-width,dest-width |
| 01100000 | 00 | 00000000 | | XFER |
| | 01 | 00000001 | | |
| | ↓ | ↓ | | not used |
| | 07 | 00000111 | | |
| PPP10001 | 08 | 00001000 | offset-lo,offset-hi,segment-lo,segment-hi | LPDI ptr-reg,immed32 |
| | 09 | 00001001 | | |
| | ↓ | ↓ | | not used |
| | 1F | 00011111 | | |
| RRR01000 | 20 | 00100000 | data-8 | ADDBI register,immed8 |
| RRR10001 | 20 | 00100000 | data-lo,data-hi | ADDI register,immed16 |
| 10001000 | 20 | 00100000 | disp-8 | JMP short-label |
| 10010001 | 20 | 00100000 | disp-lo,disp-hi | LJMP long-label |
| | 21 | 00100001 | | |
| | ↓ | ↓ | | not used |
| | 23 | 00100011 | | |
| RRR01000 | 24 | 00100100 | data-8 | ORBI register,immed8 |
| RRR10001 | 24 | 00100100 | data-lo,data-hi | ORI register,immed16 |
| | 25 | 00100101 | | |
| | ↓ | ↓ | | not used |
| | 27 | 00100111 | | |
| RRR01000 | 28 | 00101000 | data-8 | ANDBI register,immed8 |

Table 4-26. 8089 Machine Instruction Decoding Guide (Cont'd.

| Byte 1 | Byte 2 | | Bytes 3, 4, 5, 6 | ASM89 Instruction Format |
|---|---|---|---|---|
| | Hex | Binary | | |
| RRR10001 | 28 | 00101000 | data-lo,data-hi | ANDI register,immed16 |
| | 29 | 00101001 | | not used |
| | ↓ | ↓ | | |
| | 2B | 00101011 | | |
| RRR00000 | 2C | 00101100 | | NOT register |
| | 2D | 00101101 | | not used |
| | ↓ | ↓ | | |
| | 2F | 00101111 | | |
| RRR01000 | 30 | 00110000 | data-8 | MOVBI register,immed8 |
| RRR10001 | 30 | 00110000 | data-lo,data-hi | MOVI register,immed16 |
| | 31 | 00110001 | | not used |
| | ↓ | ↓ | | |
| | 37 | 00110111 | | |
| RRR00000 | 38 | 00111000 | | INC register |
| | 39 | 00111001 | | not used |
| | ↓ | ↓ | | |
| | 3B | 00111011 | | |
| RRR00000 | 3C | 00111100 | | DEC register |
| | 3D | 00111101 | | not used |
| | ↓ | ↓ | | |
| | 3F | 00111111 | | |
| RRR01000 | 40 | 01000000 | disp-8 | JNZ register,short-label |
| RRR10000 | 40 | 01000000 | disp-lo,disp-hi | LJNZ register,long-label |
| | 41 | 01000001 | | not used |
| | ↓ | ↓ | | |
| | 43 | 01000011 | | |
| RRR01000 | 44 | 01000100 | disp-8 | JZ register,short-label |
| RRR10000 | 44 | 01000100 | disp-lo,disp-hi | LJZ register,short-label |
| | 45 | 01000101 | | not used |
| | ↓ | ↓ | | |
| | 47 | 01000111 | | |
| 00100000 | 48 | 01001000 | | HLT |
| | 49 | 01001001 | | not used |
| | ↓ | ↓ | | |
| | 4B | 01001011 | | |
| 00001AA0 | 4C | 010011MM | (offset),data-8 | MOVBI mem8,immed8 |
| ↓ | ↓ | ↓ | | |
| 00001AA0 | 4F | 010011MM | | |
| 00010AA1 | 4C | 010011MM | (offset),data-lo,data-hi | MOVI mem16,immed16 |
| ↓ | ↓ | ↓ | | |
| 00010AA1 | 4F | 010011MM | | |
| | 50 | 01010000 | | not used |
| | ↓ | ↓ | | |
| | 7F | 01111111 | | |
| RRR00AA0 | 80 | 100000MM | (offset) | MOVB register,mem8 |
| ↓ | ↓ | ↓ | | |
| RRR00AA0 | 83 | 100000MM | | |

Table 4-26. 8089 Machine Instruction Decoding Guide (Cont'd.

| Byte 1 | Byte 2 | | Bytes 3, 4, 5, 6 | ASM89 Instruction Format |
|---|---|---|---|---|
| | Hex | Binary | | |
| RRR00AA1 ↓ RRR00AA1 | 80 ↓ 83 | 100000MM ↓ 100000MM | (offset) | MOV register,mem16 |
| RRR00AA0 ↓ RRR00AA0 | 84 ↓ 87 | 100001MM ↓ 100001MM | (offset) | MOVB mem8,register |
| RRR00AA1 ↓ RRR00AA1 | 84 ↓ 87 | 100001MM ↓ 100001MM | (offset) | MOV mem16,register |
| PPP00AA1 ↓ PPP00AA1 | 88 ↓ 8B | 100010MM ↓ 100010MM | (offset) | LPD ptr-reg,mem32 |
| PPP00AA1 ↓ PPP00AA1 | 8C ↓ 8F | 100011MM ↓ 100011MM | (offset) | MOVP ptr-reg,mem24 |
| 00000AA0 ↓ 00000AA0 | 90 ↓ 93 | 100100MM ↓ 100100MM | (offset),00000AA0,110011MM,(offset) | MOVB mem8,mem8 |
| 00000AA1 ↓ 00000AA1 | 90 ↓ 93 | 100100MM ↓ 100100MM | (offset),00000AA1,110011MM,(offset) | MOV mem16,mem16 |
| 00011AA0 ↓ 00011AA0 | 94 ↓ 97 | 100101MM ↓ 100101MM | (offset),data-8,disp-8 | TSL mem8,immed8,short-label |
| PPP00AA1 ↓ PPP00AA1 | 98 ↓ 9B | 100110MM ↓ 100110MM | (offset) | MOVP mem24,ptr-reg |
| 10001AA1 ↓ 10001AA1 | 9C ↓ 9F | 100111MM ↓ 100111MM | (offset),disp-8 | CALL mem24,short-label |
| 10010AA1 ↓ 10010AA1 | 9C ↓ 9F | 100111MM ↓ 100111MM | (offset),disp-lo,disp-hi | LCALL mem24,long-label |
| RRR00AA0 ↓ RRR00AA0 | A0 ↓ A3 | 101000MM ↓ 101000MM | (offset) | ADDB register,mem8 |
| RRR00AA1 ↓ RRR00AA1 | A0 ↓ A3 | 101000MM ↓ 101000MM | (offset) | ADD register,mem16 |
| RRR00AA0 ↓ RRR00AA0 | A4 ↓ A7 | 101001MM ↓ 101001MM | (offset) | ORB register,mem8 |
| RRR00AA1 ↓ RRR00AA1 | A4 ↓ A7 | 101001MM ↓ 101001MM | (offset) | OR register,mem16 |
| RRR00AA0 ↓ RRR00AA0 | A8 ↓ AB | 101010MM ↓ 101010MM | (offset) | ANDB mem8,register |

Table 4-26. 8089 Machine Instruction Decoding Guide (Cont'd.

| Byte 1 | Byte 2 | | Bytes 3, 4, 5, 6 | ASM89 Instruction Format |
|---|---|---|---|---|
| | Hex | Binary | | |
| RRR00AA1<br>↓<br>RRR00AA1 | A8<br>↓<br>AB | 101010MM<br>↓<br>101010MM | (offset) | AND mem16,register |
| RRR00AA0<br>↓<br>RRR00AA0 | AC<br>↓<br>AF | 101011MM<br>↓<br>101011MM | (offset) | NOTB register,mem8 |
| RRR00AA1<br>↓<br>RRR00AA1 | AC<br>↓<br>AF | 101011MM<br>↓<br>101011MM | (offset) | NOT register,mem16 |
| 00001AA0<br>↓<br>00001AA0 | B0<br>↓<br>B3 | 101100MM<br>↓<br>101100MM | (offset),disp-8 | JMCE mem8,short-label |
| 00010AA0<br>↓<br>00010AA0 | B0<br>↓<br>B3 | 101100MM<br>↓<br>101100MM | (offset),disp-lo,disp-hi | LJMCE mem8,long-label |
| 00001AA0<br>↓<br>00001AA0 | B4<br>↓<br>B7 | 101101MM<br>↓<br>101101MM | (offset),disp-8 | JMCNE mem8,short-label |
| 00010AA0<br>↓<br>00010AA0 | B4<br>↓<br>B7 | 101101MM<br>↓<br>101101MM | (offset),disp-lo,disp-hi | LJMCNE mem8,long-label |
| BBB01AA0<br>↓<br>BBB01AA0 | B8<br>↓<br>BB | 101110MM<br>↓<br>101110MM | (offset),disp-8 | JNBT mem8,bit-select,short-label |
| BBB10AA0<br>↓<br>BBB10AA0 | B8<br>↓<br>BB | 101110MM<br>↓<br>101110MM | (offset),disp-lo,disp-hi | LJNBT mem8,bit-select,long-label |
| BBB01AA0<br>↓<br>BBB01AA0 | BC<br>↓<br>BF | 101111MM<br>↓<br>101111MM | (offset),disp-8 | JBT mem8,bit-select,short-label |
| BBB10AA0<br>↓<br>BBB10AA0 | BC<br>↓<br>BF | 101111MM<br>↓<br>101111MM | (offset),disp-lo,disp-hi | LJBT mem8,bit-select,long-label |
| 00001AA0<br>↓<br>00001AA0 | C0<br>↓<br>C3 | 110000MM<br>↓<br>110000MM | (offset),data-8 | ADDBI mem8,immed8 |
| 00010AA1<br>↓<br>00010AA1 | C0<br>↓<br>C3 | 110000MM<br>↓<br>110000MM | (offset),data-lo,data-hi | ADDI mem16,immed16 |
| 00001AA0<br>↓<br>00001AA0 | C4<br>↓<br>C7 | 110001MM<br>↓<br>110001MM | (offset),data-8 | ORBI mem8,immed8 |
| 00010AA1<br>↓<br>00010AA1 | C4<br>↓<br>C7 | 110001MM<br>↓<br>110001MM | (offset),data-lo,data-hi | ORI mem16,immed16 |
| 00001AA0<br>↓<br>00001AA0 | C8<br>↓<br>CB | 110010MM<br>↓<br>110010MM | (offset),data-8 | ANDBI mem8,immed8 |

Table 4-26. 8089 Machine Instruction Decoding Guide (Cont'd.

| Byte 1 | Byte 2 | | Bytes 3, 4, 5, 6 | ASM89 Instruction Format |
|---|---|---|---|---|
| | Hex | Binary | | |
| 00010AA1 | C8 | 110010MM | | |
| ↓ | ↓ | ↓ | (offset),data-lo,data-hi | ANDI mem16,immed16 |
| 00010AA1 | CB | 110010MM | | |
| | CC | 11001100 | | |
| | ↓ | ↓ | | not used |
| | CF | 11001111 | | |
| RRR00AA0 | D0 | 110100MM | | |
| ↓ | ↓ | ↓ | (offset) | ADDB mem8,register |
| RRR00AA0 | D3 | 110100MM | | |
| RRR00AA1 | D0 | 110100MM | | |
| ↓ | ↓ | ↓ | (offset) | ADD mem16,register |
| RRR00AA1 | D3 | 110100MM | | |
| RRR00AA0 | D4 | 110101MM | | |
| ↓ | ↓ | ↓ | (offset) | ORB mem8,register |
| RRR00AA0 | D7 | 110101MM | | |
| RRR00AA1 | D4 | 110101MM | | |
| ↓ | ↓ | ↓ | (offset) | OR mem16,register |
| RRR00AA1 | D7 | 110101MM | | |
| RRR00AA0 | D8 | 110110MM | | |
| ↓ | ↓ | ↓ | (offset) | ANDB mem8,register |
| RRR00AA0 | DB | 110110MM | | |
| RRR00AA1 | D8 | 110110MM | | |
| ↓ | ↓ | ↓ | (offset) | AND mem16,register |
| RRR00AA1 | DB | 110110MM | | |
| RRR00AA0 | DC | 110111MM | | |
| ↓ | ↓ | ↓ | (offset) | NOTB mem8,register |
| RRR00AA0 | DF | 110111MM | | |
| RRR00AA1 | DC | 110111MM | | |
| ↓ | ↓ | ↓ | (offset) | NOT mem16,register |
| RRR00AA1 | DF | 110111MM | | |
| 00001AA0 | E0 | 111000MM | | |
| ↓ | ↓ | ↓ | (offset),disp-8 | JNZB mem8,short-label |
| 00001AA0 | E3 | 111000MM | | |
| 00001AA1 | E0 | 111000MM | | |
| ↓ | ↓ | ↓ | (offset),disp-8 | JNZ mem16,short-label |
| 00001AA1 | E3 | 111000MM | | |
| 00010AA0 | E0 | 111000MM | | |
| ↓ | ↓ | ↓ | (offset),disp-lo,disp-hi | LJNZB mem8,long-label |
| 00010AA0 | E3 | 111000MM | | |
| 00010AA1 | E0 | 111000MM | | |
| ↓ | ↓ | ↓ | (offset),disp-lo,disp-hi | LJNZ mem16,longlabel |
| 00010AA1 | E3 | 111000MM | | |
| 00001AA0 | E4 | 111001MM | | |
| ↓ | ↓ | ↓ | (offset),disp-8 | JZB mem8,short-label |
| 00001AA0 | E7 | 111001MM | | |
| 00001AA1 | E4 | 111001MM | | |
| ↓ | ↓ | ↓ | (offset),disp-8 | JZ mem16,short-label |
| 00001AA1 | E7 | 111001MM | | |

Table 4-26. 8089 Machine Instruction Decoding Guide (Cont'd.

| Byte 1 | Byte 2 | | Bytes 3, 4, 5, 6 | ASM89 Instruction Format |
|---|---|---|---|---|
| | Hex | Binary | | |
| 00010AA0 | E4 | 111001MM | | |
| ↓ | ↓ | ↓ | (offset),disp-lo,disp-hi | LJZB mem8,long-label |
| 00010AA0 | E7 | 111001MM | | |
| 00010AA1 | E4 | 111001MM | | |
| ↓ | ↓ | ↓ | (offset),disp-lo,disp-hi | LJZ mem16,long-label |
| 00010AA1 | E7 | 111001MM | | |
| 00000AA0 | E8 | 111010MM | | |
| ↓ | ↓ | ↓ | (offset) | INCB mem8 |
| 00000AA0 | EB | 111010MM | | |
| 00000AA1 | E8 | 111010MM | | |
| ↓ | ↓ | ↓ | (offset) | INC mem16 |
| 00000AA1 | EB | 111010MM | | |
| 00000AA0 | EC | 111011MM | | |
| ↓ | ↓ | ↓ | (offset) | DECB mem8 |
| 00000AA0 | EF | 111011MM | | |
| 00000AA1 | EC | 111011MM | | |
| ↓ | ↓ | ↓ | (offset) | DEC mem16 |
| 00000AA1 | EF | 111011MM | | |
| | F0 | 11110000 | | |
| | ↓ | ↓ | | not used |
| | F3 | 11110000 | | |
| BBB00AA0 | F4 | 111101MM | | |
| ↓ | ↓ | ↓ | (offset) | SETB mem8,0-7 |
| BBB00AA0 | F7 | 111101MM | | |
| BBB00AA0 | F8 | 111110MM | | |
| ↓ | ↓ | ↓ | (offset) | CLR mem8,0-7 |
| BBB00AA0 | FB | 111110MM | | |
| | FC | 11111100 | | |
| | ↓ | ↓ | | not used |
| | FF | 11111111 | | |