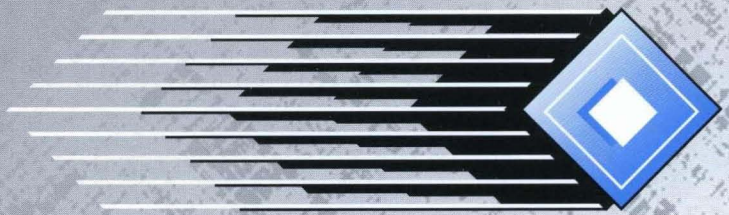
intel 80C186EB/80C188EB Microprocessor User's Manual

intel

80C186EB/80C188EB Microprocessor User's Manual



intel®

80C186EB/80C188EB Microprocessor User's Manual

February 1995

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

Intel Corporation
Literature Sales
P.O. Box 7641
Mt. Prospect, IL 60056-7641

or call 1-800-879-4683

intel

CONTENTS

FIGURES

FIGURES

FIGURES

FIGURES

TABLES

TABLES

EXAMPLES

intel®

1

Introduction

intel

CHAPTER 1
INTRODUCTION

The 8086 microprocessor was first introduced in 1978 and gained rapid support as the microcomputer engine of choice. There are literally millions of 8086/8088-based systems in the world today. The amount of software written for the 8086/8088 is rivaled by no other architecture.

By the early 1980's, however, it was clear that a replacement for the 8086/8088 was necessary. An 8086/8088 system required dozens of support chips to implement even a moderately complex design. Intel recognized the need to integrate commonly used system peripherals onto the same silicon die as the CPU. In 1982 Intel addressed this need by introducing the 80186/80188 family of embedded microprocessors. The original 80186/80188 integrated an enhanced 8086/8088 CPU with six commonly used system peripherals. A parallel effort within Intel also gave rise to the 80286 microprocessor in 1982. The 80286 began the trend toward the very high performance Intel architecture that today includes the Intel386™, Intel486™ and Pentium™ microprocessors.

As technology advanced and turned toward small geometry CMOS processes, it became clear that a new 80186 was needed. In 1987 Intel announced the second generation of the 80186 family: the 80C186/C188. The 80C186 family is pin compatible with the 80186 family, while adding an enhanced feature set. The high-performance CHMOS III process allowed the 80C186 to run at twice the clock rate of the NMOS 80186, while consuming less than one-fourth the power.

The 80186 family took another major step in 1990 with the introduction of the 80C186EB family. The 80C186EB heralded many changes for the 80186 family. First, the enhanced 8086/8088 CPU was redesigned as a static, stand-alone module known as the 80C186 Modular Core. Second, the 80186 family peripherals were also redesigned as static modules with standard interfaces. The goal behind this redesign effort was to give Intel the capability to proliferate the 80186 family rapidly, in order to provide solutions for an even wider range of customer applications.

The 80C186EB/C188EB was the first product to use the new modular capability. The 80C186EB/C188EB includes a different peripheral set than the original 80186 family. Power consumption was dramatically reduced as a direct result of the static design, power management features and advanced CHMOS IV process. The 80C186EB/C188EB has found acceptance in a wide array of portable equipment ranging from cellular phones to personal organizers.

In 1991 the 80C186 Modular Core family was again extended with the introduction of three new products: the 80C186XL, the 80C186EA and the 80C186EC. The 80C186XL/C188XL is a higher performance, lower power replacement for the 80C186/C188. The 80C186EA/C188EA combines the feature set of the 80C186 with new power management features for power-critical applications. The 80C186EC/C188EC offers the highest level of integration of any of the 80C186 Modular Core family products, with 14 on-chip peripherals (see Table 1-1).

The 80C186 Modular Core family is the direct result of ten years of Intel development. It offers the designer the peace of mind of a well-established architecture with the benefits of state-of-the-art technology.

Table 1-1. Comparison of 80C186 Modular Core Family Products

Feature	80C186XL	80C186EA	80C186EB	80C186EC
Enhanced 8086 Instruction Set				
Low-Power Static Modular CPU				
Power-Save (Clock Divide) Mode				
Powerdown and Idle Modes				
80C187 Interface				
ONCE Mode				
Interrupt Control Unit				8259 Compatible
Timer/Counter Unit				
Chip-Select Unit			Enhanced	Enhanced
DMA Unit	2 Channel	2 Channel		4 Channel
Serial Communications Unit				
Refresh Control Unit			Enhanced	Enhanced
Watchdog Timer Unit				
I/O Ports			16 Total	22 Total

1.1 HOW TO USE THIS MANUAL

This manual uses phrases such as *80C186 Modular Core Family* or *80C188 Modular Core*, as well as references to specific products such as *80C188EA*. Each phrase refers to a specific set of 80C186 family products. The phrases and the products they refer to are as follows:

80C186 Modular Core Family: This phrase refers to any device that uses the modular 80C186/C188 CPU core architecture. At this time these include the 80C186EA/C188EA, 80C186EB/C188EB, 80C186EC/C188EC and 80C186XL/C188XL.

80C186 Modular Core: Without the word *family*, this phrase refers only to the 16-bit bus members of the 80C186 Modular Core Family.

80C188 Modular Core: This phrase refers to the 8-bit bus products.

80C188EC: A specific product reference refers only to the named device. For example, *On the 80C188EC...* refers strictly to the 80C188EC and not to any other device.

Each chapter covers a specific section of the device, beginning with the CPU core. Each peripheral chapter includes programming examples intended to aid in your understanding of device operation. Please read the comments carefully, as not all of the examples include all the code necessary for a specific application.

This user's guide is a supplement to the device data sheet. Specific timing values are not discussed in this guide. When designing a system, always consult the most recent version of the device data sheet for up-to-date specifications.

1.2 RELATED DOCUMENTS

The following table lists documents and software that are useful in designing systems that incorporate the 80C186 Modular Core Family. These documents are available through Intel Literature. In the U.S. and Canada, call 1-800-548-4725 to order. In Europe and other international locations, please contact your local Intel sales office or distributor.

NOTE

If you will be transferring a design from the 80186/80188 or 80C186/80C188 to the 80C186XL/80C188XL, refer to FaxBack Document No. 2132.

Table 1-2. Related Documents and Software

Document/Software Title	Document Order No.
Embedded Microprocessors (includes 186 family data sheets)	272396
186 Embedded Microprocessor Line Card	272079
80186/80188 High-Integration 16-Bit Microprocessor Data Sheet	272430
80C186XL/C188XL-20, -12 16-Bit High-Integration Embedded Microprocessor Data Sheet	272431
80C186EA/80C188EA-20, -12 and 80L186EA/80L188EA-13, -8 (low power versions) 16-Bit High-Integration Embedded Microprocessor Data Sheet	272432
80C186EB/80C188EB-20, -13 and 80L186EB/80L188EB-13, -8 (low power versions) 16-Bit High-Integration Embedded Microprocessor Data Sheet	272433
80C186EC/80C188EC-20, -13 and 80L186EC/80L188EC-13, -8 (low power versions) 16-Bit High-Integration Embedded Microprocessor Data Sheet	272434
80C187 80-Bit Math Coprocessor Data Sheet	270640
Low Voltage Embedded Design	272324
80C186/C188, 80C186XL/C188XL Microprocessor User's Manual	272164
80C186EA/80C188EA Microprocessor User's Manual	270950
80C186EB/80C188EB Microprocessor User's Manual	270830
80C186EC/80C188EC Microprocessor User's Manual	272047
8086/8088/8087/80186/80188 Programmer's Pocket Reference Guide	231017

Table 1-2. Related Documents and Software (Continued)

Document/Software Title	Document Order No.
8086/8088 User's Manual Programmer's and Hardware Reference Manual	240487
*Ap*BUILDER Software	272216
80C186EA Hypertext Manual	272275
80C186EB Hypertext Manual	272296
80C186EC Hypertext Manual	272298
80C186XL Hypertext Manual	272630
ZCON - Z80 Code Converter	Available on BBS

1.3 CUSTOMER SERVICE

This section provides telephone numbers and describes various customer services.

- Customer Support (U.S. and Canada) 800-628-8686
- Customer Training (U.S. and Canada) 800-234-8806
- Literature Fulfillment
  - 800-548-4725 (U.S. and Canada)
  - +44(0)793-431155 (Europe)
- FaxBack* Service
  - 800-628-2283 (U.S. and Canada)
  - +44(0)793-496646 (Europe)
  - 916-356-3105 (worldwide)
- Application Bulletin Board System
  - 916-356-3600 (worldwide, up to 14.4-Kbaud line)
  - 916-356-7209 (worldwide, dedicated 2400-baud line)
  - +44(0)793-496340 (Europe)

Intel provides 24-hour automated technical support through the use of our FaxBack service and our centralized Intel Application Bulletin Board System (BBS). The FaxBack service is a simple-to-use information system that lets you order technical documents by phone for immediate delivery to your fax machine. The BBS is a centralized computer bulletin board system that provides updated application-specific information about Intel products.

1.3.1 How to Use Intel's FaxBack Service

Think of the FaxBack service as a library of technical documents that you can access with your phone. Just dial the telephone number (see page 1-4) and respond to the system prompts. After you select a document, the system sends a copy to your fax machine.

Each document is assigned an order number and is listed in a subject catalog. First-time users should order the appropriate subject catalogs to get a complete listing of document order numbers.

The following catalogs and information packets are available:

1. Microcontroller, Flash, and iPLD catalog
2. Development tool catalog
3. System catalog
4. DVI and multimedia catalog
5. BBS catalog
6. Microprocessor and peripheral catalog
7. Quality and reliability catalog
8. Technical questionnaire

1.3.2 How to Use Intel's Application BBS

The Application Bulletin Board System (BBS) provides centralized access to information, software drivers, firmware upgrades, and revised software. Any user with a modem and computer can access the BBS. Use the following modem settings.

- 14400, N, 8, 1

If your modem does not support 14.4K baud, the system provides auto configuration support for 1200- through 14.4K-baud modems.

To access the BBS, just dial the telephone number (see page 1-4) and respond to the system prompts. During your first session, the system asks you to register with the system operator by entering your name and location. The system operator will then set up your access account within 24 hours. At that time, you can access the files on the BBS. For a listing of files, call the FaxBack service and order catalog #6 (the BBS catalog).

If you encounter any difficulty accessing our high-speed modem, try our dedicated 2400-baud modem (see page 1-4). Use the following modem settings.

- 2400 baud, N, 8, 1

1.3.3 How to Find the Latest *Ap*BUILDER Files, Hypertext Manuals, and Data Sheets on the BBS

The latest *Ap*BUILDER files and hypertext manuals and data sheets are available first from the BBS. To access the files:

1. Select [F] from the BBS Main menu.
2. Select [L] from the Intel Apps Files menu.
3. The BBS displays the list of all area levels and prompts for the area number.
4. Select [25] to choose the *Ap*BUILDER / Hypertext area.
5. Area level 25 has four sublevels: (1) General, (2) 196 Files, (3) 186 Files, and (4) 8051 Files.
6. Select [1] to find the latest *Ap*BUILDER files or the number of the appropriate product-family sublevel to find the hypertext manuals and data sheets.
7. Enter the file number to tag the files you wish to download. The BBS displays the approximate download time for tagged files.

intel®

2

Overview of the 80C186 Family Architecture

intel

CHAPTER 2
OVERVIEW OF THE 80C186 FAMILY ARCHITECTURE

The 80C186 Modular Microprocessor Core shares a common base architecture with the 8086, 8088, 80186, 80188, 80286, Intel386™ and Intel486™ processors. The 80C186 Modular Core maintains full object-code compatibility with the 8086/8088 family of 16-bit microprocessors, while adding hardware and software performance enhancements. Most instructions require fewer clocks to execute on the 80C186 Modular Core because of hardware enhancements in the Bus Interface Unit and the Execution Unit. Several additional instructions simplify programming and reduce code size (see Appendix A, "80C186 Instruction Set Additions and Extensions").

2.1 ARCHITECTURAL OVERVIEW

The 80C186 Modular Microprocessor Core incorporates two separate processing units: an Execution Unit (EU) and a Bus Interface Unit (BIU). The Execution Unit is functionally identical among all family members. The Bus Interface Unit is configured for a 16-bit external data bus for the 80C186 core and an 8-bit external data bus for the 80C188 core. The two units interface via an instruction prefetch queue.

The Execution Unit executes instructions; the Bus Interface Unit fetches instructions, reads operands and writes results. Whenever the Execution Unit requires another opcode byte, it takes the byte out of the prefetch queue. The two units can operate independently of one another and are able, under most circumstances, to overlap instruction fetches and execution.

The 80C186 Modular Core family has a 16-bit Arithmetic Logic Unit (ALU). The Arithmetic Logic Unit performs 8-bit or 16-bit arithmetic and logical operations. It provides for data movement between registers, memory and I/O space.

The 80C186 Modular Core family CPU allows for high-speed data transfer from one area of memory to another using string move instructions and between an I/O port and memory using block I/O instructions. The CPU also provides many conditional branch and control instructions.

The 80C186 Modular Core architecture features 14 basic registers grouped as general registers, segment registers, pointer registers and status and control registers. The four 16-bit general-purpose registers (AX, BX, CX and DX) can be used as operands for most arithmetic operations as either 8- or 16-bit units. The four 16-bit pointer registers (SI, DI, BP and SP) can be used in arithmetic operations and in accessing memory-based variables. Four 16-bit segment registers (CS, DS, SS and ES) allow simple memory partitioning to aid modular programming. The status and control registers consist of an Instruction Pointer (IP) and the Processor Status Word (PSW) register, which contains flag bits. Figure 2-1 is a simplified CPU block diagram.

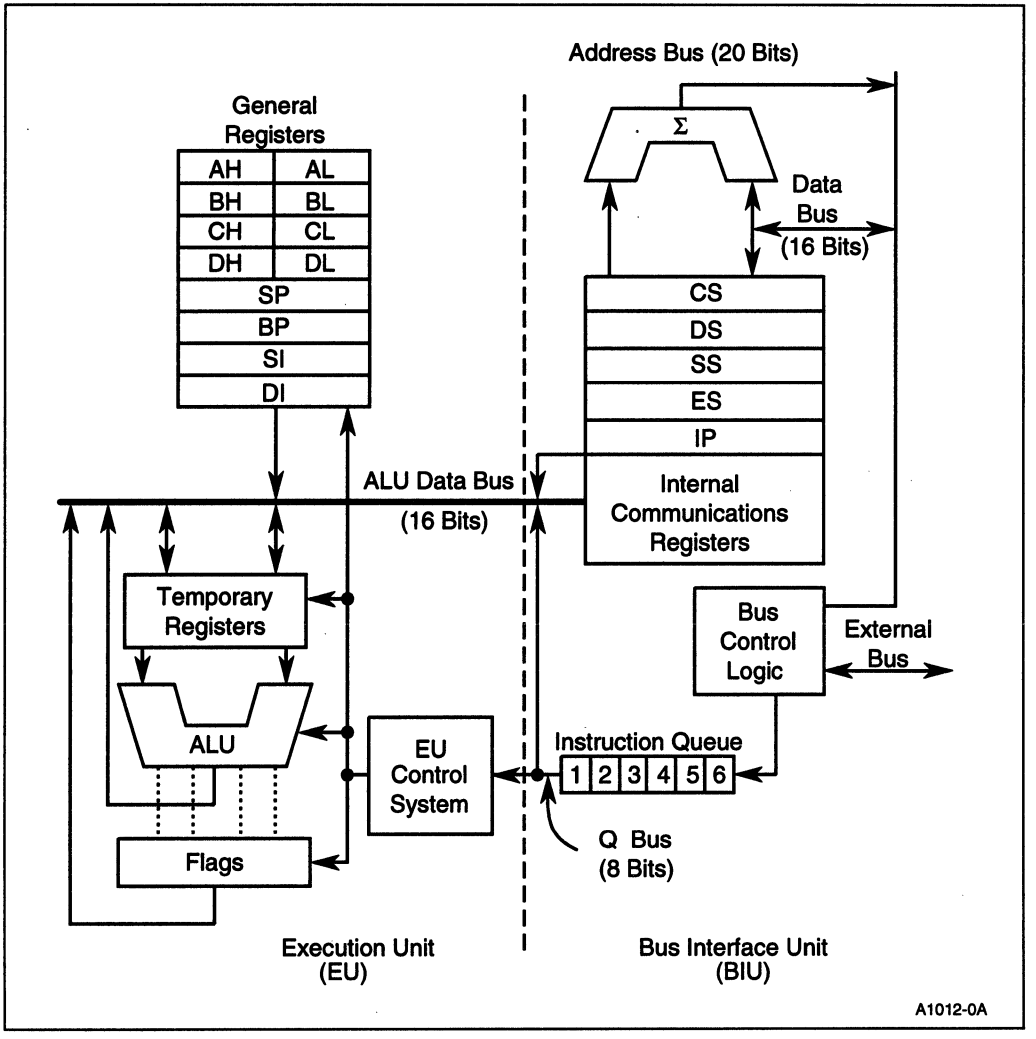


Figure 2-1. Simplified Functional Block Diagram of the 80C186 Family CPU

2.1.1 Execution Unit

The Execution Unit executes all instructions, provides data and addresses to the Bus Interface Unit and manipulates the general registers and the Processor Status Word. The 16-bit ALU within the Execution Unit maintains the CPU status and control flags and manipulates the general registers and instruction operands. All registers and data paths in the Execution Unit are 16 bits wide for fast internal transfers.

The Execution Unit does not connect directly to the system bus. It obtains instructions from a queue maintained by the Bus Interface Unit. When an instruction requires access to memory or a peripheral device, the Execution Unit requests the Bus Interface Unit to read and write data. Addresses manipulated by the Execution Unit are 16 bits wide. The Bus Interface Unit, however, performs an address calculation that allows the Execution Unit to access the full megabyte of memory space.

To execute an instruction, the Execution Unit must first fetch the object code byte from the instruction queue and then execute the instruction. If the queue is empty when the Execution Unit is ready to fetch an instruction byte, the Execution Unit waits for the Bus Interface Unit to fetch the instruction byte.

2.1.2 Bus Interface Unit

The 80C186 Modular Core and 80C188 Modular Core Bus Interface Units are functionally identical. They are implemented differently to match the structure and performance characteristics of their respective system buses. The Bus Interface Unit executes all external bus cycles. This unit consists of the segment registers, the Instruction Pointer, the instruction code queue and several miscellaneous registers. The Bus Interface Unit transfers data to and from the Execution Unit on the ALU data bus.

The Bus Interface Unit generates a 20-bit physical address in a dedicated adder. The adder shifts a 16-bit segment value left 4 bits and then adds a 16-bit offset. This offset is derived from combinations of the pointer registers, the Instruction Pointer and immediate values (see Figure 2-2). Any carry from this addition is ignored.

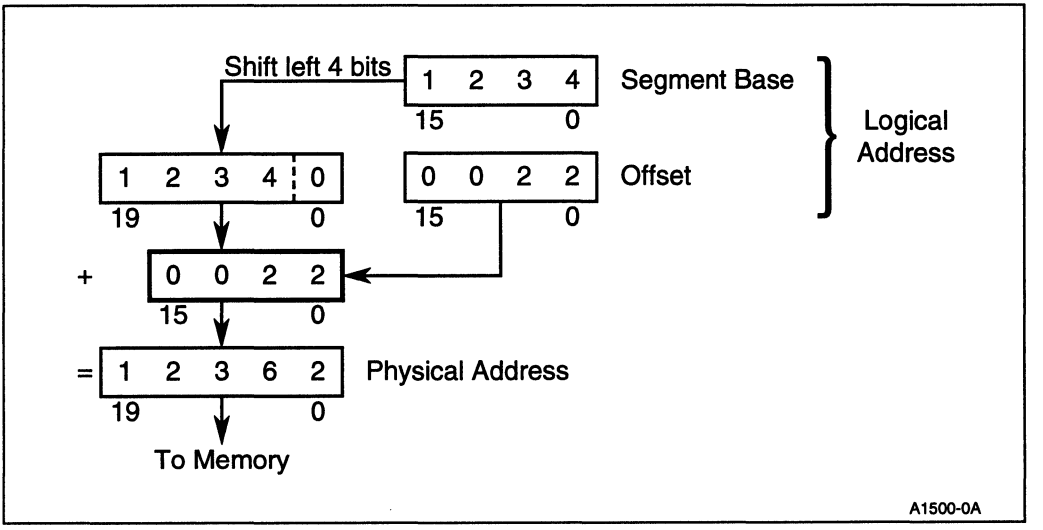


Figure 2-2. Physical Address Generation

During periods when the Execution Unit is busy executing instructions, the Bus Interface Unit sequentially prefetches instructions from memory. As long as the prefetch queue is partially full, the Execution Unit fetches instructions.

2.1.3 General Registers

The 80C186 Modular Core family CPU has eight 16-bit general registers (see Figure 2-3). The general registers are subdivided into two sets of four registers. These sets are the data registers (also called the H & L group for high and low) and the pointer and index registers (also called the P & I group).

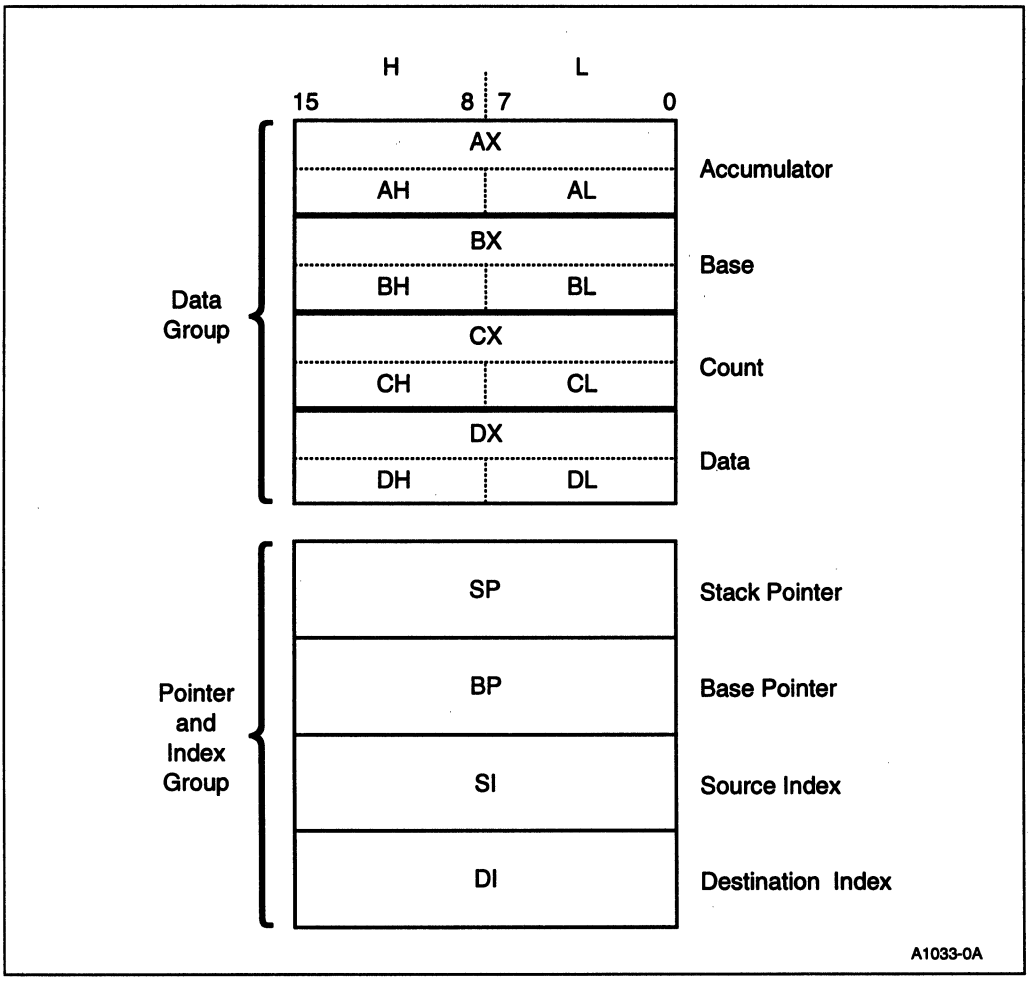


Figure 2-3. General Registers

The data registers can be addressed by their upper or lower halves. Each data register can be used interchangeably as a 16-bit register or two 8-bit registers. The pointer registers are always accessed as 16-bit values. The CPU can use data registers without constraint in most arithmetic and logic operations. Arithmetic and logic operations can also use the pointer and index registers. Some instructions use certain registers implicitly (see Table 2-1), allowing compact encoding.

Table 2-1. Implicit Use of General Registers

Register	Operations
AX	Word Multiply, Word Divide, Word I/O
AL	Byte Multiply, Byte Divide, Byte I/O, Translate, Decimal Arithmetic
AH	Byte Multiply, Byte Divide
BX	Translate
CX	String Operations, Loops
CL	Variable Shift and Rotate
DX	Word Multiply, Word Divide, Indirect I/O
SP	Stack Operations
SI	String Operations
DI	String Operations

The contents of the general-purpose registers are undefined following a processor reset.

2.1.4 Segment Registers

The 80C186 Modular Core family memory space is 1 Mbyte in size and divided into logical segments of up to 64 Kbytes each. The CPU has direct access to four segments at a time. The segment registers contain the base addresses (starting locations) of these memory segments (see Figure 2-4). The CS register points to the current code segment, which contains instructions to be fetched. The SS register points to the current stack segment, which is used for all stack operations. The DS register points to the current data segment, which generally contains program variables. The ES register points to the current extra segment, which is typically used for data storage. The CS register initializes to 0FFFFH, and the SS, DS and ES registers initialize to 0000H. Programs can access and manipulate the segment registers with several instructions.

64K

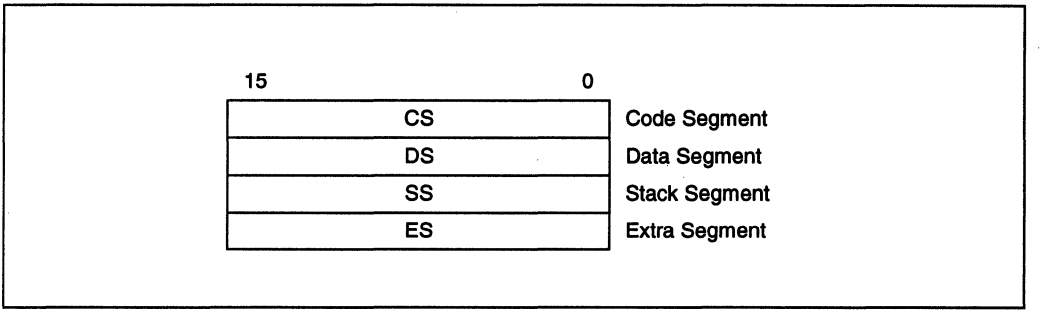


Figure 2-4. Segment Registers

2.1.5 Instruction Pointer

The Bus Interface Unit updates the 16-bit Instruction Pointer (IP) register so it contains the offset of the next instruction to be fetched. Programs do not have direct access to the Instruction Pointer, but it can change, be saved or be restored as a result of program execution. For example, if the Instruction Pointer is saved on the stack, it is first automatically adjusted to point to the next instruction to be executed.

Reset initializes the Instruction Pointer to 0000H. The CS and IP values comprise a starting execution address of 0FFFF0H (see "Logical Addresses" on page 2-10 for a description of address formation).

2.1.6 Flags

The 80C186 Modular Core family has six status flags (see Figure 2-5) that the Execution Unit posts as the result of arithmetic or logical operations. Program branch instructions allow a program to alter its execution depending on conditions flagged by a prior operation. Different instructions affect the status flags differently, generally reflecting the following states:

- If the Auxiliary Flag (AF) is set, there has been a carry out from 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.
- If the Carry Flag (CF) is set, there has been a carry out of or a borrow into the high-order bit of the instruction result (8- or 16-bit). This flag is used by instructions that add or subtract multibyte numbers. Rotate instructions can also isolate a bit in memory or a register by placing it in the Carry Flag.
- If the Overflow Flag (OF) is set, an arithmetic overflow has occurred. 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.
- If the Sign Flag (SF) is set, the high-order bit of the result is a 1. Since negative binary numbers are represented in standard two's complement notation, SF indicates the sign of the result (0 = positive, 1 = negative).
- If the Parity Flag (PF) is set, the result has even parity, an even number of 1 bits. This flag can be used to check for data transmission errors.
- If the Zero Flag (ZF) is set, the result of the operation is zero.

Additional control flags (see Figure 2-5) can be set or cleared by programs to alter processor operations:

- Setting the Direction Flag (DF) causes string operations to auto-decrement. Strings are processed from high address to low address (or "right to left"). Clearing DF causes string operations to auto-increment. Strings are processed from low address to high address (or "left to right").
- Setting the Interrupt Enable Flag (IF) allows the CPU to recognize maskable external or internal interrupt requests. Clearing IF disables these interrupts. The Interrupt Enable Flag has no effect on software interrupts or non-maskable interrupts.
- Setting the Trap Flag (TF) bit puts the processor into single-step mode for debugging. In this mode, the CPU automatically generates an interrupt after each instruction. This allows a program to be inspected instruction by instruction during execution.

The status and control flags are contained in a 16-bit Processor Status Word (see Figure 2-5). Reset initializes the Processor Status Word to 0F000H.

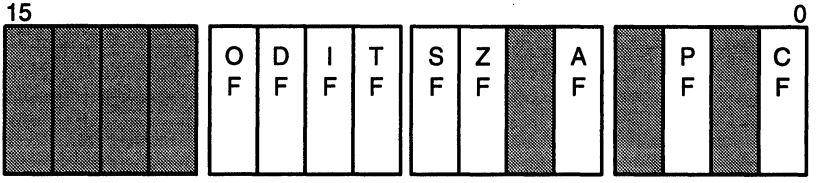
2.1.7 Memory Segmentation

Programs for the 80C186 Modular Core family view the 1 Mbyte memory space as a group of user-defined segments. A segment is a logical unit of memory that can be up to 64 Kbytes long. Each segment is composed of contiguous memory locations. Segments are independent and separately addressable. Software assigns every segment a base address (starting location) in memory space. All segments begin on 16-byte memory boundaries. There are no other restrictions on segment locations. Segments can be adjacent, disjoint, partially overlapped or fully overlapped (see Figure 2-6). A physical memory location can be mapped into (covered by) one or more logical segments.

Register Name: Processor Status Word

Register Mnemonic: PSW (FLAGS)

Register Function: Posts CPU status information.



A1035-0A

Bit Mnemonic	Bit Name	Reset State	Function
OF	Overflow Flag	0	If OF is set, an arithmetic overflow has occurred.
DF	Direction Flag	0	If DF is set, string instructions are processed high address to low address. If DF is clear, strings are processed low address to high address.
IF	Interrupt Enable Flag	0	If IF is set, the CPU recognizes maskable interrupt requests. If IF is clear, maskable interrupts are ignored.
TF	Trap Flag	0	If TF is set, the processor enters single-step mode.
SF	Sign Flag	0	If SF is set, the high-order bit of the result of an operation is 1, indicating it is negative.
ZF	Zero Flag	0	If ZF is set, the result of an operation is zero.
AF	Auxiliary Flag	0	If AF is set, there has been a carry from the low nibble to the high or a borrow from the high nibble to the low nibble of an 8-bit quantity. Used in BCD operations.
PF	Parity Flag	0	If PF is set, the result of an operation has even parity.
CF	Carry Flag	0	If CF is set, there has been a carry out of, or a borrow into, the high-order bit of the result of an instruction.

NOTE: Reserved register bits are shown with gray shading. Reserved bits must be written to a logic zero to ensure compatibility with future Intel products.

Figure 2-5. Processor Status Word

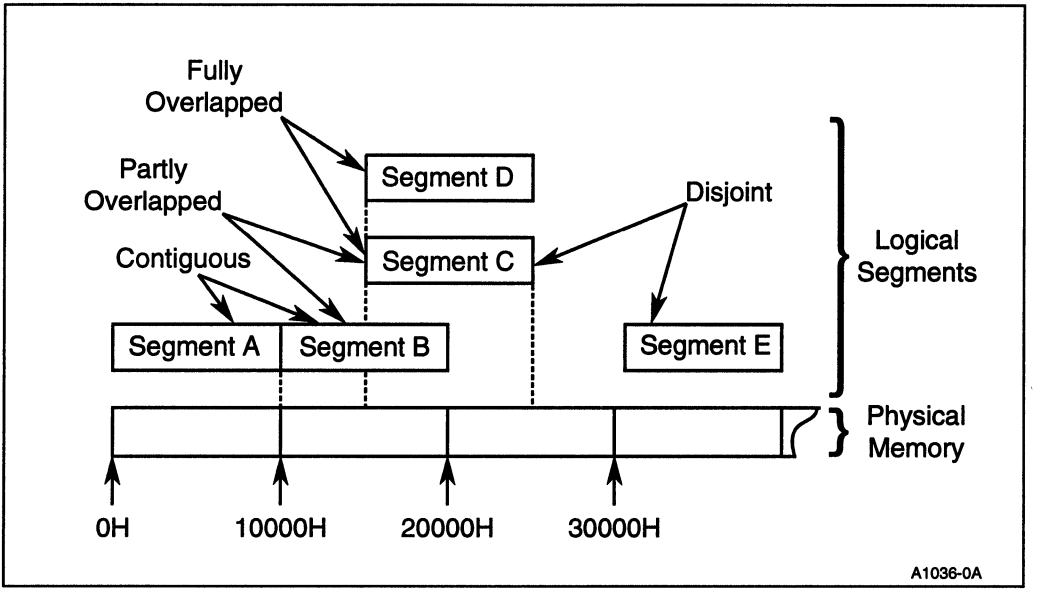


Figure 2-6. Segment Locations in Physical Memory

The four segment registers point to four "currently addressable" segments (see Figure 2-7). The currently addressable segments provide a work space consisting of 64 Kbytes for code, a 64 Kbytes for stack and 128 Kbytes for data storage. Programs access code and data in another segment by updating the segment register to point to the new segment.

2.1.8 Logical Addresses

It is useful to think of every memory location as having two kinds of addresses, physical and logical. A physical address is a 20-bit value that identifies a unique byte location in the memory space. Physical addresses range from 0H to 0FFFFFH. All exchanges between the CPU and memory use physical addresses.

Programs deal with logical rather than physical addresses. Program code can be developed without prior knowledge of where the code will be located in memory. 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 segment. The offset value represents the distance, in bytes, of the target location from the beginning of the segment. Segment base and offset values are unsigned 16-bit quantities. Many different logical addresses can map to the same physical location. In Figure 2-8, physical memory location 2C3H is contained in two different overlapping segments, one beginning at 2B0H and the other at 2C0H.

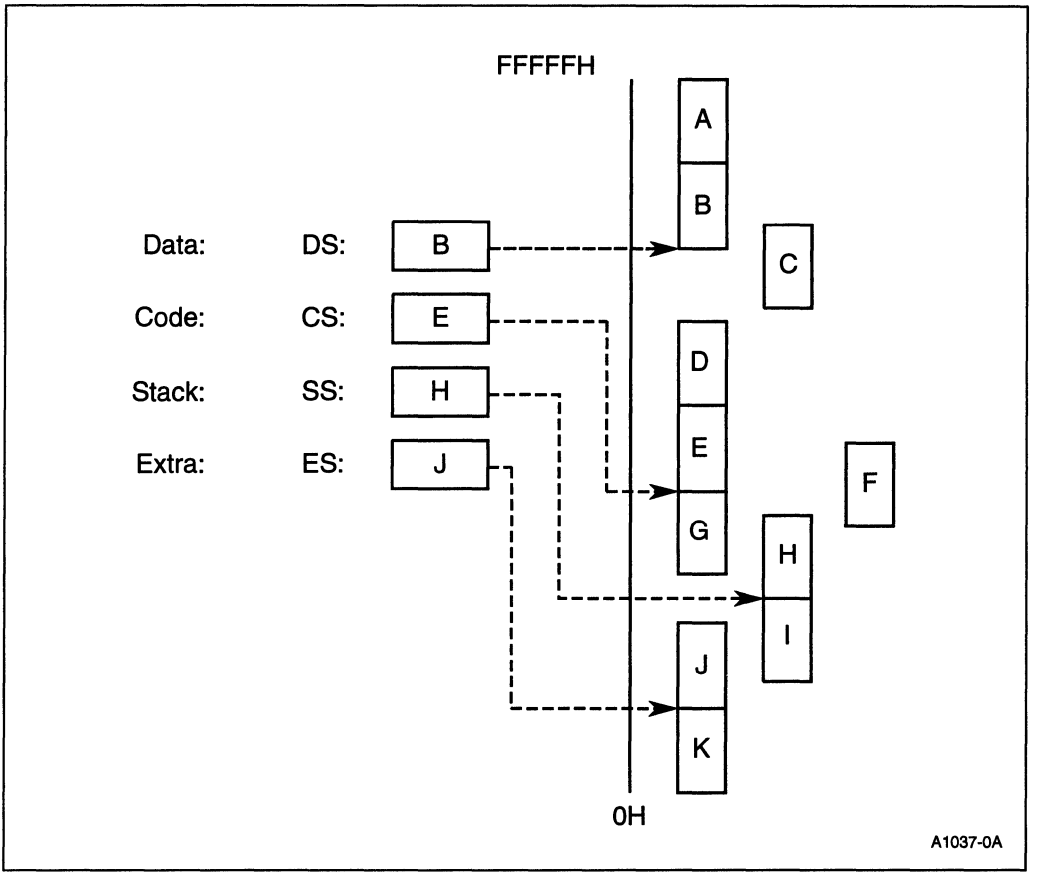


Figure 2-7. Currently Addressable Segments

The segment register is automatically selected according to the rules in Table 2-2. All information in one segment type generally shares the same logical attributes (e.g., code or data). This leads to programs that are shorter, faster and better structured.

The Bus Interface Unit must obtain the logical address before generating the physical address. The logical address of a memory location can come from different sources, depending on the type of reference that is being made (see Table 2-2).

Segment registers always hold the segment base addresses. The Bus Interface Unit determines which segment register contains the base address according to the type of memory reference made. However, the programmer can explicitly direct the Bus Interface Unit to use any currently addressable segment (except for the destination operand of a string instruction). In assembly language, this is done by preceding an instruction with a segment override prefix.

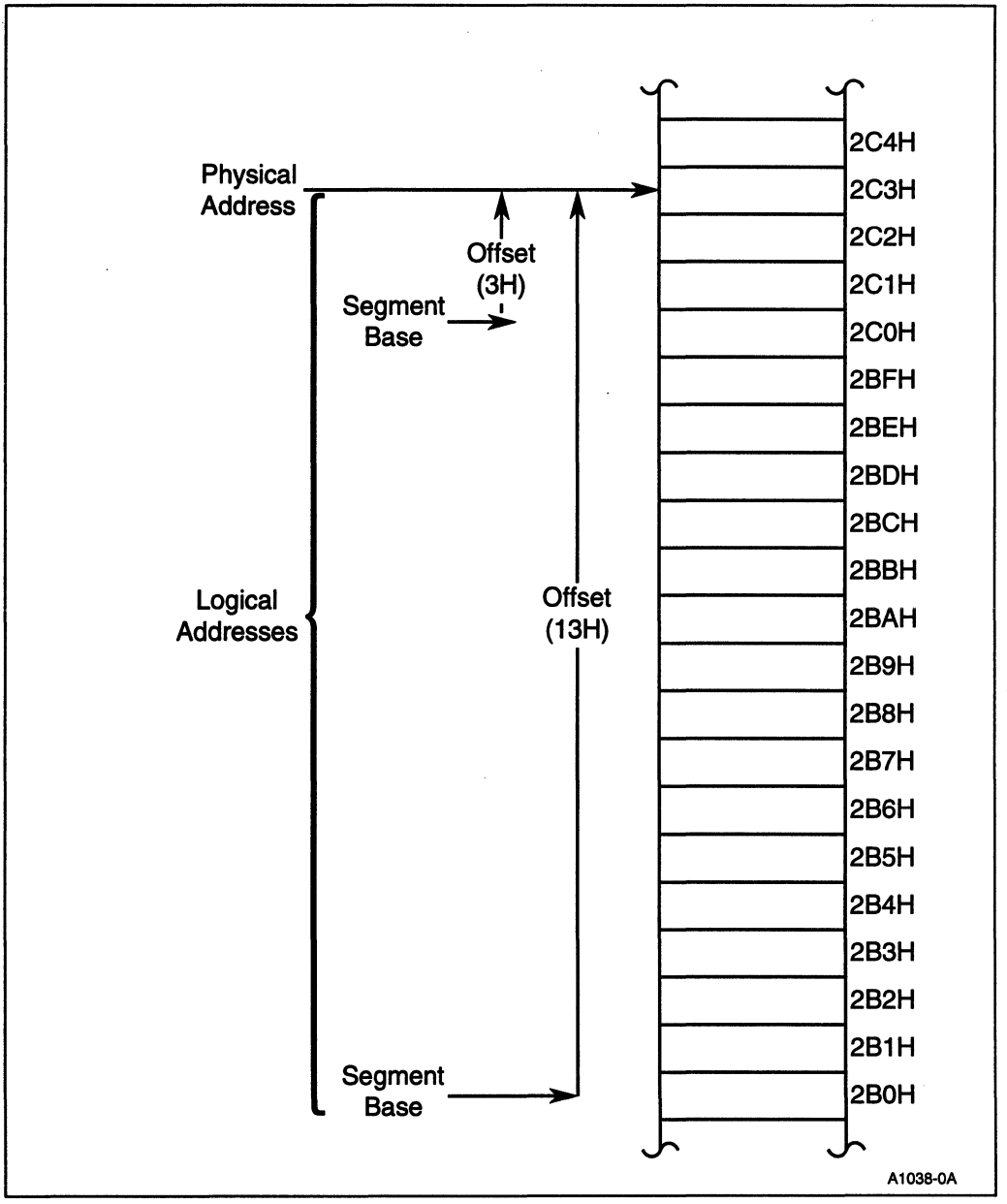


Figure 2-8. Logical and Physical Address

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

Instructions are always fetched from the current code segment. The IP register contains the instruction's offset from the beginning of the segment. Stack instructions always operate on the current stack segment. The Stack Pointer (SP) register contains the offset of the top of the stack from the base of the stack. Most variables (memory operands) are assumed to reside in the current data segment, but a program can instruct the Bus Interface Unit to override this assumption. Often, the offset of a memory variable is not directly available and must be calculated at execution time. The addressing mode specified in the instruction determines how this offset is calculated (see "Addressing Modes" on page 2-27). The result is called the operand's Effective Address (EA).

Strings are addressed differently than other variables. The source operand of a string instruction is assumed to lie in the current data segment. However, the program can use another currently addressable segment. The operand's offset is taken from the Source Index (SI) register. The destination operand of a string instruction always resides in the current extra segment. The destination's offset is taken from the Destination Index (DI) register. The string instructions automatically adjust the SI and DI registers as they process the strings one byte or word at a time.

When an instruction designates the Base Pointer (BP) register as a base register, the variable is assumed to reside in the current stack segment. The BP register provides a convenient way to access data on the stack. The BP register can also be used to access data in any other currently addressable segment.

2.1.9 Dynamically Relocatable Code

The segmented memory structure of the 80C186 Modular Core family allows creation of dynamically relocatable (position-independent) programs. Dynamic relocation allows a multiprogramming or multitasking system to make effective use of available memory. The processor can write inactive programs to a disk and reallocate the space they occupied to other programs. A disk-resident program can then be read back into available memory locations and restarted whenever it is needed. If a program needs a large contiguous block of storage and the total amount is available only in non-adjacent fragments, other program segments can be compacted to free enough continuous space. This process is illustrated in Figure 2-9.

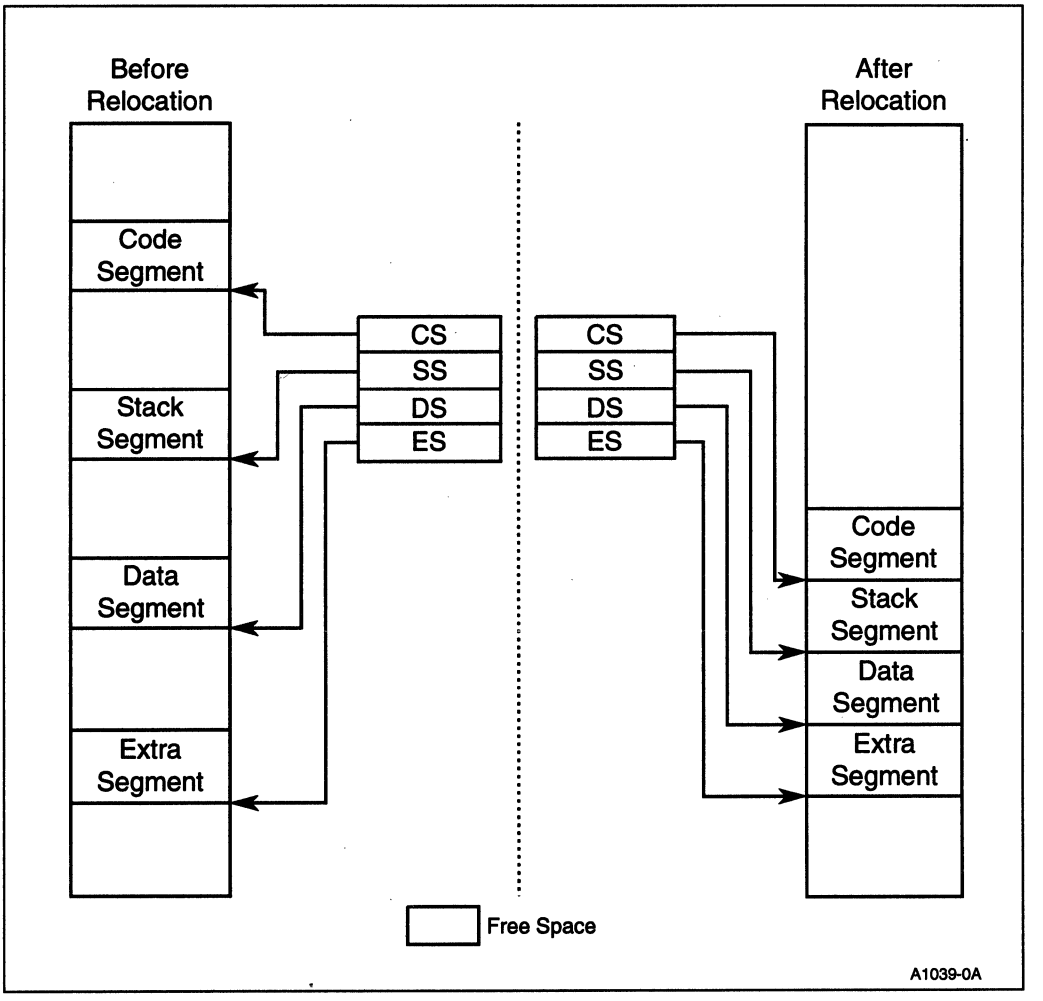


Figure 2-9. Dynamic Code Relocation

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. All program offsets must be relative to the segment registers. This allows the program to be moved anywhere in memory, provided that the segment registers are updated to point to the new base addresses.

2.1.10 Stack Implementation

Stacks in the 80C186 Modular Core family reside in memory space. They are located by the Stack Segment register (SS) and the Stack Pointer (SP). A system can have multiple stacks, but only one stack is directly addressable at a time. A stack can be up to 64 Kbytes long, the maximum length of a segment. Growing a stack segment beyond 64 Kbytes overwrites the beginning of the segment. The SS register contains the base address of the current stack. The top of the stack, not the base address, is the origination point of the stack. The SP register contains an offset that points to the Top of Stack (TOS).

Stacks are 16 bits wide. Instructions operating on a stack add and remove stack elements one word at a time. An element is pushed onto the stack (see Figure 2-10) by first decrementing the SP register by 2 and then writing the data word. An element is popped off the stack by copying it from the top of the stack and then incrementing the SP register by 2. The stack grows down in memory toward its base address. Stack operations never move or erase elements on the stack. The top of the stack changes only as a result of updating the stack pointer.

2.1.11 Reserved Memory and I/O Space

Two specific areas in memory and one area in I/O space are reserved in the 80C186 Core family.

- Locations 0H through 3FFH in low memory are used for the Interrupt Vector Table. Programs should not be loaded here.
- Locations 0FFFF0H through 0FFFFFH in high memory are used for system reset code because the processor begins execution at 0FFFF0H.
- Locations 0F8H through 0FFH in I/O space are reserved for communication with other Intel hardware products and must not be used. On the 80C186 core, these addresses are used as I/O ports for the 80C187 numerics processor extension.

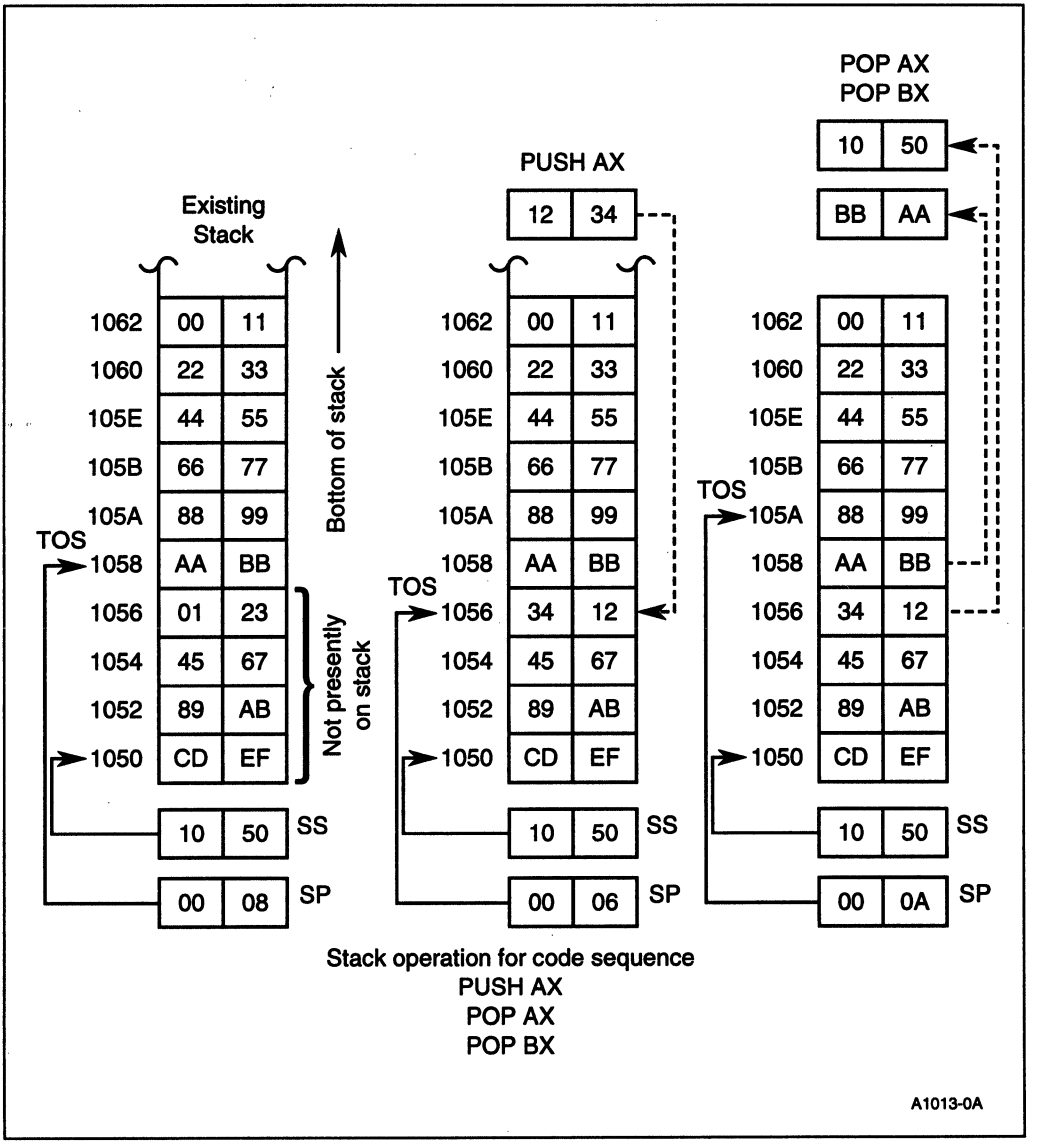


Figure 2-10. Stack Operation

2.2 SOFTWARE OVERVIEW

All 80C186 Modular Core family members execute the same instructions. This includes all the 8086/8088 instructions plus several additions and enhancements (see Appendix A, "80C186 Instruction Set Additions and Extensions"). The following sections describe the instructions by category and provide a detailed discussion of the operand addressing modes.

Software for 80C186 core family systems need not be written in assembly language. The processor provides direct hardware support for programs written in the many high-level languages available. The hardware addressing modes provide 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 allow efficient character data manipulation. Finally, routines with critical performance requirements can be written in assembly language and linked with high-level code.

2.2.1 Instruction Set

The 80C186 Modular Core family instructions treat different types of operands uniformly. Nearly every instruction can operate on either byte or word data. Register, memory and immediate operands can be specified interchangeably in most instructions. Immediate values are exceptions: they must serve as source operands and not destination operands. Memory variables can be manipulated (added to, subtracted from, shifted, compared) without being moved into 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 can produce faster and shorter object programs.

The 80C186 Modular Core family instruction set can be viewed as existing on two levels. One is the assembly level and the other is the machine level. To the assembly language programmer, the 80C186 Modular Core family appears to have about 100 instructions. One MOV (data move) instruction, for example, transfers a byte or a word from a register, a memory location or an immediate value to either a register or a memory location. The 80C186 Modular Core family CPUs, however, recognize 28 different machine versions of the MOV instruction.

The two levels of instruction sets address two requirements: efficiency and simplicity. Approximately 300 forms of machine-level instructions 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. Incrementing a register, however, requires less information, so the instruction can be shorter. The 80C186 Core family has eight single-byte machine-level instructions that increment different 16-bit registers.

The assembly level instructions simplify the programmer's view of the instruction set. The programmer writes one form of an INC (increment) instruction and the assembler examines the operand to determine which machine level instruction to generate. The following paragraphs provide a functional description of the assembly-level instructions.

2.2.1.1 Data Transfer Instructions

The instruction set contains 14 data transfer instructions. These instructions move single bytes and words between memory and registers. They also move single bytes and words between the AL or AX register and I/O ports. Table 2-3 lists the four types of data transfer instructions and their functions.

Table 2-3. Data Transfer Instructions

General-Purpose	
MOV	Move byte or word
PUSH	Push word onto stack
POP	Pop word off stack
PUSHA	Push registers onto stack
POPA	Pop registers off stack
XCHG	Exchange byte or word
XLAT	Translate byte
Input/Output	
IN	Input byte or word
OUT	Output byte or word
Address Object and Stack Frame	
LEA	Load effective address
LDS	Load pointer using DS
LES	Load pointer using ES
ENTER	Build stack frame
LEAVE	Tear down stack frame
Flag Transfer	
LAHF	Load AH register from flags
SAHF	Store AH register in flags
PUSHF	Push flags from stack
POPF	Pop flags off stack

Data transfer instructions are categorized as general purpose, input/output, address object and flag transfer. The stack manipulation instructions, used for transferring flag contents and instructions used for loading segment registers are also included in this group. Figure 2-11 shows the flag storage formats. The address object instructions manipulate the addresses of variables instead of the values of the variables.

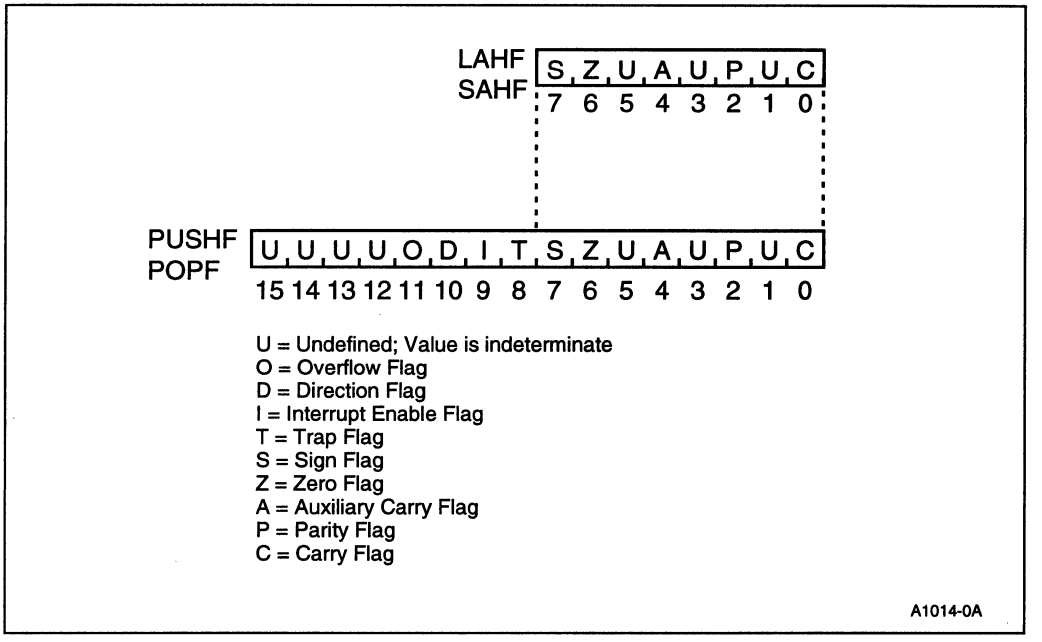


Figure 2-11. Flag Storage Format

2.2.1.2 Arithmetic Instructions

The arithmetic instructions (see Table 2-4) operate on four types of numbers:

- Unsigned binary
- Signed binary (integers)
- Unsigned packed decimal
- Unsigned unpacked decimal

Table 2-5 shows the interpretations of various bit patterns according to number type. Binary numbers can 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 assumes that the operands in arithmetic instructions contain data that represents valid numbers for that instruction. Invalid data may produce unpredictable results. The Execution Unit analyzes the results of arithmetic instructions and adjusts status flags accordingly.

Table 2-4. 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 multiplication
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 double-word

Table 2-5. 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

2.2.1.3 Bit Manipulation Instructions

There are three groups of instructions for manipulating bits within bytes and words. These three groups are logical, shifts and rotates. Table 2-6 lists the bit manipulation instructions and their functions.

Table 2-6. 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 instructions include the Boolean operators NOT, AND, OR and exclusive OR (XOR), as well as a TEST instruction. The TEST instruction sets the flags as a result of a Boolean AND operation but does not alter either of its operands.

Individual bits in bytes and words can be shifted either arithmetically or logically. Up to 32 shifts can be performed, according to the value of the count operand coded in the instruction. The count can be specified as an immediate value or as a variable in the CL register. This allows the shift count to be a supplied at execution time. Arithmetic shifts can be used to multiply and divide binary numbers by powers of two. Logical shifts can be used to isolate bits in bytes or words.

Individual bits in bytes and words can also be rotated. The processor does not discard the bits rotated out of an operand. The bits circle back to the other end of the operand. The number of bits to be rotated is taken from the count operand, which can specify either an immediate value or the CL register. The carry flag can act as an extension of the operand in two of the rotate instructions. This allows a bit to be isolated in the Carry Flag (CF) and then tested by a JC (jump if carry) or JNC (jump if not carry) instruction.

2.2.1.4 String Instructions

Five basic string operations process strings of bytes or words, one element (byte or word) at a time. Strings of up to 64 Kbytes can be manipulated with these instructions. Instructions are available to move, compare or scan for a value, as well as to move string elements to and from the accumulator. Table 2-7 lists the string instructions. These basic operations can be preceded by a one-byte prefix that causes the instruction to be repeated by the hardware, allowing long strings to be processed much faster than is possible with a software loop. The repetitions can be terminated by a variety of conditions. Repeated operations can be interrupted and resumed.

Table 2-7. String Instructions

REP	Repeat
REPE/REPZ	Repeat while equal/zero
REPNE/REPNZ	Repeat while not equal/not zero
MOVSB/MOVSW	Move byte string/word string
MOVS	Move byte or word string
INS	Input byte or word string
OUTS	Output 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

String instructions operate similarly in many respects (see Table 2-8). A string instruction can 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 can override this assumption. A destination string must be in the current extra segment. The assembler does not use the operand names to address strings. Instead, the contents of the Source Index (SI) register are used as an offset to address the current element of the source string. The contents of the Destination Index (DI) register are taken as the offset of the current destination string element. These registers must be initialized to point to the source and destination strings before executing the string instructions. The LDS, LES and LEA instructions are useful in performing this function.

String instructions automatically update the SI register, the DI register, or both, before processing the next string element. The Direction Flag (DF) determines whether the index registers are auto-incremented (DF = 0) or auto-decremented (DF = 1). The processor adjusts the DI, SI, or both registers by one for byte strings or by two for word strings.

If a repeat prefix is used, the count register (CX) is decremented by one after each repetition of the string instruction. The CX register must be initialized to the number of repetitions before the string instruction is executed. If the CX register is 0, the string instruction is not executed and control goes to the following instruction.

Table 2-8. String Instruction Register and Flag Use

SI	Index (offset) for source string
DI	Index (offset) for destination string
CX	Repetition counter
AL/AX	Scan value
	Destination for LODS
	Source for STOS
DF	Direction Flag
	0 = auto-increment SI, DI
	1 = auto-decrement SI, DI
ZF	Scan/compare terminator

2.2.1.5 Program Transfer Instructions

The contents of the Code Segment (CS) and Instruction Pointer (IP) registers determine the instruction execution sequence in the 80C186 Modular Core family. The CS register contains the base address of the current code segment. The Instruction Pointer register points to the memory location of the next instruction to be fetched. In most operating conditions, the next instruction will already have been fetched and will be waiting in the CPU instruction queue. Program transfer instructions operate on the IP and CS registers. Changing the contents of these registers causes normal sequential operation to be altered. When a program transfer occurs, the queue no longer contains the correct instruction. The Bus Interface Unit obtains the next instruction from memory using the new IP and CS values. It then passes the instruction directly to the Execution Unit and begins refilling the queue from the new location.

The 80C186 Modular Core family offers four groups of program transfer instructions (see Table 2-9). These are unconditional transfers, conditional transfers, iteration control instructions and interrupt-related instructions.

Unconditional transfer instructions can transfer control either to a target instruction within the current code segment (intrasegment transfer) or to a different code segment (intersegment transfer). The assembler terms an intrasegment transfer SHORT or NEAR and an intersegment transfer FAR. The transfer is made unconditionally when the instruction is executed. CALL, RET and JMP are all unconditional transfers.

CALL is used to transfer the program to a procedure. A CALL can be NEAR or FAR. A NEAR CALL stacks only the Instruction Pointer, while a FAR CALL stacks both the Instruction Pointer and the Code Segment register. The RET instruction uses the information pushed onto the stack to determine where to return when the procedure finishes. Note that the RET and CALL instructions must be the same type. This can be a problem when the CALL and RET instructions are in separately assembled programs. The JMP instruction does not push any information onto the stack. A JMP instruction can be NEAR or FAR.

Conditional transfer instructions are jumps that may or may not transfer control, depending on the state of the CPU flags when the instruction is executed. Each conditional transfer instruction tests a different combination of flags for a condition (see Table 2-10). If the condition is logically TRUE, control is transferred to the target specified in the instruction. If the condition is FALSE, control passes to the instruction following the conditional jump. All conditional jumps are SHORT. The target must be in the current code segment within –128 to +127 bytes of the next instruction's first byte. For example, JMP 00H causes a jump to the first byte of the next instruction. Jumps are 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.

Table 2-9. Program Transfer Instructions

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
Unconditional Transfers	
CALL	Call procedure
RET	Return from procedure
JMP	Jump
Iteration Control	
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
BOUND	Interrupt if out of array bounds
IRET	Interrupt return

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 can transfer only to targets that are within –128 to +127 bytes of themselves. They are SHORT transfers.

The interrupt instructions allow programs and external hardware devices to activate interrupt service routines. The effect of a software interrupt is similar to that of a hardware-initiated interrupt. The processor cannot execute an interrupt acknowledge bus cycle if the interrupt originates in software or with an NMI (Non-Maskable Interrupt).

Table 2-10. 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: The terms *above* and *below* refer to the relationship of two unsigned values; *greater* and *less* refer to the relationship of two signed values.

2.2.1.6 Processor Control Instructions

Processor control instructions (see Table 2-11) allow programs to control various CPU functions. Seven of these instructions update flags, four of them are used to synchronize the microprocessor with external events, and the remaining instruction causes the CPU to do nothing. Except for flag operations, processor control instructions do not affect the flags.

Table 2-11. 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

2.2.2 Addressing Modes

The 80C186 Modular Core family members access instruction operands in several ways. Operands can be contained either in registers, in the instruction itself, in memory or at I/O ports. Addresses of memory and I/O port operands can be calculated in many ways. These addressing modes greatly extend the flexibility and convenience of the instruction set. The following paragraphs briefly describe register and immediate modes of operand addressing. A detailed description of the memory and I/O addressing modes is also provided.

2.2.2.1 Register and Immediate Operand Addressing Modes

Usually, the fastest, most compact operand addressing forms specify only register operands. This is because the register operand addresses are encoded in instructions in just a few bits and no bus cycles are run (the operation occurs within the CPU). Registers can serve as source operands, destination operands, or both.

Immediate operands are constant data contained in an instruction. Immediate data can be either 8 or 16 bits in length. Immediate operands are available directly from the instruction queue and can be accessed quickly. As with a register operand, no bus cycles need to be run to get an immediate operand. Immediate operands can be only source operands and must have a constant value.

2.2.2.2 Memory Addressing Modes

Although the Execution Unit has direct access to register and immediate operands, memory operands must be transferred to and from the CPU over the bus. When the Execution Unit needs to read or write a memory operand, it must pass an offset value to the Bus Interface Unit. The Bus Interface Unit adds the offset to the shifted contents of a segment register, producing a 20-bit physical address. One or more bus cycles are then run to access the operand.

The offset that the Execution Unit calculates for memory operand is called the operand's Effective Address (EA). This address 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 Execution Unit can calculate the effective address in several ways. Information encoded in the second byte of the instruction tells the Execution Unit how to calculate the effective address of each memory operand. A compiler or assembler derives this information from the instruction written by the programmer. Assembly language programmers have access to all addressing modes.

The Execution Unit calculates the Effective Address by summing a displacement, the contents of a base register and the contents of an index register (see Figure 2-12). Any combination of these can be present in a given instruction. This allows a variety of memory addressing modes.

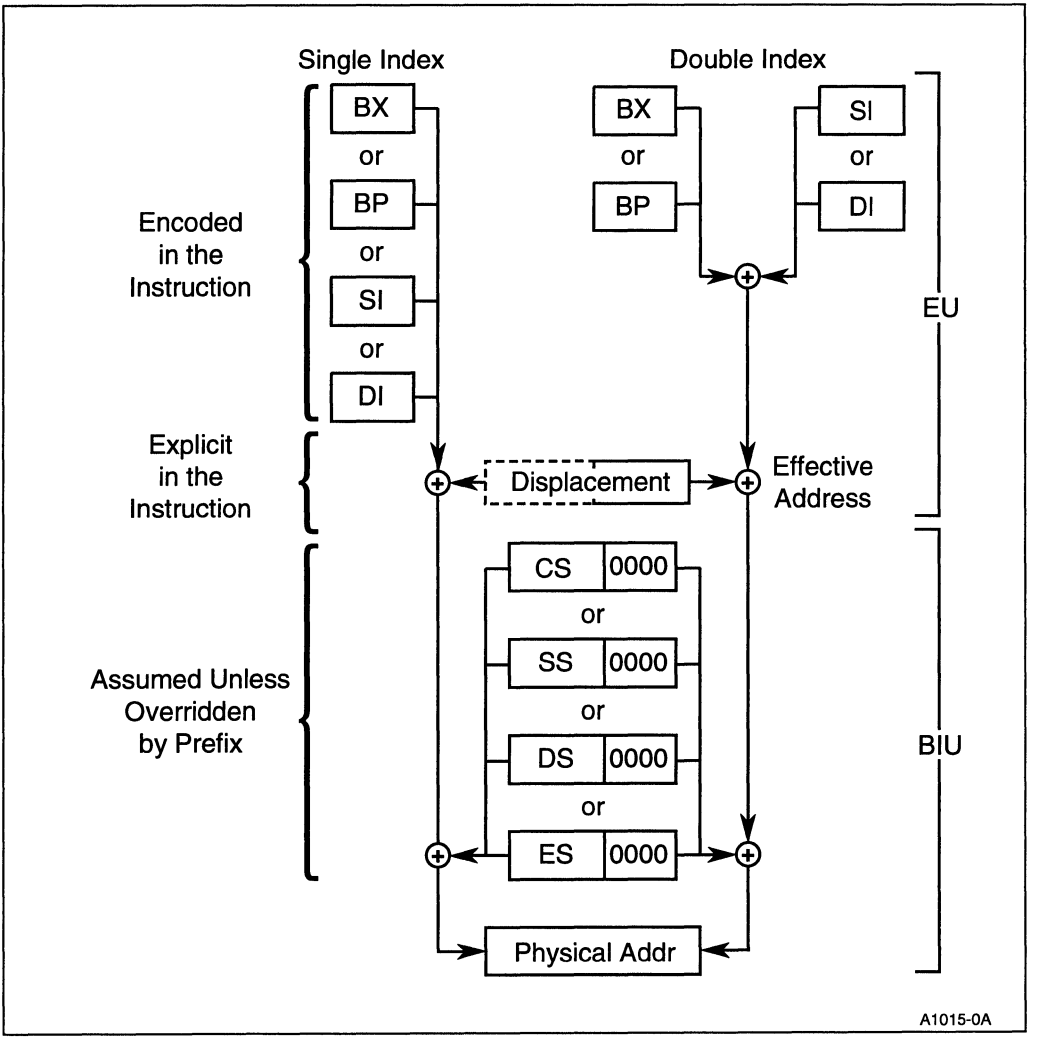


Figure 2-12. Memory Address Computation

The displacement is an 8- or 16-bit number contained in the instruction. The displacement generally is derived from the position of the operand's name (a variable or label) in the program. The programmer can modify this value or explicitly specify the displacement.

The BX or BP register can be specified as the base register for an effective address calculation. Similarly, either the SI or the DI register can be specified as the index register. The displacement value is a constant. The contents of the base and index registers can change during execution. This allows one instruction to access different memory locations depending upon the current values in the base or base and index registers. The default base register for effective address calculations with the BP register is SS, although DS or ES can be specified.

Direct addressing is the simplest memory addressing mode (see Figure 2-13). No registers are involved, and the effective address is taken directly from the displacement of the instruction. Programmers typically use direct addressing to access scalar variables.

With register indirect addressing, the effective address of a memory operand can be taken directly from one of the base or index registers (see Figure 2-14). One instruction can operate on various memory locations if the base or index register is updated accordingly. Any 16-bit general register can be used for register indirect addressing with the JMP or CALL instructions.

In based addressing, the effective address is the sum of a displacement value and the contents of the BX or BP register (see Figure 2-15). Specifying the BP register as a base register directs the Bus Interface Unit to obtain the operand from the current stack segment (unless a segment override prefix is present). This makes based addressing with the BP register a convenient way to access stack data.

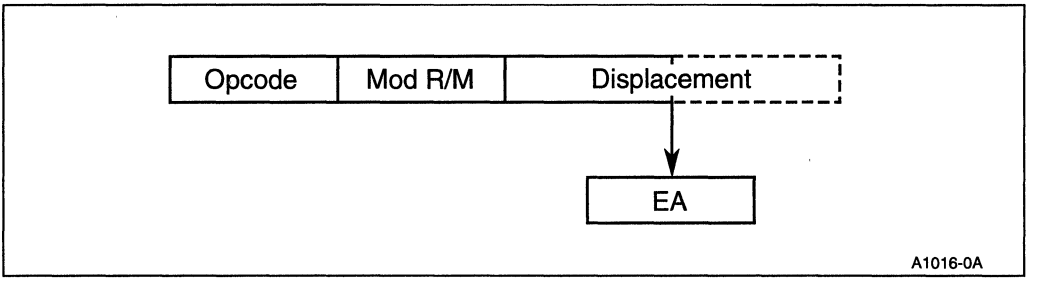


Figure 2-13. Direct Addressing

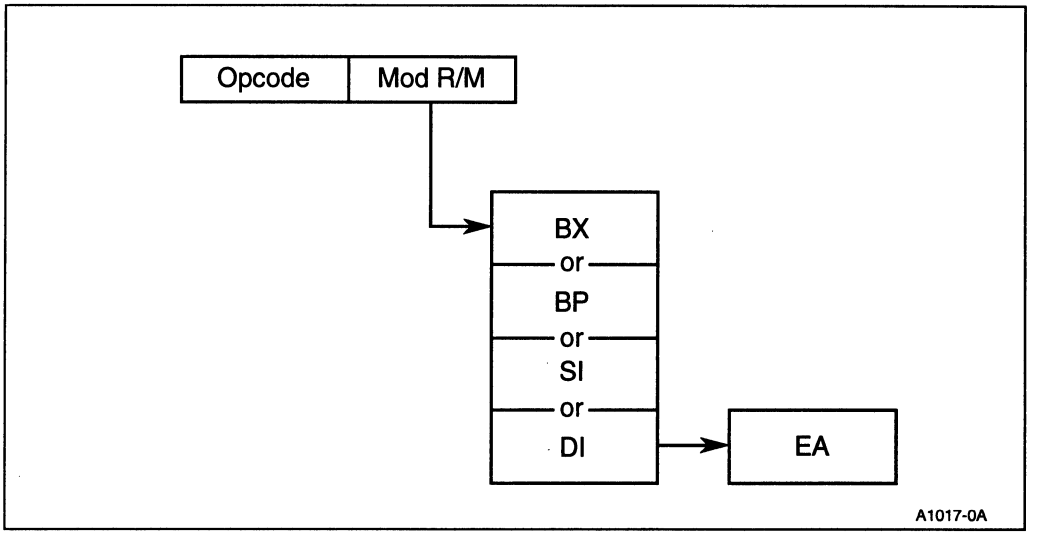


Figure 2-14. Register Indirect Addressing

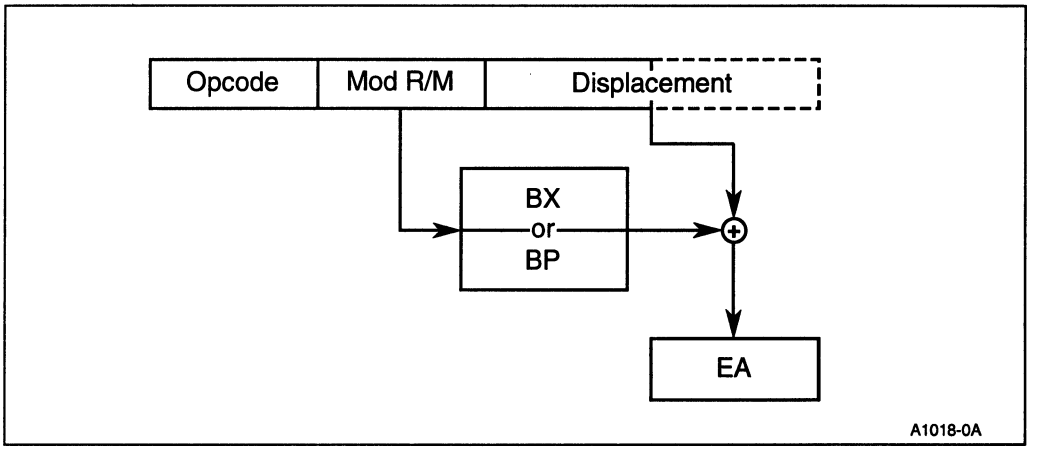


Figure 2-15. Based Addressing

Based addressing provides a simple way to address data structures that may be located in different places in memory (see Figure 2-16). A base register can be pointed at the structure. Elements of the structure can then be addressed by their displacements. Different copies of the same structure can be accessed by simply changing the base register.

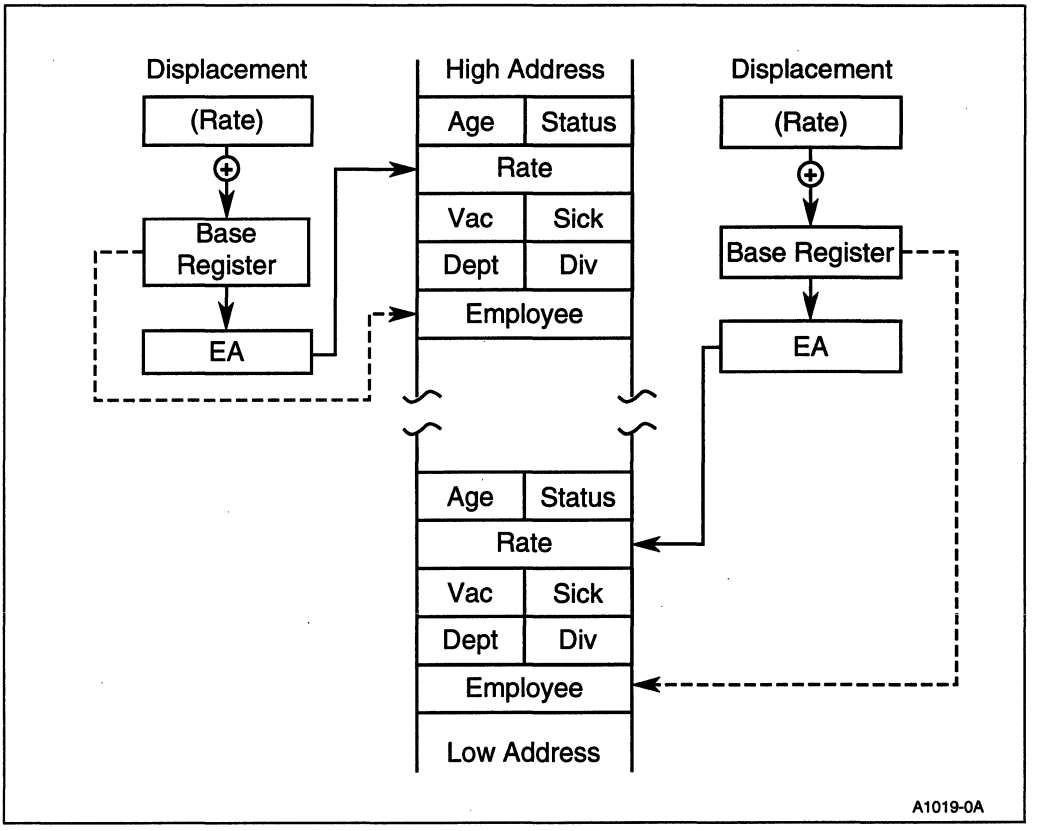


Figure 2-16. Accessing a Structure with Based Addressing

With indexed addressing, the effective address is calculated by summing a displacement and the contents of an index register (SI or DI, see Figure 2-17). Indexed addressing is often used to access elements in an array (see Figure 2-18). The displacement locates the beginning of the array, and the value of the index register selects one element. If the index register contains 0000H, the processor selects the first element. Since all array elements are the same length, simple arithmetic on the register can select any element.

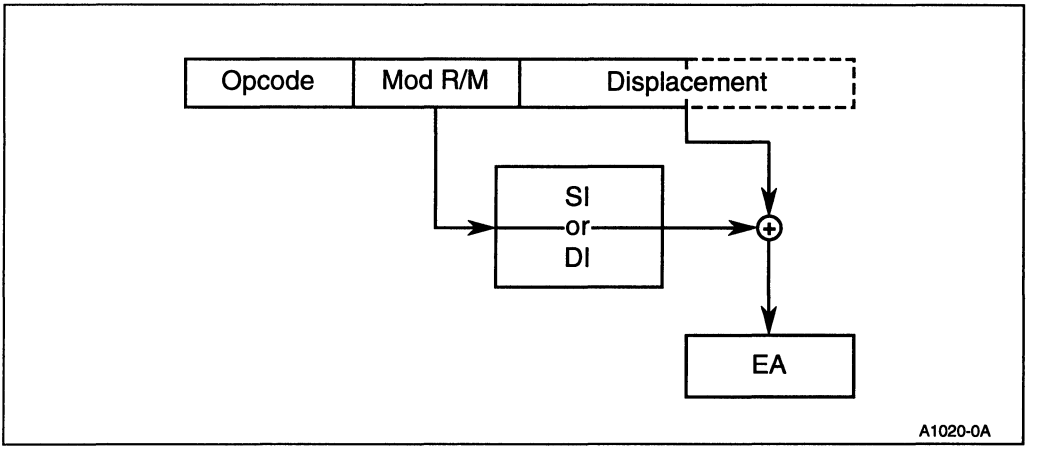


Figure 2-17. Indexed Addressing

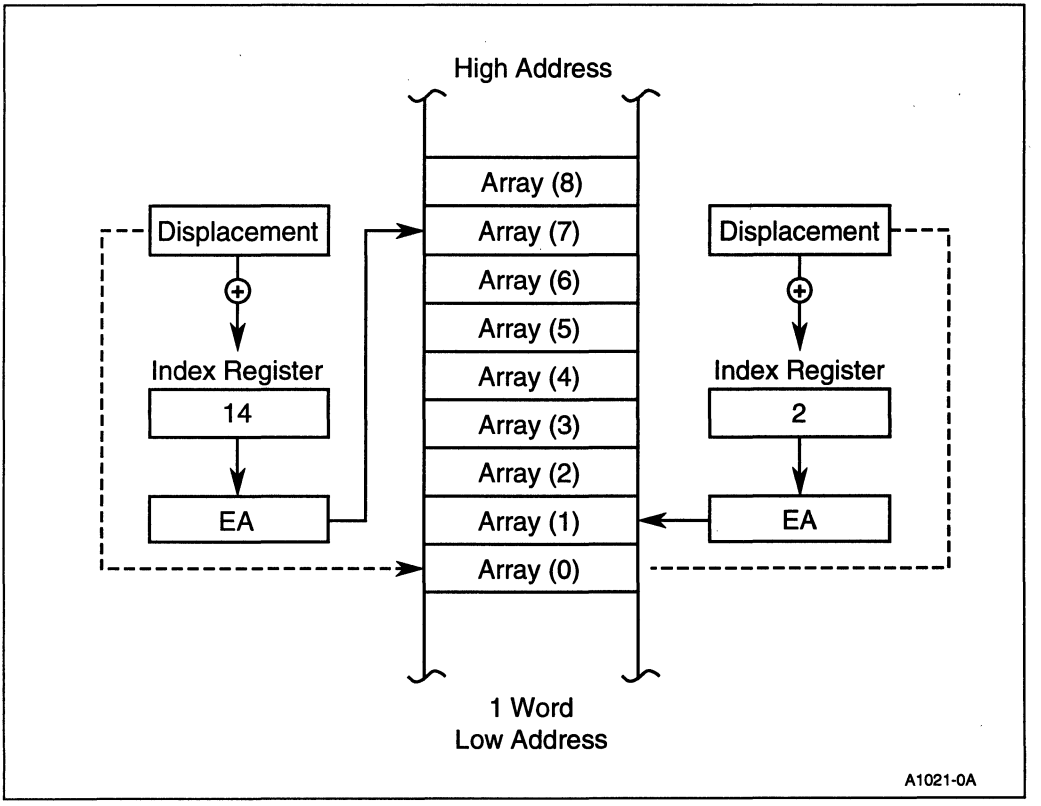


Figure 2-18. Accessing an Array with Indexed Addressing

Based index addressing generates an effective address that is the sum of a base register, an index register and a displacement (see Figure 2-19). The two address components can be determined at execution time, making this a very flexible addressing mode.

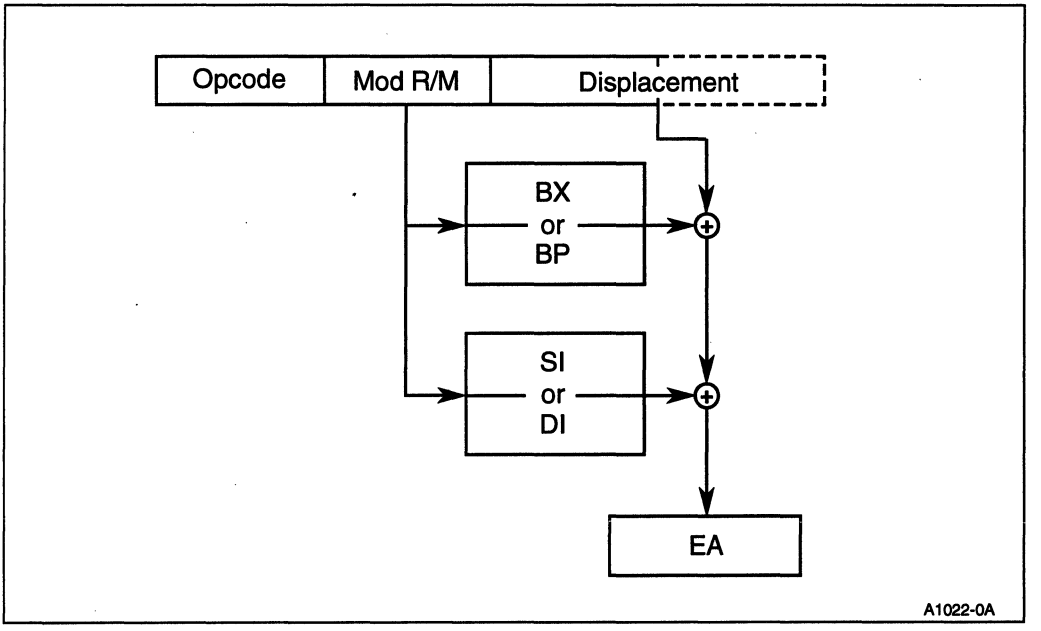


Figure 2-19. Based Index Addressing

Based index addressing provides a convenient way for a procedure to address an array located on a stack (see Figure 2-20). The BP register can contain the offset of a reference point on the stack. This is 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. The index register can be used to access individual array elements. Arrays contained in structures and matrices (two-dimensional arrays) can also be accessed with based indexed addressing.

String instructions do not use normal memory addressing modes to access operands. Instead, the index registers are used implicitly (see Figure 2-21). When a string instruction executes, the SI register must point to the first byte or word of the source string, and the DI register must point to the first byte or word of the destination string. In a repeated string operation, the CPU will automatically adjust the SI and DI registers to obtain subsequent bytes or words. For string instructions, the DS register is the default segment register for the SI register and the ES register is the default segment register for the DI register. This allows string instructions to operate on data located anywhere within the 1 Mbyte address space.

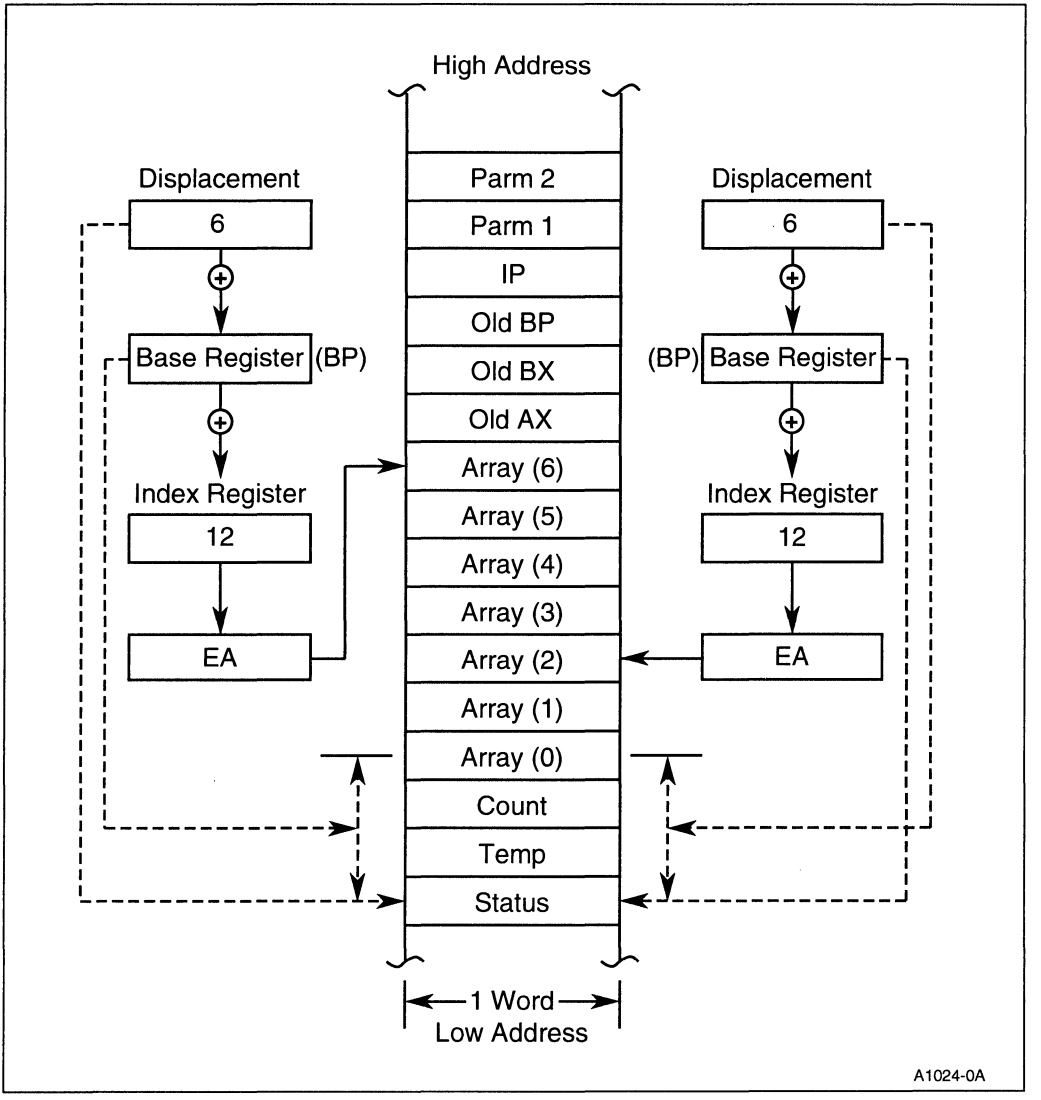


Figure 2-20. Accessing a Stacked Array with Based Index Addressing

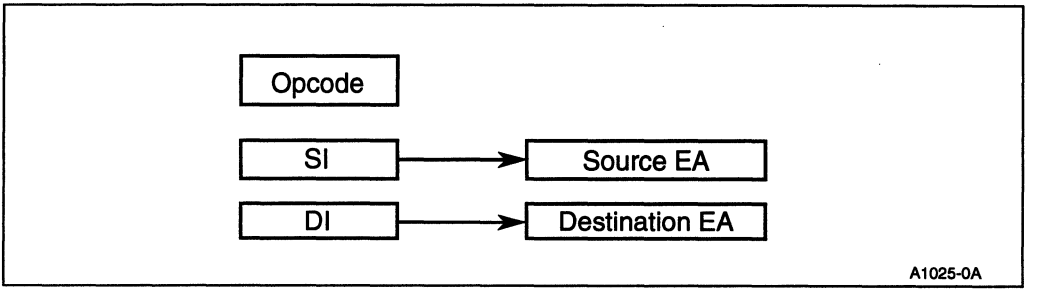


Figure 2-21. String Operand

2.2.2.3 I/O Port Addressing

Any memory operand addressing modes can be used to access an I/O port if the port is memory-mapped. String instructions can also be used to transfer data to memory-mapped ports with an appropriate hardware interface.

Two addressing modes can be used to access ports located in the I/O space (see Figure 2-22). For direct I/O port addressing, the port number is an 8-bit immediate operand. This allows fixed access to ports numbered 0 to 255. Indirect I/O port addressing is similar to register indirect addressing of memory operands. The DX register contains the port number, which can range from 0 to 65,535. Adjusting the contents of the DX register allows one instruction to 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 of the DX register.

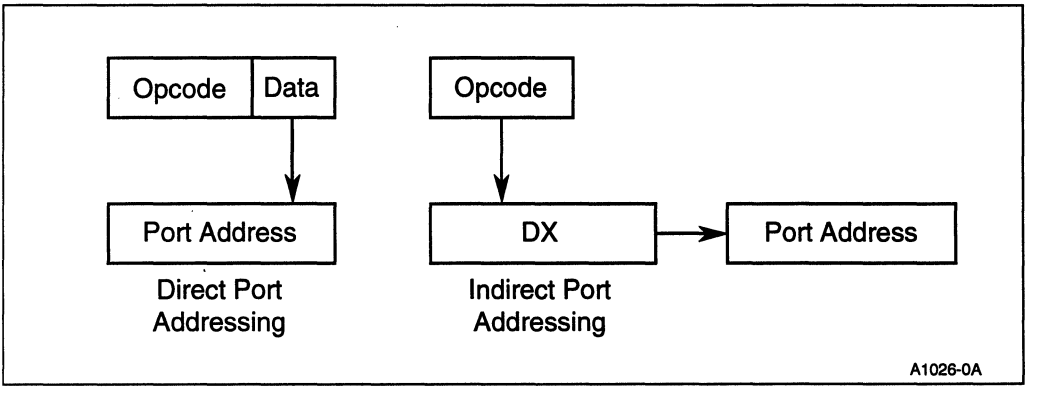


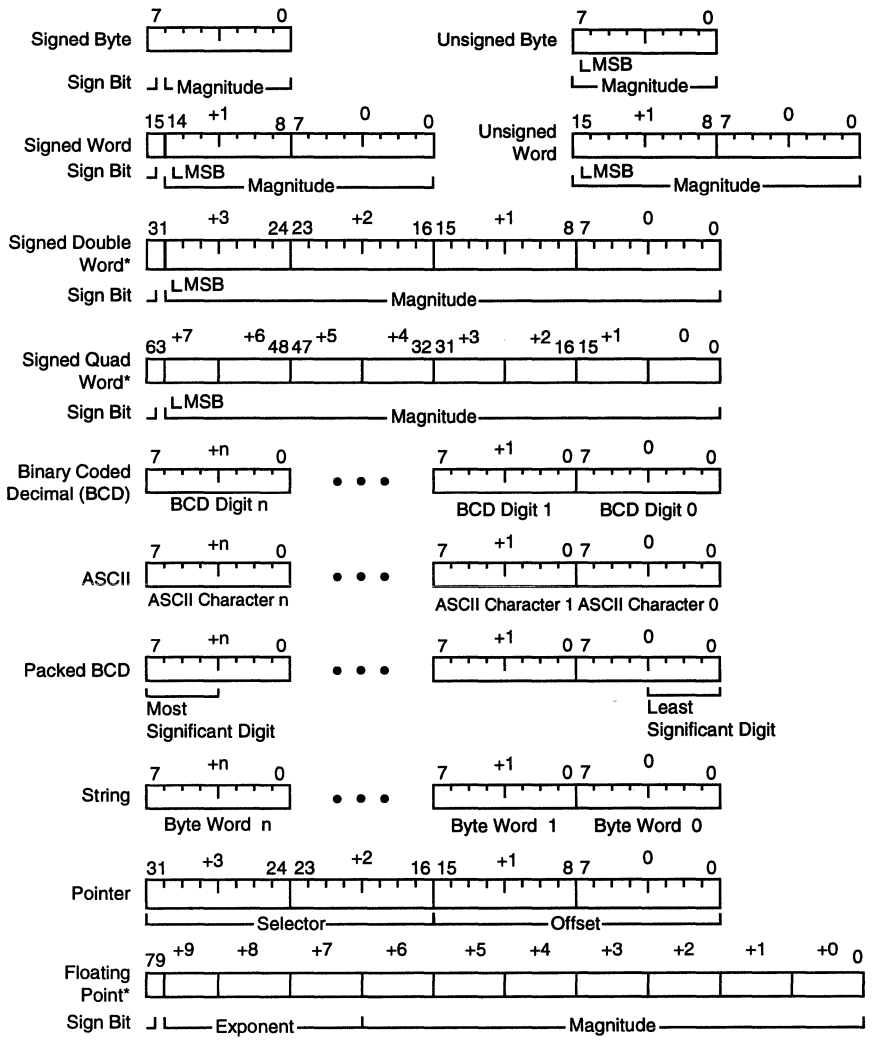
Figure 2-22. I/O Port Addressing

2.2.2.4 Data Types Used in the 80C186 Modular Core Family

The 80C186 Modular Core family supports the data types described in Table 2-12 and illustrated in Figure 2-23. In general, individual data elements must fit within defined segment limits.

Table 2-12. Supported Data Types

Type	Description
Integer	A signed 8- or 16-bit binary numeric value (signed byte or word). All operations assume a 2's complement representation. The 80C187 numerics processor extension, when added to an 80C186 Modular Core system, directly supports signed 32- and 64-bit integers (signed double-words and quad-words). The 80C188 Modular Core does not support the 80C187.
Ordinal	An unsigned 8- or 16-bit binary numeric value (unsigned byte or word).
BCD	A byte (unpacked) representation of a single decimal digit (0-9).
ASCII	A byte representation of alphanumeric and control characters using the ASCII standard.
Packed BCD	A byte (packed) representation of two decimal digits (0-9).One digit is stored in each nibble (4 bits) of the byte.
String	A contiguous sequence of bytes or words. A string can contain from 1 byte to 64 Kbytes.
Pointer	A 16- or 32-bit quantity. A 16-bit pointer consists of a 16-bit offset component; a 32-bit pointer consists of the combination of a 16-bit base component (selector) plus a 16-bit offset component.
Floating Point	A signed 32-, 64-, or 80-bit real number representation. The 80C187 numerics processor extension, when added to an 80C186 Modular Core system, directly supports floating point operands. The 80C188 Modular Core does not support the 80C187.



NOTE: *Directly supported if the system contains an 80C187.

A1027-0B

Figure 2-23. 80C186 Modular Core Family Supported Data Types

2.3 INTERRUPTS AND EXCEPTION HANDLING

Interrupts and exceptions alter program execution in response to an external event or an error condition. An interrupt handles asynchronous external events, for example an NMI. Exceptions result directly from the execution of an instruction, usually an instruction fault. The user can cause a software interrupt by executing an "INT*n*" instruction. The CPU processes software interrupts in the same way that it handles exceptions.

The 80C186 Modular Core responds to interrupts and exceptions in the same way for all devices within the 80C186 Modular Core family. However, devices within the family may have different Interrupt Control Units. The Interrupt Control Unit handles all external interrupt sources and presents them to the 80C186 Modular Core via one maskable interrupt request (see Figure 2-24). This discussion covers only those areas of interrupts and exceptions that are common to the 80C186 Modular Core family. The Interrupt Control Unit is proliferation-dependent; see Chapter 8, "Interrupt Control Unit," for additional information.

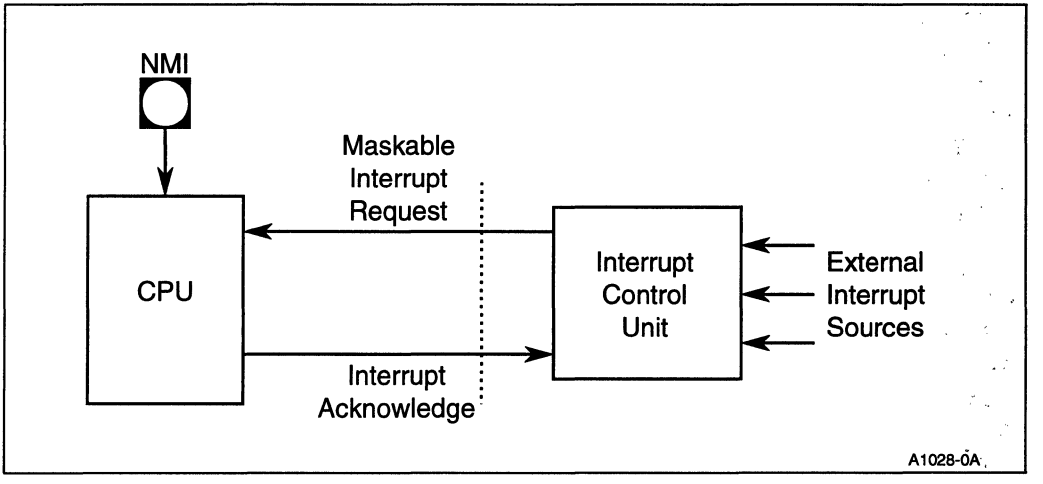
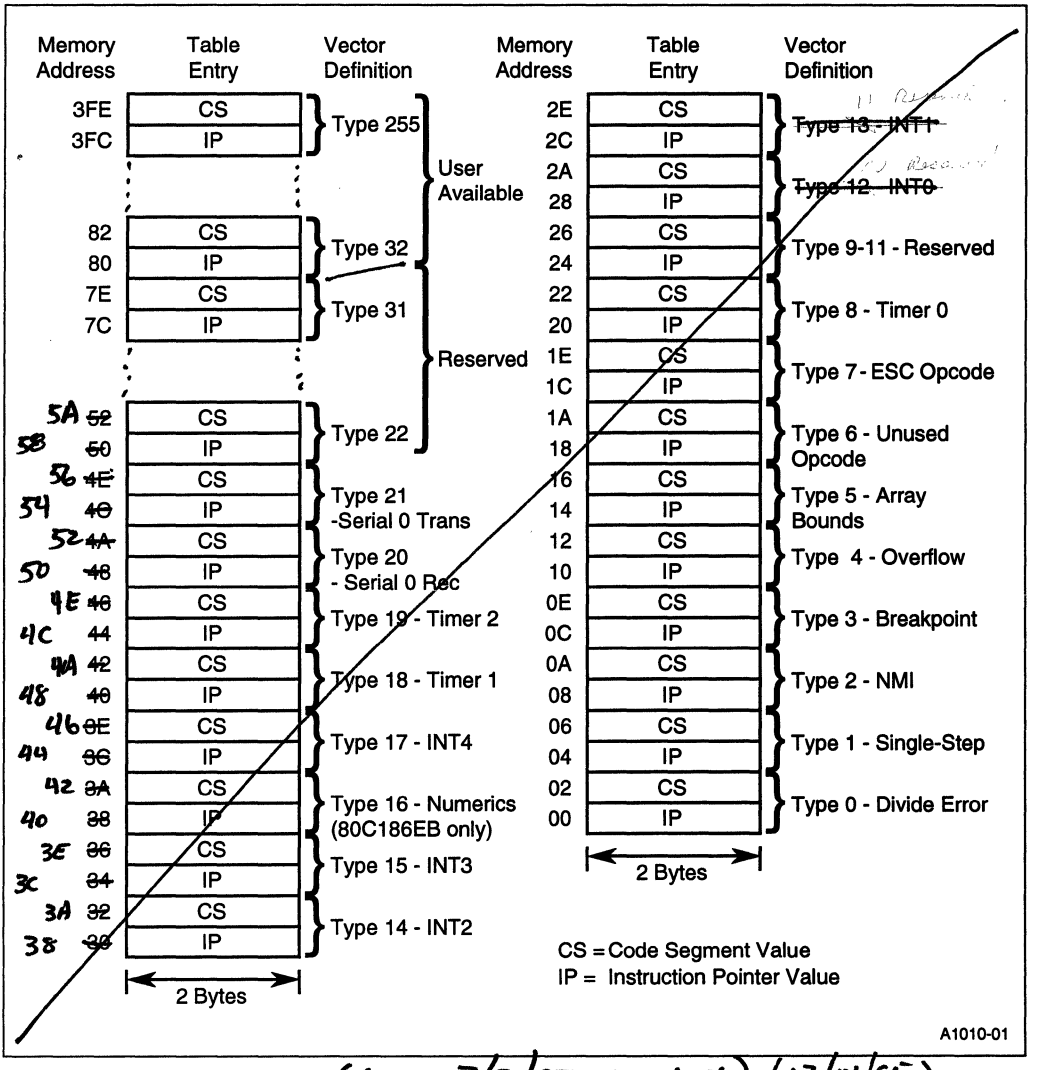


Figure 2-24. Interrupt Control Unit

2.3.1 Interrupt/Exception Processing

The 80C186 Modular Core can service up to 256 different interrupts and exceptions. A 256-entry Interrupt Vector Table (Figure 2-25) contains the pointers to interrupt service routines. Each entry consists of four bytes, which contain the Code Segment (CS) and Instruction Pointer (IP) of the first instruction in the interrupt service routine. Each interrupt or exception is given a type number, 0 through 255, corresponding to its position in the Interrupt Vector Table. Note that interrupt types 0–31 are reserved for Intel and should **not** be used by an application program.



(SEE 7/31/95 ERRATA) (12/14/95 ERRATA)

Figure 2-25. Interrupt Vector Table

When an interrupt is acknowledged, a common event sequence (Figure 2-26) allows the processor to execute the interrupt service routine.

1. The processor saves a partial machine status by pushing the Processor Status Word onto the stack.

2. The Trap Flag bit and Interrupt Enable bit are cleared in the Processor Status Word. This prevents maskable interrupts or single step exceptions from interrupting the processor during the interrupt service routine.
3. The current CS and IP are pushed onto the stack.
4. The CPU fetches the new CS and IP for the interrupt vector routine from the Interrupt Vector Table and begins executing from that point.

The CPU is now executing the interrupt service routine. The programmer must save (usually by pushing onto the stack) all registers used in the interrupt service routine; otherwise, their contents will be lost. To allow nesting of maskable interrupts, the programmer must set the Interrupt Enable bit in the Processor Status Word.

When exiting an interrupt service routine, the programmer must restore (usually by popping off the stack) the saved registers and execute an IRET instruction, which performs the following steps.

1. Loads the return CS and IP by popping them off the stack.
2. Pops and restores the old Processor Status Word from the stack.

The CPU now executes from the point at which the interrupt or exception occurred.

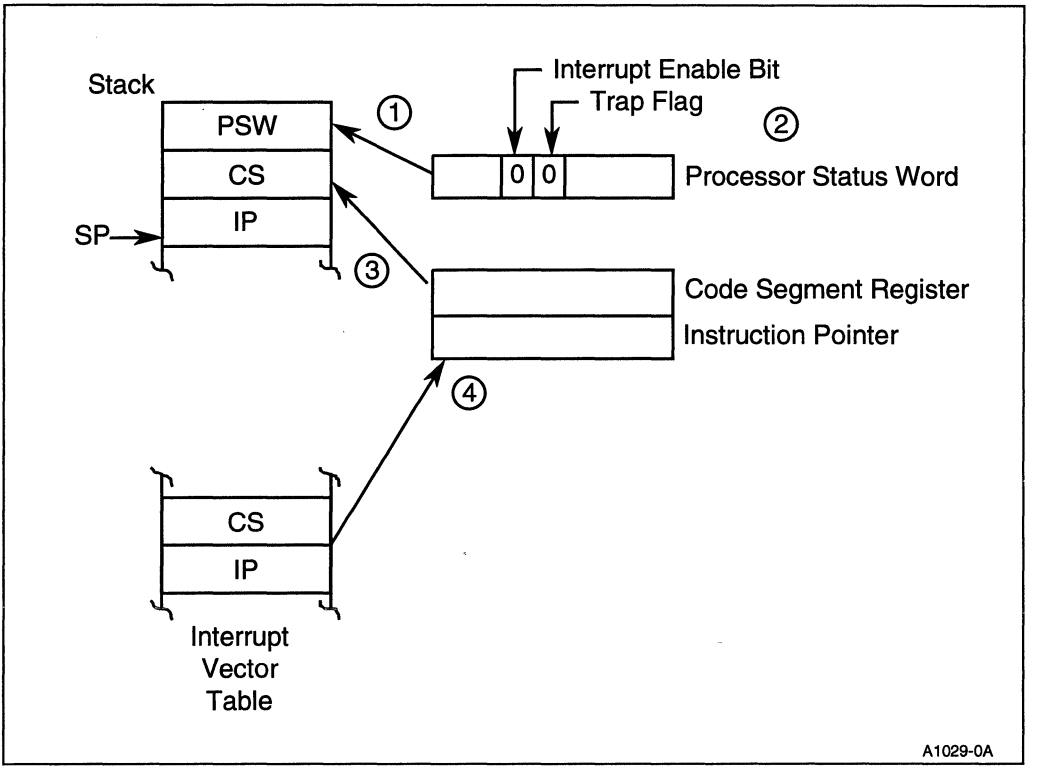


Figure 2-26. Interrupt Sequence

2.3.1.1 Non-Maskable Interrupts

The Non-Maskable Interrupt (NMI) is the highest priority interrupt. It is usually reserved for a catastrophic event such as impending power failure. An NMI cannot be prevented (or masked) by software. When the NMI input is asserted, the interrupt processing sequence begins after execution of the current instruction completes (see "Interrupt Latency" on page 2-45). The CPU automatically generates a type 2 interrupt vector.

The NMI input is asynchronous. Setup and hold times are given only to guarantee recognition on a specific clock edge. To be recognized, NMI must be asserted for at least one CLKOUT period and meet the correct setup and hold times. NMI is edge-triggered and level-latched. Multiple NMI requests cause multiple NMI service routines to be executed. NMI can be nested in this manner an infinite number of times.

2.3.1.2 Maskable Interrupts

Maskable interrupts are the most common way to service external hardware interrupts. Software can globally enable or disable maskable interrupts. This is done by setting or clearing the Interrupt Enable bit in the Processor Status Word.

The Interrupt Control Unit processes the multiple sources of maskable interrupts and presents them to the core via a single maskable interrupt input. The Interrupt Control Unit provides the interrupt vector type to the 80C186 Modular Core. The Interrupt Control Unit differs among members of the 80C186 Modular Core family; see Chapter 8, "Interrupt Control Unit," for information.

2.3.1.3 Exceptions

Exceptions occur when an unusual condition prevents further instruction processing until the exception is corrected. The CPU handles software interrupts and exceptions in the same way. The interrupt type for an exception is either predefined or supplied by the instruction.

Exceptions are classified as either faults or traps, depending on when the exception is detected and whether the instruction that caused the exception can be restarted. Faults are detected and serviced **before** the faulting instruction can be executed. The return address pushed onto the stack in the interrupt processing instruction points to the beginning of the faulting instruction. This allows the instruction to be restarted. Traps are detected and serviced immediately **after** the instruction that caused the trap. The return address pushed onto the stack during the interrupt processing points to the instruction following the trapping instruction.

Divide Error — Type 0

A Divide Error trap is invoked when the quotient of an attempted division exceeds the maximum value of the destination. A divide-by-zero is a common example.

Single Step — Type 1

The Single Step trap occurs after the CPU executes one instruction with the Trap Flag (TF) bit set in the Processor Status Word. This allows programs to execute one instruction at a time. Interrupts are not generated after prefix instructions (e.g., REP), after instructions that modify segment registers (e.g., POP DS) or after the WAIT instruction. Vectoring to the single-step interrupt service routine clears the Trap Flag bit. An IRET instruction in the interrupt service routine restores the Trap Flag bit to logic "1" and transfers control to the next instruction to be single-stepped.

Breakpoint Interrupt — Type 3

The Breakpoint Interrupt is a single-byte version of the INT instruction. It is commonly used by software debuggers to set breakpoints in RAM. Because the instruction is only one byte long, it can substitute for any instruction.

Interrupt on Overflow — Type 4

The Interrupt on Overflow trap occurs if the Overflow Flag (OF) bit is set in the Processor Status Word and the INT0 instruction is executed. Interrupt on Overflow is a common method for handling arithmetic overflows conditionally.

Array Bounds Check — Type 5

An Array Bounds trap occurs when the array index is outside the array bounds during execution of the BOUND instruction (see Appendix A, "80C186 Instruction Set Additions and Extensions").

Invalid Opcode — Type 6

Execution of an undefined opcode causes an Invalid Opcode trap.

Escape Opcode — Type 7

The Escape Opcode fault is used for floating point emulation. With 80C186 Modular Core family members, this fault is enabled by setting the Escape Trap (ET) bit in the Relocation Register (see Chapter 4, "Peripheral Control Block"). When a floating point instruction is executed with the Escape Trap bit set, the Escape Opcode fault occurs, and the Escape Opcode service routine emulates the floating point instruction. If the Escape Trap bit is cleared, the CPU sends the floating point instruction to an external 80C187.

80C188 Modular Core Family members do not support the 80C187 interface and always generate the Escape Opcode Fault.

Numerics Coprocessor Fault — Type 16

The Numerics Coprocessor fault is caused by an external 80C187 numerics coprocessor. The 80C187 reports the exception by asserting the $\overline{\text{ERROR}}$ pin. The 80C186 Modular Core checks the $\overline{\text{ERROR}}$ pin only when executing a numerics instruction. A Numerics Coprocessor Fault indicates that the **previous** numerics instruction caused the exception. The 80C187 saves the address of the floating point instruction that caused the exception. The return address pushed onto the stack during the interrupt processing points to the numerics instruction that detected the exception. This way, the last numerics instruction can be restarted.

2.3.2 Software Interrupts

A Software Interrupt is caused by executing an "INT*n*" instruction. The *n* parameter corresponds to the specific interrupt type to be executed. The interrupt type can be any number between 0 and 255. If the *n* parameter corresponds to an interrupt type associated with a hardware interrupt (NMI, Timers), the vectors are fetched and the routine is executed, but the corresponding bits in the Interrupt Status register **are not altered**.

The CPU processes software interrupts and exceptions in the same way. Software interrupts, exceptions and traps cannot be masked.

2.3.3 Interrupt Latency

Interrupt latency is the amount of time it takes for the CPU to recognize the existence of an interrupt. The CPU generally recognizes interrupts only between instructions or on instruction boundaries. Therefore, the current instruction must finish executing before an interrupt can be recognized.

The worst-case 80C186 instruction execution time is an integer divide instruction with segment override prefix. The instruction takes 69 clocks, assuming an 80C186 Modular Core family member and a zero wait-state external bus. The execution time for an 80C188 Modular Core family member may be longer, depending on the queue.

This is one factor in determining interrupt latency. In addition, the following are also factors in determining maximum latency:

1. The CPU does not recognize the Maskable Interrupt unless the Interrupt Enable bit is set.
2. The CPU does not recognize interrupts during HOLD.
3. Once communication is completely established with an 80C187, the CPU does not recognize interrupts until the numerics instruction is finished.

The CPU can recognize interrupts only on valid instruction boundaries. A valid instruction boundary usually occurs when the current instruction finishes. The following is a list of exceptions:

1. MOVs and POPs referencing a segment register delay the servicing of interrupts until after the following instruction. The delay allows a 32-bit load to the SS and SP without an interrupt occurring between the two loads.
2. The CPU allows interrupts between repeated string instructions. If multiple prefixes precede a string instruction and the instruction is interrupted, only the one prefix preceding the string primitive is restored.
3. The CPU can be interrupted during a WAIT instruction. The CPU will return to the WAIT instruction.

2.3.4 Interrupt Response Time

Interrupt response time is the time from the CPU recognizing an interrupt until the first instruction in the service routine is executed. Interrupt response time is less for interrupts or exceptions which supply their own vector type. The maskable interrupt has a longer response time because the vector type must be supplied by the Interrupt Control Unit (see Chapter 8, "Interrupt Control Unit").

Figure 2-27 shows the events that dictate interrupt response time for the interrupts that supply their type. Note that an on-chip bus master, such as the DRAM Refresh Unit, can make use of idle bus cycles. This can increase interrupt response time.

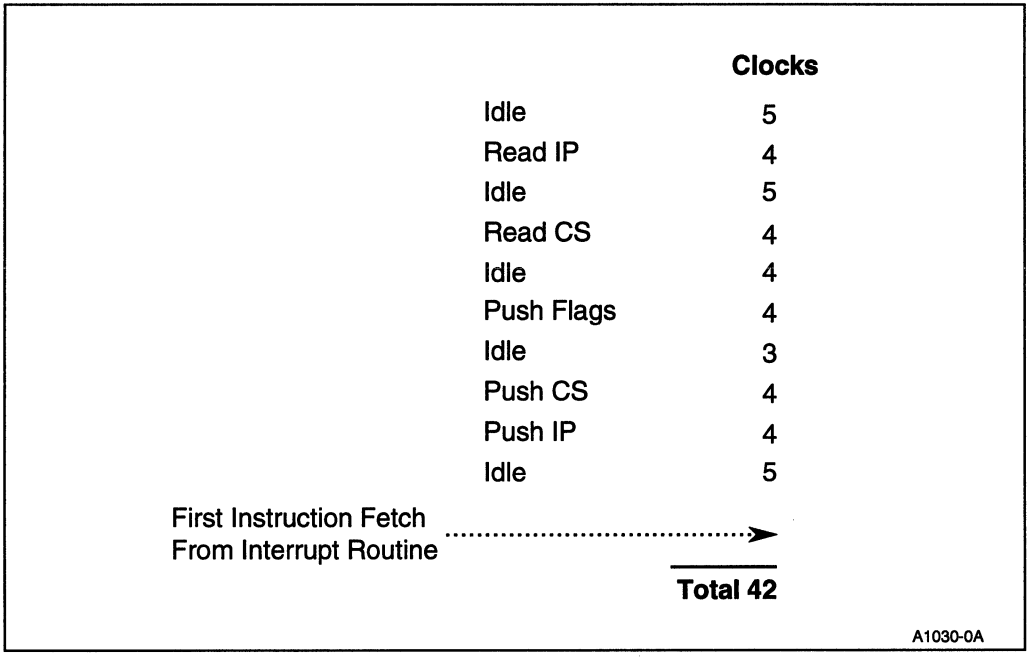


Figure 2-27. Interrupt Response Factors

2.3.5 Interrupt and Exception Priority

Interrupts can be recognized only on valid instruction boundaries. If an NMI and a maskable interrupt are both recognized on the same instruction boundary, NMI has precedence. The maskable interrupt will not be recognized until the Interrupt Enable bit is set and it is the highest priority.

Only the single step exception can occur concurrently with another exception. At most, two exceptions can occur at the same instruction boundary and one of those exceptions must be the single step. Single step is a special case; it is discussed on page 2-48. Ignoring single step (for now), only one exception can occur at any given instruction boundary.

An exception has priority over both NMI and the maskable interrupt. However, a pending NMI can interrupt the CPU at any valid instruction boundary. Therefore, NMI can interrupt an exception service routine. If an exception and NMI occur simultaneously, the exception vector is taken, then is followed immediately by the NMI vector (see Figure 2-28). While the exception has higher priority at the instruction boundary, the NMI interrupt service routine is executed first.

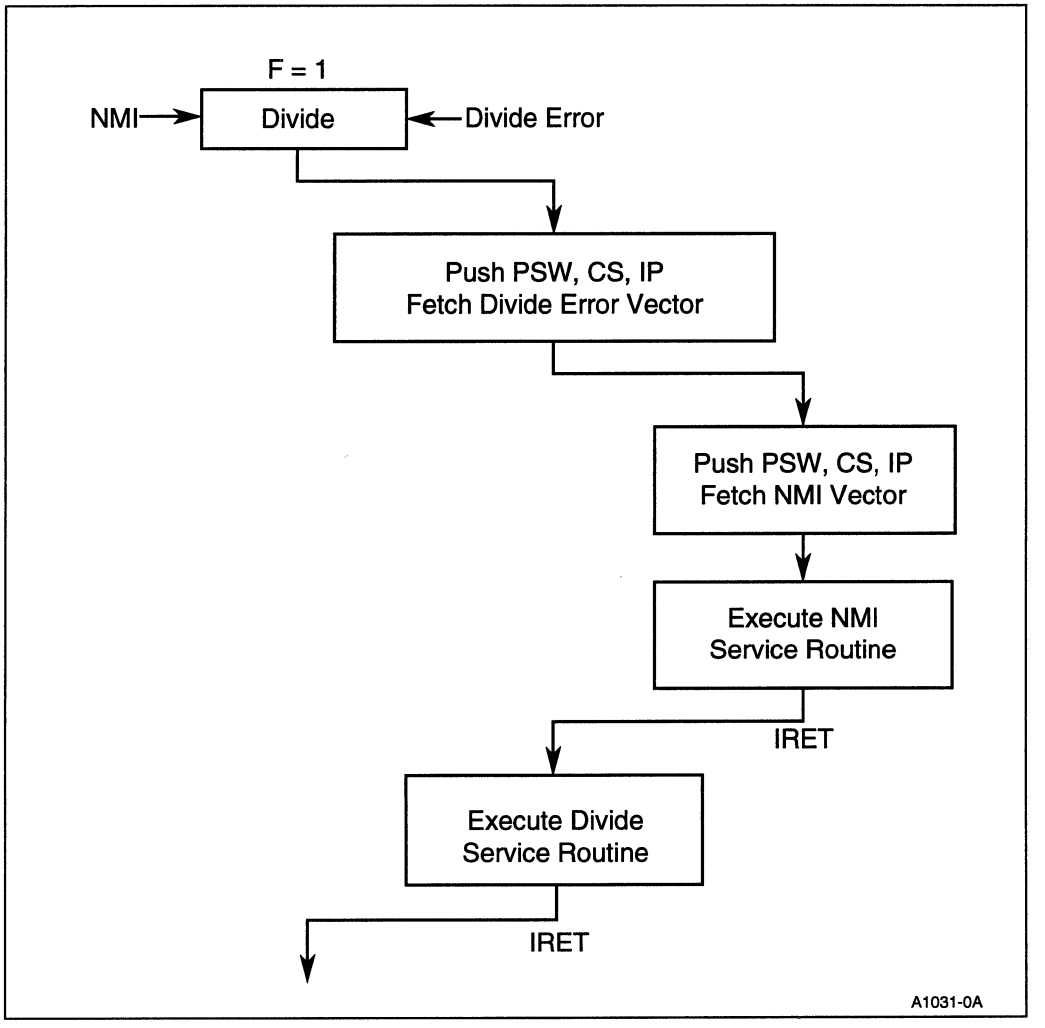


Figure 2-28. Simultaneous NMI and Exception

Single step priority is a special case. If an interrupt (NMI or maskable) occurs at the same instruction boundary as a single step, the interrupt vector is taken first, then is followed immediately by the single step vector. However, the single step service routine is executed before the interrupt service routine (see Figure 2-29). If the single step service routine re-enables single step by executing the IRET, the interrupt service routine will also be single stepped. This can severely limit the real-time response of the CPU to an interrupt.

To prevent the single-step routine from executing before a maskable interrupt, disable interrupts while single stepping an instruction, then enable interrupts in the single step service routine. The maskable interrupt is serviced from within the single step service routine and that interrupt service routine is not single-stepped. To prevent single stepping before an NMI, the single-step service routine must compare the return address on the stack to the NMI vector. If they are the same, return to the NMI service routine immediately without executing the single step service routine.

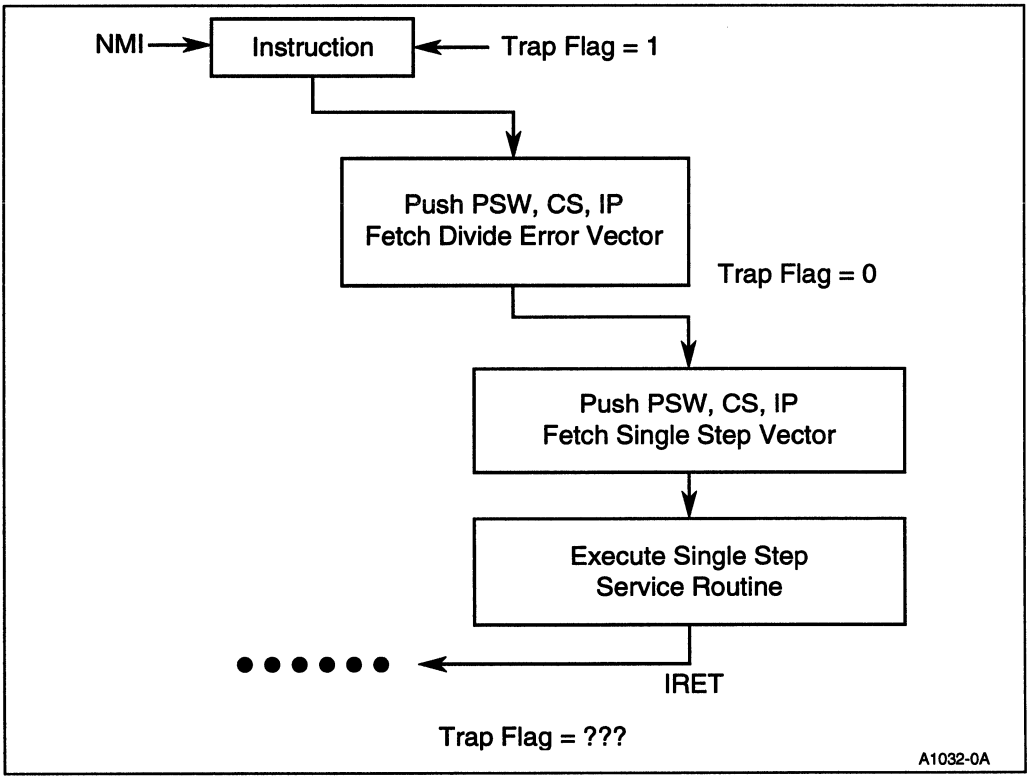


Figure 2-29. Simultaneous NMI and Single Step Interrupts

The most complicated case is when an NMI, a maskable interrupt, a single step and another exception are pending on the same instruction boundary. Figure 2-30 shows how this case is prioritized by the CPU. Note that if the single-step routine sets the Trap Flag (TF) bit before executing the IRET instruction, the NMI routine will also be single stepped.

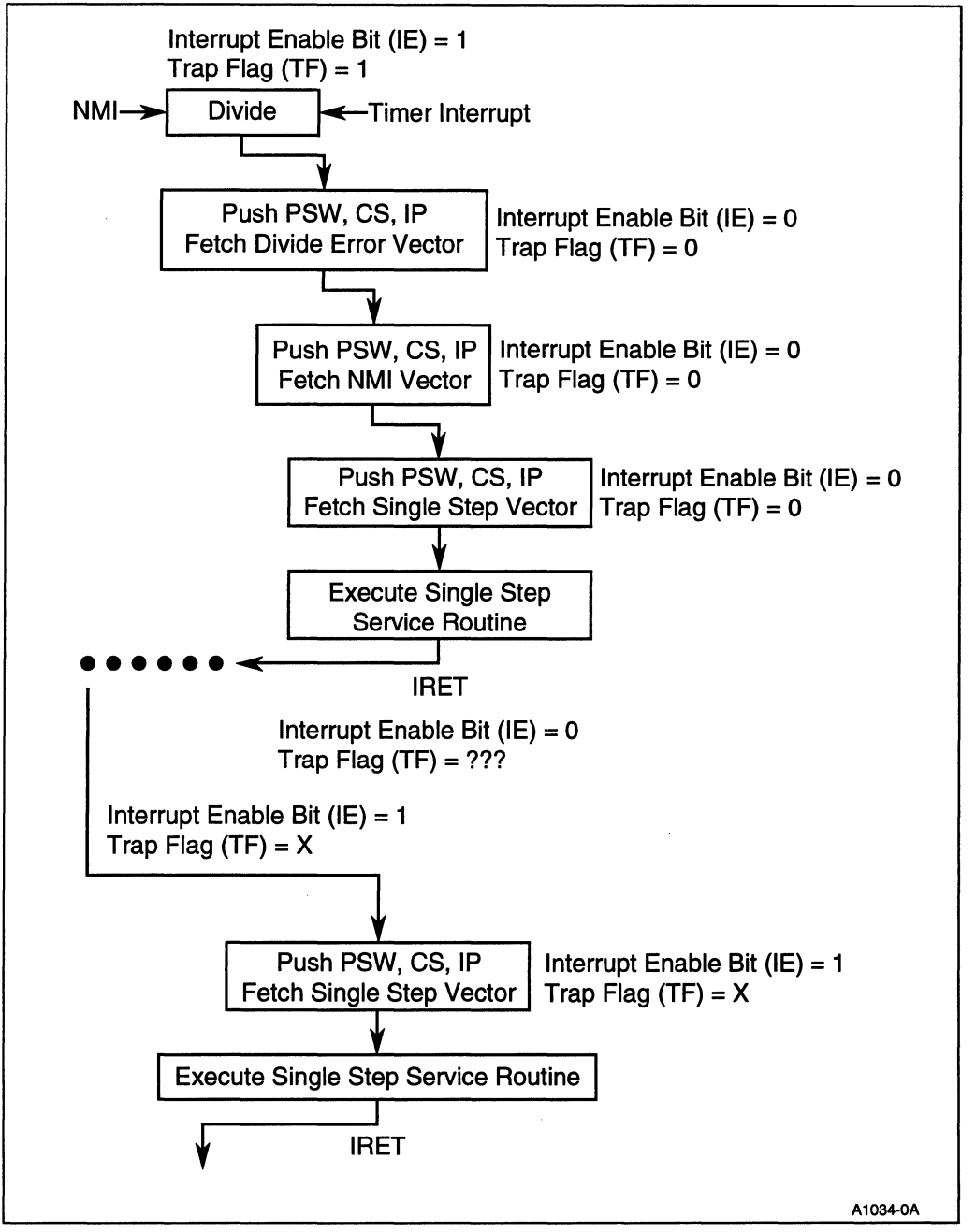


Figure 2-30. Simultaneous NMI, Single Step and Maskable Interrupt

intel®

3

Bus Interface Unit

CHAPTER 3
BUS INTERFACE UNIT

The Bus Interface Unit (BIU) generates bus cycles that prefetch instructions from memory, pass data to and from the execution unit, and pass data to and from the integrated peripheral units.

The BIU drives address, data, status and control information to define a bus cycle. The start of a bus cycle presents the address of a memory or I/O location and status information defining the type of bus cycle. Read or write control signals follow the address and define the direction of data flow. A read cycle requires data to flow from the selected memory or I/O device to the BIU. In a write cycle, the data flows from the BIU to the selected memory or I/O device. Upon termination of the bus cycle, the BIU latches read data or removes write data.

3.1 MULTIPLEXED ADDRESS AND DATA BUS

The BIU has a combined address and data bus, commonly referred to as a time-multiplexed bus. Time multiplexing address and data information makes the most efficient use of device package pins. A system with address latching provided within the memory and I/O devices can directly connect to the address/data bus (or *local bus*). The local bus can be demultiplexed with a single set of address latches to provide non-multiplexed address and data information to the system.

3.2 ADDRESS AND DATA BUS CONCEPTS

The programmer views the memory or I/O address space as a sequence of bytes. Memory space consists of 1 Mbyte, while I/O space consists of 64 Kbytes. Any byte can contain an 8-bit data element, and any two consecutive bytes can contain a 16-bit data element (identified as a word). The discussions in this section apply to both memory and I/O bus cycles. For brevity, memory bus cycles are used for examples and illustration.

3.2.1 16-Bit Data Bus

The memory address space on a 16-bit data bus is physically implemented by dividing the address space into two banks of up to 512 Kbytes each (see Figure 3-1). One bank connects to the lower half of the data bus and contains even-addressed bytes (A0=0). The other bank connects to the upper half of the data bus and contains odd-addressed bytes (A0=1). Address lines A19:1 select a specific byte within each bank. A0 and Byte High Enable ($\overline{BHE}$) determine whether one bank or both banks participate in the data transfer.

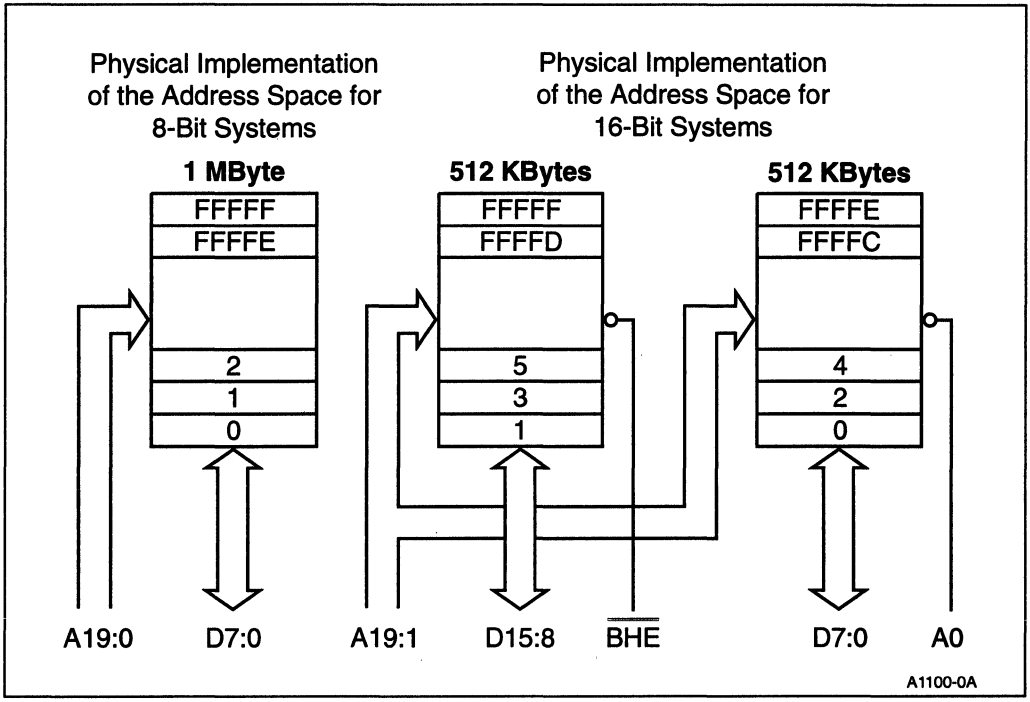


Figure 3-1. Physical Data Bus Models

Byte transfers to even addresses transfer information over the lower half of the data bus (see Figure 3-2). A0 low enables the lower bank, while $\overline{BHE}$ high disables the upper bank. The data value from the upper bank is ignored during a bus read cycle. $\overline{BHE}$ high prevents a write operation from destroying data in the upper bank.

Byte transfers to odd addresses transfer information over the upper half of the data bus (see Figure 3-2). $\overline{BHE}$ low enables the upper bank, while A0 high disables the lower bank. The data value from the lower bank is ignored during a bus read cycle. A0 high prevents a write operation from destroying data in the lower bank.

To access even-addressed 16-bit words (two consecutive bytes with the least-significant byte at an even address), information is transferred over both halves of the data bus (see Figure 3-3). A19:1 select the appropriate byte within each bank. A0 and $\overline{BHE}$ drive low to enable both banks simultaneously.

Odd-addressed word accesses require the BIU to split the transfer into two byte operations (see Figure 3-4). The first operation transfers data over the upper half of the bus, while the second operation transfers data over the lower half of the bus. The BIU automatically executes the two-byte sequence whenever an odd-addressed word access is performed.

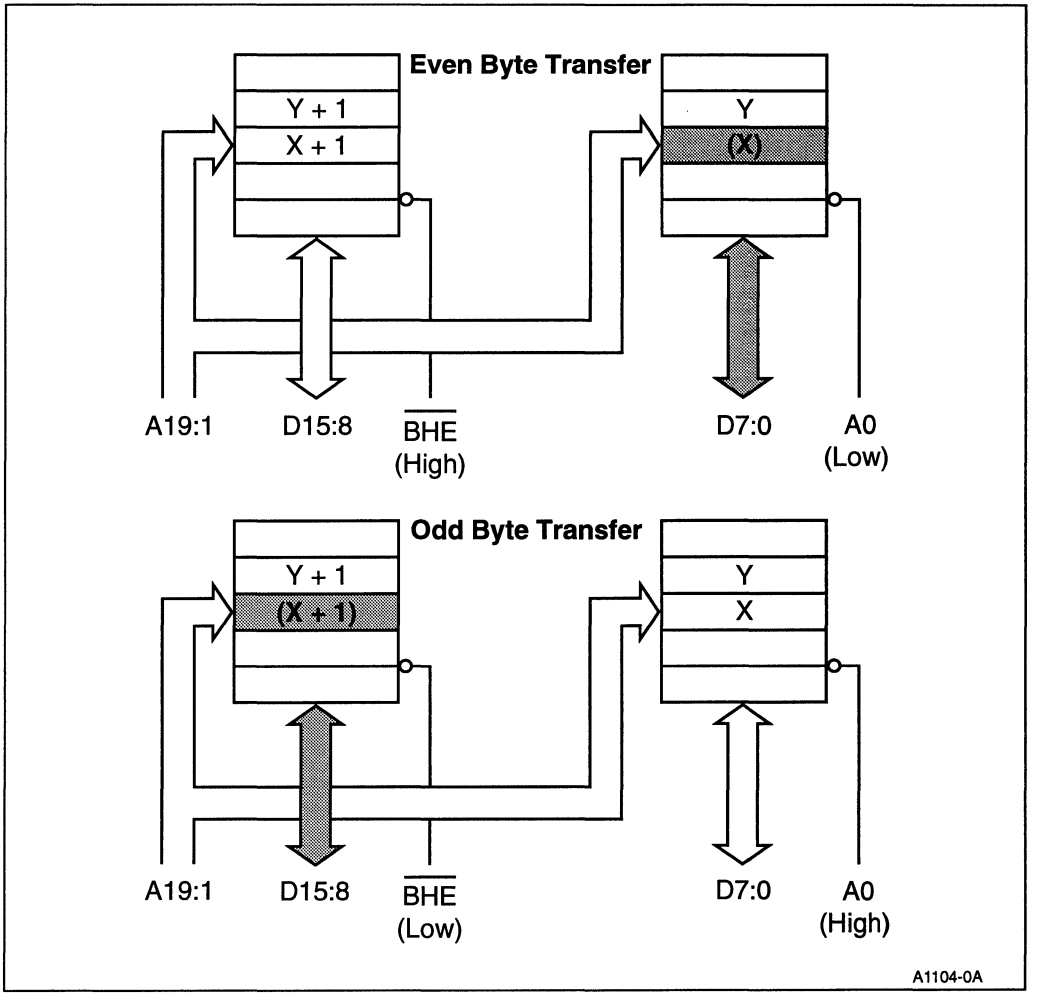


Figure 3-2. 16-Bit Data Bus Byte Transfers

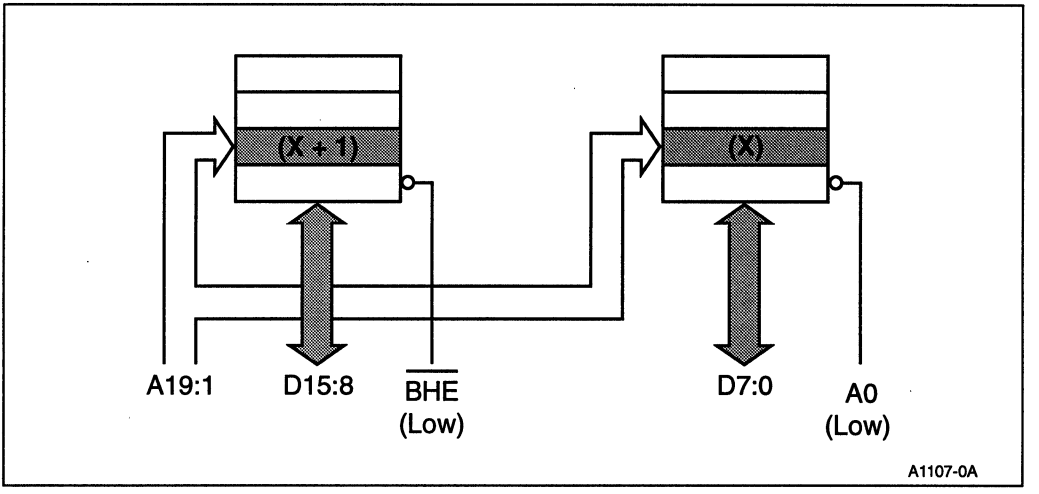


Figure 3-3. 16-Bit Data Bus Even Word Transfers

During a byte read operation, the BIU floats the entire 16-bit data bus, even though the transfer occurs on only one half of the bus. This action simplifies the decoding requirements for read-only devices (e.g., ROM, EPROM, Flash). During the byte read, an external device can drive **both halves** of the bus, and the BIU automatically accesses the correct half. During the byte write operation, the BIU drives both halves of the bus. Information on the half of the bus not involved in the transfer is indeterminate. This action requires that the appropriate bank (defined by $\overline{BHE}$ or A0 high) be disabled to prevent destroying data.

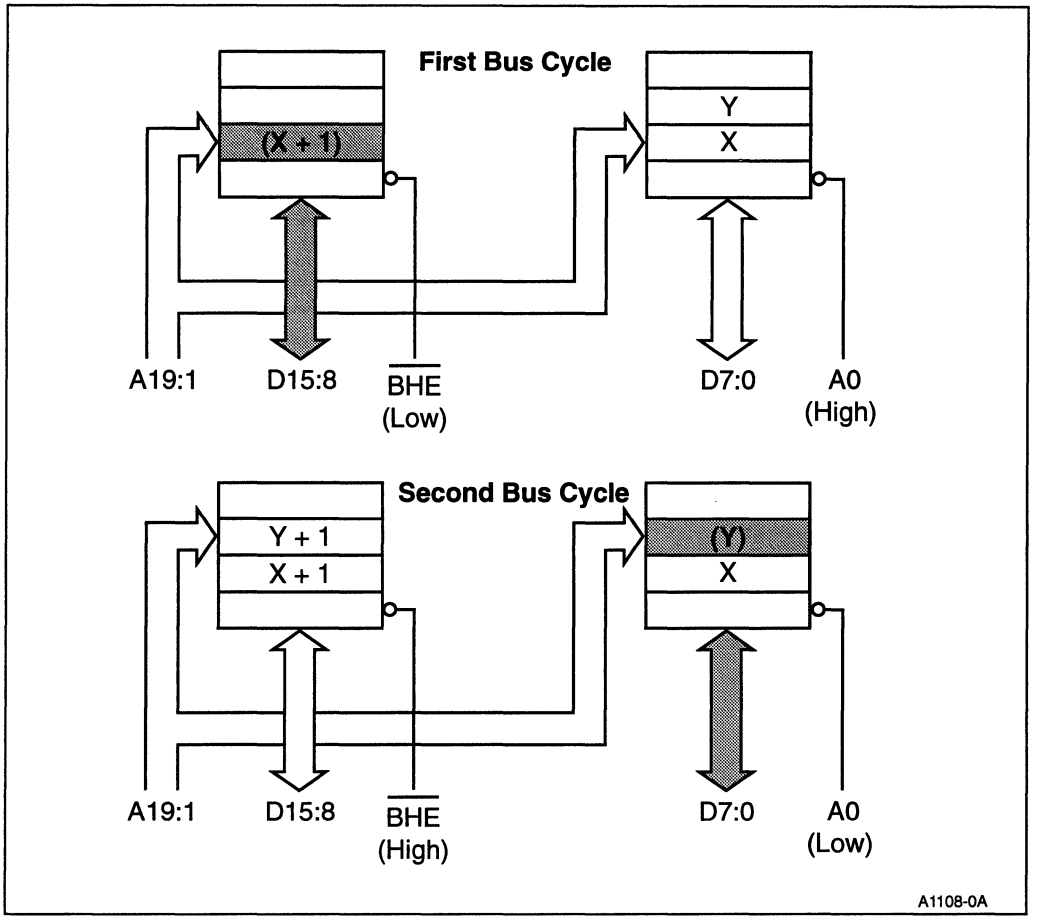


Figure 3-4. 16-Bit Data Bus Odd Word Transfers

3.2.2 8-Bit Data Bus

The memory address space on an 8-bit data bus is physically implemented as one bank of 1 Mbyte (see Figure 3-1 on page 3-2). Address lines A19:0 select a specific byte within the bank. Unlike transfers with a 16-bit bus, byte and word transfers (to even or odd addresses) all transfer data over the same 8-bit bus.

Byte transfers to even or odd addresses transfer information in one bus cycle. Word transfers to even or odd addresses transfer information in two bus cycles. The BIU automatically converts the word access into two consecutive byte accesses, making the operation transparent to the programmer.

For word transfers, the word address defines the first byte transferred. The second byte transfer occurs from the word address plus one. Figure 3-5 illustrates a word transfer on an 8-bit bus interface.

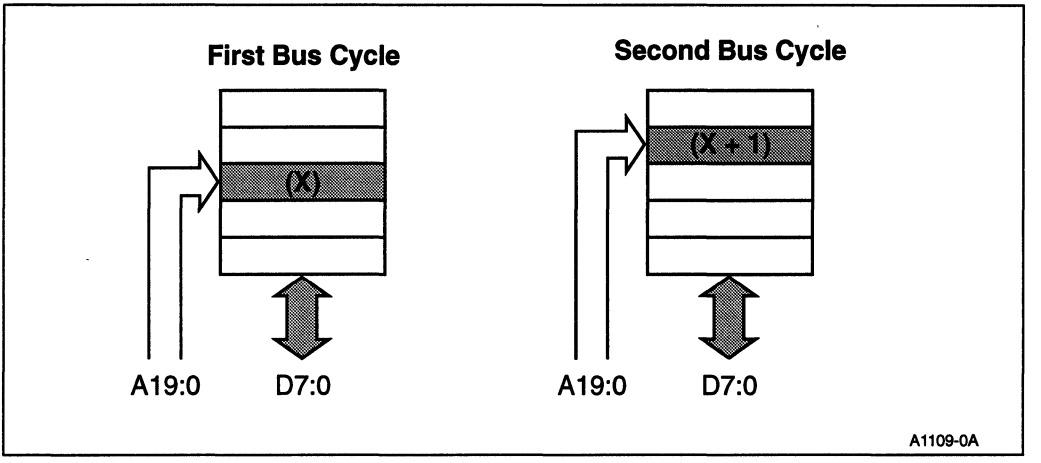


Figure 3-5. 8-Bit Data Bus Word Transfers

3.3 MEMORY AND I/O INTERFACES

The CPU can interface with 8- and 16-bit memory and I/O devices. Memory devices exchange information with the CPU during memory read, memory write and instruction fetch bus cycles. I/O (peripheral) devices exchange information with the CPU during memory read, memory write, I/O read, I/O write and interrupt acknowledge bus cycles. Memory-mapped I/O refers to peripheral devices that exchange information during memory cycles. Memory-mapped I/O allows the full power of the instruction set to be used when communicating with peripheral devices.

I/O read and I/O write bus cycles use a separate I/O address space. Only IN and OUT instructions can access I/O address space, and information must be transferred between the peripheral device and the AX register. The first 256 bytes (0–255) of I/O space can be accessed directly by the I/O instructions. The entire 64 Kbyte I/O address space can be accessed only indirectly, through the DX register. I/O instructions always force address bits A19:16 to zero.

Interrupt acknowledge, or $\overline{\text{INTA}}$, bus cycles access an I/O device intended to increase interrupt input capability. Valid address information is **not** generated as part of the $\overline{\text{INTA}}$ bus cycle, and data is transferred only over the lower bank (16-bit device).

3.3.1 16-Bit Bus Memory and I/O Requirements

A 16-bit bus has certain assumptions that must be met to operate properly. Memory used to store instruction operands (i.e., the program) and immediate data must be 16 bits wide. Instruction prefetch bus cycles require that **both banks** be used. The lower bank contains the even bytes of code and the upper bank contains the odd bytes of code.

Memory used to store interrupt vectors and stack data must be 16 bits wide. Memory address space between 0H and 3FFH (1 Kbyte) holds the starting location of an interrupt routine. In response to an interrupt, the BIU fetches two consecutive, even-addressed words from this 1 Kbyte address space. Stack pushes and pops always write or read even-addressed word data.

3.3.2 8-Bit Bus Memory and I/O Requirements

An 8-bit bus interface has no restrictions on implementing the memory or I/O interfaces. All transfers, bytes and words, occur over the single 8-bit bus. Operations requiring word transfers automatically execute two consecutive byte transfers.

3.4 BUS CYCLE OPERATION

The BIU executes a bus cycle to transfer data between any of the integrated units and any external memory or I/O devices (see Figure 3-6). A bus cycle consists of a minimum of four CPU clocks known as "T-states." A T-state is bounded by one falling edge of CLKOUT to the next falling edge of CLKOUT (see Figure 3-7). Phase 1 represents the low time of the T-state and starts at the high-to-low transition of CLKOUT. Phase 2 represents the high time of the T-state and starts at the low-to-high transition of CLKOUT. Address, data and control signals generated by the BIU go active and inactive at different phases within a T-state.

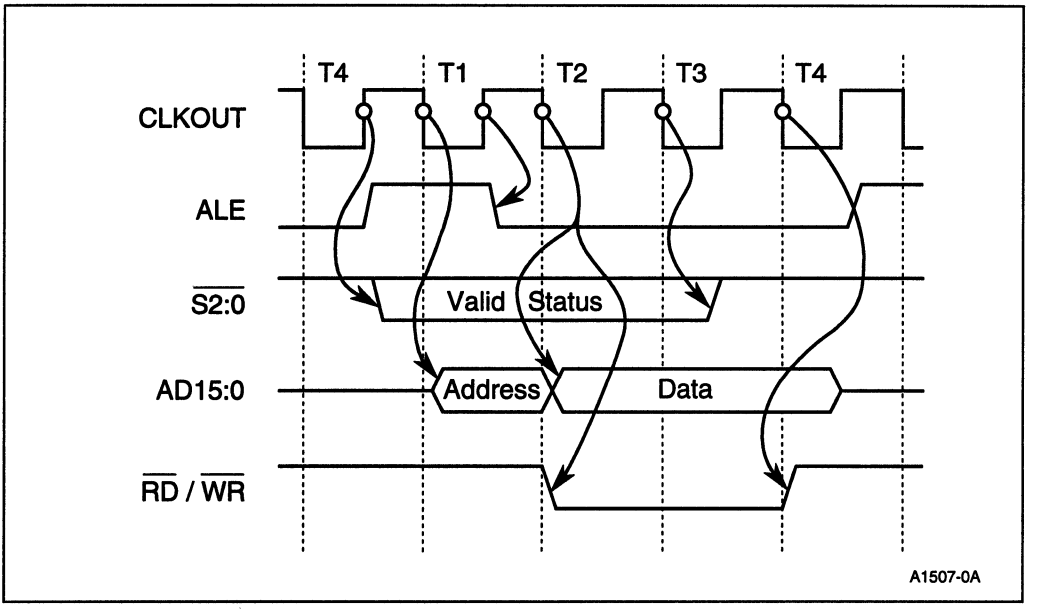


Figure 3-6. Typical Bus Cycle

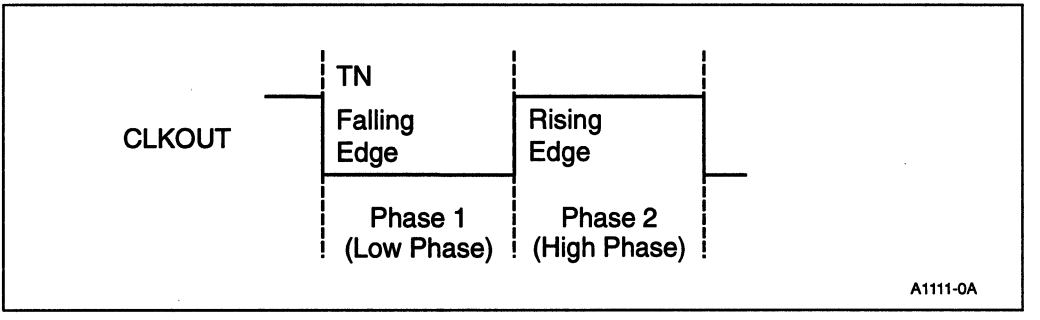


Figure 3-7. T-State Relation to CLKOUT

Figure 3-8 shows the BIU state diagram. Typically a bus cycle consists of four consecutive T-states labeled T1, T2, T3 and T4. A TI (idle) state occurs when no bus cycle is pending. Multiple T3 states occur to generate wait states. The TW symbol represents a wait state.

The operation of a bus cycle can be separated into two phases:

- Address/Status Phase
- Data Phase

The address/status phase starts just before T1 and continues through T1. The data phase starts at T2 and continues through T4. Figure 3-9 illustrates the T-state relationship of the two phases.

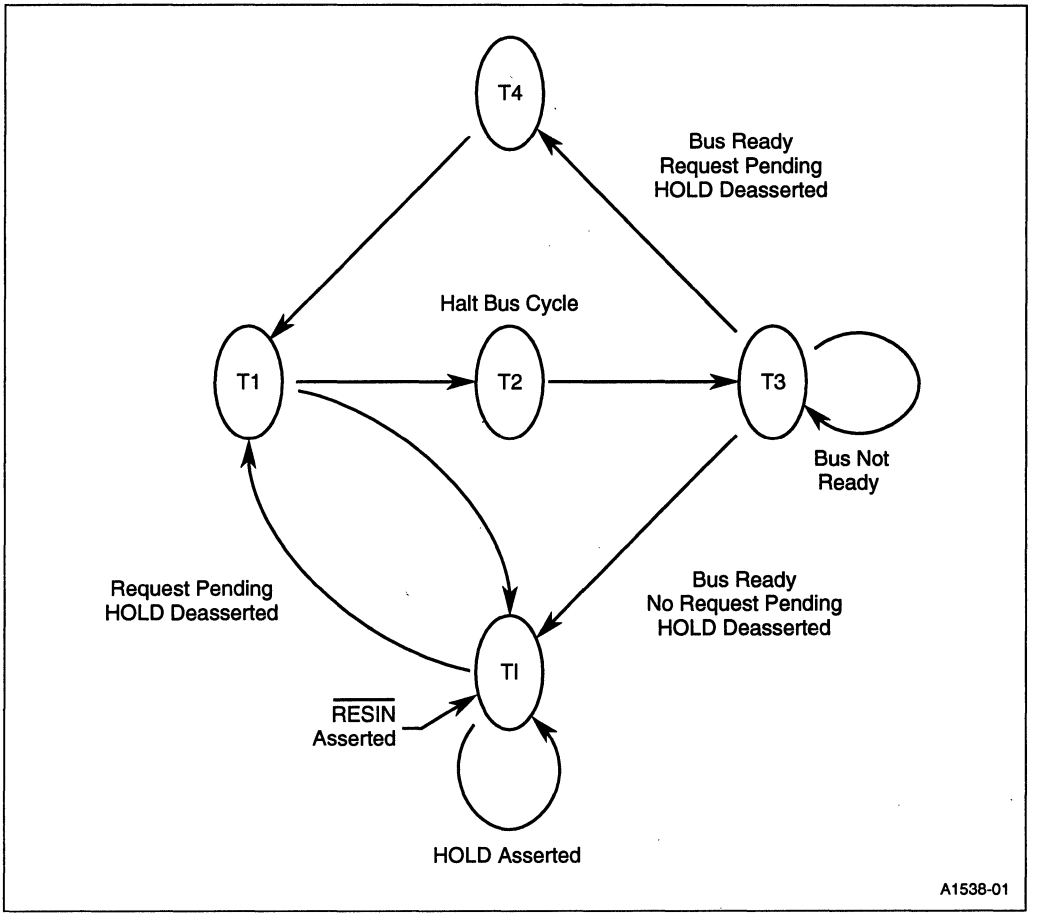


Figure 3-8. BIU State Diagram

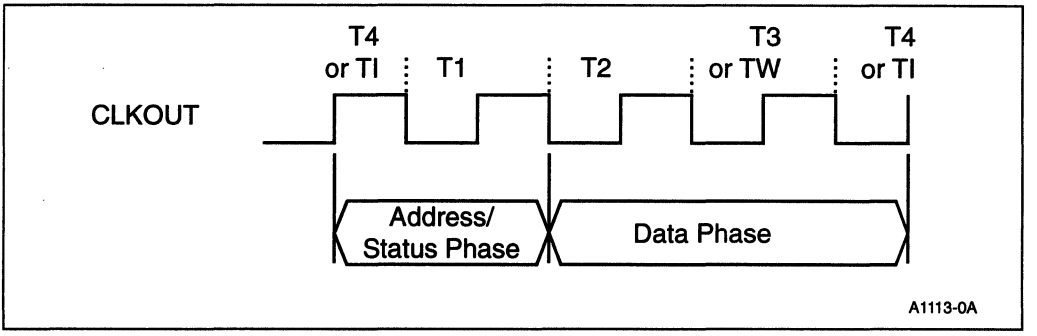


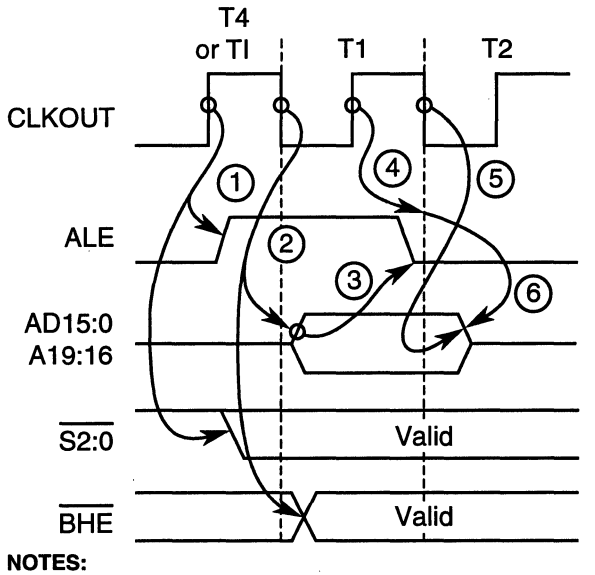
Figure 3-9. T-State and Bus Phases

3.4.1 Address/Status Phase

Figure 3-10 shows signal timing relationships for the address/status phase of a bus cycle. A bus cycle begins with the transition of ALE and $\overline{S2:0}$. These signals transition during phase 2 of the T-state just prior to T1. Either T4 or TI precedes T1, depending on the operation of the previous bus cycle (see Figure 3-8 on page 3-9).

ALE provides a strobe to latch physical address information. Address is presented on the multiplexed address/data bus during T1 (see Figure 3-10). The falling edge of ALE occurs during the middle of T1 and provides a strobe to latch the address. Figure 3-11 presents a typical circuit for latching addresses.

The status signals ($\overline{S2:0}$) define the type of bus cycle (Table 3-1). $\overline{S2:0}$ remain valid until phase 1 of T3 (or the last TW, when wait states occur). The circuit shown in Figure 3-11 can also be used to extend $\overline{S2:0}$ beyond the T3 (or TW) state.



NOTES:

1. T_{CHOV} : Clock high to ALE high, $\overline{S2:0}$ valid.
2. T_{CLOV} : Clock low to address valid, $\overline{BHE}$ valid.
3. T_{AVLL} : Address valid to ALE low (address setup to ALE).
4. T_{CHOV} : Clock high to ALE low.
5. T_{CLOF} : Clock low to address invalid (address hold from clock low).
6. T_{LLAX} : ALE low to address invalid (address hold from ALE).

A1101-0A

Figure 3-10. Address/Status Phase Signal Relationships

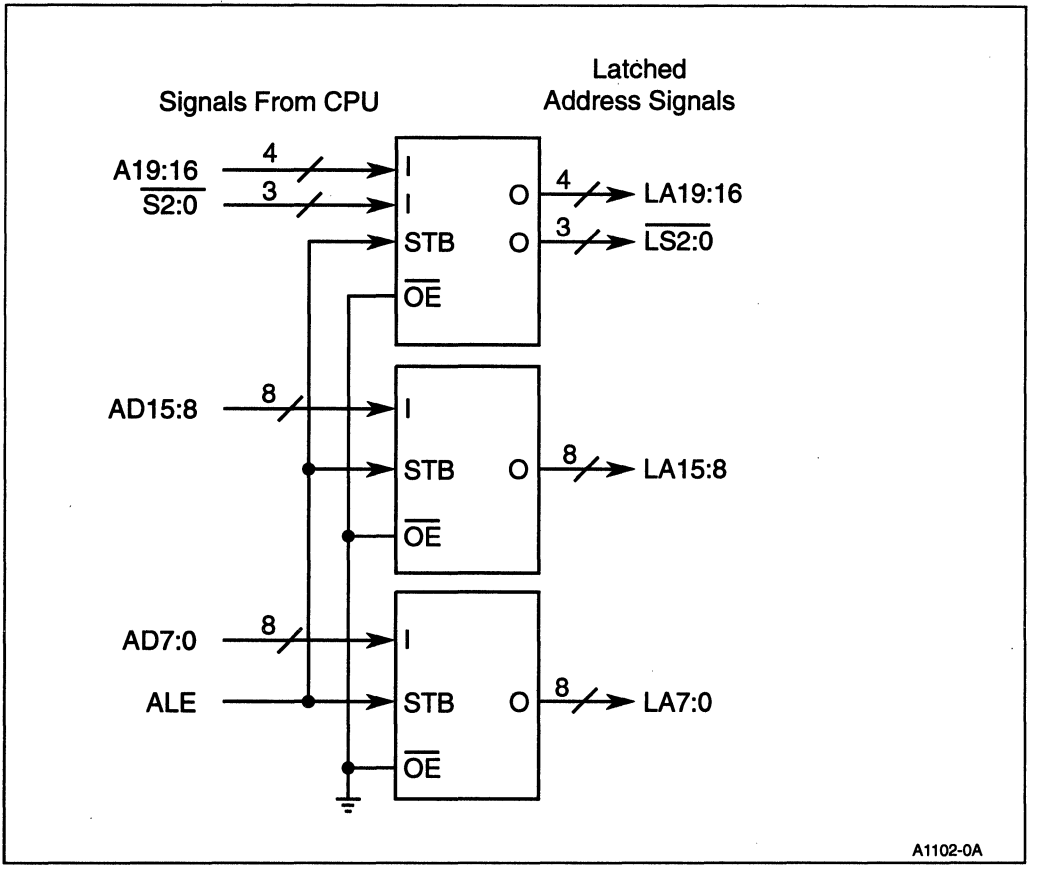


Figure 3-11. Demultiplexing Address Information

Table 3-1. Bus Cycle Types

Status Bit			Operation
S2	S1	S0	
0	0	0	Interrupt Acknowledge
0	0	1	I/O Read
0	1	0	I/O Write
0	1	1	Halt
1	0	0	Instruction Prefetch
1	0	1	Memory Read
1	1	0	Memory Write
1	1	1	Idle (passive)

3.4.2 Data Phase

Figure 3-12 shows the timing relationships for the data phase of a bus cycle. The only bus cycle type that does not have a data phase is a bus halt. During the data phase, the bus transfers information between the internal units and the memory or peripheral device selected during the address/status phase. Appropriate control signals become active to coordinate the transfer of data.

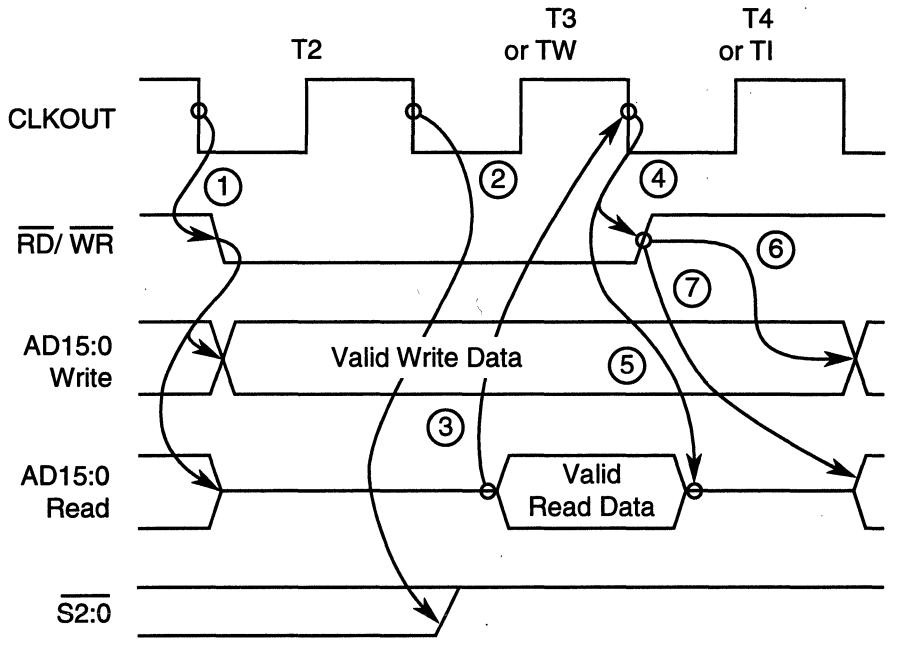
The data phase begins at phase 1 of T2 and continues until phase 2 of T4 or TI. The length of the data phase varies depending on the number of wait states. Wait states occur after T3 and before T4 or TI.

3.4.3 Wait States

Wait states extend the data phase of the bus cycle. Memory and I/O devices that cannot provide or accept data in the minimum four CPU clocks require wait states. Figure 3-13 shows a typical bus cycle with wait states inserted.

The READY input and the Chip-Select Unit control bus cycle wait states. Only the READY input is described in this chapter. (See Chapter 6, "Chip-Select Unit," for additional information.)

Figure 3-14 shows a simplified block diagram of the READY input. To avoid wait states, READY must be active (high) within a specified setup time prior to phase 2 of T2. To insert wait states, READY must be inactive (low) within a specified setup time to phase 2 of T2 or phase 1 of T3. Depending on the size and characteristics of the system, ready implementation can take one of two approaches: normally not-ready or normally ready.



NOTES:

1. T_{CLOV} : Clock low to valid $\overline{RD}$/ $\overline{WR}$ active; Write data valid
2. T_{CLOV} : Clock low to status inactive
3. T_{CLIS} : Data input valid to clock low
4. T_{CLOV} : Clock valid to $\overline{RD}$/ $\overline{WR}$ inactive
5. T_{CLIH} : Data input HOLD from clock low
6. T_{WHDX} : Output data HOLD from $\overline{WR}$ high
7. T_{RHAV} : Bus no longer floating from $\overline{RD}$ high

A1103-0A

Figure 3-12. Data Phase Signal Relationships

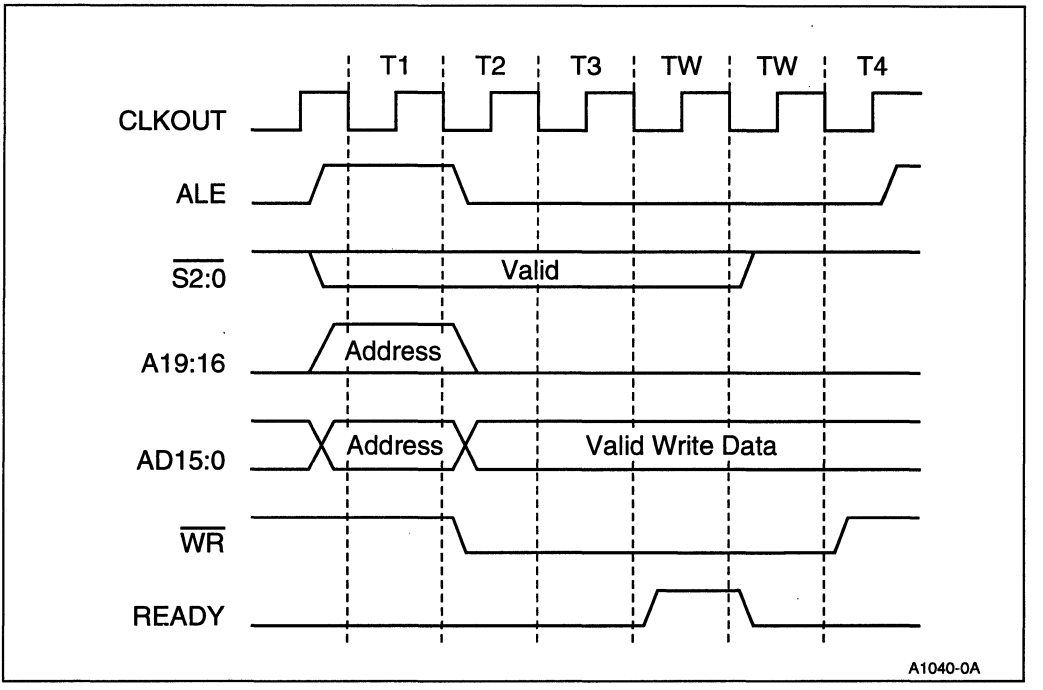


Figure 3-13. Typical Bus Cycle with Wait States

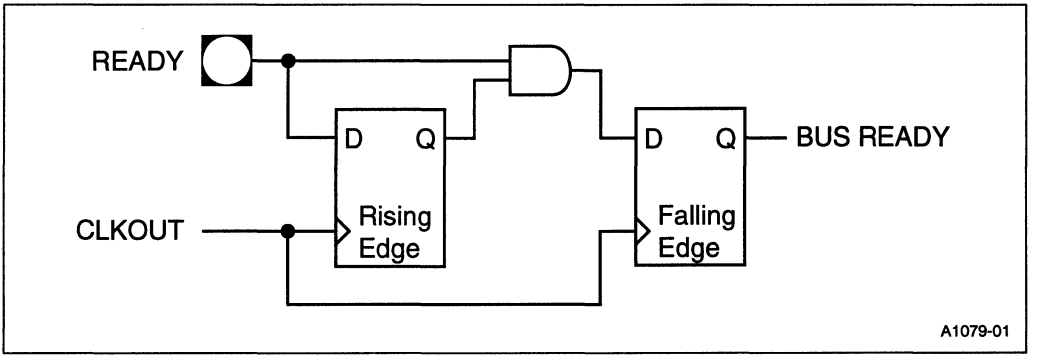


Figure 3-14. READY Pin Block Diagram

A normally not-ready system is one in which READY remains low at all times except to signal a ready condition. For any bus cycle, only the selected device drives the READY input high to complete the bus cycle. The circuit shown in Figure 3-15 illustrates a simple circuit to generate a normally not-ready signal. **Note that if no device is selected the bus remains not-ready indefinitely.** Systems with many slow devices that cannot operate at the maximum bus bandwidth usually implement a normally not-ready signal.

The start of a bus cycle clears the wait state module and forces READY low. After every rising edge of CLKOUT, INPUT1 and INPUT2 are shifted through the module and eventually drive READY high. Assuming INPUT1 and INPUT2 are valid prior to phase 2 of T2, no delay through the module causes one wait state. Each additional clock delay through the module generates one additional wait state. Two inputs are used to establish different wait state conditions.

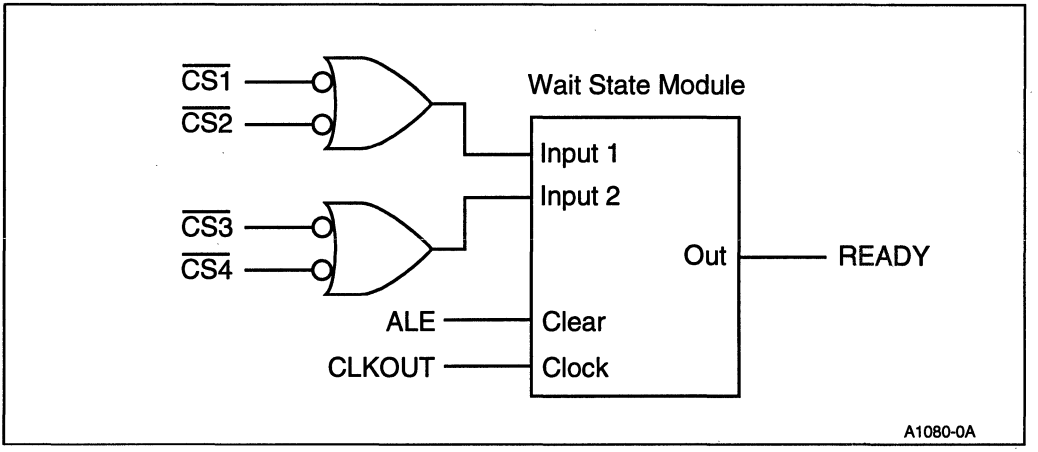


Figure 3-15. Generating a Normally Not-Ready Bus Signal

A normally ready signal remains high at all times except when the selected device needs to signal a not-ready condition. For any bus cycle, only the selected device drives the READY input low to delay the completion of the bus cycle. The circuit shown in Figure 3-16 illustrates a simple circuit to generate a normally ready signal. **Note that if no device is selected the bus remains ready.** Systems that have few or no devices requiring wait states usually implement a normally ready signal.

The start of a bus cycle preloads a zero shifter and forces READY active (high). READY remains active if neither $\overline{CS1}$ or $\overline{CS2}$ goes low. Should either $\overline{CS1}$ or $\overline{CS2}$ go low, zeros are shifted out on every rising edge of CLKOUT, causing READY to go inactive. At the end of the shift pattern, READY is forced active again. Assuming $\overline{CS1}$ and $\overline{CS2}$ are active just prior to phase 2 of T2, shifting one zero through the module causes two wait states. Each additional zero shifted through the module generates one wait state.

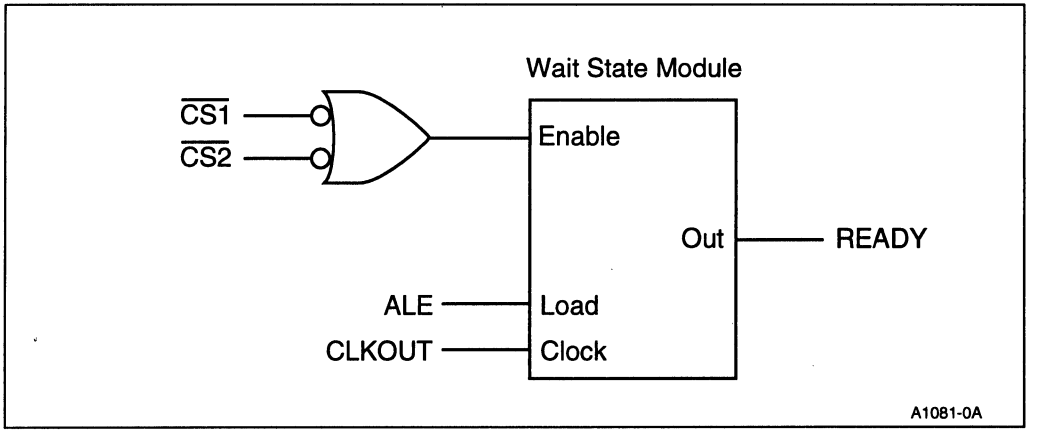


Figure 3-16. Generating a Normally Ready Bus Signal

The READY input has two major timing concerns that can affect whether a normally ready or normally not-ready signal may be required. Two latches capture the state of the READY input (see Figure 3-14 on page 3-15). The first latch captures READY on the phase 2 clock edge. The second latch captures READY and the result of first latch on the phase 1 clock edge. The following items define the requirements of the READY input to meet ready or not-ready bus conditions.

- The bus is **ready** if **both** of these two conditions are true:
  - — READY is active prior to the phase 2 clock edge, **and**
  - — READY remains active after the phase 1 clock edge.
- The bus is **not-ready** if **either** of these two conditions is true:
  - — READY is inactive prior to the phase 2 clock edge, **or**
  - — READY is inactive prior to the phase 1 clock edge.

A normally not-ready system must generate a valid READY input at phase 2 of T2 to prevent wait states. If it cannot, then running without wait states requires a normally ready system. Figure 3-17 illustrates the timing necessary to prevent wait states in a normally not-ready system. Figure 3-17 also shows how to terminate a bus cycle with wait states in a normally not-ready system.

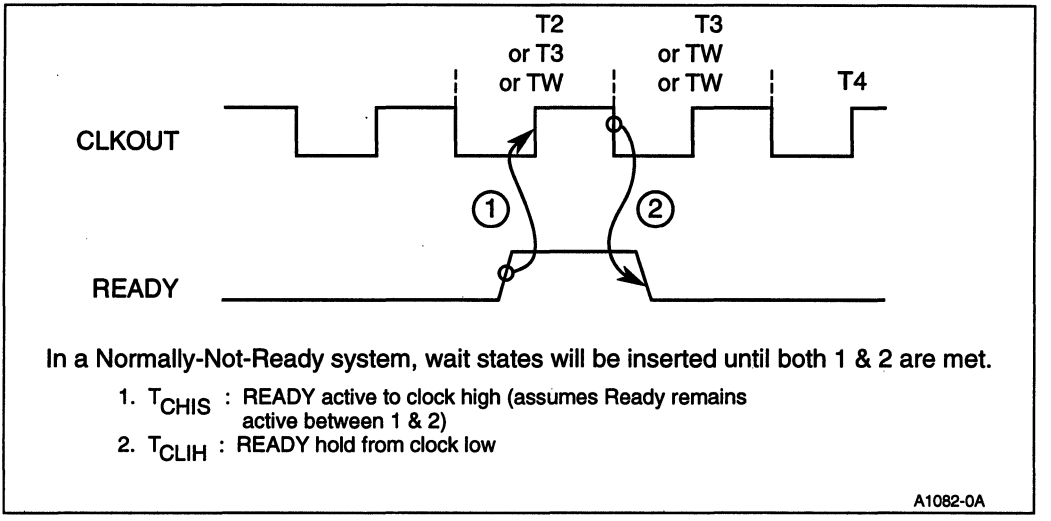


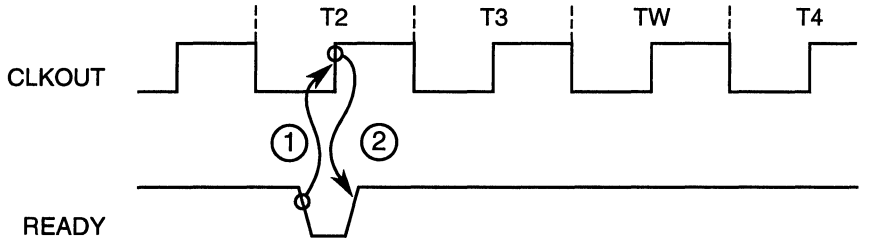
Figure 3-17. Normally Not-Ready System Timing

A valid not-ready input can be generated as late as phase 1 of T3 to insert wait states in a normally ready system. A normally not-ready system must run wait states if the not-ready condition cannot be met in time. Figure 3-18 illustrates the minimum and maximum timing necessary to insert wait states in a normally ready system. Figure 3-18 also shows how to terminate a bus cycle with wait states in a normally ready system.

The BIU can execute an indefinite number of wait states. However, bus cycles with large numbers of wait states limit the performance of the CPU and the integrated peripherals. CPU performance suffers because the instruction prefetch queue cannot be kept full. Integrated peripheral performance suffers because the maximum bus bandwidth decreases.

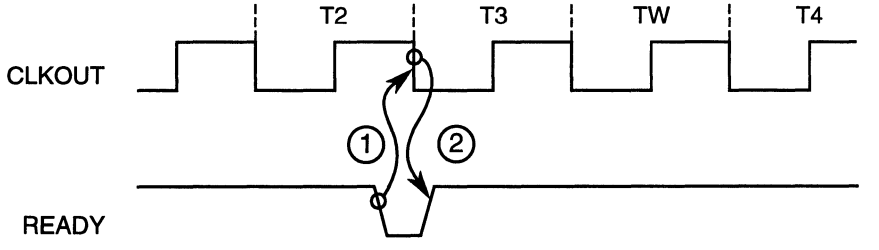
3.4.4 Idle States

Under most operating conditions, the BIU executes consecutive (back-to-back) bus cycles. However, several conditions cause the BIU to become idle. An idle condition occurs between bus cycles (see Figure 3-8 on page 3-9) and may last an indefinite period of time, depending on the instruction sequence.



In a Normally-Ready system, a wait state will be inserted when 1 & 2 are met.

1. T_{CHIS} : READY low to clock high
2. T_{CHIH} : READY hold from clock high



Alternatively, in a Normally-Ready system, a wait state will be inserted when1 & 2 are met.

1. T_{CLIS} : READY low to clock low
2. T_{CLIH} : READY hold from clock low

Failure to meet READY setup and hold can cause a device failure (i.e., the bus hangs or operates inappropriately).

A1083-0A

Figure 3-18. Normally Ready System Timings

Conditions causing the BIU to become idle include the following.

- The instruction prefetch queue is full.
- An effective address calculation is in progress.
- The bus cycle inherently requires idle states (e.g., interrupt acknowledge, locked operations).
- Instruction execution forces idle states (e.g., HLT, WAIT).

An idle bus state may or may not drive the bus. An idle bus state following a bus read cycle continues to float the bus. An idle bus state following a bus write cycle continues to drive the bus. The BIU drives no control strobes active in an idle state except to indicate the start of another bus cycle.

3.5 BUS CYCLES

There are four basic types of bus cycles: read, write, interrupt acknowledge and halt. Interrupt acknowledge and halt bus cycles define special bus operations and require separate discussions. Read bus cycles include memory, I/O and instruction prefetch bus operations. Write bus cycles include memory and I/O bus operations. All read and write bus cycles have the same basic format.

The following sections present timing equations containing symbols found in the data sheet. The timing equations provide information necessary to start a worst-case design analysis.

3.5.1 Read Bus Cycles

Figure 3-19 illustrates a typical read cycle. Table 3-2 lists the three types of read bus cycles.

Table 3-2. Read Bus Cycle Types

Status Bit			Bus Cycle Type
S2	S1	S0	
0	0	1	Read I/O — Initiated by the Execution Unit or the Refresh Control Unit. A19:16 are driven to zero.
1	0	0	Instruction Prefetch — Initiated by the BIU. Data read from the bus fills the prefetch queue.
1	0	1	Read Memory — Initiated by the Execution Unit or the Refresh Control Unit. A19:0 select the desired byte or word memory location.

Figure 3-20 illustrates a typical 16-bit interface connection to a read-only device interface. The same example applies to an 8-bit bus system, except that no devices connect to an upper bus. Four parameters (Table 3-3) must be evaluated when determining the compatibility of a memory (or I/O) device. T_{ADLTCH} defines the delay through the address latch.

Table 3-3. Read Cycle Critical Timing Parameters

Memory Device Parameter	Description	Equation
T_{OE}	Output enable ($\overline{RD}$ low) to data valid	$2T - T_{CLOV2} - T_{CLIS}$
T_{ACC}	Address valid to data valid	$3T - T_{CLOV2} - T_{ADLTCH} - T_{CLIS}$
T_{CE}	Chip enable ($\overline{UCS}$) to data valid	$3T - T_{CLOV2} - T_{CLIS}$
T_{DF}	Output disable ($\overline{RD}$ high) to output float	T_{RHAX}

T_{OE}, T_{ACC} and T_{CE} define the maximum data access requirements for the memory device. These device parameters must be **less** than the value calculated in the equation column. An equal to or greater than result indicates that wait states must be inserted into the bus cycle.

T_{DF} determines the maximum time the memory device can float its outputs before the next bus cycle begins. A T_{DF} value greater than the equation result indicates a buffer fight. A buffer fight means two (or more) devices are driving the bus **at the same time**. This can lead to short circuit conditions, resulting in large current spikes and possible device damage.

T_{RHAX} cannot be lengthened (other than by slowing the clock rate). To resolve a buffer fight condition, choose a faster device or buffer the AD bus (see "Buffering the Data Bus" on page 3-35).

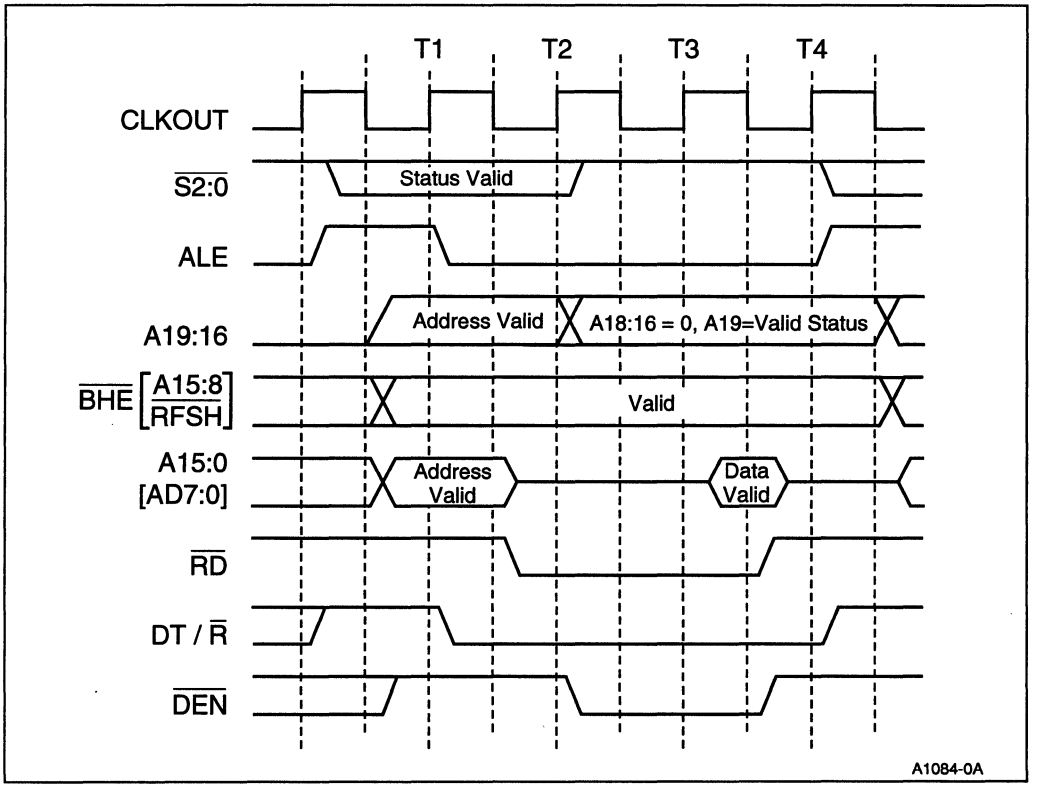


Figure 3-19. Typical Read Bus Cycle

3.5.1.1 Refresh Bus Cycles

A refresh bus cycle operates similarly to a normal read bus cycle except for the following:

- For a 16-bit data bus, address bit A0 and $\overline{BHE}$ drive to a 1 (high) and the data value on the bus is ignored.
- For an 8-bit data bus, address bit A0 drives to a 1 (high) and $\overline{RFSH}$ is driven active (low). The data value on the bus is ignored. $\overline{RFSH}$ has the same bus timing as $\overline{BHE}$.

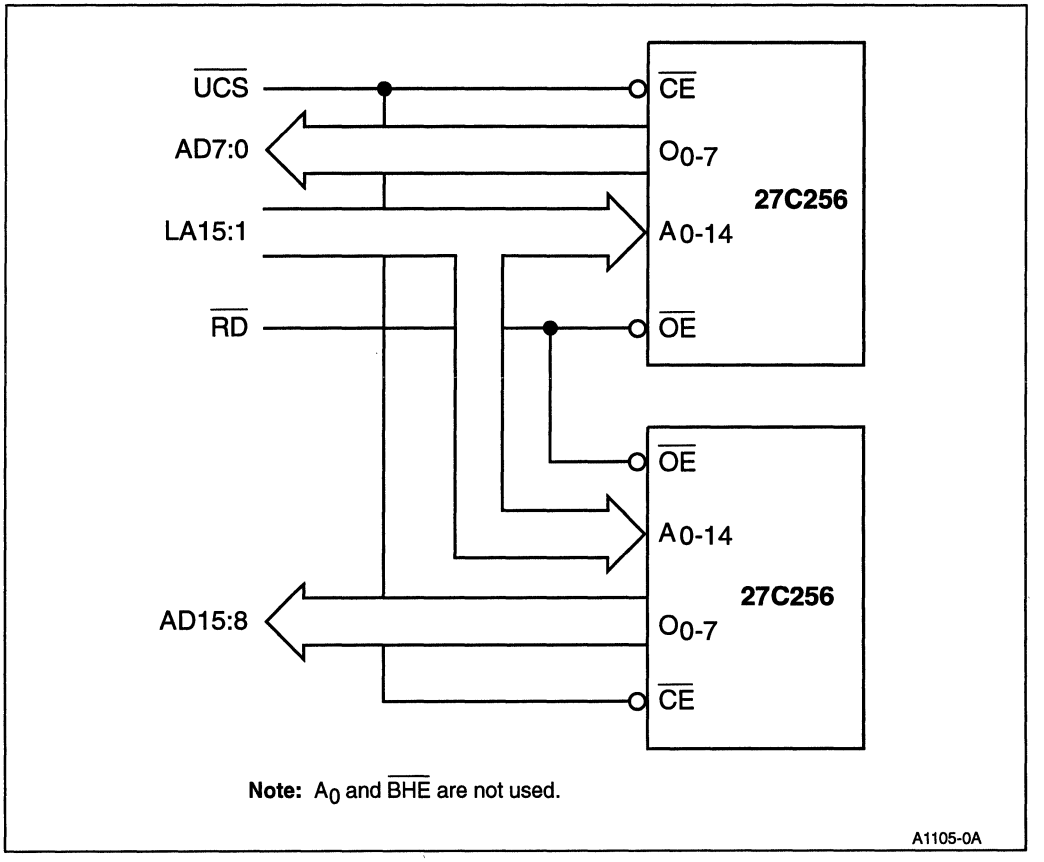


Figure 3-20. Read-Only Device Interface

3.5.2 Write Bus Cycles

Figure 3-21 illustrates a typical write bus cycle. The bus cycle starts with the transition of ALE high and the generation of valid status bits $\overline{S2:0}$. The bus cycle ends when $\overline{WR}$ transitions high (inactive), although data remains valid for one additional clock. Table 3-4 lists the two types of write bus cycles.

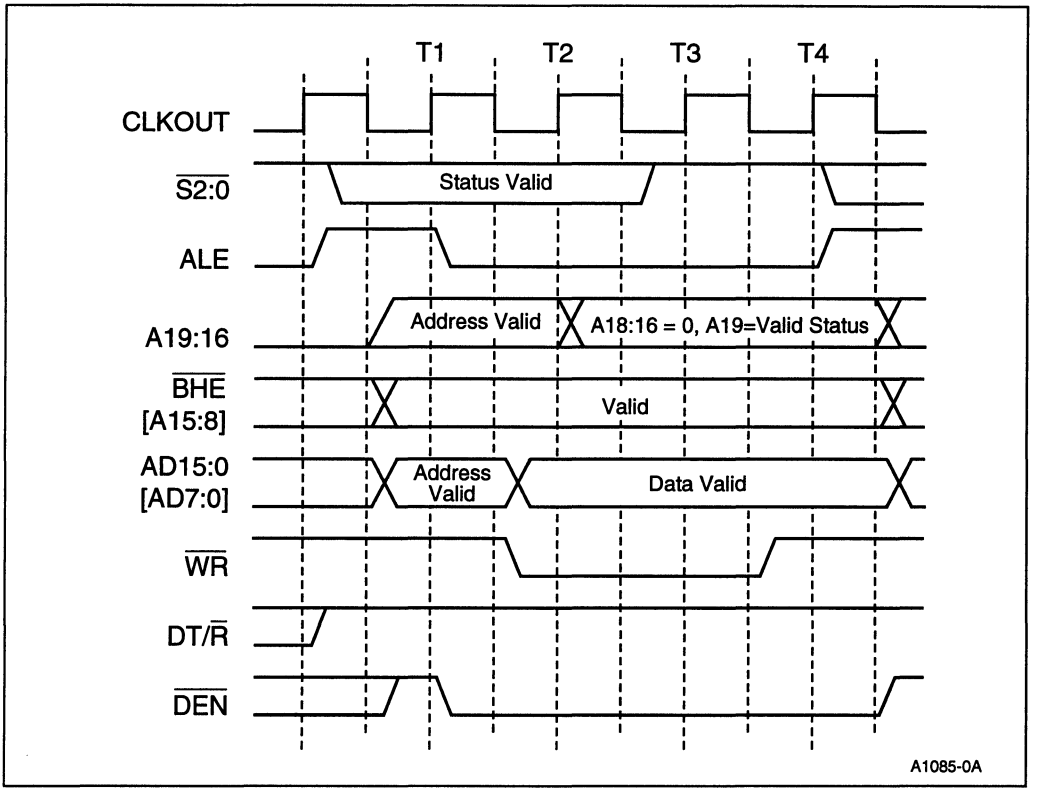


Figure 3-21. Typical Write Bus Cycle

Table 3-4. Write Bus Cycle Types

Status Bits			Bus Cycle Type
S2	S1	S0	
0	1	0	Write I/O — Initiated by executing IN, OUT, INS, OUTS instructions. A15:0 select the desired I/O port. A19:16 are driven to zero.
1	1	0	Write Memory —Initiated by any of the Byte/ Word memory instructions. A19:0 selects the desired byte or word memory location.

Figure 3-22 illustrates a typical 16-bit interface connection to a read/write device. Write bus cycles have many parameters that must be evaluated in determining the compatibility of a memory (or I/O) device. Table 3-5 lists some critical write bus cycle parameters.

Most memory and peripheral devices latch data on the rising edge of the write strobe. Address, chip-select and data must be valid (set up) prior to the rising edge of $\overline{WR}$. T_{AW}, T_{CW} and T_{DW} define the minimum data setup requirements. The value calculated by their respective equations must be greater than the device requirements. To increase the calculated value, insert wait states.

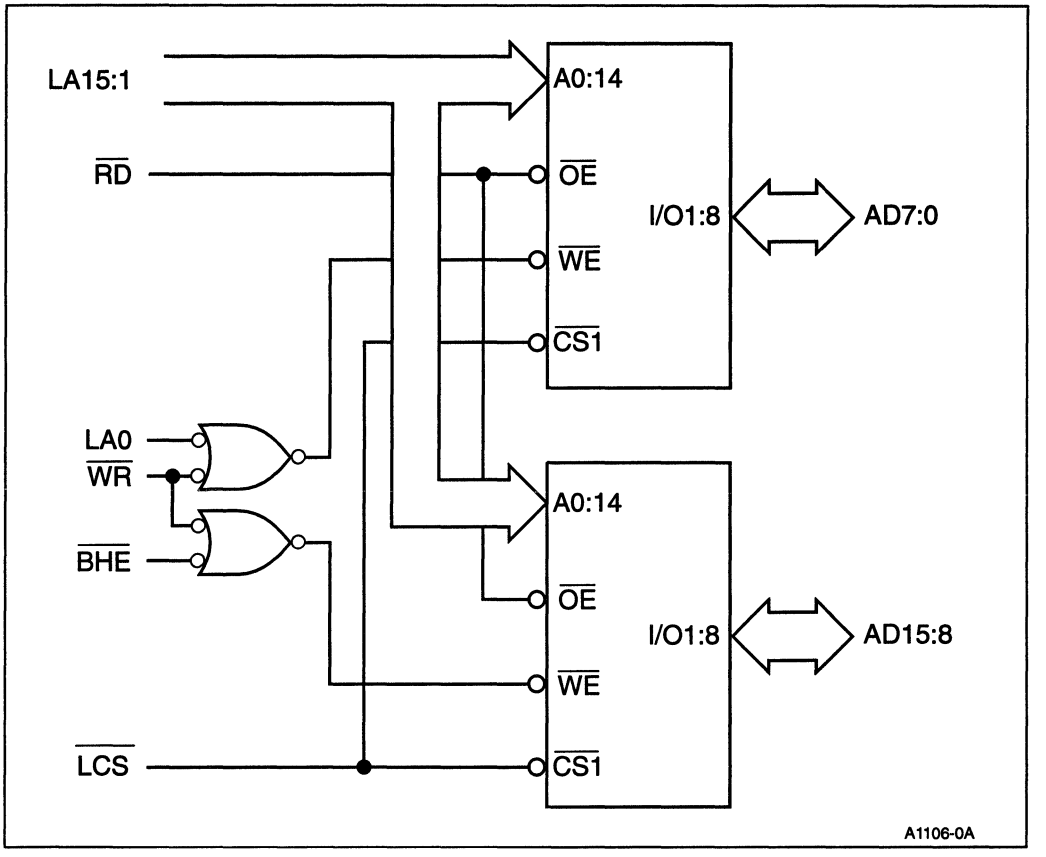


Figure 3-22. 16-Bit Bus Read/Write Device Interface

The minimum device data hold time (from $\overline{WR}$ high) is defined by T_{DH}. The calculated value must be greater than the minimum device requirements; however, the value can be changed only by decreasing the clock rate.

Table 3-5. Write Cycle Critical Timing Parameters

Memory Device Parameter	Description	Equation
T_{WC}	Write cycle time	4T
T_{AW}	Address valid to end of write strobe ($\overline{WR}$ high)	$3T - T_{ADLTCH}$
T_{CW}	Chip enable ($\overline{LCS}$) to end of write strobe ($\overline{WR}$ high)	3T
T_{WR}	Write recover time	T_{WHLH}
T_{DW}	Data valid to write strobe ($\overline{WR}$ high)	2T
T_{DH}	Data hold from write strobe ($\overline{WR}$ high)	T_{WHDX}
T_{WP}	Write pulse width	T_{WLWH}

T_{WC} and T_{WP} define the minimum time (maximum frequency) a device can process write bus cycles. T_{WR} determines the minimum time from the end of the current write cycle to the start of the next write cycle. All three parameters require that calculated values be greater than device requirements. The calculated T_{WC} and T_{WP} values increase with the insertion of wait states. The calculated T_{WR} value, however, can be changed only by decreasing the clock rate.

3.5.3 Interrupt Acknowledge Bus Cycle

Interrupt expansion is accomplished by interfacing the Interrupt Control Unit with a peripheral device such as the 82C59A Programmable Interrupt Controller. (See Chapter 8, "Interrupt Control Unit," for more information.) The BIU controls the bus cycles required to fetch vector information from the peripheral device, then passes the information to the CPU. These bus cycles, collectively known as Interrupt Acknowledge bus cycles, operate similarly to read bus cycles. However, instead of generating $\overline{RD}$ to enable the peripheral, the $\overline{INTA}$ signal is used. Figure 3-23 illustrates a typical Interrupt Acknowledge (or $\overline{INTA}$) bus cycle.

An Interrupt Acknowledge bus cycle consists of two consecutive bus cycles. $\overline{LOCK}$ is generated to indicate the sequential bus operation. The second bus cycle strobes vector information only from the lower half of the bus (D7:0).In a 16-bit bus system, D15:13 contain cascade address information and D12:8 float. .

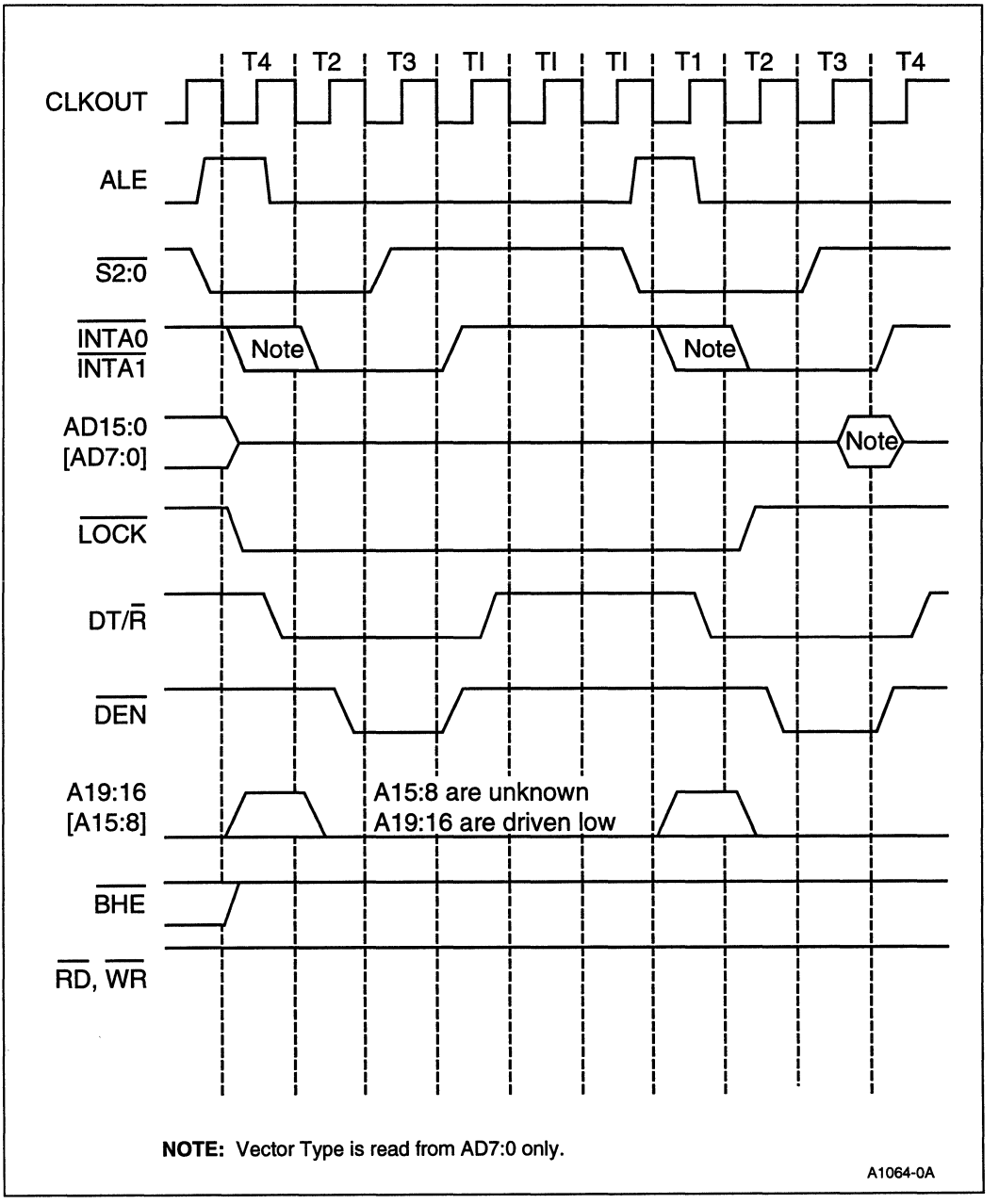


Figure 3-23. Interrupt Acknowledge Bus Cycle

Figure 3-24 shows a typical 82C59A interface example. Bus ready must be provided to terminate both bus cycles in the interrupt acknowledge sequence.

NOTE

Due to an internal condition, external ready is ignored if the device is configured in Cascade mode and the Peripheral Control Block (PCB) is located at 0000H in I/O space. In this case, wait states **cannot** be added to interrupt acknowledge bus cycles. However, you **can** add wait states to interrupt acknowledge cycles if the PCB is located at any other address.

3.5.3.1 System Design Considerations

Although ALE is generated for both bus cycles, the BIU does not drive valid address information. Actually, all address bits except A19:16 float during the time ALE becomes active (on both 8- and 16-bit bus devices). Address-decoding circuitry must be disabled for Interrupt Acknowledge bus cycles to prevent erroneous operation.

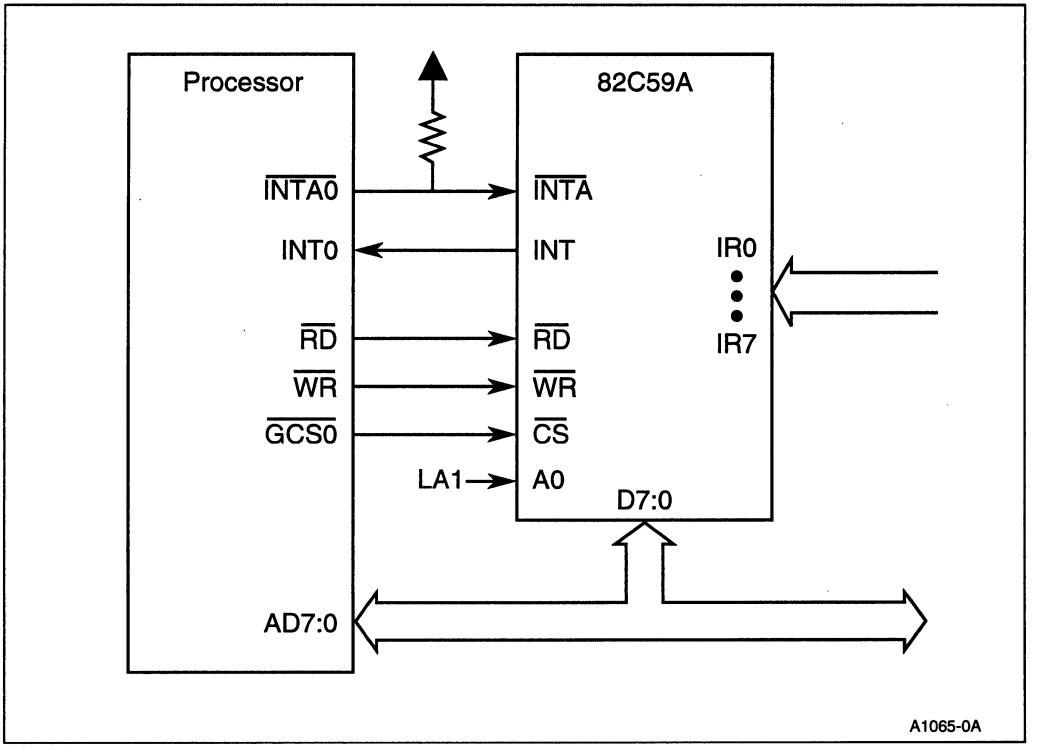


Figure 3-24. Typical 82C59A Interface

3.5.4 HALT Bus Cycle

Suspending the CPU reduces device power consumption and potentially reduces interrupt latency time. The HLT instruction initiates two events:

1. Suspends the Execution Unit.
2. Instructs the BIU to execute a HALT bus cycle.

The Idle or Powerdown power management mode (or the absence of both of them, known as Active Mode) affects the operation of the bus HALT cycle. The effects relating to BIU operation and the HALT bus cycle are described in this chapter. Chapter 5, "Clock Generation and Power Management," discusses the concepts of Active, Idle and Powerdown power management modes.

After executing a HALT bus cycle, the BIU suspends operation until one of the following events occurs:

- An interrupt is generated.
- A bus HOLD is generated (except when Powerdown mode is enabled).
- A refresh request is generated (except when Powerdown mode is enabled).

Figure 3-25 shows the operation of a HALT bus cycle. The address/data bus either floats or drives during T1, depending on the next bus cycle to be executed by the BIU. Under most instruction sequences, the BIU floats the address/data bus because the next operation would most likely be an instruction prefetch. However, if the HALT occurs just after a bus write operation, the address/data bus drives either data or address information during T1. A19:16 continue to drive the previous bus cycle information under most instruction sequences (otherwise, they drive the next prefetch address). The BIU always operates in the same way for any given instruction sequence.

The Chip-Select Unit prevents a programmed chip-select from going active during a HALT bus cycle. However, chip-selects generated by external decoder circuits must be disabled for HALT bus cycles.

After several TI bus states, all address/data, address/status and bus control pins drive to a known state when Powerdown or Idle Mode is enabled. The address/data and address/status bus pins force a low (0) state. Bus control pins force their inactive state. Figure 3-3 lists the state of each pin after entering the HALT bus state.

Table 3-6. HALT Bus Cycle Pin States

Pin(s)	Pin State	
	No Powerdown or Idle Mode	Powerdown or Idle Mode
AD15:0 (AD7:0 for 8-bit)	Float	Drive Zero
A15:8 (8-bit)	Drive Address	Drive Zero
A19:16	Drive 8H or Zero	Drive Zero
$\overline{BHE}$ (16-bit)	Drive Last Value	Drive One
$\overline{RD}$, $\overline{WR}$, $\overline{DEN}$, DT/$\overline{R}$, $\overline{RFSH}$ (8-bit), $\overline{S2:0}$	Drive One	Drive One

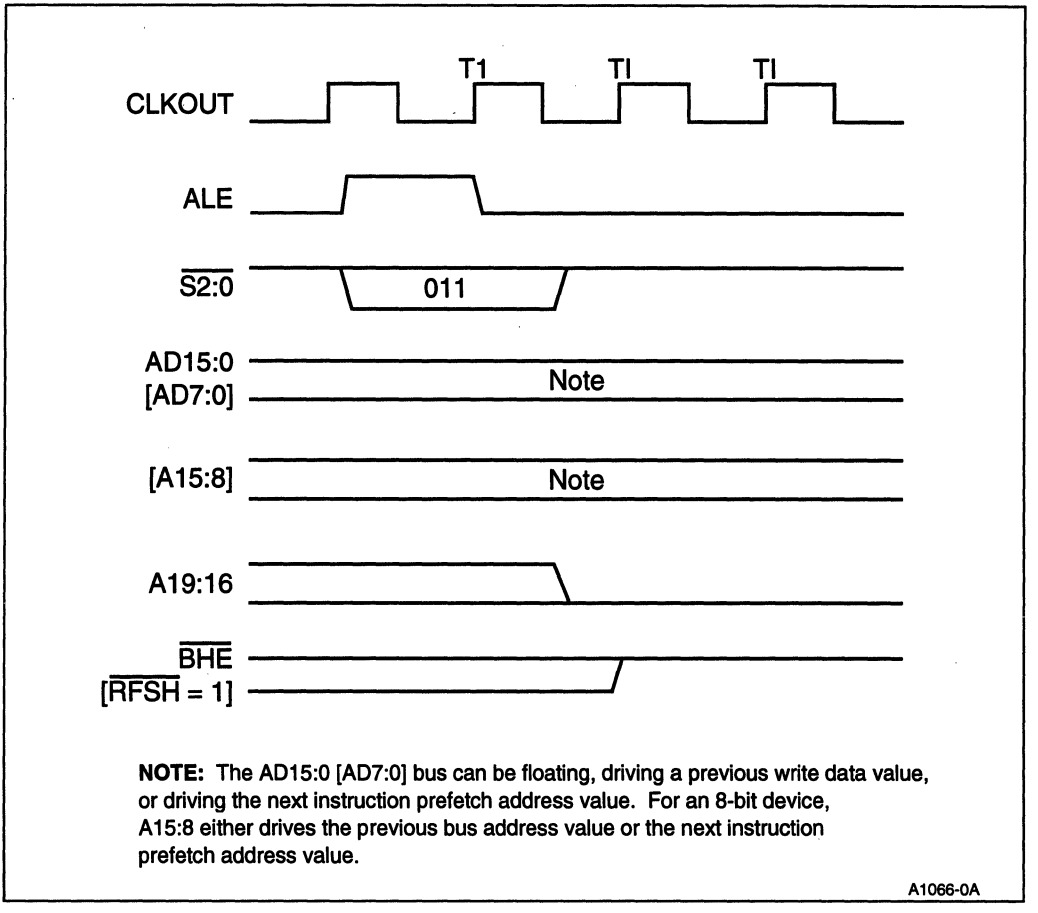


Figure 3-25. HALT Bus Cycle

3.5.5 Temporarily Exiting the HALT Bus State

A refresh request or bus hold request causes the BIU to exit the HALT bus state temporarily. This can occur only when in the Active or Idle power management mode. The BIU returns to the HALT bus **state** after it completes the desired bus operation. However, the BIU **does not** execute another bus HALT **cycle** (i.e., ALE and bus cycle status are not regenerated). Figures 3-26 and 3-27 illustrate how the BIU temporarily exits and then returns to the HALT bus state.

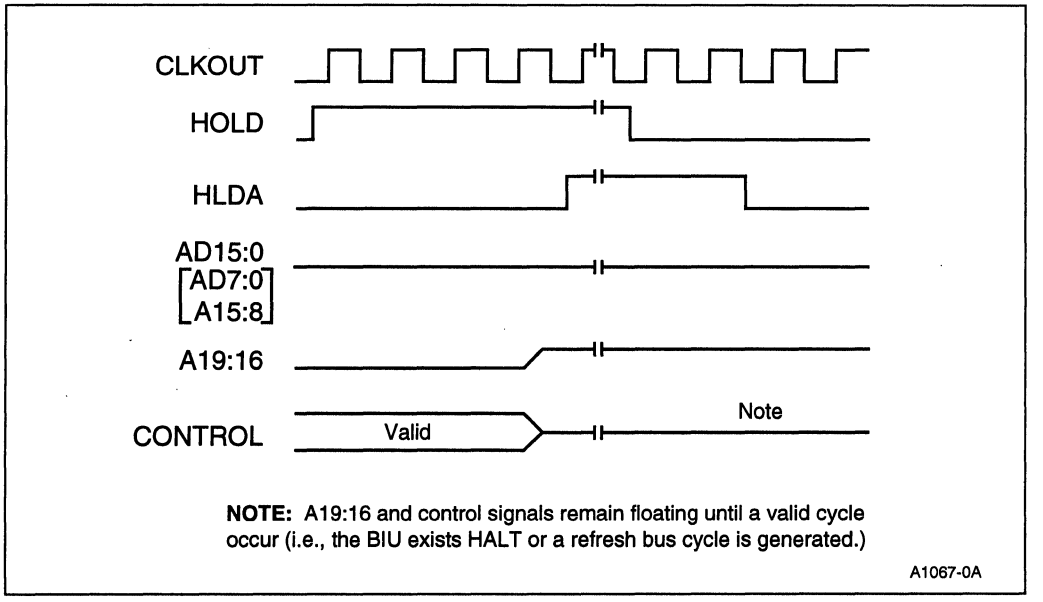
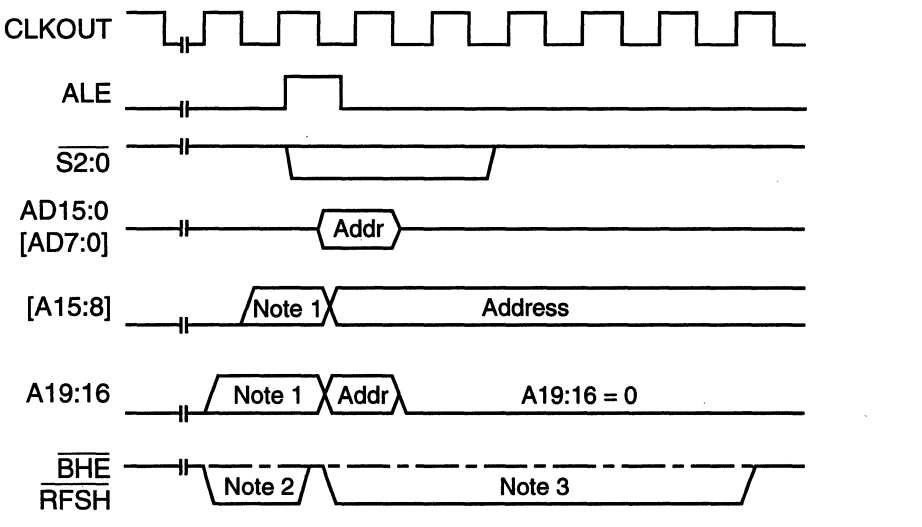


Figure 3-26. Returning to HALT After a HOLD/HLDA Bus Exchange



NOTE:

1. Previous bus cycle value.
2. Only occurs for $\overline{BHE}$ on the first refresh bus cycle after entering HALT.
3. $\overline{BHE}$ = 1 for 16-bit device, $\overline{RFSH}$ = 0 for 8-bit device.

A1068-0A

Figure 3-27. Returning to HALT After a Refresh Bus Cycle

3.5.6 Exiting HALT

Any NMI or maskable interrupt forces the BIU to exit the HALT bus state (in any power management mode). The first bus operations to occur after exiting HALT are read cycles to reload the CS:IP registers. Figure 3-28 and Figure 3-29 show how the HALT bus state is exited when an NMI or INT*n* occurs.

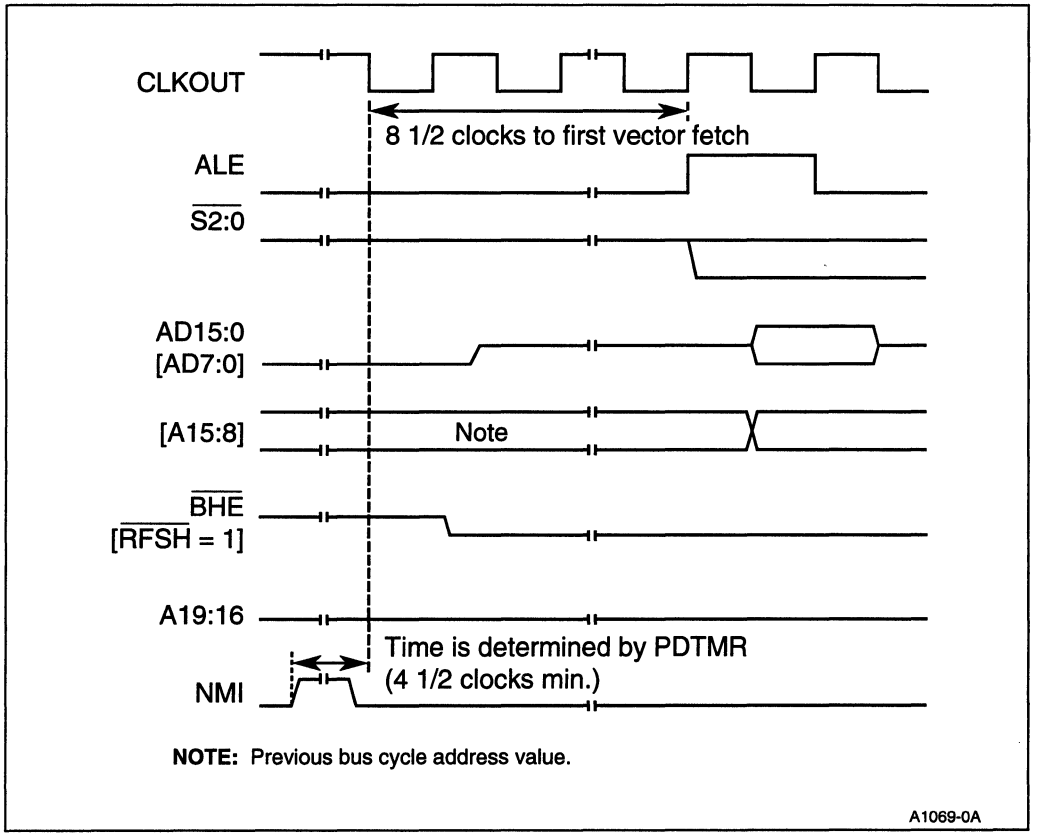
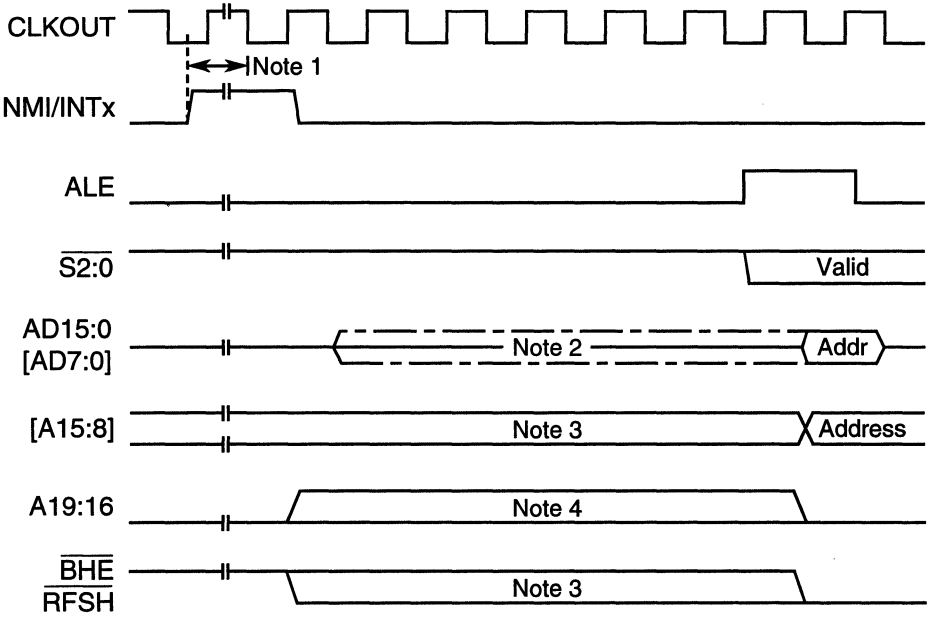


Figure 3-28. Exiting HALT (Powerdown Mode)



NOTE:
1. For NMI, delay = 4 1/2 clocks. For INTx, delay = 7 1/2 clocks (min).
2. If previous bus cycle was a read, bus will float. If previous bus cycle was a write, bus will drive data value.
3. Previous bus cycle value.
4. If previous bus cycle was a refresh cycle, value will be 8H (A19 = 1); otherwise, value will be 0.

A1070-0A

Figure 3-29. Exiting HALT (Active/Idle Mode)

3.6 SYSTEM DESIGN ALTERNATIVES

Most system designs require no signals other than those already provided by the BIU. However, heavily loaded bus conditions, slow memory or peripheral device performance and off-board device interfaces may not be supported directly without modifying the BIU interface. The following sections deal with topics to enhance or modify the operation of the BIU.

3.6.1 Buffering the Data Bus

The BIU generates two control signals, $\overline{DEN}$ and DT/$\overline{R}$, to control bidirectional buffers or transceivers. The timing relationship of $\overline{DEN}$ and DT/$\overline{R}$ is shown in Figure 3-30. The following conditions require transceivers:

- The capacitive load on the address/data bus gets too large.
- The current load on the address/data bus exceeds device specifications.
- Additional V_{OL} and V_{OH} drive is required.
- A memory or I/O device cannot float its outputs in time to prevent bus contention, even at reset.

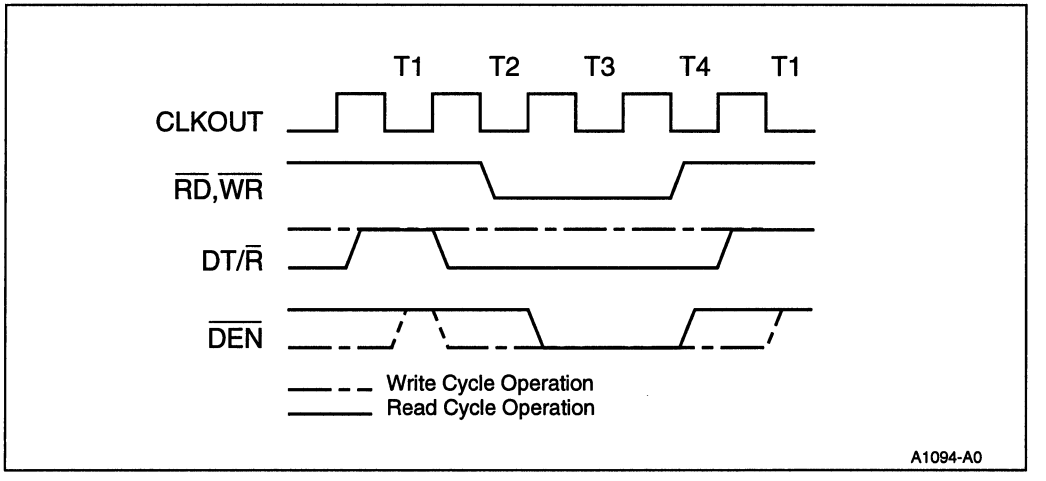


Figure 3-30. $\overline{DEN}$ and DT/$\overline{R}$ Timing Relationships

The circuit shown in Figure 3-31 illustrates how to use transceivers to buffer the address/data bus. The connection between the processor and the transceiver is known as the *local bus*. A connection between the transceiver and other memory or I/O devices is known as the *buffered bus*. A *fully buffered* system has **no** devices attached to the local bus. A *partially buffered* system has devices on both the local and buffered buses.

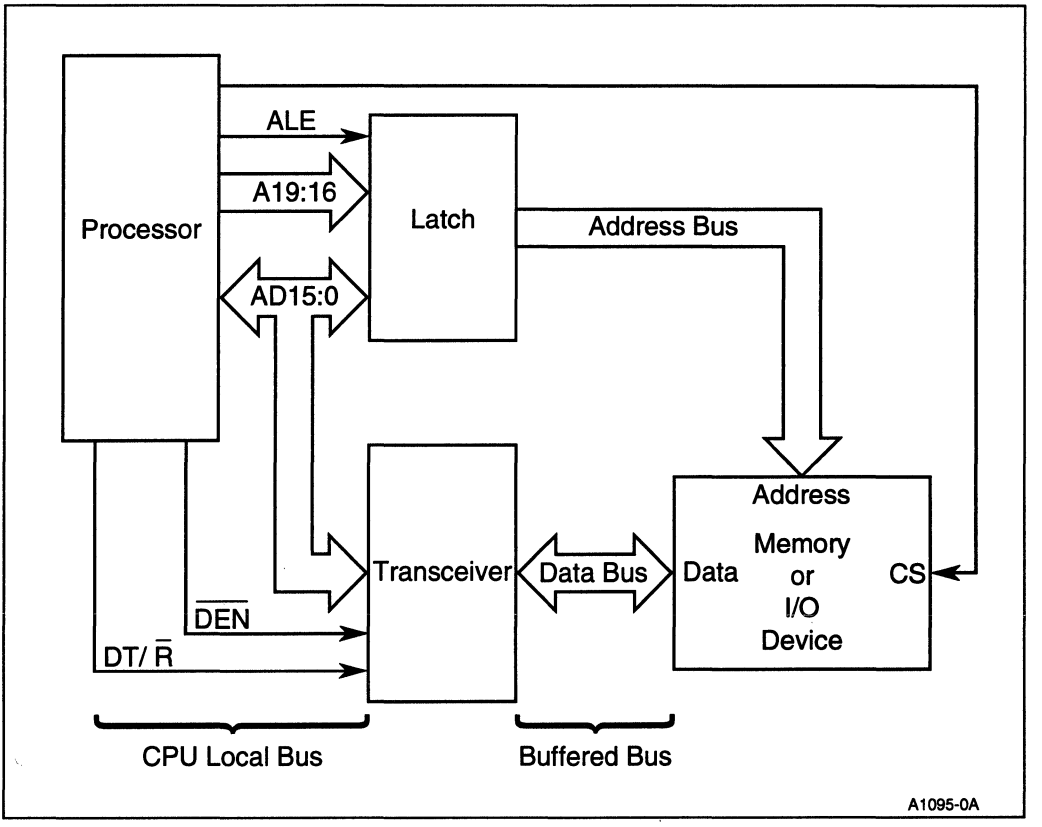


Figure 3-31. Buffered AD Bus System

In a fully buffered system, $\overline{DEN}$ directly drives the transceiver output enable. A partially buffered system requires that $\overline{DEN}$ be qualified with another signal to prevent the transceiver from going active for local bus accesses. Figure 3-32 illustrates how to use chip-selects to qualify $\overline{DEN}$.

DT/$\overline{R}$ always connects directly to the transceiver. However, an inverter may be required if the polarity of DT/$\overline{R}$ does not match the transceiver. DT/$\overline{R}$ goes low (0) only for memory and I/O read, instruction prefetch and interrupt acknowledge bus cycles.

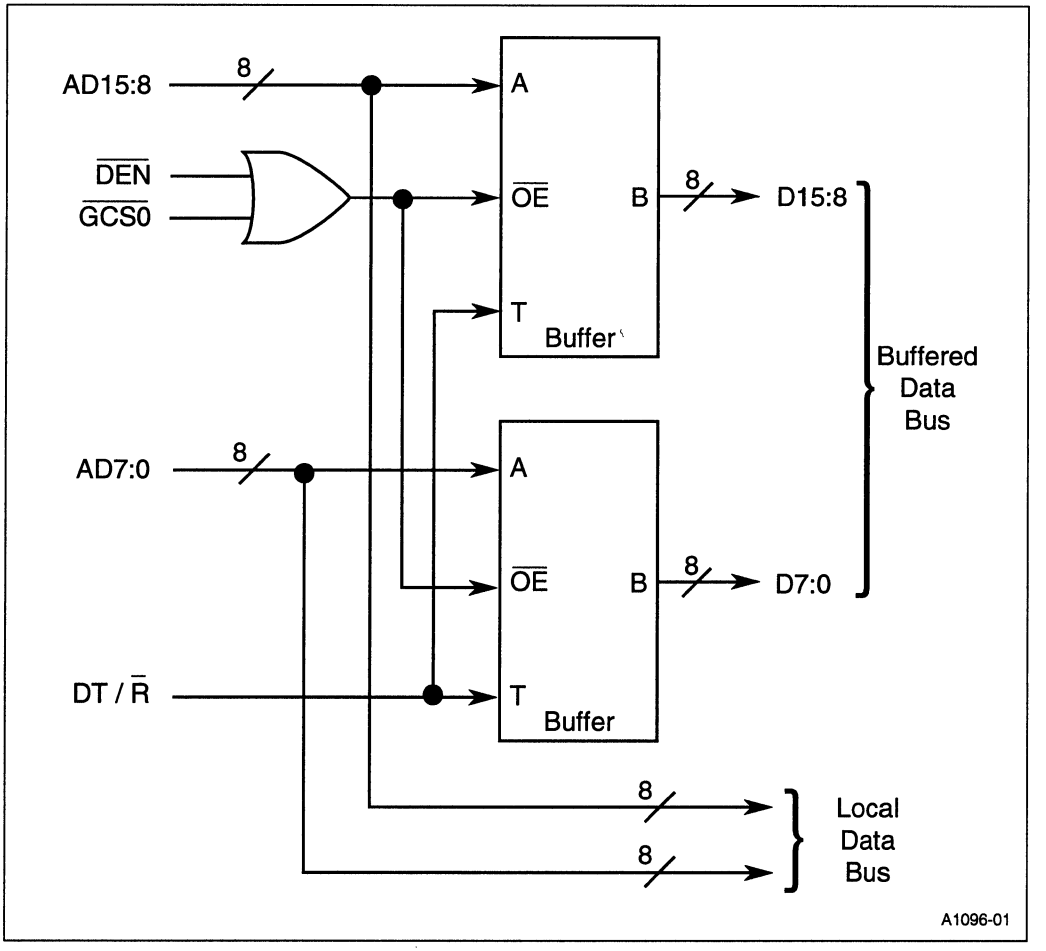


Figure 3-32. Qualifying $\overline{DEN}$ with Chip-Selects

3.6.2 Synchronizing Software and Hardware Events

The execution sequence of a program and hardware events occurring within a system are often asynchronous to each other. In some systems there may be a requirement to suspend program execution until an event (or events) occurs, then continue program execution.

One way to synchronize software execution with hardware events requires the use of interrupts. Executing a HALT instruction suspends program execution until an unmasked interrupt occurs. However, there is a delay associated with servicing the interrupt before program execution can proceed. Using the WAIT instruction removes the delay associated with servicing interrupts.

The WAIT instruction suspends program execution until one of two events occurs: an interrupt is generated, or the $\overline{TEST}$ input pin is sampled low. Unlike interrupts, the $\overline{TEST}$ input pin does not require that program execution be transferred to a new location (i.e., an interrupt routine is not executed). In processing the WAIT instruction, program execution remains suspended as long as $\overline{TEST}$ remains high (at least until an interrupt occurs). When $\overline{TEST}$ is sampled low, program execution resumes.

The $\overline{TEST}$ input and WAIT instruction provide a mechanism to delay program execution until a hardware event occurs, without having to absorb the delay associated with servicing an interrupt.

3.6.3 Using a Locked Bus

To address the problems of controlling accesses to shared resources, the BIU provides a hardware $\overline{LOCK}$ output. The execution of a LOCK prefix instruction activates the $\overline{LOCK}$ output.

$\overline{LOCK}$ goes active in phase 1 of T1 of the first bus cycle following execution of the LOCK prefix instruction. It remains active until phase 1 of T1 of the first bus cycle following the execution of the instruction following the LOCK prefix. To provide bus access control in multiprocessor systems, the $\overline{LOCK}$ signal should be incorporated into the system bus arbitration logic residing in the CPU.

During normal multiprocessor system operation, priority of the shared system bus is determined by the arbitration circuits on a cycle by cycle basis. As each CPU requires a transfer over the system bus, it requests access to the bus via its resident bus arbitration logic. When the CPU gains priority (determined by the system bus arbitration scheme and any associated logic), it takes control of the bus, performs its bus cycle and either maintains bus control, voluntarily releases the bus or is forced off the bus by the loss of priority.

The lock mechanism prevents the CPU from losing bus control (either voluntarily or by force) and guarantees that the CPU can execute multiple bus cycles without intervention and possible corruption of the data by another CPU. A classic use of the mechanism is the "TEST and SET semaphore," during which a CPU must read from a shared memory location and return data to the location without allowing another CPU to reference the same location during the test and set operations.

Another application of $\overline{LOCK}$ for multiprocessor systems consists of a locked block move, which allows high speed message transfer from one CPU's message buffer to another. During the locked instruction (i.e., while $\overline{LOCK}$ is active), a bus hold or refresh request is recorded, but is not acknowledged until completion of the locked instruction. However, $\overline{LOCK}$ has no effect on interrupts. As an example, a locked HALT instruction causes bus hold or refresh bus requests to be ignored, but still allows the CPU to exit the HALT state on an interrupt.

In general, prefix bytes (such as LOCK) are considered extensions of the instructions they precede. Interrupts and refresh requests that occur during execution of the prefix are not acknowledged until the instruction following the prefix completes (except for instructions that are servicing interrupts during their execution, such as HALT, WAIT and repeated string primitives). Note that multiple prefix bytes can precede an instruction.

Another example is a string primitive preceded by the repetition prefix (REP), which can be interrupted after each execution of the string primitive, even if the REP prefix is combined with the LOCK prefix. This prevents interrupts from being locked out during a block move or other repeated string operations. However, bus hold and refresh requests remain locked out until $\overline{\text{LOCK}}$ is removed (either when the block operation completes or after an interrupt occurs).

3.7 MULTI-MASTER BUS SYSTEM DESIGNS

The BIU supports protocols for transferring control of the local bus between itself and other devices capable of acting as bus masters. To support such a protocol, the BIU uses a hold request input (HOLD) and a hold acknowledge output (HLDA) as bus transfer handshake signals. To gain control of the bus, a device asserts the HOLD input, then waits until the HLDA output goes active before driving the bus. After HLDA goes active, the requesting device can take control of the local bus and remains in control of the bus until HOLD is removed.

3.7.1 Entering Bus HOLD

In responding to the hold request input, the BIU floats the entire address and data bus, and many of the control signals. Figure 3-33 illustrates the timing sequence when acknowledging the hold request. Table 3-7 lists the states of the BIU pins when HLDA is asserted. All device pins not mentioned in Table 3-7 or shown in Figure 3-33 remain either active (e.g., CLKOUT and T1OUT) or inactive (e.g., $\overline{\text{UCS}}$ and $\overline{\text{INTA}}$). Refer to the data sheet for specific details of pin functions during a bus hold.

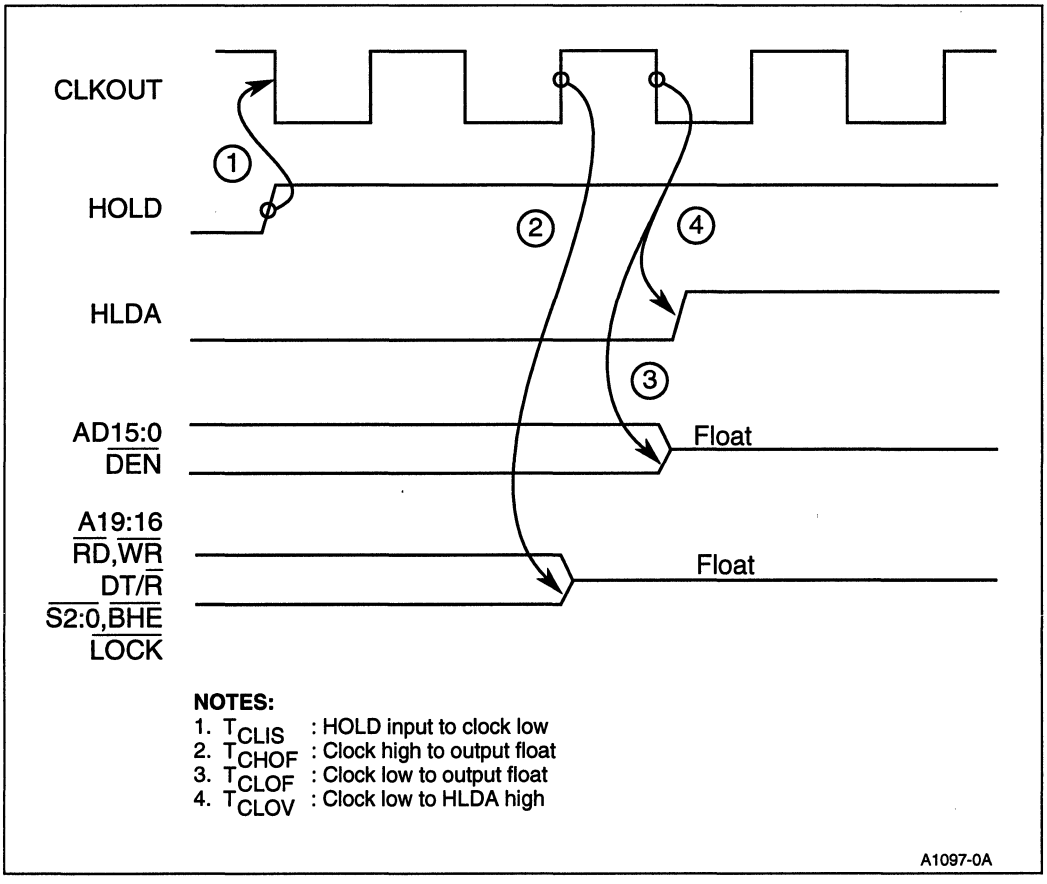


Figure 3-33. Timing Sequence Entering HOLD

Table 3-7. Signal Condition Entering HOLD

Signal	HOLD Condition
A19:16, $\overline{S2:0}$, $\overline{RD}$, $\overline{WR}$, DT/$\overline{R}$, $\overline{BHE}$ ($\overline{RFSH}$), $\overline{LOCK}$	These signals float one-half clock before HLDA is generated (i.e., phase 2).
AD15:0 (16-bit), AD7:0 (8-bit), A15:8 (8-bit), $\overline{DEN}$	These signals float during the same clock in which HLDA is generated (i.e., phase 1).

3.7.1.1 HOLD Bus Latency

The duration between the time that the external device asserts HOLD and the time that the BIU asserts HLDA is known as *bus latency*. In Figure 3-33, the two-clock delay between HOLD and HLDA represents the shortest bus latency. Normally this occurs only if the bus is idle or halted or if the bus hold request occurs just before the BIU begins another bus cycle.

The major factors that influence bus latency are listed below (in order from longest delay to shortest delay).

1. Bus Not Ready — As long as the bus remains not ready, a bus hold request cannot be serviced.
2. Locked Bus Cycle — As long as $\overline{\text{LOCK}}$ remains asserted, a bus hold request cannot be serviced. Performing a locked move string operation can take several thousands of clocks.
3. Completion of Current Bus Cycle — A bus hold request cannot be serviced until the current bus cycle completes. A bus hold request will not separate bus cycles required to move odd-aligned word data. Also, bus cycles with long wait states will delay the servicing of a bus hold request.
4. Interrupt Acknowledge Bus Cycle — A bus hold request is not serviced until after an $\overline{\text{INTA}}$ bus cycle has completed. An $\overline{\text{INTA}}$ bus cycle drives $\overline{\text{LOCK}}$ active.
5. Refresh Bus Cycles — A bus hold request is not serviced until after the refresh bus cycle has completed. Refresh bus cycles have a higher priority than hold bus requests.

3.7.1.2 Refresh Operation During a Bus HOLD

Under normal operating conditions, once HLDA has been asserted it remains asserted until HOLD is removed. However, when a refresh bus request is generated, the HLDA output is removed (driven low) to signal the need for the BIU to regain control of the local bus. The BIU does not gain control of the bus until HOLD is removed. This procedure prevents the BIU from just arbitrarily regaining control of the bus.

Figure 3-34 shows the timing associated with the occurrence of a refresh request while HLDA is active. Note that HLDA can be as short as one clock in duration. This happens when a refresh request occurs just after HLDA is granted. A refresh request has higher priority than a bus hold request; therefore, when the two occur simultaneously, the refresh request occurs before HLDA becomes active.

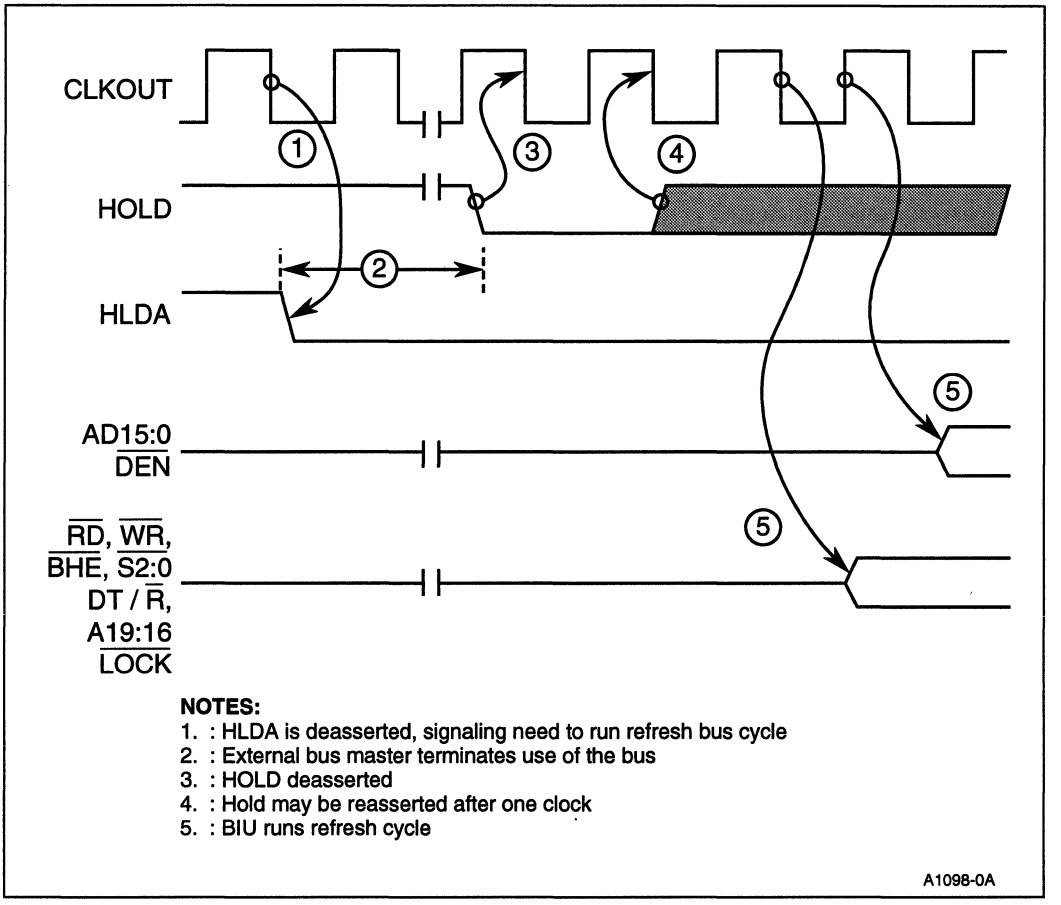


Figure 3-34. Refresh Request During HOLD

The device requesting a bus hold must be able to detect a HLDA pulse that is one clock in duration. A bus lockup (hang) condition can result if the requesting device fails to detect the short HLDA pulse and continues to wait for HLDA to be asserted while the BIU waits for HOLD to be deasserted. The circuit shown in Figure 3-35 can be used to latch HLDA.

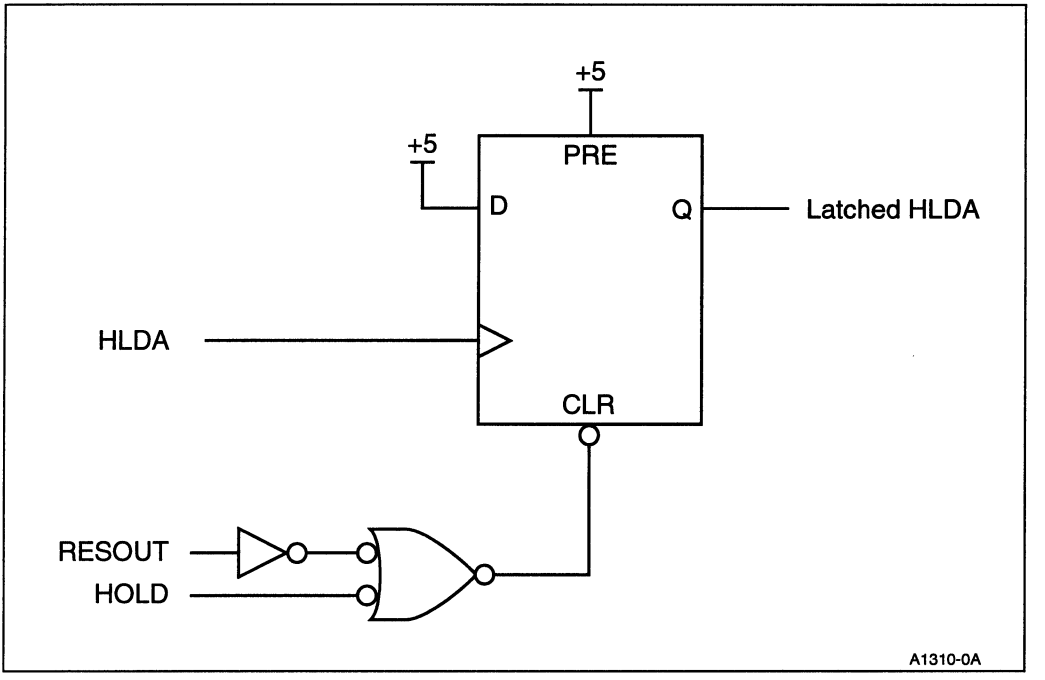


Figure 3-35. Latching HLDA

The removal of HOLD must be detected for at least one clock cycle to allow the BIU to regain the bus and execute a refresh bus cycle. Should HOLD go active before the refresh bus cycle is complete, the BIU will release the bus and generate HLDA.

3.7.2 Exiting HOLD

Figure 3-36 shows the timing associated with exiting the bus hold state. Normally a bus operation (e.g., an instruction prefetch) occurs just after HOLD is released. However, if no bus cycle is pending when leaving a bus hold state, the bus and associated control signals remain floating, if the system is in normal operating mode. (For signal states associated with Idle and Powerdown modes, see "Temporarily Exiting the HALT Bus State" on page 3-30).

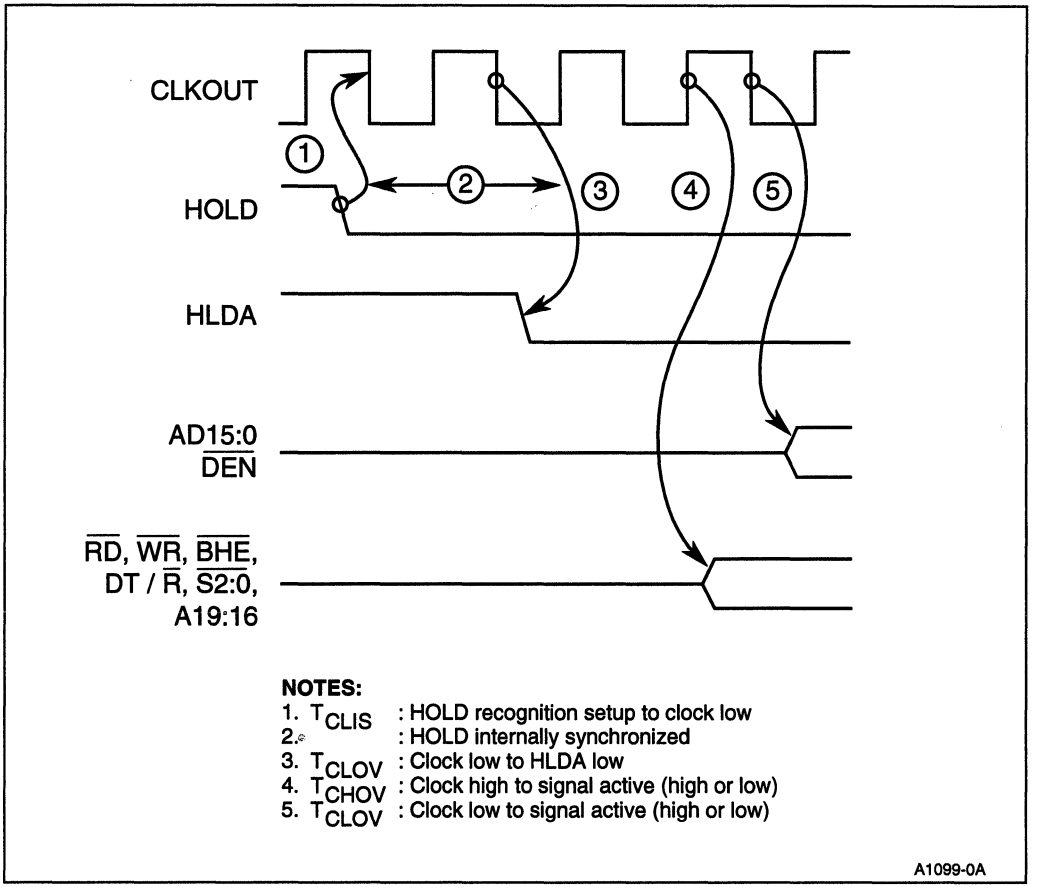


Figure 3-36. Exiting HOLD

3.8 BUS CYCLE PRIORITIES

The BIU arbitrates requests for bus cycles from the Execution Unit, the integrated peripherals (e.g., Interrupt Control Unit) and external bus masters (i.e., bus hold requests). The list below summarizes the priorities for all bus cycle requests (from highest to lowest).

1. Instruction execution read/write following a non-pipelined effective address calculation.
2. Refresh bus cycles.
3. Bus hold request.
4. Single step interrupt vectoring sequence.
5. Non-Maskable interrupt vectoring sequence.

6. Internal error (e.g., divide error, overflow) interrupt vectoring sequence.
7. Hardware (e.g., INT0) interrupt vectoring sequence.
8. 80C187 Math Coprocessor error interrupt vectoring sequence.
9. General instruction execution. This category includes read/write operations following a pipelined effective address calculation, vectoring sequences for software interrupts and numerics code execution. The following points apply to sequences of related execution cycles.
   - — The second read/write cycle of an odd-addressed word operation is inseparable from the first bus cycle.
   - — The second read/write cycle of an instruction with both load and store accesses (e.g., XCHG) can be separated from the first cycle by other bus cycles.
   - — Successive bus cycles of string instructions (e.g., MOVS) can be separated by other bus cycles.
   - — When a locked instruction begins, its associated bus cycles become the highest priority and cannot be separated (or preempted) until completed.
10. Bus cycles necessary to fill the prefetch queue.

intel®

4

Peripheral Control Block

CHAPTER 4
PERIPHERAL CONTROL BLOCK

All integrated peripherals in the 80C186 Modular Core family are controlled by sets of registers within an integrated Peripheral Control Block (PCB). The peripheral control registers are physically located in the peripheral devices they control, but they are addressed as a single block of registers. The Peripheral Control Block encompasses 256 contiguous bytes and can be located on any 256-byte boundary of memory or I/O space. The PCB Relocation Register, which is also located within the Peripheral Control Block, controls the location of the PCB.

4.1 PERIPHERAL CONTROL REGISTERS

Each of the integrated peripherals' control and status registers is located at a fixed offset above the programmed base location of the Peripheral Control Block (see Table 4-1). These registers are described in the chapters that cover the associated peripheral. "Accessing the Peripheral Control Block" on page 4-4 discusses how the registers are accessed and outlines considerations for reading and writing them.

4.2 PCB RELOCATION REGISTER

In addition to control registers for the integrated peripherals, the Peripheral Control Block contains the PCB Relocation Register (Figure 4-1). The Relocation Register is located at a fixed offset within the Peripheral Control Block (Table 4-1). If the Peripheral Control Block is moved, the Relocation Register also moves.

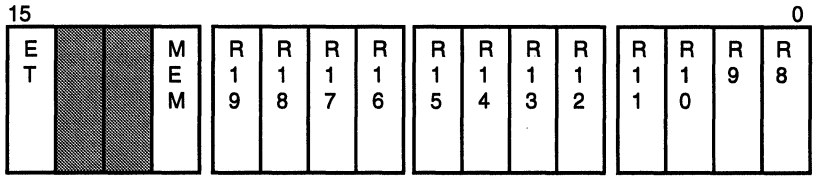
The PCB Relocation Register allows the Peripheral Control Block to be relocated to any 256-byte boundary within memory or I/O space. The Memory I/O bit (MEM) selects either memory space or I/O space, and the R19:8 bits specify the starting (base) address of the PCB. The remaining bit, Escape Trap (ET), controls access to the math coprocessor interface.

"Setting the PCB Base Location" on page 4-6 describes how to set the base location and outlines some restrictions on the Peripheral Control Block location.

Register Name: PCB Relocation Register

Register Mnemonic: RELREG

Register Function: Relocates the PCB within memory or I/O space.



A1263-0A

Bit Mnemonic	Bit Name	Reset State	Function
ET	Escape Trap	0	The ET bit controls access to the math coprocessor. If ET is set, the CPU will trap (resulting in a Type 7 interrupt) when an ESC instruction is executed. **NOTE**: The 8-bit bus version of the device automatically traps an ESC opcode to the Type 7 interrupt, regardless of the state of the ET bit.
MEM	Memory I/O	0	The MEM bit specifies the PCB location. Set MEM to locate the PCB in memory space, or clear it to locate the PCB in I/O space.
R19:8	PCB Base Address Upper Bits	0FFH	R19:8 define the upper address bits of the PCB base address. All lower bits are zero. R19:16 are ignored when the PCB is mapped to I/O space.

NOTE: Reserved register bits are shown with gray shading. Reserved bits must be written to a logic zero to ensure compatibility with future Intel products.

Figure 4-1. PCB Relocation Register

Table 4-1. Peripheral Control Block

PCB Offset	Function	PCB Offset	Function	PCB Offset	Function	PCB Offset	Function
00H	Reserved	40H	T2CNT	80H	GCS0ST	C0H	Reserved
02H	EOI	42H	T2CMPA	82H	GCS0SP	C2H	Reserved
04H	POLL	44H	Reserved	84H	GCS1ST	C4H	Reserved
06H	POLLSTS	46H	T2CON	86H	GCS1SP	C6H	Reserved
08H	IMASK	48H	Reserved	88H	GCS2ST	C8H	Reserved
0AH	PRIMSK	4AH	Reserved	8AH	GCS2SP	CAH	Reserved
0CH	INSERV	4CH	Reserved	8CH	GCS3ST	CCH	Reserved
0EH	REQST	4EH	Reserved	8EH	GCS3SP	CEH	Reserved
10H	INSTS	50H	P1DIR	90H	GCS4ST	D0H	Reserved
12H	TCUCON	52H	P1PIN	92H	GCS4SP	D2H	Reserved
14H	SCUCON	54H	P1CON	94H	GCS5ST	D4H	Reserved
16H	I4CON	56H	P1LTCH	96H	GCS5SP	D6H	Reserved
18H	I0CON	58H	P2DIR	98H	GCS6ST	D8H	Reserved
1AH	I1CON	5AH	P2PIN	9AH	GCS6SP	DAH	Reserved
1CH	I2CON	5CH	P2CON	9CH	GCS7ST	DCH	Reserved
1EH	I3CON	5EH	P2LTCH	9EH	GCS7SP	DEH	Reserved
20H	Reserved	60H	B0CMP	A0H	LCSST	E0H	Reserved
22H	Reserved	62H	B0CNT	A2H	LCSSP	E2H	Reserved
24H	Reserved	64H	S0CON	A4H	UCSST	E4H	Reserved
26H	Reserved	66H	S0STS	A6H	UCSSP	E6H	Reserved
28H	Reserved	68H	S0RBUF	A8H	RELREG	E8H	Reserved
2AH	Reserved	6AH	S0TBUF	AAH	Reserved	EAH	Reserved
2CH	Reserved	6CH	Reserved	ACH	Reserved	ECH	Reserved
2EH	Reserved	6EH	Reserved	AEH	Reserved	EEH	Reserved
30H	T0CNT	70H	B1CMP	B0H	RFBASE	F0H	Reserved
32H	T0CMPA	72H	B1CNT	B2H	RFTIME	F2H	Reserved
34H	T0CMPB	74H	S1CON	B4H	RFCON	F4H	Reserved
36H	T0CON	76H	S1STS	B6H	RFADDR	F6H	Reserved
38H	T1CNT	78H	S1RBUF	B8H	PWRCON	F8H	Reserved
3AH	T1CMPA	7AH	S1TBUF	BAH	Reserved	FAH	Reserved
3CH	T1CMPB	7CH	Reserved	BCH	STEPID	FCH	Reserved
3EH	T1CON	7EH	Reserved	BEH	Reserved	FEH	Reserved

4.3 RESERVED LOCATIONS

Many locations within the Peripheral Control Block are not assigned to any peripheral. Unused locations are reserved. Reading from these locations yields an undefined result. If reserved registers are written (for example, during a block MOV instruction) they must be set to 0H.

NOTE

Failure to follow this guideline could result in incompatibilities with future 80C186 Modular Core family products.

4.4 ACCESSING THE PERIPHERAL CONTROL BLOCK

All communication between integrated peripherals and the Modular CPU Core occurs over a special bus, called the *F-Bus*, which always carries 16-bit data. The Peripheral Control Block, like all integrated peripherals, is always accessed 16 bits at a time.

4.4.1 Bus Cycles

The processor runs an external bus cycle for any memory or I/O cycle accessing a location within the Peripheral Control Block. Address, data and control information is driven on the external pins as with an ordinary bus cycle. Information returned by an external device is ignored, even if the access does not correspond to the location of an integrated peripheral control register. This is also true for the 80C188 Modular Core family, except that word accesses made to integrated registers are performed in two bus cycles.

4.4.2 READY Signals and Wait States

The processor generates an internal READY signal whenever an integrated peripheral is accessed. External READY is ignored. READY is also generated if an access is made to a location within the Peripheral Control Block that does not correspond to an integrated peripheral control register. For accesses to timer control and counting registers, the processor inserts one wait state. This is required to properly multiplex processor and counter element accesses to the timer control registers. For accesses to the remaining locations in the Peripheral Control Block, the processor does not insert wait states.

4.4.3 F-Bus Operation

The F-Bus functions differently than the external data bus for byte and word accesses. All write transfers on the F-Bus occur as words, regardless of how they are encoded. For example, the instruction OUT DX, AL (DX is even) will write the entire AX register to the Peripheral Control Block register at location [DX]. If DX were an odd location, AL would be placed in [DX] and AH would be placed at [DX–1]. A word operation to an odd address would write [DX] and [DX–1] with AL and AH, respectively. This differs from normal external bus operation where unaligned word writes modify [DX] and [DX+1]. In summary, do not use odd-aligned byte or word writes to the PCB.

Aligned word reads work normally. Unaligned word reads work differently. For example, IN AX, DX (DX is odd) will transfer [DX] into AL and [DX–1] into AH. Byte reads from even or odd addresses work normally, but only a byte will be read. For example, IN AL, DX will **not** transfer [DX] into AX (only AL is modified).

No problems will arise if the following recommendations are adhered to.

Word reads

Aligned word reads of the PCB work normally. Access only even-aligned words with IN AX, DX or MOV *word register, even PCB address.*

Byte reads

Byte reads of the PCB work normally. Beware of reading word-wide PCB registers that may change value between successive reads (e.g., timer count value).

Word writes

Always write even-aligned words to the PCB. Writing an odd-aligned word will give unexpected results.

For the 80C186 Modular Core, use either
– OUT DX, AX or
– OUT DX, AL or
– MOV *even PCB address, word register.*

For the 80C188 Modular Core, using OUT DX, AX will perform an unnecessary bus cycle and is not recommended. Use either
– OUT DX, AL or
– MOV *even-aligned byte PCB address, byte register low byte.*

Byte writes

Always use even-aligned byte writes to the PCB. Even-aligned byte writes will modify the entire word PCB location. Do **not** perform unaligned byte writes to the PCB.

4.4.3.1 Writing the PCB Relocation Register

Whenever mapping the Peripheral Control Block to another location, the user should program the Relocation Register with a **byte write** (i.e., OUT DX, AL). Internally, the Relocation Register is written with 16 bits of the AX register, while externally the Bus Interface Unit runs a single 8-bit bus cycle. If a word instruction (i.e., OUT DX, AX) is used with an 80C188 Modular Core family member, the Relocation Register is written on the first bus cycle. The Bus Interface Unit then runs an unnecessary second bus cycle. The address of the second bus cycle is no longer within the control block, since the Peripheral Control Block was moved on the first cycle. External READY must now be generated to complete the cycle. For this reason, we recommend byte operations for the Relocation Register.

4.4.3.2 Accessing the Peripheral Control Registers

Byte instructions should be used for the registers in the Peripheral Control Block of an 80C188 Modular Core family member. This requires half the bus cycles of word operations. Byte operations are valid only for even-addressed writes to the Peripheral Control Block. A word read (e.g., IN AX, DX) must be performed to read a 16-bit Peripheral Control Block register when possible.

4.4.3.3 Accessing Reserved Locations

Unused locations are reserved. If a write is made to these locations, a bus cycle occurs, but data is not stored. If a subsequent read is made to the same location, the value written is not read back. If reserved registers are written (for example, during a block MOV instruction) they must be cleared to 0H.

NOTE

Failure to follow this guideline could result in incompatibilities with future 80C186 Modular Core family products.

4.5 SETTING THE PCB BASE LOCATION

Upon reset, the PCB Relocation Register (see Figure 4-1 on page 4-2) contains the value 00FFH, which causes the Peripheral Control Block to be located at the top of I/O space (0FF00H to 0FFFFH). Writing the PCB Relocation Register allows the user to change that location.

As an example, to relocate the Peripheral Control Block to the memory range 10000-100FFH, the user would program the PCB Relocation Register with the value 1100H. Since the Relocation Register is part of the Peripheral Control Block, it relocates to word 10000H plus its fixed offset.

NOTE

Due to an internal condition, external ready is ignored if the device is configured in Cascade mode and the Peripheral Control Block (PCB) is located at 0000H in I/O space. In this case, wait states **cannot** be added to interrupt acknowledge bus cycles. However, you **can** add wait states to interrupt acknowledge cycles if the PCB is located at any other address.

4.5.1 Considerations for the 80C187 Math Coprocessor Interface

Systems using the 80C187 math coprocessor interface must **not** relocate the Peripheral Control Block to location 0000H in I/O space. The 80C187 interface uses I/O locations 0F8H through 0FFH. If the Peripheral Control Block resides in these locations, the processor communicates with the Peripheral Control Block, **not** the 80C187 interface circuitry.

NOTE

If the PCB is located at 0000H in I/O space and access to the math coprocessor interface is enabled (the Escape Trap bit is clear), a numerics (ESC) instruction causes indeterminate system operation.

Since the 8-bit bus version of the device does not support the 80C187, it automatically traps an ESC instruction to the Type 7 interrupt, regardless of the state of the Escape Trap (ET) bit.

For details on the math coprocessor interface, see Chapter 12, "Math Coprocessing."

intel®

5

Clock Generation and Power Management

CHAPTER 5
CLOCK GENERATION AND POWER MANAGEMENT

The clock generation and distribution circuits provide uniform clock signals for the Execution Unit, the Bus Interface Unit and all integrated peripherals. The 80C186 Modular Core Family processors have additional logic that controls the clock signals to provide power management functions.

5.1 CLOCK GENERATION

The clock generation circuit (Figure 5-1) includes a crystal oscillator, a divide-by-two counter, and reset circuitry. See "Power Management" on page 5-10 for a discussion of power management options.

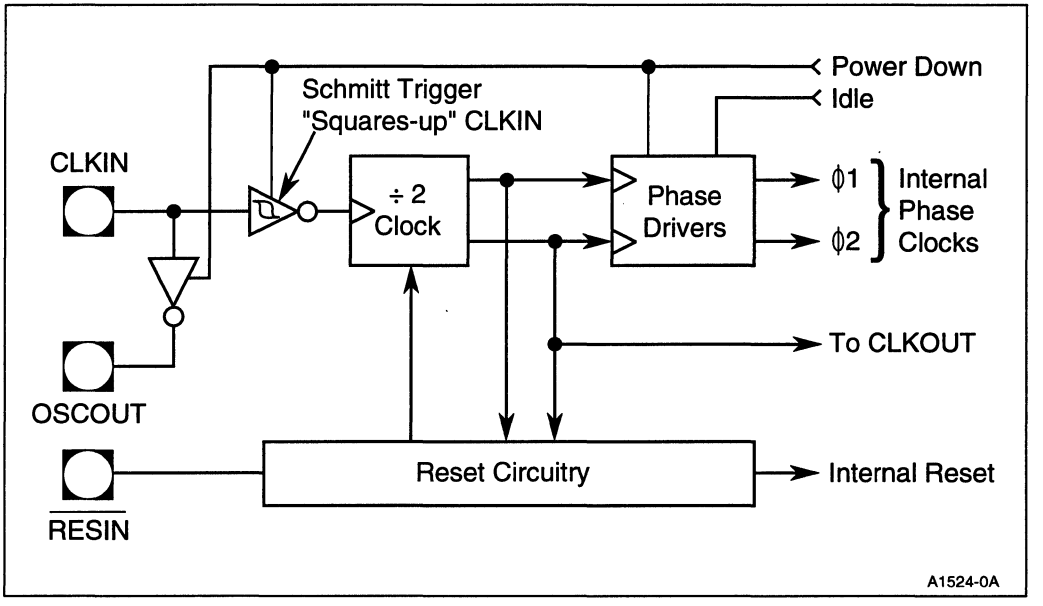


Figure 5-1. Clock Generator

5.1.1 Crystal Oscillator

The internal oscillator is a parallel resonant Pierce oscillator, a specific form of the common phase shift oscillator.

5.1.1.1 Oscillator Operation

A phase shift oscillator operates through positive feedback, where a non-inverted, amplified version of the input connects back to the input. A 360° phase shift around the loop will sustain the feedback in the oscillator. The on-chip inverter provides a 180° phase shift. The combination of the inverter's output impedance and the first load capacitor (see Figure 5-2) provides another 90° phase shift. At resonance, the crystal becomes primarily resistive. The combination of the crystal and the second load capacitor provides the final 90° phase shift. Above and below resonance, the crystal is reactive and forces the oscillator back toward the crystal's nominal frequency.

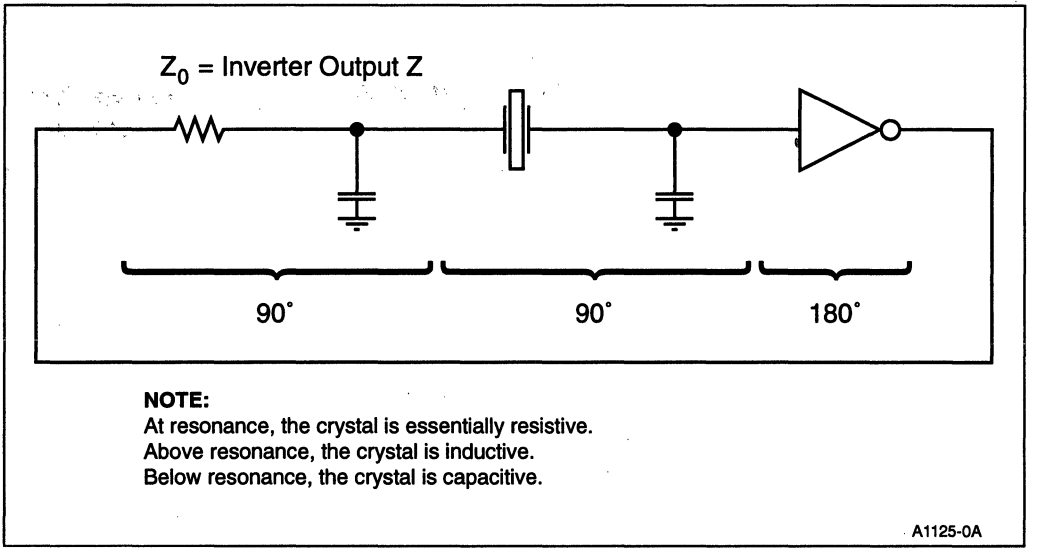


Figure 5-2. Ideal Operation of Pierce Oscillator

Figure 5-3 shows the actual microprocessor crystal connections. For low frequencies, crystal vendors offer fundamental mode crystals. At higher frequencies, a third overtone crystal is the only choice. The external capacitors, C_{X1} at CLKIN and C_{X2} at OSCOUT, together with stray capacitance, form the load. A third overtone crystal requires an additional inductor L_1 and capacitor C_1 to select the third overtone frequency and reject the fundamental frequency. See "Selecting Crystals" on page 5-5 for a more detailed discussion of crystal vibration modes.

Choose C_1 and L_1 component values in the third overtone crystal circuit to satisfy the following conditions:

- The LC components form an equivalent series resonant circuit at a frequency below the fundamental frequency. This criterion makes the circuit inductive at the fundamental frequency. The inductive circuit cannot make the 90° phase shift and oscillations do not take place.
- The LC components form an equivalent parallel resonant circuit at a frequency about halfway between the fundamental frequency and the third overtone frequency. This criterion makes the circuit capacitive at the third overtone frequency, necessary for oscillation.
- The two capacitors and inductor at OSCOUT, plus some stray capacitance, approximately equal the 20 pF load capacitor, C_{X2}, used alone in the fundamental mode circuit.

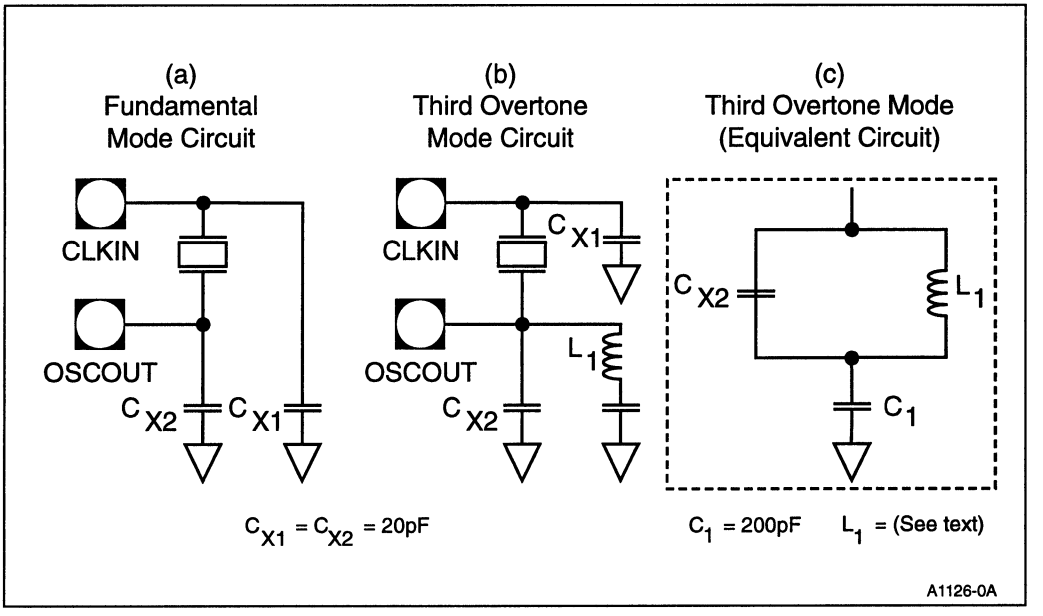


Figure 5-3. Crystal Connections to Microprocessor

Choosing C_1 as 200 pF (at least 10 times the value of the load capacitor) simplifies the circuit analysis. At the series resonance, the capacitance connected to L_1 is 200 pF in series with 20 pF. The equivalent capacitance is still about 20 pF and the equation in Figure 5-4(a) yields the series resonant frequency.

To examine the parallel resonant frequency, refer to Figure 5-3(c), an equivalent circuit to Figure 5-3(b). The capacitance connected to L_1 is 200 pF in parallel with 20 pF. The equivalent capacitance is still about 200 pF (within 10%) and the equation in Figure 5-4(a) now yields the parallel resonant frequency.

(a) Series or Parallel Resonant Frequency

$$f = \frac{1}{2\pi\sqrt{L_1 C_1}}$$

(b) Equivalent Capacitance

$$C_{eq} = \frac{\omega^2 C_1 C_{x2} L_1 - C_1 - C_{x2}}{\omega^2 C_1 L_1 - 1}$$

Figure 5-4. Equations for Crystal Calculations

The equation in Figure 5-4(b) yields the equivalent capacitance C_{eq} at the operation frequency. The desired operation frequency is the third overtone frequency marked on the crystal. Optimizing equations for the above three criteria yields Table 5-1. This table shows suggested standard inductor values for various processor frequencies. The equivalent capacitance is about 15 pF.

Table 5-1. Suggested Values for Inductor L_1 in Third Overtone Oscillator Circuit

CLKOUT Frequency (MHz)	Third-Overtone Crystal Frequency (MHz)	Inductor L_1 Values (µH)
13.04	26.08	6.8, 8.2, 10.0
16	32	3.9, 4.7, 5.6
20	40	2.2, 2.7, 3.3

5.1.1.2 Selecting Crystals

When specifying crystals, consider these parameters:

- Resonance and Load Capacitance — Crystals carry a parallel or series resonance specification. The two types do not differ in construction, just in test conditions and expected circuit application. Parallel resonant crystals carry a test load specification, with typical load capacitance values of 15, 18 or 22 pF. Series resonant crystals do not carry a load capacitance specification. You may use a series resonant crystal with the microprocessor, even though the circuit is parallel resonant. However, it will vibrate at a frequency slightly (on the order of 0.1%) higher than its calibration frequency.
- Vibration Mode — The vibration mode is either fundamental or third overtone. Crystal thickness varies inversely with frequency. Vendors furnish third or higher overtone crystals to avoid manufacturing very thin, fragile quartz crystal elements. At a given frequency, an overtone crystal is thicker and more rugged than its fundamental mode counterpart. Below 20 MHz, most crystals are fundamental mode. In the 20 to 32 MHz range, you can purchase both modes. You must know the vibration mode to know whether to add the LC circuit at OSCOUT.
- Equivalent Series Resistance (ESR) — ESR is proportional to crystal thickness, inversely proportional to frequency. A lower value gives a faster startup time, but the specification is usually not important in microprocessor applications.
- Shunt Capacitance — A lower value reduces ESR, but typical values such as 7 pF will work fine.
- Drive Level — Specifies the maximum power dissipation for which the manufacturer calibrated the crystal. It is proportional to ESR, frequency, load and V_{CC}. Disregard this specification unless you use a third overtone crystal whose ESR and frequency will be relatively high. Several crystal manufacturers stock a standard microprocessor crystal line. Specifying a "microprocessor grade" crystal should ensure that the rated drive level is a couple of milliwatts with 5-volt operation.
- Temperature Range — Specifies an operating range over which the frequency will not vary beyond a stated limit. Specify the temperature range to match the microprocessor temperature range.
- Tolerance — The allowable frequency deviation at a particular calibration temperature, usually 25° C. Quartz crystals are more accurate than microprocessor applications call for; do not pay for a tighter specification than you need. Vendors quote frequency tolerance in percentage or parts per million (ppm). Standard microprocessor crystals typically have a frequency tolerance of 0.01% (100 ppm). If you use these crystals, you can usually disregard all the other specifications; these crystals are ideal for the 80C186 Modular Core family.

An important consideration when using crystals is that the oscillator **start** correctly over the voltage and temperature ranges expected in operation. Observe oscillator startup in the laboratory. Varying the load capacitors (within about ± 50%) can optimize startup characteristics versus stability. In your experiments, consider stray capacitance and scope loading effects.

For help in selecting external oscillator components for unusual circumstances, count on the crystal manufacturer as your best resource. Using low-cost ceramic resonators in place of crystals is possible if your application will tolerate less precise frequencies.

5.1.2 Using an External Oscillator

The microprocessor's on-board clock oscillator allows the use of a relatively low cost crystal. However, the designer may also use a "canned oscillator" or other external frequency source. Connect the external frequency input (EFI) signal directly to the oscillator CLKIN input. Leave OSCOUT unconnected. This oscillator input drives the internal divide-by-two counter directly, generating the CPU clock signals. The external frequency input can have practically any duty cycle, provided it meets the minimum high and low times stated in the data sheet. Selecting an external clock oscillator is more straightforward than selecting a crystal.

5.1.3 Output from the Clock Generator

The crystal oscillator output drives a divide-by-two circuit, generating a 50% duty cycle clock for the processor's integrated components. All processor timings refer to this clock, available externally at the CLKOUT pin. CLKOUT changes state on the high-to-low transition of the CLKIN signal, even during reset and bus hold. CLKOUT is also available during Idle mode, but not during Powerdown mode. (See "Idle Mode" on page 5-11 and "Powerdown Mode" on page 5-16.)

In a CMOS circuit, significant current flows only during logic level transitions. Since the microprocessor consists mostly of clocked circuitry, the clock distribution is the basis of power management.

5.1.4 Reset and Clock Synchronization

The clock generator provides a system reset signal (RESOUT). The $\overline{\text{RESIN}}$ input generates RESOUT and the clock generator synchronizes it to the CLKOUT signal.

A Schmitt trigger in the $\overline{\text{RESIN}}$ input ensures that the switch point for a low-to-high transition is greater than the switch point for a high-to-low transition. The processor must remain in reset a minimum of 4 CLKOUT cycles after V_{CC} and CLKOUT stabilize. The hysteresis allows a simple RC circuit to drive the $\overline{\text{RESIN}}$ input (see Figure 5-5). Typical applications can use about 100 milliseconds as an RC time constant.

Reset may be either cold (power-up) or warm. Figure 5-6 illustrates a cold reset. Assert the $\overline{\text{RESIN}}$ input during power supply and oscillator startup. The processor's pins assume their reset pin states a maximum of 28 CLKIN periods after CLKIN and V_{CC} stabilize. Assert $\overline{\text{RESIN}}$ 4 additional CLKIN periods after the device pins assume their reset states.

Applying $\overline{\text{RESIN}}$ when the device is running constitutes a warm reset (see Figure 5-7). In this case, assert $\overline{\text{RESIN}}$ for at least 4 CLKOUT periods. The device pins will assume their reset states on the second falling edge of CLKIN following the assertion of $\overline{\text{RESIN}}$.

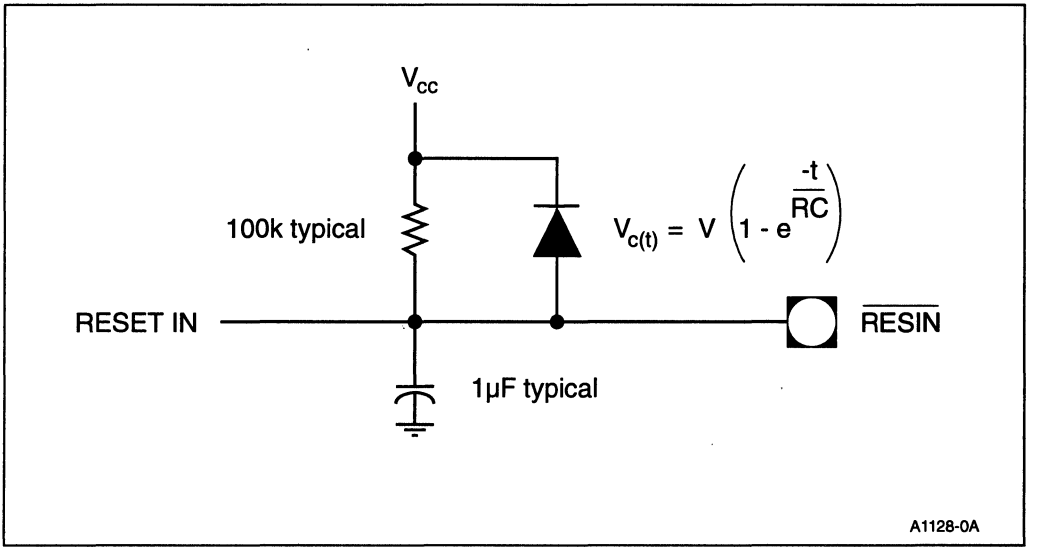
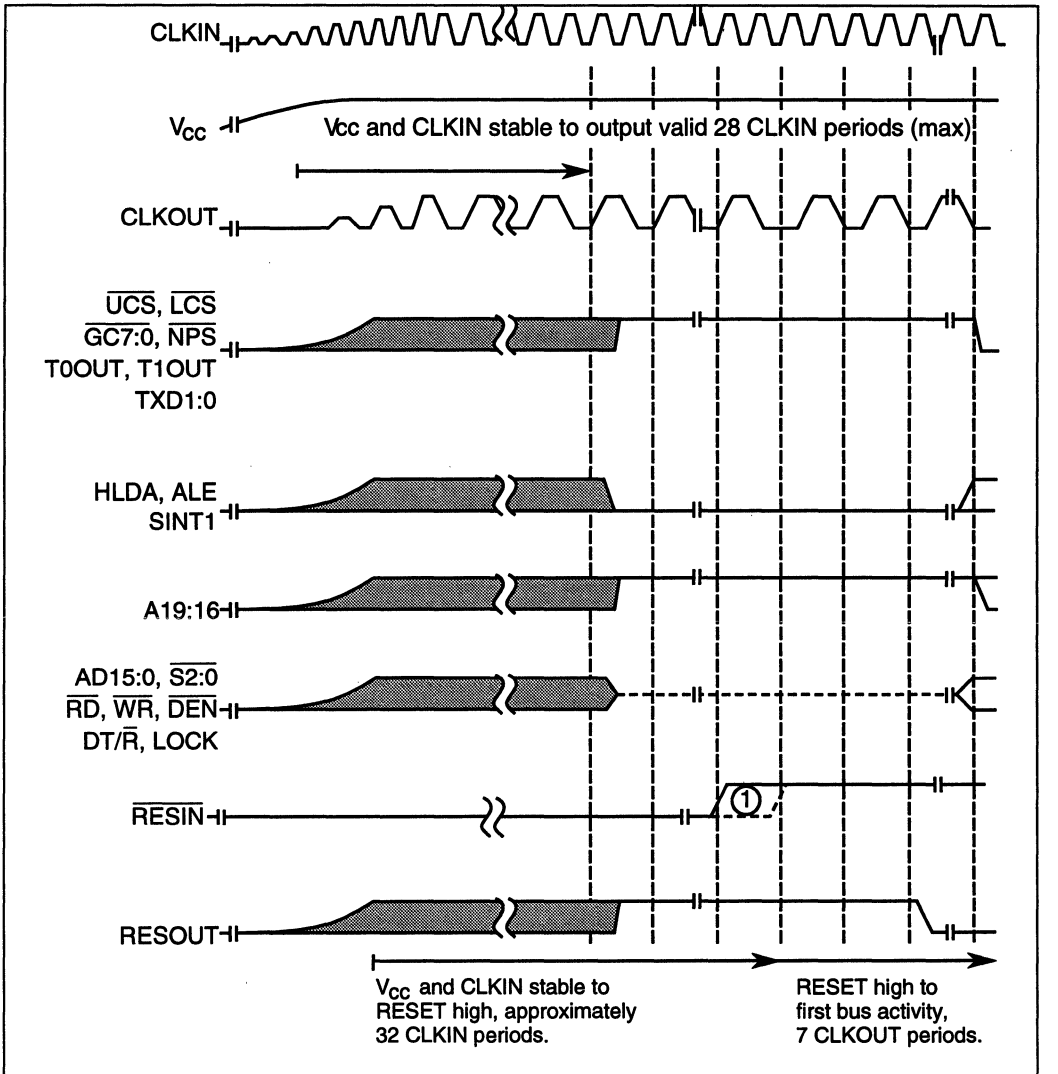


Figure 5-5. Simple RC Circuit for Powerup Reset

The processor exits reset identically in both cases. The rising $\overline{\text{RESIN}}$ edge generates an internal RESYNC pulse (see Figure 5-8), resynchronizing the divide-by-two internal phase clock. The clock generator samples $\overline{\text{RESIN}}$ on the falling CLKIN edge. If $\overline{\text{RESIN}}$ is sampled high while CLKOUT is high, the processor forces CLKOUT low for the next two CLKIN cycles. The clock essentially "skips a beat" to synchronize the internal phases. If $\overline{\text{RESIN}}$ is sampled high while CLKOUT is low, CLKOUT is already in phase.



NOTES:

1. CLKOUT synchronization occurs on the rising edge of RESIN. If RESIN is sampled high while CLKOUT is high (solid line), then CLKOUT will remain low for two CLKIN periods. If RESIN is sampled high while CLKOUT is low (dashed line), the CLKOUT will not be affected.

A1134-0A

Figure 5-6. Cold Reset Waveform

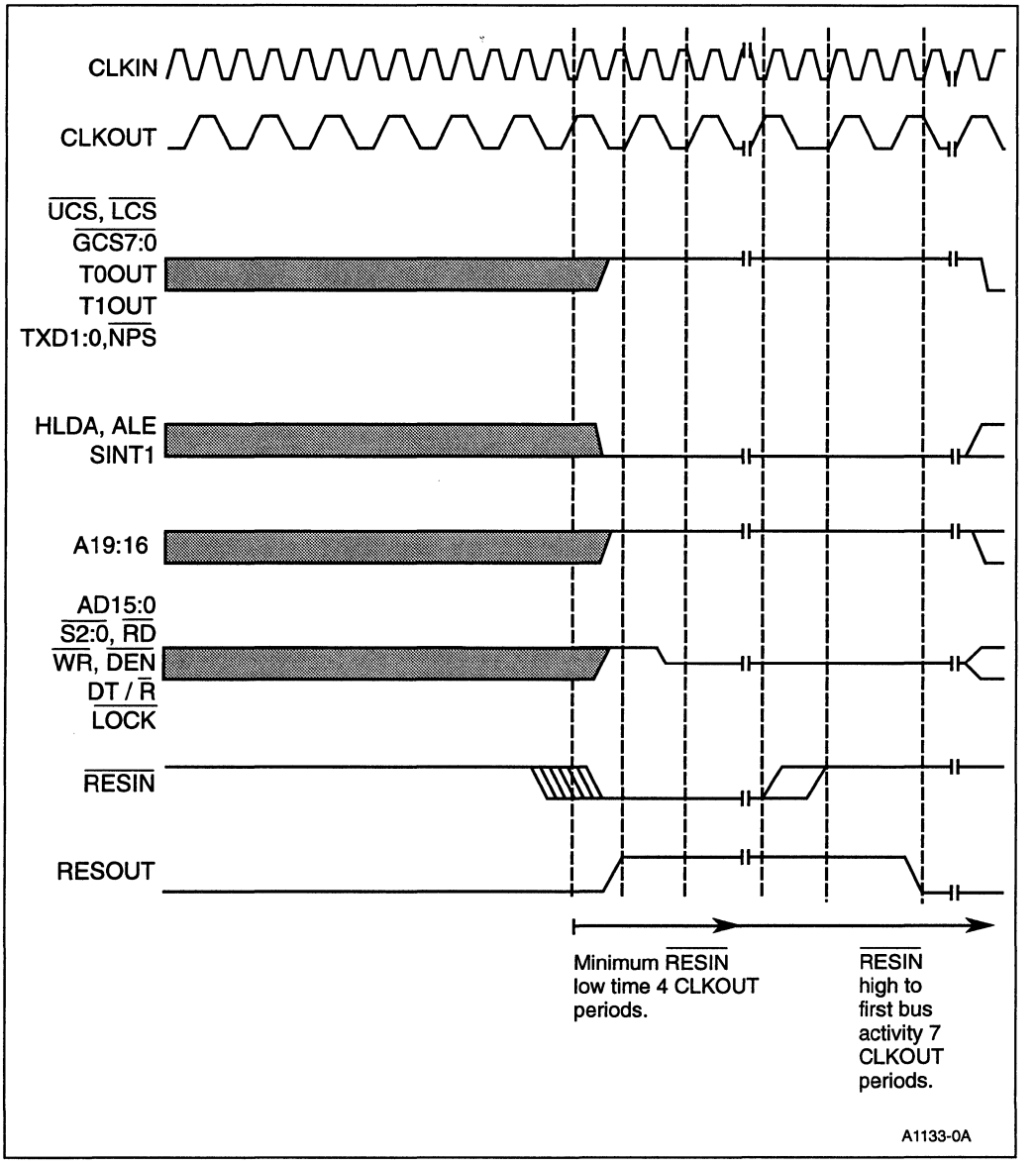
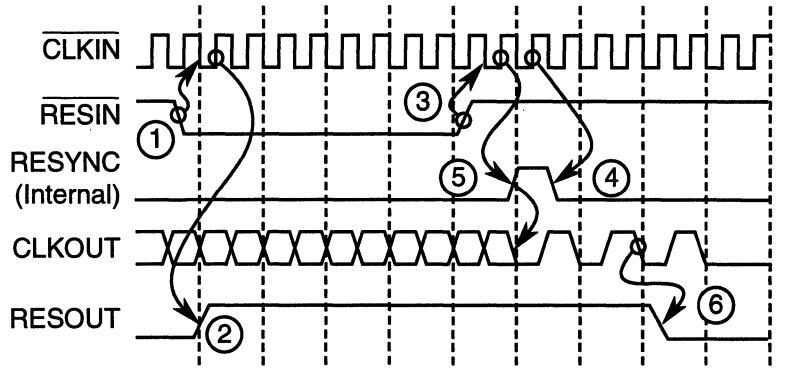


Figure 5-7. Warm Reset Waveform

At the second falling CLKOUT edge after the internal clocks resynchronize, the processor deasserts RESOUT. Bus activity starts seven CLKOUT periods after recognition of $\overline{\text{RESIN}}$ in the logic high state. If an alternate bus master asserts HOLD during reset, the processor immediately asserts HLDA and will not prefetch instructions.



NOTES:
1. Setup of RESIN to falling CLKIN.
2. RESOUT goes active.
3. RESIN allowed to go inactive after minimum 4 CLKOUT cycles.
4. RESYNC pulse generated.
5. RESYNC pulse drives CLKOUT low, resynchronizing the clock generator.
6. RESOUT goes inactive on the second falling CLKOUT edge following CLKOUT resynchronization.

A1117-0A

Figure 5-8. Clock Synchronization at Reset

5.2 POWER MANAGEMENT

Many VLSI devices available today use dynamic circuitry. A dynamic circuit uses a capacitor (usually parasitic gate or diffusion capacitance) to store information. The stored charge decays over time due to leakage currents in the silicon. If the device does not use the stored information before it decays, the state of the entire device may be lost. Circuits must periodically refresh dynamic RAMs, for example, to ensure data retention. Any microprocessor that has a minimum clock frequency has dynamic logic. On a dynamic microprocessor, if you stop or slow the clock, the dynamic nodes within it begin discharging. With a long enough delay, the processor is likely to lose its present state, needing a reset to resume normal operation.

An 80C186 Modular Core microprocessor is fully **static**. The CPU stores its current state in flip-flops, not capacitive nodes. The clock signal to both the CPU core and the peripherals can stop without losing any internal information, provided the design maintains power. When the clock restarts, the device will execute from its previous state. When the processor is inactive for significant periods, special power management hardware takes advantage of static operation to achieve major power savings.

There are two power management modes: Idle and Powerdown. Power management is a clock distribution function. For this discussion, Active mode is the condition of no programmed power management. Active mode operation feeds the clock signal to the CPU core and all the integrated peripherals and power consumption reaches its maximum for the application. The processor defaults to Active mode at reset.

5.2.1 Idle Mode

During Idle mode operation, the clock signal is routed only to the integrated peripheral devices. CLKOUT continues toggling. The clocks to the CPU core (Execution and Bus Interface Units) freeze in a logic low state. Idle mode reduces current consumption by about a third, depending on the activity in the peripheral units.

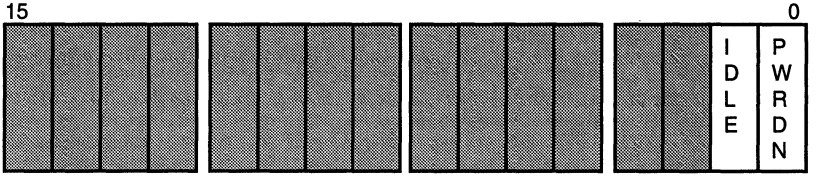
5.2.1.1 Entering Idle Mode

Setting the appropriate bit in the Power Control Register (Figure 5-9) prepares for Idle mode. The processor enters Idle mode when it executes the HLT (halt) instruction. If the program arms both Idle mode and Powerdown mode by mistake, the device halts but remains in Active mode. See Chapter 3, "Bus Interface Unit," for detailed information on HALT bus cycles. Figure 5-10 shows some internal and external waveforms during entry into Idle mode.

Register Name: Power Control Register

Register Mnemonic: PWRCON

Register Function: Arms power management functions.



A1129-0A

Bit Mnemonic	Bit Name	Reset State	Function
IDLE	Idle Mode	0	Setting the IDLE bit forces the CPU to enter the Idle mode when the HLT instruction is executed. The PWRDN bit must be cleared when setting the IDLE bit, otherwise Idle mode is not armed.
PWRDN	Powerdown Mode	0	Setting the PWRDN bit forces the CPU to enter the Powerdown mode when the next HLT instruction is executed. The IDLE bit must be cleared when setting the PWRDN bit, otherwise Powerdown mode is not armed.

NOTE: Reserved register bits are shown with gray shading. Reserved bits must be written to a logic zero to ensure compatibility with future Intel products.

Figure 5-9. Power Control Register

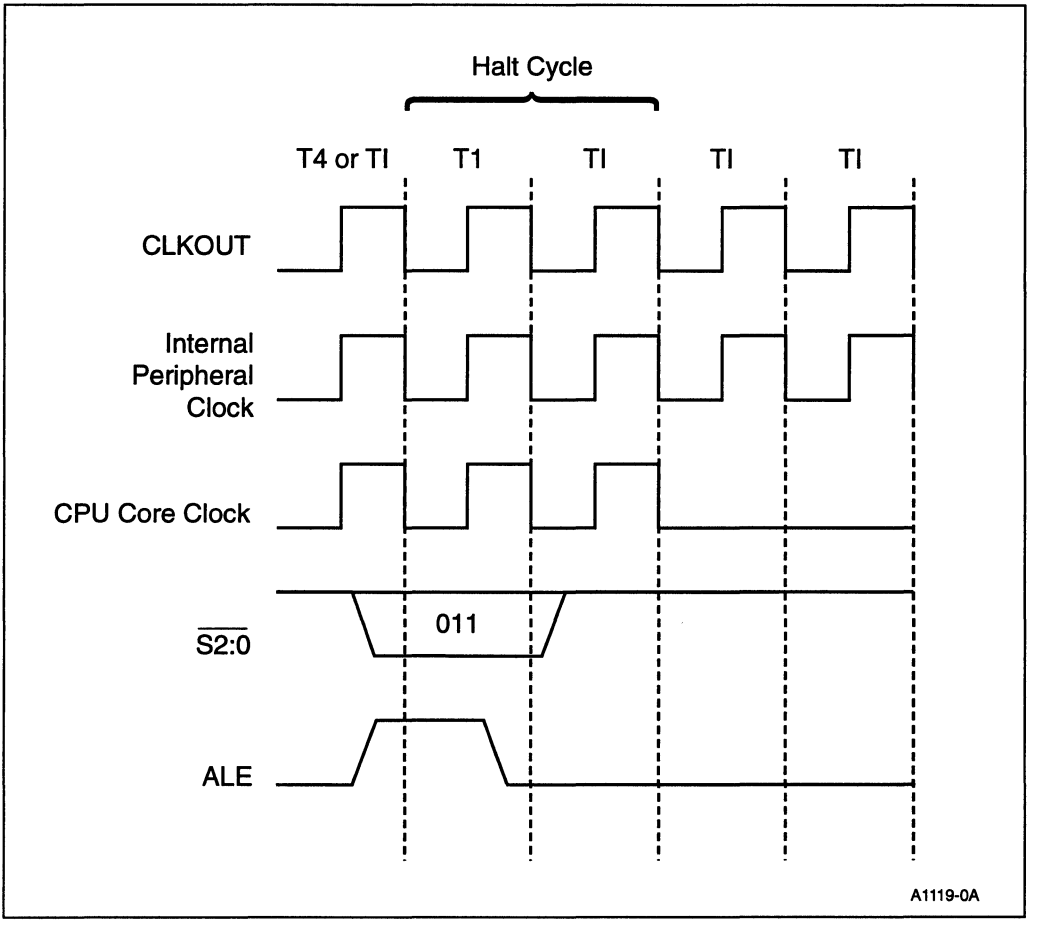


Figure 5-10. Entering Idle Mode

5.2.1.2 Bus Operation During Idle Mode

DRAM refresh requests and HOLD requests temporarily turn on the core clocks. If the processor needs to run a refresh cycle during Idle mode, the internal core clock begins to toggle on the falling CLKOUT edge immediately after the down-counter reaches zero. After one idle T-state, the processor runs the refresh cycle. As with all other bus cycles, the BIU uses the ready, wait state generation and chip-select circuitry as necessary for refresh cycles during Idle mode. There is one idle T-state after T4 before the internal core clock shuts off again.

A HOLD request from an external bus master turns on the core clock as long as HOLD is active (see Figure 5-11). The core clock restarts one CLKOUT cycle after the bus processor samples HOLD high. The microprocessor asserts HLDA one cycle after the core clock starts. The core clock turns off and the processor deasserts HLDA one cycle after the external bus master deasserts HOLD.

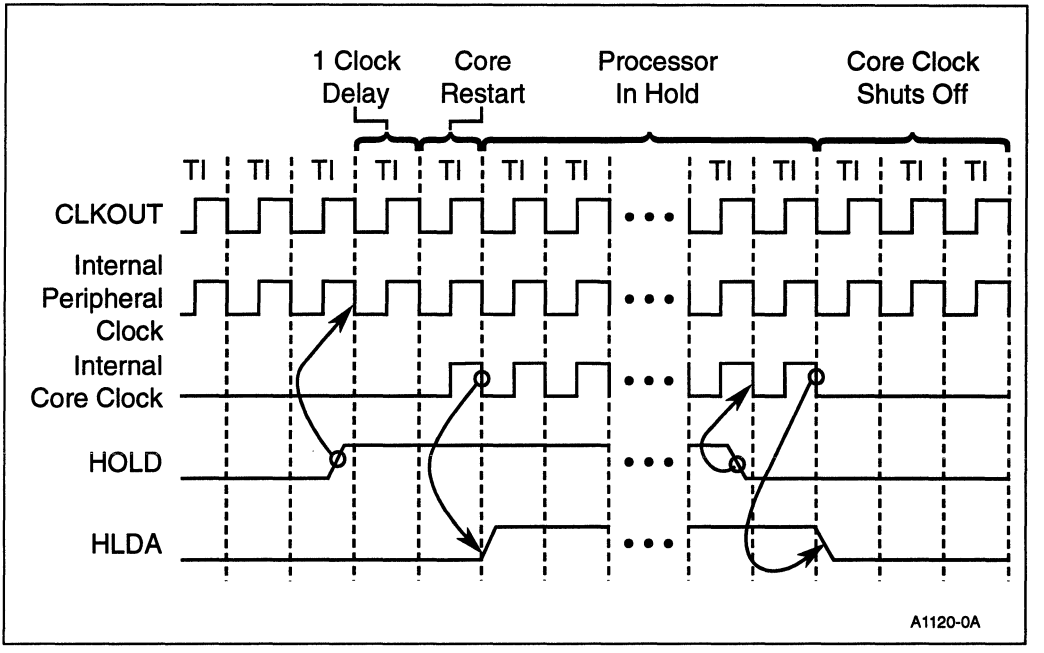


Figure 5-11. HOLD/HLDA During Idle Mode

As in Active mode, refresh requests will force the BIU to drop HLDA during bus hold. (For more information on refresh cycles during hold, see "Refresh Operation During a Bus HOLD" on page 3-41 and "Refresh Operation and Bus HOLD" on page 7-13.)

5.2.1.3 Leaving Idle Mode

Any unmasked interrupt or non-maskable interrupt (NMI) will return the processor to Active mode. Reset also returns the processor to Active mode, but the device loses its prior state.

Any **unmasked** interrupt received by the core will return the processor to Active mode. Interrupt requests pass through the Interrupt Control Unit with an interrupt resolution time for mask and priority level checking. Then, after 1½ clocks, the core clock begins toggling. It takes an additional 6 CLKOUT cycles for the core to begin the interrupt vectoring sequence.

After execution of the IRET (interrupt return) instruction in the interrupt service routine, the CS:IP will point to the instruction following the HALT. Interrupt execution does not modify the Power Control Register. Unless the programmer intentionally reprograms the register after exiting Idle mode, the processor will re-enter Idle mode at the next HLT instruction.

Like an unmasked interrupt, an NMI will return the core to Active mode from Idle mode. It takes two CLKOUT cycles to restart the core clock after an NMI occurs. The NMI signal does not need the mask and priority checks that a maskable interrupt does. This results in a considerable difference in clock restart time between an NMI and an unmasked interrupt. The core begins the interrupt response six cycles after the core clock restarts when it fetches the NMI vector from location 00008H. NMI does not clear the IDLE bit in the Power Control Register.

Resetting the microprocessor will return the device to Active mode. Unlike interrupts, a reset clears the Power Control Register. Execution begins as it would following a warm reset (see "Reset and Clock Synchronization" on page 5-6).

5.2.1.4 Example Idle Mode Initialization Code

Example 5-1 illustrates programming the Power Control Register and entering Idle mode upon HLT. The interrupts from the serial port and timers are not masked. Assume that the serial port connects to a keyboard controller. At every keystroke, the keyboard sends a data byte, and the processor wakes up to service the interrupt. After acting on the keystroke, the core will go back into Idle mode. The example excludes the actual keystroke processing.

```
$mod186
name            example_80C186_power_management_code

;FUNCTION:      This function reduces CPU power consumption.
;  SYNTAX:      extern void far power_mgt(int mode);
;  INPUTS:      mode - 00 -> Active Mode
;                      01 -> Powerdown Mode
;                      02 -> Idle Mode
;                      03 -> Active Mode
; OUTPUTS:      None
;    NOTE:      Parameters are passed on the stack as required
;               by high-level languages

PWRCON equ xxxxH                                ;substitute PWRCON register
                                                ;offset
lib_80C186      segment public 'code'
                assume cs:lib_80C186
                public _power_mgt
_power_mgt      proc far

                push bp                         ;save caller's bp
                mov  bp, sp                     ;get current top of stack
                push ax                         ;save registers that will
                push dx                         ;be modified

_mode           equ  word ptr[bp+6]             ;get parameter off the
                                                ;stack
                mov  dx, PWRCON                 ;select Power Control Reg
                mov  ax, _mode                  ;get mode
                and  ax, 3                      ;mask off unwanted bits
                out  dx, ax
                hlt                             ;enter mode
                pop  dx                         ;restore saved registers
                pop  ax
                pop  bp                         ;restore caller's bp
                ret
_power_mgt      endp
lib_80C186      ends
                end
```

Example 5-1. Initializing the Power Management Unit for Idle or Powerdown Mode

5.2.2 Powerdown Mode

Powerdown mode freezes the clock to the entire device (core and peripherals) and disables the crystal oscillator. All internal devices (registers, state machines, etc.) maintain their states as long as V_{CC} is applied. The BIU will not honor DRAM refresh and HOLD requests in Powerdown mode because the clocks for those functions are off. CLKOUT freezes in a logic high state. Current consumption in Powerdown mode consists of just transistor leakage (typically less than 100 microamps).

5.2.2.1 Entering Powerdown Mode

Powerdown mode is entered by executing the HLT instruction after setting the PWRDN bit in the Power Control Register (see Figure 5-9 on page 5-12). The HALT cycle turns off both the core and peripheral clocks and disables the crystal oscillator. See Chapter 3, "Bus Interface Unit," for detailed information on HALT bus cycles. Figure 5-12 shows the internal and external waveforms during entry into Powerdown mode.

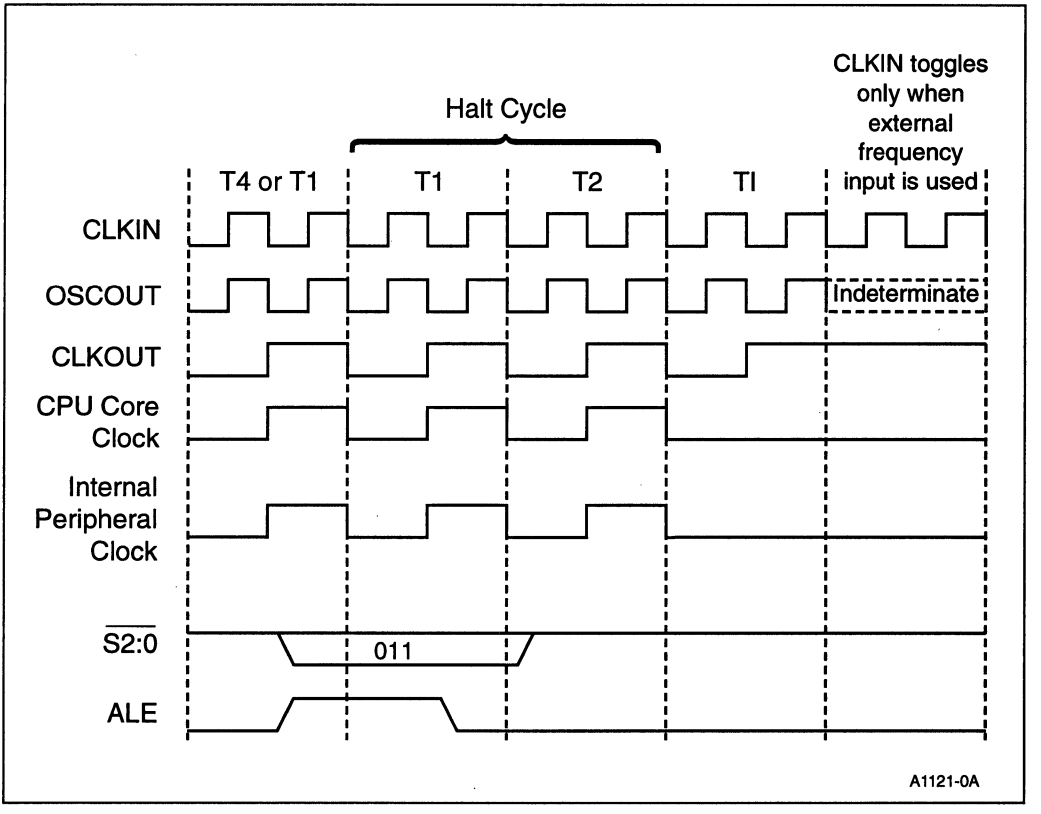


Figure 5-12. Entering Powerdown Mode

During the T2 phase of the HLT instruction, the core generates a signal called Enter_Powerdown. Enter_Powerdown immediately disables the internal CPU core and peripheral clocks. The processor disables the oscillator inverter during the next CLKOUT cycle. If the design uses a crystal oscillator, the oscillator stops immediately. When CLKIN originates from an external frequency input (EFI), Powerdown isolates the signal on the CLKIN pin from the internal circuitry. Therefore, the circuit may drive CLKIN during Powerdown mode, although it will not clock the device.

5.2.2.2 Leaving Powerdown Mode

An NMI or reset returns the processor to Active mode. If the device leaves Powerdown mode by an NMI, a delay must follow the interrupt request to allow the crystal oscillator to stabilize before gating it to the internal phase clocks. An external timing pin sets this delay as described below. Leaving Powerdown by an NMI does not clear the PWRDN bit in the Power Control Register. A reset also takes the processor out of Powerdown mode. Since the oscillator is off, the user should follow the oscillator cold start guidelines (see "Reset and Clock Synchronization" on page 5-6).

The Powerdown timer circuit (Figure 5-13) has a PDTMR pin. Connecting this pin to an external capacitor gives the user control over the gating of the crystal oscillator to the internal clocks. The strong P-channel device is always on except during exit from Powerdown mode. This pullup keeps the powerdown capacitor C_{PD} charged up to V_{CC}. C_{PD} discharges slowly. At the same time, the circuit turns on the feedback inverter on the crystal oscillator and oscillation starts.

The Schmitt trigger connected to the PDTMR pin asserts the internal OSC_OK signal when the voltage at the pin drops below its switching threshold. The OSC_OK signal gates the crystal oscillator output to the internal clock circuitry. One CLKOUT cycle runs before the internal clocks turn back on. It takes two additional CLKOUT cycles for an NMI request to reach the CPU and another six clocks for the vector to be fetched.

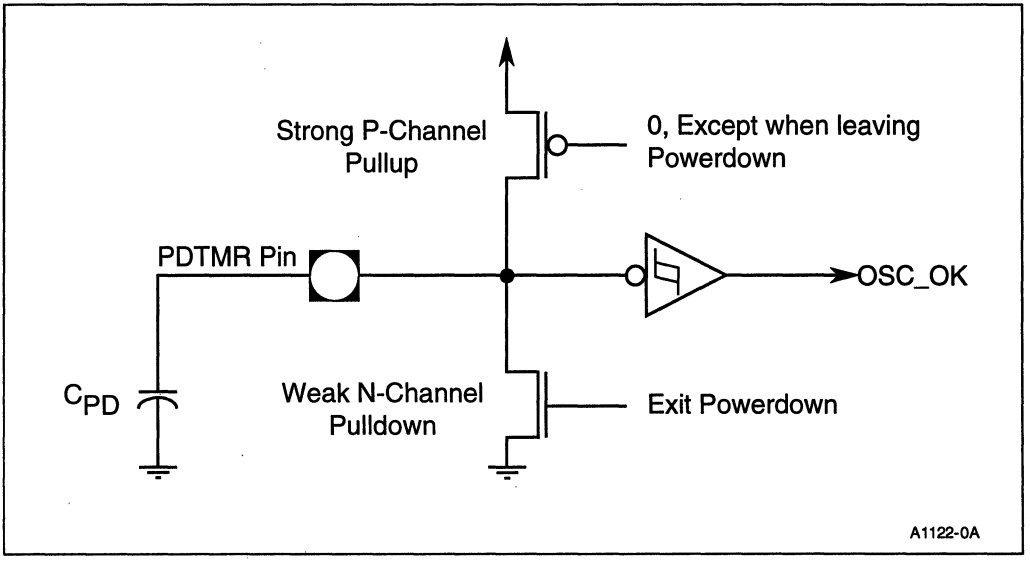


Figure 5-13. Powerdown Timer Circuit

The first step in determining the proper C_{PD} value is startup time characterization for the crystal oscillator circuit. This step can be done with a storage oscilloscope if you compensate for scope probe loading effects. Characterize startup over the full range of operating voltages and temperatures. The oscillator starts up on the order of a couple of milliseconds. After determining the oscillator startup time, refer to "PDTMR Pin Delay Calculation" in the data sheet. Multiply the startup time (in seconds) by the given constant to get the C_{PD} value. Typical values are less than 1μF.

If the design uses an external oscillator instead of a crystal, the external oscillator continues running during Powerdown mode. Leave the PDTMR pin unconnected and the processor can exit Powerdown mode immediately.

5.2.3 Implementing a Power Management Scheme

Table 5-2 summarizes the power management options available to the user. Overall power consumption has two parts: switching power dissipated by driving loads such as the address/data bus, and device power dissipated internally by the microprocessor whether or not it is connected to external devices. A power management scheme should consider loading as well as the raw specifications in the processor's data sheet.

Table 5-2. Summary of Power Management Modes

Mode	Relative Power	Typical Power	User Overhead	Chief Advantage
Active	Full	250 mW at 16 MHz	—	Full-speed operation
Idle	Low	175 mW at 16 MHz	Low	Peripherals are unaffected
Powerdown	Lowest	250 μW	Low to Moderate	Long battery life

NOTE

If an NMI or external maskable interrupt service routine is used to enter a power management mode, the interrupt request signal should be deasserted before entering the power management mode.

intel®

6

Chip-Select Unit

intel

CHAPTER 6
CHIP-SELECT UNIT

Every system requires some form of component-selection mechanism to enable the CPU to access a specific memory or peripheral device. The signal that selects the memory or peripheral device is referred to as a chip-select. Besides selecting a specific device, each chip-select can be used to control the number of wait states inserted into the bus cycle. Devices that are too slow to keep up with the maximum bus bandwidth can use wait states to slow the bus down.

6.1 COMMON METHODS FOR GENERATING CHIP-SELECTS

One method of generating chip-selects uses latched address signals directly. An example interface is shown in Figure 6-1(A). In the example, an inverted A16 is connected to an SRAM device with an active-low chip-select. Any bus cycle with an address between 10000H and 1FFFFH (A16 = 1) enables the SRAM device. Also note that any bus cycle with an address starting at 30000H, 50000H, 70000H and so on also selects the SRAM device.

Decoding more address bits solves the problem of a chip-select being active over multiple address ranges. In Figure 6-1(B), a one-of-eight decoder is connected to the uppermost address bits. Each decoded output is active for one-eighth of the 1 Mbyte address space. However, each chip-select has a fixed starting address and range. Future system memory changes could require circuit changes to accommodate the additional memory.

6.2 CHIP-SELECT UNIT FEATURES AND BENEFITS

The Chip-Select Unit overcomes limitations of the designs shown in Figure 6-1 and has the following features:

- Ten chip-select outputs
- Programmable start and stop addresses
- Memory or I/O bus cycle decoder
- Programmable wait-state generator
- Provision to disable a chip-select
- Provision to override bus ready

Figure 6-2 illustrates the logic blocks that generate a chip-select. Each chip-select has a duplicate set of logic.

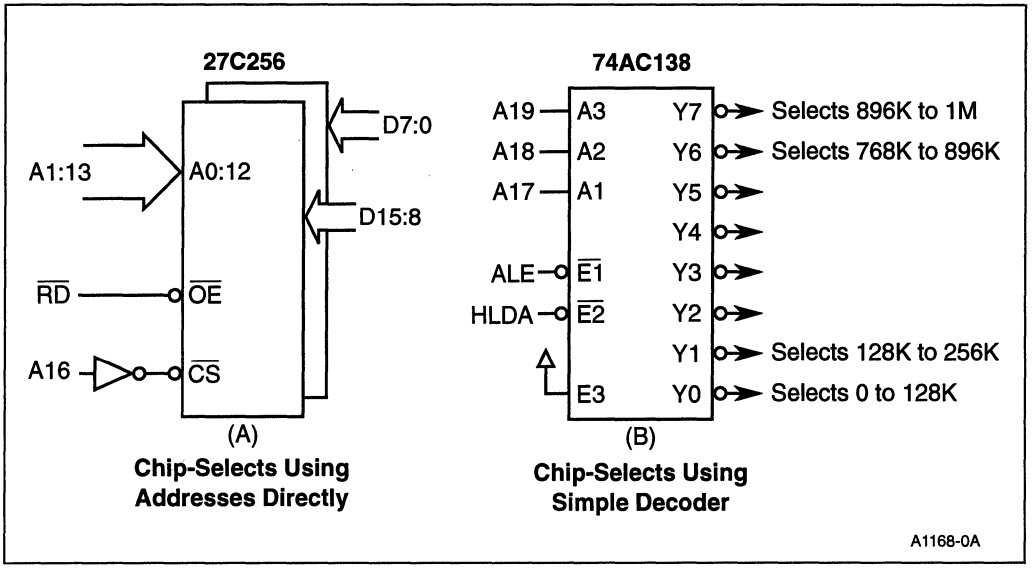


Figure 6-1. Common Chip-Select Generation Methods

6.3 CHIP-SELECT UNIT FUNCTIONAL OVERVIEW

The Chip-Select Unit (CSU) decodes bus cycle address and status information and enables the appropriate chip-select. Figure 6-3 illustrates the timing of a chip-select during a bus cycle. Note that the chip-select goes active in the same bus state as address goes active, eliminating any delay through address latches and decoder circuits. The Chip-Select Unit activates a chip-select for bus cycles initiated by the CPU or Refresh Control Unit.

Any of the ten chip-selects can map into either memory or I/O address space. A memory-mapped chip-select can start and end on any 1 Kbyte address location. An I/O-mapped chip-select can start and end on any 64 byte address location. The chip-selects typically associate with memory and peripheral devices as follows:

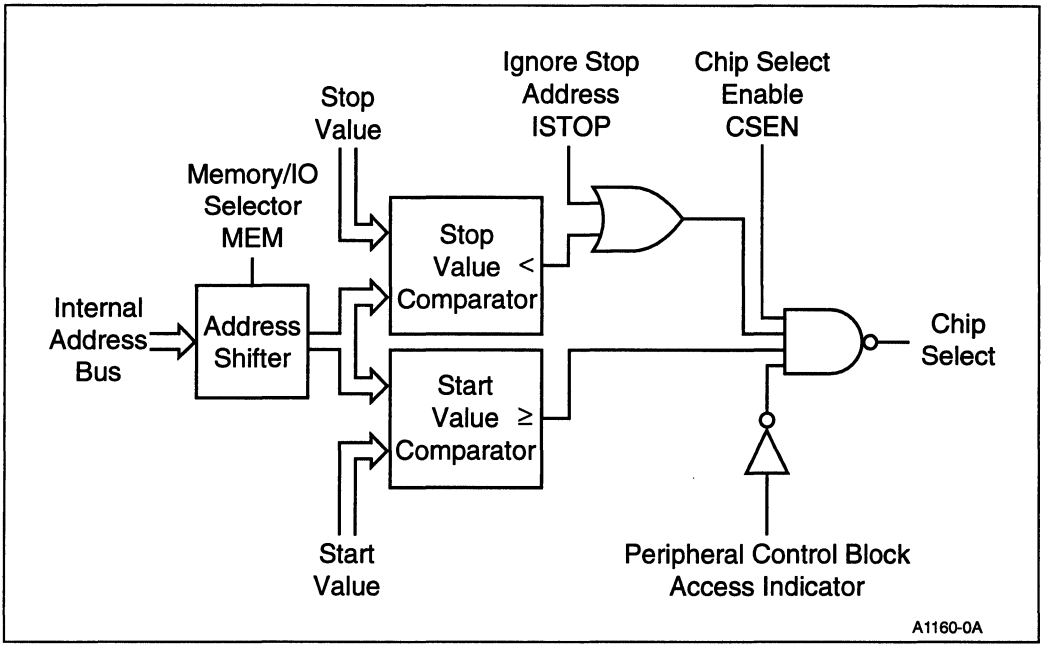


Figure 6-2. Chip-Select Block Diagram

$\overline{\text{UCS}}$ Mapped to the upper memory address space; selects the BOOT memory device (EPROM or Flash memory types).

$\overline{\text{LCS}}$ Mapped to the lower memory address space; selects a static memory (SRAM) device that stores the interrupt vector table, local stack, local data, and scratch pad data.

$\overline{\text{GCS7:0}}$ Mapped to memory or I/O address space; selects additional SRAM memory, DRAM memory, local peripherals, system bus, etc.

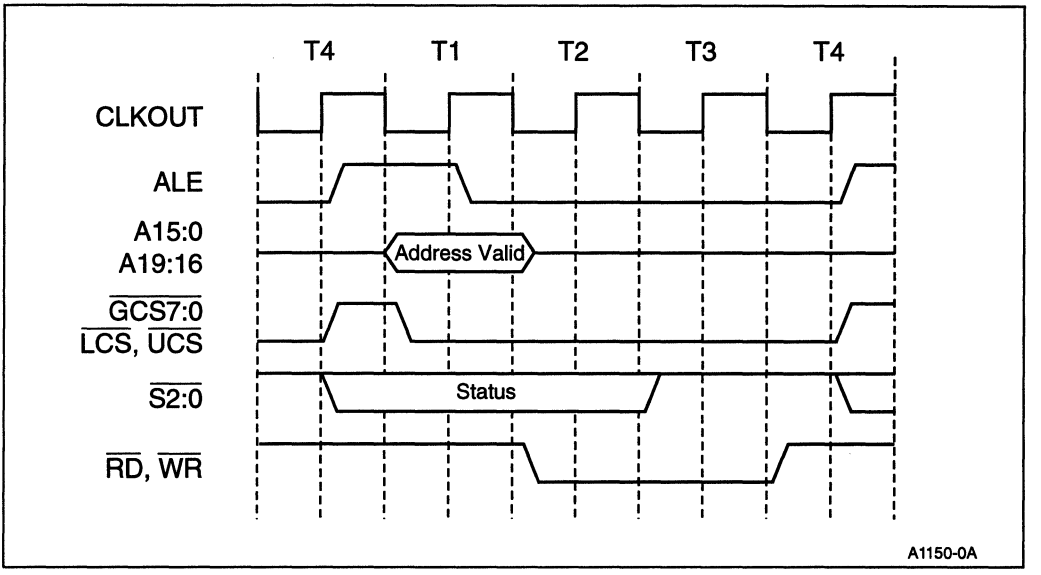


Figure 6-3. Chip-Select Relative Timings

A chip-select goes active when it meets all of the following criteria:

1. The chip-select is enabled.
2. The bus cycle status matches the programmed type (memory or I/O).
3. The bus cycle address is equal to or greater than the start address value.
4. The bus cycle address is less than the stop address value or the stop address is ignored.
5. The bus cycle is **not** accessing the Peripheral Control Block.

A memory address applies to memory read, memory write and instruction prefetch bus cycles. An I/O address applies to I/O read and I/O write bus cycles. Interrupt acknowledge and HALT bus cycles never activate a chip-select, regardless of the address generated.

After power-on or system reset, only the $\overline{\text{UCS}}$ chip-select is initialized and active (see Figure 6-4).

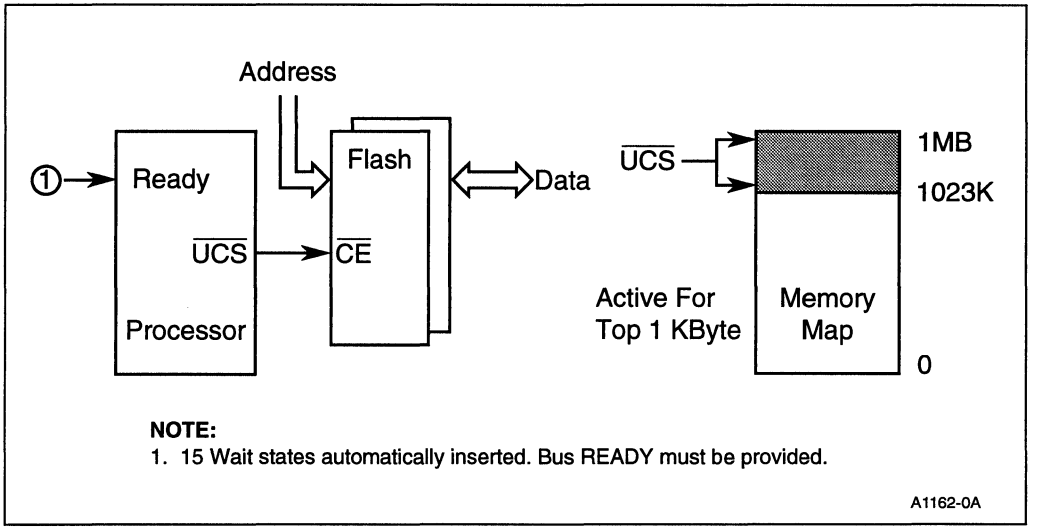


Figure 6-4. $\overline{\text{UCS}}$ Reset Configuration

6.4 PROGRAMMING

Two registers, START and STOP, determine the operating characteristics of each chip-select. The Peripheral Control Block defines the location of the Chip-Select Unit registers. Table 6-1 lists the registers and their associated programming names.

Table 6-1. Chip-Select Unit Registers

START Register Mnemonic	STOP Register Mnemonic	Chip-Select Affected
GCS0ST	GCS0SP	$\overline{\text{GCS0}}$
GCS1ST	GCS1SP	$\overline{\text{GCS1}}$
GCS2ST	GCS2SP	$\overline{\text{GCS2}}$
GCS3ST	GCS3SP	$\overline{\text{GCS3}}$
GCS4ST	GCS4SP	$\overline{\text{GCS4}}$
GCS5ST	GCS5SP	$\overline{\text{GCS5}}$
GCS6ST	GCS6SP	$\overline{\text{GCS6}}$
GCS7ST	GCS7SP	$\overline{\text{GCS7}}$
UCSST	UCSSP	$\overline{\text{UCS}}$
LCSST	LCSSP	$\overline{\text{LCS}}$

The START register (Figure 6-5) defines the starting address and the wait state requirements. The STOP register (Figure 6-6) defines the ending address and the bus ready, bus cycle and enable requirements.

6.4.1 Initialization Sequence

Chip-selects do not have to be initialized in any specific order. However, the following guidelines help prevent a system failure.

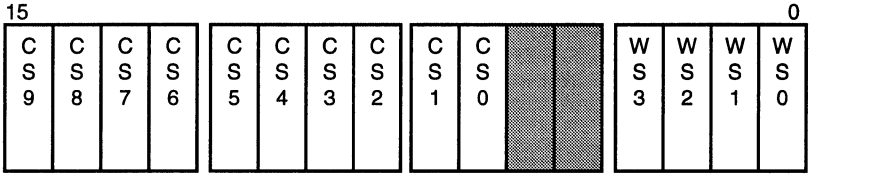
1. Initialize local memory chip-selects
2. Initialize local peripheral chip-selects
3. Perform local diagnostics
4. Initialize off-board memory and peripheral chip-selects
5. Complete system diagnostics

An unmasked interrupt or NMI must not occur until the interrupt vector addresses have been written to memory. Failure to prevent an interrupt from occurring during initialization will cause a system failure. Use external logic to generate the chip-select if interrupts cannot be masked prior to initialization.

Register Name: Chip-Select Start Register

Register Mnemonic: UCSST, LCSST, GCSxST (x=0-7)

Register Function: Defines chip-select start address and number of bus wait states.



A1163-0A

Bit Mnemonic	Bit Name	Reset State	Function
CS9:0	Start Address	3FFH	Defines the starting (base) address for the chip-select. CS9:0 are compared with the A19:10 (memory bus cycles) or A15:6 (I/O bus cycles) address bits. An equal to or greater than result enables the chip-select.
WS3:0	Wait State Value	0FH	WS3:0 define the minimum number of wait states inserted into the bus cycle. A zero value means no wait states. Additional wait states can be inserted into the bus cycle using bus ready.

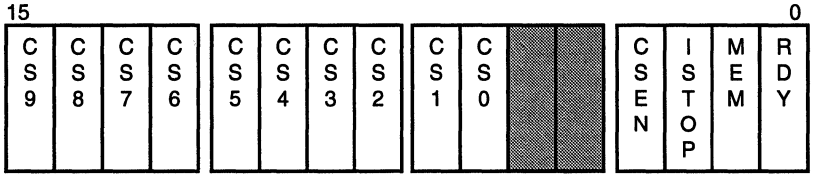
NOTE: Reserved register bits are shown with gray shading. Reserved bits must be written to a logic zero to ensure compatibility with future Intel products.

Figure 6-5. START Register Definition

Register Name: Chip-Select Stop Register

Register Mnemonic: UCSSP, LCSSP, GCSxSP (x=0-7)

Register Function: Defines chip-select stop address and other control functions.



A1164-0A

Bit Mnemonic	Bit Name	Reset State	Function
CS9:0	Stop Address	3FFH	Defines the ending address for the chip-select. CS9:0 are compared with the A19:10 (memory bus cycles) or A15:6 (I/O bus cycles) address bits. A less than result enables the chip-select. CS9:0 are ignored if ISTOP is set.
CSEN	Chip-Select Enable	0 (Note)	Disables the chip-select when cleared. Setting CSEN enables the chip-select.
ISTOP	Ignore Stop Address	0 (Note)	Setting this bit disables stop address checking, which automatically sets the ending address at 0FFFFFH (memory) or 0FFFFH (I/O). When ISTOP is cleared, the stop address requirements must be met to enable the chip-select.

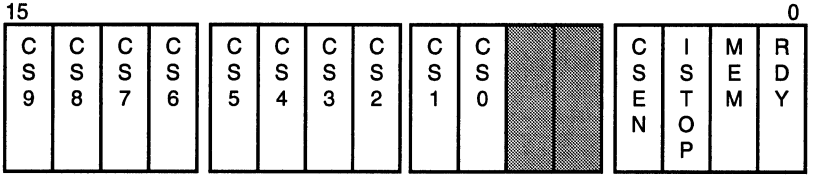
NOTE: Reserved register bits are shown with gray shading. Reserved bits must be written to a logic zero to ensure compatibility with future Intel products. The reset state of CSEN and ISTOP is '1' for the UCSSP register.

Figure 6-6. STOP Register Definition

Register Name: Chip-Select Stop Register

Register Mnemonic: UCSSP, LCSSP, GCSxSP (x=0-7)

Register Function: Defines chip-select stop address and other control functions.



A1164-0A

Bit Mnemonic	Bit Name	Reset State	Function
MEM	Bus Cycle Selector	1	When MEM is set, the chip-select goes active for memory bus cycles. Clearing MEM activates the chip-select for I/O bus cycles. MEM defines which address bits are used by the start and stop address comparators. When MEM is cleared, address bits A15:6 are routed to the comparators. When MEM is set, address bits A19:10 are routed to the comparators.
RDY	Bus Ready Enable	1	Setting RDY requires that bus ready be active to complete a bus cycle. Bus ready is ignored when RDY is cleared. RDY must be set to extend wait states beyond the number determined by WS3:0.

NOTE: Reserved register bits are shown with gray shading. Reserved bits must be written to a logic zero to ensure compatibility with future Intel products. The reset state of CSEN and ISTOP is '1' for the UCSSP register.

Figure 6-6. STOP Register Definition (Continued)

The correct sequence to program a non-enabled chip-select is as follows. (If the chip-select is already enabled, either reverse the sequence or disable the chip-select before reprogramming it.)

1. Program the START register
2. Program the STOP register

6.4.2 Start Address

The START register of each chip-select defines its starting (base) address. The start address value is compared to the ten most-significant address bits of the bus cycle. A bus cycle whose ten most-significant address bits are equal to or greater than the start address value causes the chip-select to go active. Table 6-2 defines the address bits that are compared with the start address value for memory and I/O bus cycles.

It is **not** possible to have a chip-select start on any arbitrary byte boundary. A chip-select configured for memory accesses can start only on multiples of 1 Kbyte. A chip-select configured for I/O accesses can start only on multiples of 64 bytes. The equations below calculate the physical start address for a given start address value.

For memory accesses:Start Value (Decimal) × 1024 = Physical Start Address (Decimal)

For I/O accesses:Start Value (Decimal) × 64= Physical Start Address (Decimal)

Table 6-2. Memory and I/O Compare Addresses

Address Space	Address Range	Number of Bits	Comparator Input	Resolution
Memory	1 Mbyte	20	A19:A10	1 Kbyte
I/O	64 Kbyte	16	A15:A6	64 Bytes

6.4.3 Stop Address

The STOP register of each chip-select defines its ending address. The stop address value is compared to the ten most-significant address bits of the bus cycle. A bus cycle whose ten most-significant bits of address are less than the stop address value causes the chip-select to go active. Table 6-2 defines the address bits that are compared with the stop address value for memory and I/O bus cycles.

It is **not** possible to have a chip-select end on any arbitrary byte boundary. A chip-select configured for memory accesses can end only on multiples of 1 Kbyte. A chip-select configured for I/O accesses can end only on multiples of 64 bytes. The equations below define the ending address for the chip-select.

For memory accesses:(Stop Value (Decimal) × 1024) – 1= Physical Ending Address (Decimal)

For I/O accesses:(Stop Value (Decimal) × 64) – 1= Physical Ending Address (Decimal)

In the previous equations, a stop value of 1023 (03FFH) results in a physical ending address of 0FFBFFH (memory) or 0FFBFH (I/O). These addresses do **not** represent the top of the memory or I/O address space. To have a chip-select enabled to the end of the physical address space, the ISTOP control bit must be set. The ISTOP control bit overrides the stop address comparator output (see Figure 6-2 on page 6-3).

6.4.4 Enabling and Disabling Chip-Selects

The ability to enable or disable a chip-select is important when multiple memory devices share (or can share) the same physical address space. Examples of where two or more devices would occupy the same address space include shadowed memory, bank switching and paging.

The STOP register holds the CSEN control bit, which determines whether the chip-select should go active. A chip-select never goes active if its CSEN control bit is cleared.

Chip-selects can be disabled by programming the stop address value less than the start address value or by programming the start address value greater than the stop address value. However, the ISTOP control bit cannot be set when chip-selects are disabled in this manner.

6.4.5 Bus Wait State and Ready Control

Normally, the bus ready input must be inactive at the appropriate time to insert wait states into the bus cycle. The Chip-Select Unit can ignore the state of the bus ready input to extend and complete the bus cycle automatically. Most memory and peripheral devices operate properly using fifteen or fewer wait states. However, accessing such devices as a dual-port memory, an expansion bus interface, a system bus interface or remote peripheral devices can require more than fifteen wait states to complete a bus cycle.

The START register holds a four-bit value (WS3:0) that defines the number of wait states to insert into the bus cycle. Figure 6-7 shows a simplified logic diagram of the wait state and ready control functions.

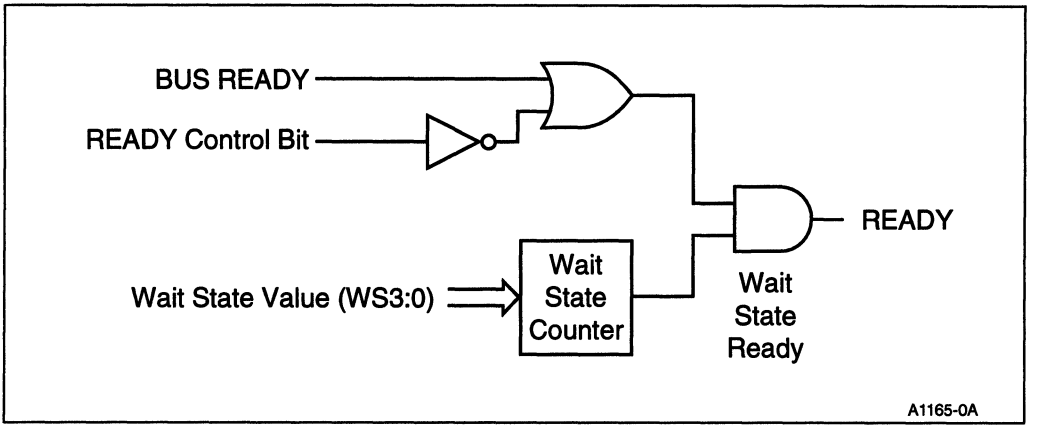


Figure 6-7. Wait State and Ready Control Functions

The STOP register defines the RDY control bit to extend bus cycles beyond fifteen wait states. The RDY control bit determines whether the bus cycle should complete normally (i.e., require bus ready) or unconditionally (i.e., ignore bus ready). Chip-selects connected to devices requiring fifteen wait states or fewer can program RDY inactive to automatically complete the bus cycle. Devices that may require more than fifteen wait states must program RDY active.

A bus cycle with wait states automatically inserted cannot be shortened. A bus cycle that ignores bus ready cannot be lengthened.

6.4.6 Overlapping Chip-Selects

The Chip-Select Unit activates all enabled chip-selects programmed to cover the same physical address space. This is true if any portion of the chip-selects' address ranges overlap (i.e., chip-selects' ranges do not need to overlap completely to all go active). There are various reasons for overlapping chip-selects. For example, a system might have a need for overlapping a portion of read-only memory with read/write memory or copying data to two devices simultaneously.

If overlapping chip-selects do not have identical wait state and bus ready programming, the Chip-Select Unit will adjust itself based on the criteria shown in Figure 6-8.

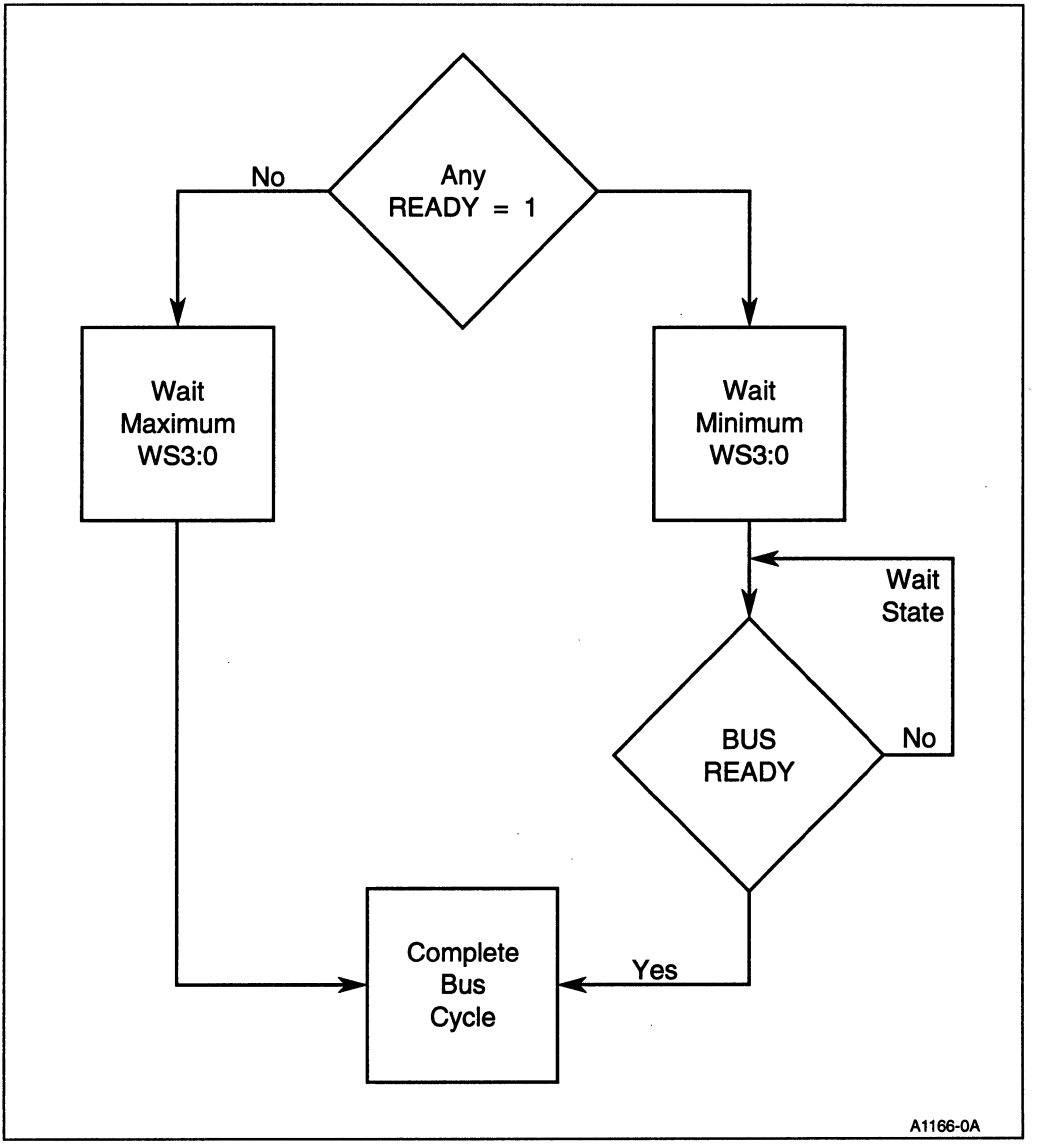


Figure 6-8. Overlapping Chip-Selects

Table 6-3 lists example wait state and bus ready requirements for overlapping chip-selects and the resulting requirements for accesses to the overlapped region.

Table 6-3. Example Adjustments for Overlapping Chip-Selects

Chip-Select X		Chip-Select Y		Overlapped Region Access	
Wait States	**Bus Ready**	**Wait States**	**Bus Ready**	**Wait States**	**Bus Ready**
3	ignored	9	ignored	9	ignored
5	required	0	ignored	0	required
2	required	2	required	2	required

Be cautious when overlapping chip-selects with different wait state or bus ready programming. The following two conditions require special attention to ensure proper system operation:

1. When all overlapping chip-selects ignore bus ready but have different wait states, verify that each chip-select still works properly using the highest wait state value. A system failure may result when too few or too many wait states occur in the bus cycle.
2. If one or more of the overlapping chip-selects requires bus ready, verify that all chip-selects that **ignore** bus ready still work properly using both the smallest wait state value and the longest possible bus cycle. A system failure may result when too few or too many wait states occur in the bus cycle.

6.4.7 Memory or I/O Bus Cycle Decoding

The Chip-Select Unit decodes bus cycle status and address information to determine whether a chip-select goes active. The MEM control bit in the STOP register defines whether memory or I/O address space is decoded. Memory address space accesses consist of memory read, memory write and instruction prefetch bus cycles. I/O address space accesses consist of I/O read and I/O write bus cycles.

Chip-selects go active for bus cycles initiated by the CPU and Refresh Control Unit.

6.4.8 Programming Considerations

When programming chip-selects active for I/O bus cycles, remember that eight bytes of I/O are reserved by Intel. These eight bytes (locations 00F8H through 00FFH) control the interface to an 80C187 math coprocessor. A chip-select can overlap this reserved space provided there is no intention of using the 80C187. However, to avoid possible future compatibility issues, Intel recommends that **no** chip-select start at I/O address location 00C0H.

The $\overline{GCS}$ chip-select outputs are multiplexed with output port functions. The register that controls the multiplexed outputs resides in the I/O Port Unit. (See Table 11-1 on page 11-7 and Figure 11-5 on page 11-8.)

6.5 CHIP-SELECTS AND BUS HOLD

The Chip-Select Unit decodes only internally generated address and bus state information. An external bus master cannot make use of the Chip-Select Unit. During HLDA, all chip-selects remain inactive.

The circuit shown in Figure 6-9 allows an external bus master to access a device during bus HOLD.

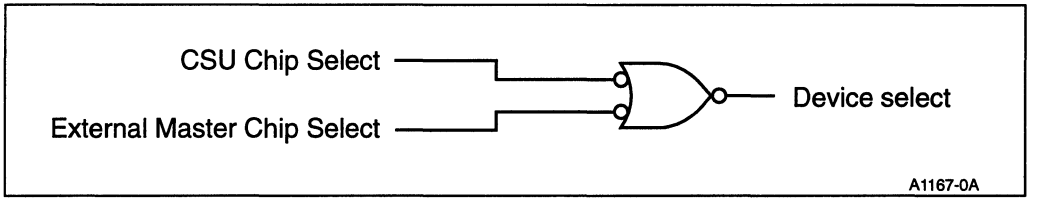


Figure 6-9. Using Chip-Selects During HOLD

6.6 EXAMPLES

The following sections provide examples of programming the Chip-Select Unit to meet the needs of a particular application. The examples do not go into hardware analysis or design issues.

6.6.1 Example 1: Typical System Configuration

Figure 6-10 illustrates a block diagram of a typical system design with a 128 Kbyte EPROM and a 32 Kbyte SRAM. The peripherals are mapped to I/O address space. Example 6.1 shows a program template for initializing the Chip-Select Unit.

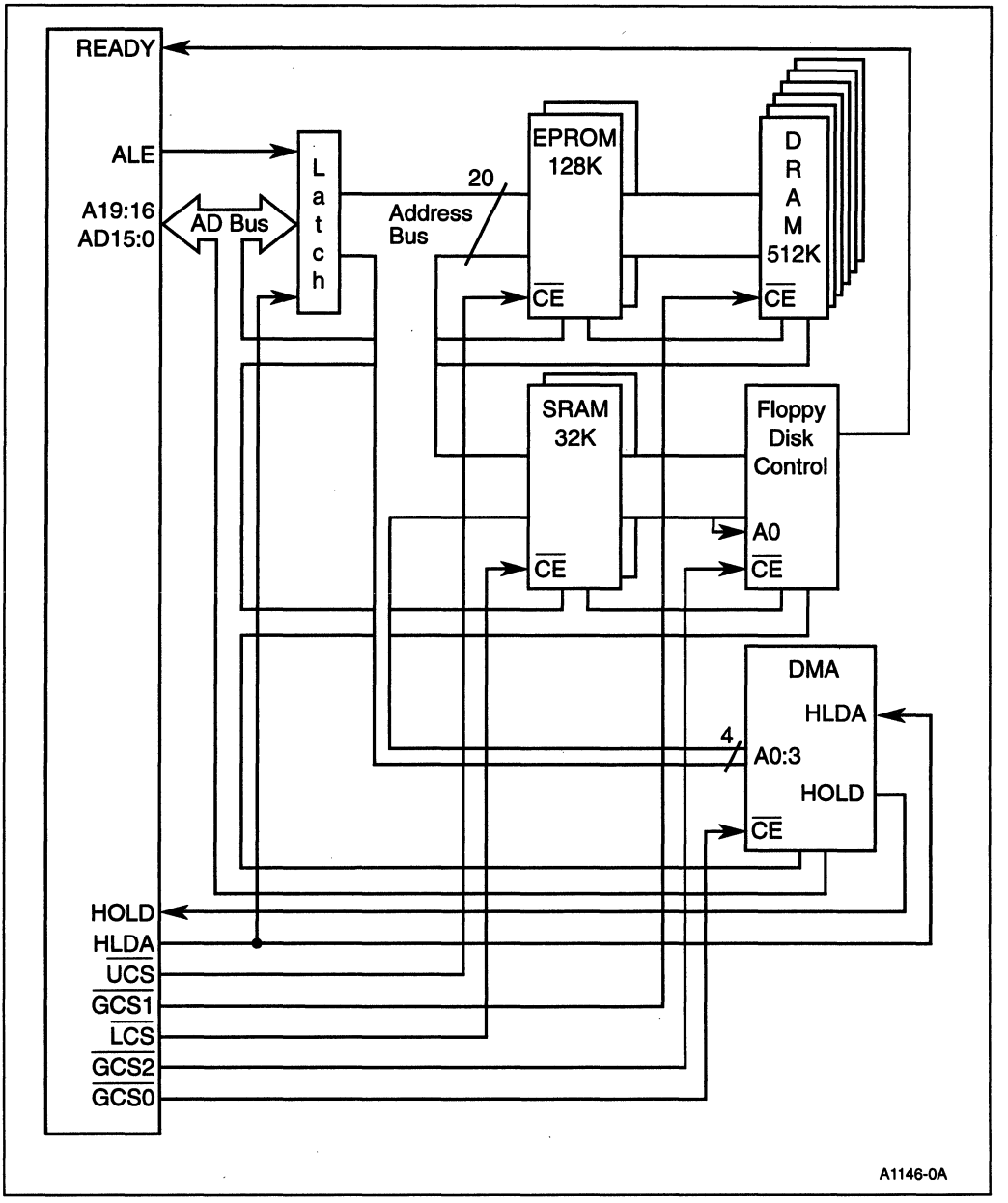


Figure 6-10. Typical System

```
$           TITLE           (Chip-Select Unit Initialization)
$           MOD186XREF
            NAME            CSU_EXAMPLE_1

; External reference from this module

$           include(PCBMAP.INC)             ;File declares Register
                                            ;Locations and names.

; Module equates

; Configuration equates

            TRUE  EQU       0FFH
            FALSE EQU       NOT TRUE
            READY EQU       0001H           ;Bus ready control modifier
            CSEN  EQU       0008H           ;Chip-Select enable modifier
            ISTOP EQU       0004H           ;Stop address modifier
            MEM   EQU       0002H           ;Memory select modifier
            IO    EQU       0000H           ;I/O select modifier

;Below is a list of the default system memory and I/O environment. These
;defaults configure the Chip-Select Unit for proper system operation.

;EPROM memory is located from 0E0000 to 0FFFFF (128 Kbytes).
;Wait states are calculated assuming 16MHz operation.
;UCS# controls the accesses to EPROM memory space.

            EPROM_SIZEEQU  128              ;Size in Kbytes
            EPROM_BASEEQU  1024 - EPROM_SIZE;Start address in Kbytes
            EPROM_WAITEQU  1                ;Wait states

;The UCS# START and STOP register values are calculated using the above system
;constraints and the equations below.

            UCSST_VALEQU   (EPROM_BASE SHL 6) OR (EPROM_WAIT)
            UCSSP_VALEQU   (CSEN) OR (ISTOP) OR (MEM)

;SRAM memory starts at 0H and continues to 7FFFH (32 Kbytes).
;Wait states are calculated assuming 16MHz operation.
;LCS# controls the accesses to SRAM memory space.

            SRAM_SIZEEQU   32               ;Size in Kbytes
            SRAM_BASEEQU   0                ;Start address in Kbytes
            SRAM_WAITEQU   0                ;Wait states

;The LCS# START and STOP register values are calculated using the above system
;constraints and the equations below

            LCSST_VALEQU   (SRAM_BASE SHL 6) OR (SRAM_WAIT)
            LCSSP_VALEQU   (((SRAM_BASE) OR (SRAM_SIZE)) SHL 6) OR
&                          (CSEN) OR (MEM)

;A DRAM interface is selected by the GCS1# chip-select. The BASE value defines
;the starting address of the DRAM window.The SIZE value (along with the BASE
;value) defines the ending address. Zero wait state performance is assumed. The
;Refresh Control Unit uses DRAM_BASE to properly configure refresh operation.
```

Example 6-1. Initializing the Chip-Select Unit

```
            DRAM_BASEEQU    128             ;Window start address in Kbytes
            DRAM_SIZEEQU    512             ;Window size in Kbytes
            DRAM_WAITEQU    0               ;Wait states (change to match
                                            ;system)

;The GCS1# START and STOP register values are calculated using the above system
;constraints and the equations below

            GCS1ST_VALEQU   (DRAM_BASE SHL 6) OR (DRAM_WAIT)
            GCS1SP_VALEQU   (((DRAM_BASE) OR (DRAM_SIZE)) SHL 6) OR
&                           (CSEN) OR (MEM)

;I/O is selected using the GCS2# chip-select.  Wait states assume operation at
;16MHz.  The SIZE and BASE values must be modulo 64 bytes. For this example, the
;Floppy Disk Controller is connected to GCS2# and GCS0# is connected to
;the DMA Controller.

            IO_SIZEEQU      64              ;Size in bytes
            IO_BASEEQU      256             ;Start address in bytes
            IO_WAITEQU      4               ;Wait states

            DMA_BASEEQU     512             ;Start address in bytes
            DMA_WAITEQU     0               ;Wait states
                                            ;Size assumed to be 64 bytes

;The GCS0# and GCS2# START and STOP register values are calculated using the
;above system contraints and the equations below.

            GCS2ST_VALEQU   ((IO_BASE/64) SHL 6) OR (IO_WAIT)
            GCS2SP_VALEQU   (((IO_BASE/64) OR (IO_SIZE/64)) SHL 6) OR
&                           (CSEN) OR (IO)

            GCS0ST_VALEQU   ((DMA_BASE/64) SHL 6) OR (DACK_WAIT)
            GCS0SP_VALEQU   (((DMA_BASE/64) + 1) SHL 6) OR (CSEN) OR (IO)

;The following statements define the default assumptions for SEGMENT locations.

            ASSUMECS:CODE
            ASSUMEDS:DATA
            ASSUMESS:DATA
            ASSUMEES:DATA

            CODE  SEGMENT PUBLIC  'CODE'

;ENTRY POINT ON POWER UP:
;The power-on or reset code does a jump here after the UCS register is
;programmed.

            FW_STARTLABEL  FAR              ;Forces far jump

            CLI                             ;Make sure interrupts are
                                            ;globally disabled

;Place register initialization code here
```

Example 6-1. Initializing the Chip-Select Unit (Continued)

```
;SET UP CHIP SELECTS

;  UCS#  - EPROM  Select
;  LCS#  - SRAM   Select
;  GCS1# - DRAM   Select
;  GCS2# - FLOPPY Select
;  GCS0# - DMA    Select

            MOV   DX, UCSSP                 ;Finish setting up UCS#
            MOV   AX, UCSSP_VAL
            OUT   DX, AL                    ;Remember, byte writes work ok

            MOV   DX, LCSST                 ;Set up LCS#
            MOV   AX, LCSST_VAL
            OUT   DX, AL
            MOV   DX, LCSSP
            MOV   AX, LCSSP_VAL
            OUT   DX, AL                    ;Remember, byte writes work ok

            MOV   DX, GCS1ST                ;Set up GCS1#
            MOV   AX, GCS1ST_VAL
            OUT   DX, AL
            MOV   AX, GCS1SP_VAL
            MOV   DX, GCS1SP
            OUT   DX, AL                    ;Remember, byte writes work ok

            MOV   DX, GCS0ST                ;Set up GCS0#
            MOV   AX, GCS0ST_VAL
            OUT   DX, AL
            MOV   AX, GCS0SP_VAL
            MOV   DX, GCS0SP
            OUT   DX, AL                    ;Remember, byte writes work ok

            MOV   DX, GCS2ST                ;Set up GCS2#
            MOV   AX, GCS2ST_VAL
            OUT   DX, AL
            MOV   DX, GCS2SP
            MOV   AX, GCS2SP_VAL
            OUT   DX, AL                    ;Remember, byte writes work ok

;Place remaining User Code here.

            CODE  ENDS

;POWER ON RESET CODE TO GET STARTED

            ASSUME CS:POWER_ON

            POWER_ONSEGMENT AT 0FFFFH

            MOV   DX, UCSST                 ;Point to UCS register
            MOV   AX, UCSST_VAL             ;Reprogram UCS# for EPROM size
            OUT   DX, AL
            JMP   FW_START                  ;Jump to start of init code

            POWER_ON ENDS
```

Example 6-1. Initializing the Chip-Select Unit (Continued)

```
; DATA SEGMENT
            DATA  SEGMENT PUBLIC  'DATA'
            DD    256 DUP (?)               ;Reserved for Interrupt Vectors
;Place additional memory variable here
            DW    500 DUP (?)               ;Stack allocation
STACK_TOP   LABEL   WORD
DATA        ENDS
;Program Ends
            END
```

Example 6-1. Initializing the Chip-Select Unit (Continued)

6.6.2 Example 2: Detecting Attempts to Access Guarded Memory

A chip-select is configured to set an interrupt when the bus accesses a physical address region that does not contain a valid memory or peripheral device. Figure 6-11 illustrates how a simple circuit detects the errant bus cycle and generates an NMI. System software then deals with the error. The purpose of using the chip-select is to generate a bus ready and prevent a bus "hang" condition.

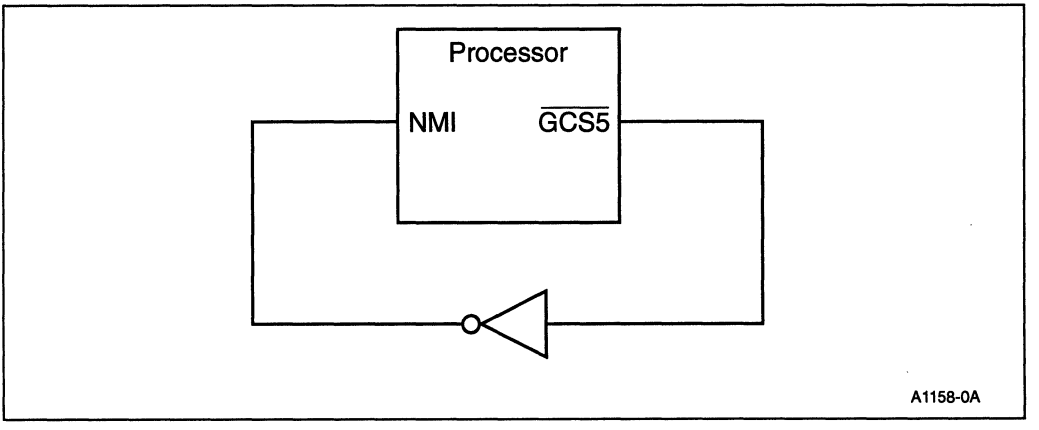


Figure 6-11. Guarded Memory Detector

intel®

7

Refresh Control Unit

CHAPTER 7
REFRESH CONTROL UNIT

The Refresh Control Unit (RCU) simplifies dynamic memory controller design with its integrated address and clock counters. Figure 7-1 shows the relationship between the Bus Interface Unit and the Refresh Control Unit. Integrating the Refresh Control Unit into the processor allows an external DRAM controller to use chip-selects, wait state logic and status lines.

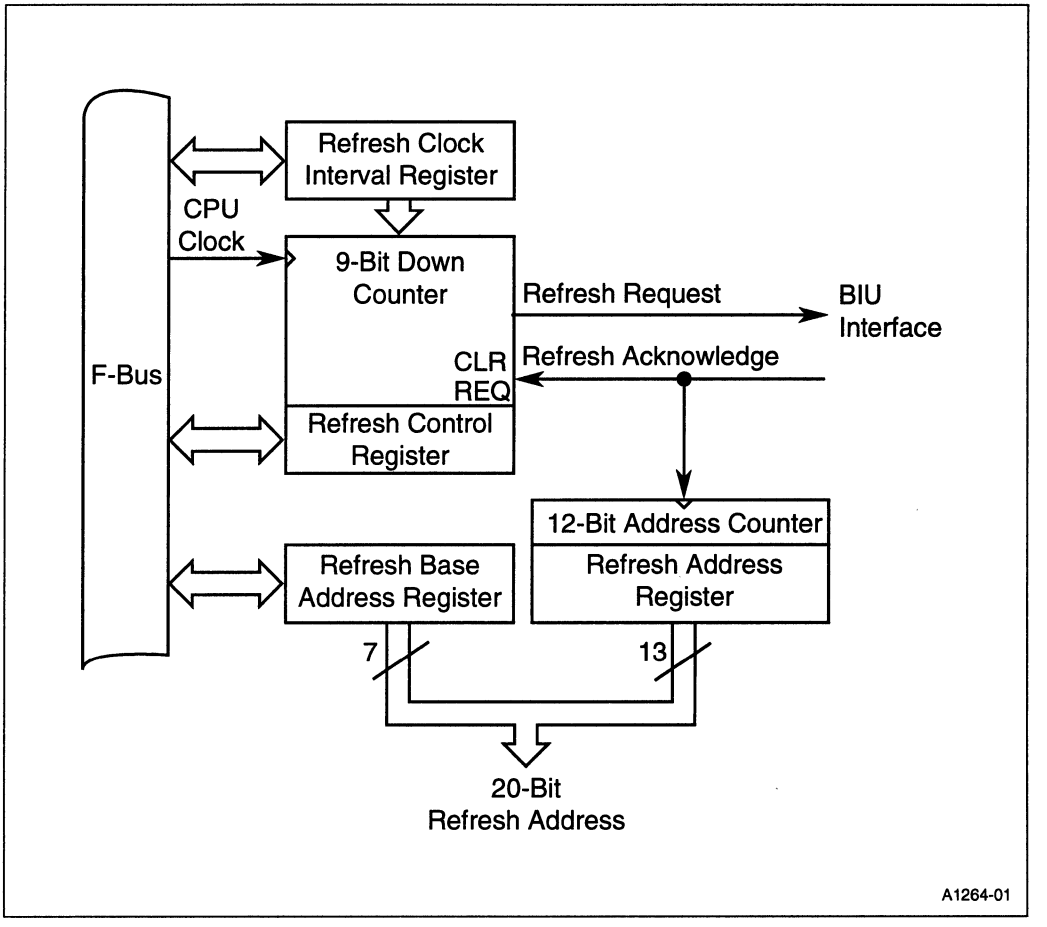


Figure 7-1. Refresh Control Unit Block Diagram

7.1 THE ROLE OF THE REFRESH CONTROL UNIT

Like a DMA controller, the Refresh Control Unit runs bus cycles independent of CPU execution. Unlike a DMA controller, however, the Refresh Control Unit does not run bus cycle bursts nor does it transfer data. The DRAM refresh process freshens individual DRAM rows in "dummy read" cycles, while cycling through all necessary addresses.

The microprocessor interface to DRAMs is more complicated than other memory interfaces. A complete DRAM controller requires circuitry beyond that provided by the processor even in the simplest configurations. This circuitry must respond correctly to reads, writes and DRAM refresh cycles. The external DRAM controller generates the Row Address Strobe ($\overline{RAS}$), Column Address Strobe ($\overline{CAS}$) and other DRAM control signals.

Pseudo-static RAMs use dynamic memory cells but generate address strobes and refresh addresses internally. The address counters still need external timing pulses. These pulses are easy to derive from the processor's bus control signals. Pseudo-static RAMs do not need a full DRAM controller.

7.2 REFRESH CONTROL UNIT CAPABILITIES

A 12-bit address counter forms the refresh addresses, supporting any dynamic memory devices with up to 12 rows of memory cells (12 refresh address bits). This includes all practical DRAM sizes for the processor's 1 Mbyte address space.

7.3 REFRESH CONTROL UNIT OPERATION

Figure 7-2 illustrates Refresh Control Unit counting, address generation and BIU bus cycle generation in flowchart form.

The nine-bit down-counter loads from the Refresh Interval Register on the falling edge of CLKOUT. Once loaded, it decrements every falling CLKOUT edge until it reaches one. Then the down-counter reloads and starts counting again, simultaneously triggering a refresh request. Once enabled, the DRAM refresh process continues indefinitely until the user reprograms the Refresh Control Unit, a reset occurs, or the processor enters Powerdown mode.

The refresh request remains active until the bus becomes available. When the bus is free, the BIU will run its "dummy read" cycle. Refresh bus requests have higher priority than most CPU bus cycles, all DMA bus cycles and all interrupt vectoring sequences. Refresh bus cycles also have a higher priority than the HOLD/$\overline{HLDA}$ bus arbitration protocol (see "Refresh Operation and Bus HOLD" on page 7-13).

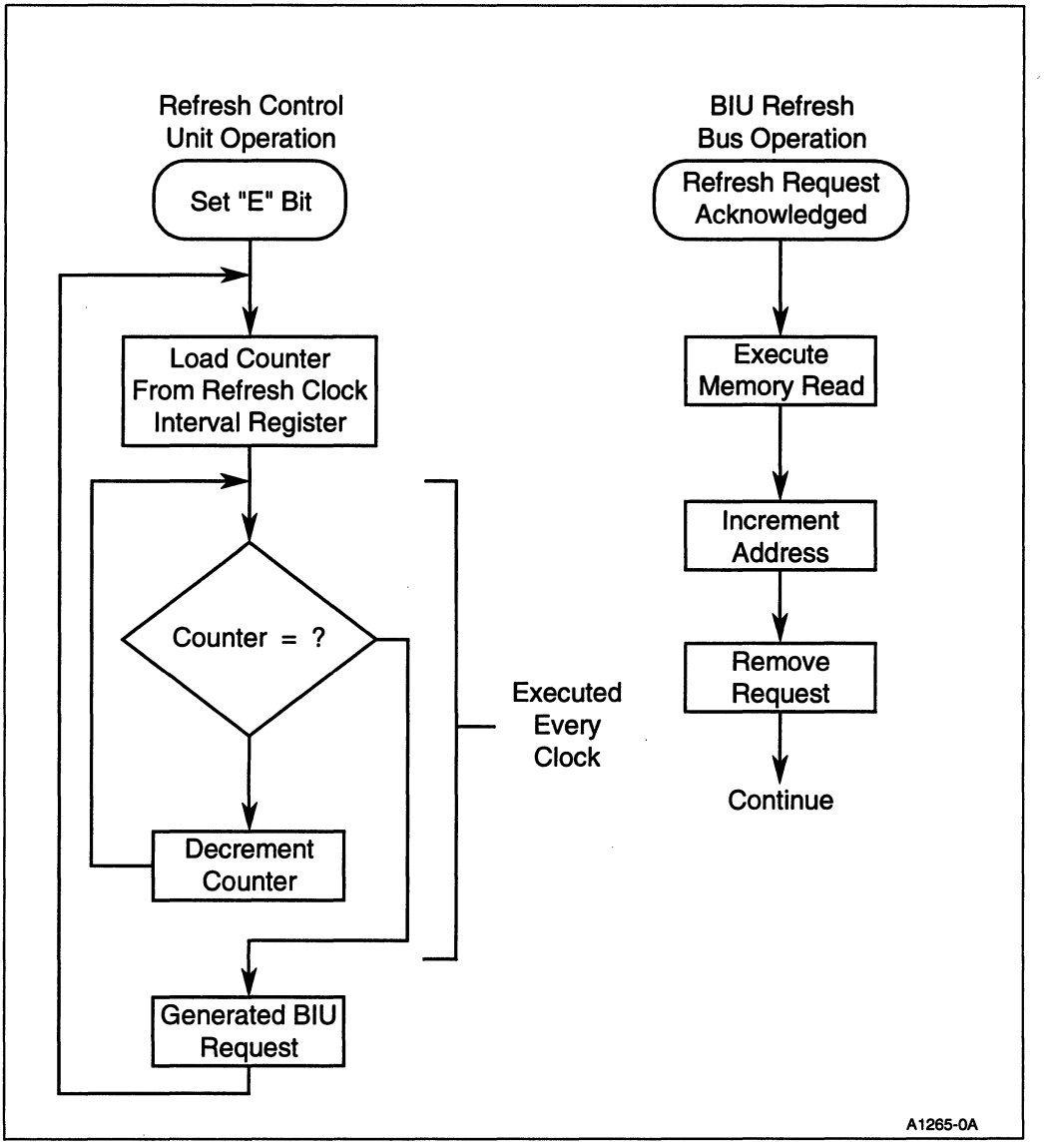


Figure 7-2. Refresh Control Unit Operation Flow Chart

The nine-bit refresh clock counter does not wait until the BIU services the refresh request to continue counting. This operation ensures that refresh requests occur at the correct interval. Otherwise, the time between refresh requests would be a function of varying bus activity. When the BIU services the refresh request, it clears the request and increments the refresh address.

The BIU does not queue DRAM refresh requests. If the Refresh Control Unit generates another request before the BIU handles the present request, the BIU loses the present request. However, the address associated with the request is not lost. The refresh address changes only after the BIU runs a refresh bus cycle. If a DRAM refresh cycle is excessively delayed, there is still a chance that the processor will successfully refresh the corresponding row of cells in the DRAM, retaining the data.

7.4 REFRESH ADDRESSES

Figure 7-3 shows the physical address generated during a refresh bus cycle. This figure applies to both the 8-bit and 16-bit data bus microprocessor versions. Refresh address bits RA19:13 come from the Refresh Base Address Register. (See "Refresh Base Address Register" on page 7-7.)

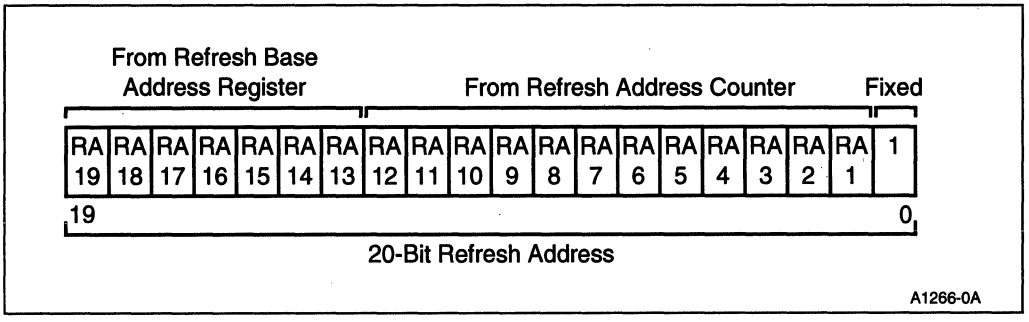


Figure 7-3. Refresh Address Formation

A linear-feedback shift counter generates address bits RA12:1 and RA0 is always one. The counter does not count linearly from 0 through FFFH. However, the counting algorithm cycles uniquely through all possible 12-bit values. It matters only that each row of DRAM memory cells is refreshed at a specific interval. The order of the rows is unimportant.

Address bit A0 is fixed at one during all refresh operations. In applications based on a 16-bit data bus processor, A0 typically selects memory devices placed on the low (even) half of the bus. Applications based on an 8-bit data bus processor typically use A0 as a true address bit. The DRAM controller must **not** route A0 to row address pins on the DRAMs.

7.5 REFRESH BUS CYCLES

Refresh bus cycles look exactly like ordinary memory read bus cycles except for the control signals listed in Table 7-1. These signals can be ANDed in a DRAM controller to detect a refresh bus cycle. The 16-bit bus processor drives both the $\overline{BHE}$ and A0 pins high during refresh cycles. The 8-bit bus version replaces the $\overline{BHE}$ pin with $\overline{RFSH}$, which has the same timings. The 8-bit bus processor drives $\overline{RFSH}$ low and A0 high during refresh cycles.

Table 7-1. Identification of Refresh Bus Cycles

Data Bus Width	$\overline{BHE}/\overline{RFSH}$	A0
16-Bit Device	1	1
8-Bit Device	0	1

7.6 GUIDELINES FOR DESIGNING DRAM CONTROLLERS

The basic DRAM access method consists of four phases:

1. The DRAM controller supplies a row address to the DRAMs.
2. The DRAM controller asserts a Row Address Strobe ($\overline{RAS}$), which latches the row address inside the DRAMs.
3. The DRAM controller supplies a column address to the DRAMs.
4. The DRAM controller asserts a Column Address Strobe ($\overline{CAS}$), which latches the column address inside the DRAMs.

Most 80C186 Modular Core family DRAM interfaces use only this method. Others are not discussed here.

The DRAM controller's purpose is to use the processor's address, status and control lines to generate the multiplexed addresses and strobes. These signals must be appropriate for three bus cycle types: read, write and refresh. They must also meet specific pulse width, setup and hold timing requirements. DRAM interface designs need special attention to transmission line effects, since DRAMs represent significant loads on the bus.

DRAM controllers may be either clocked or unclocked. An unclocked DRAM controller requires a tapped digital delay line to derive the proper timings.

Clocked DRAM controllers may use either discrete or programmable logic devices. A state machine design is appropriate, especially if the circuit must provide wait state control (beyond that possible with the processor's Chip-Select Unit). Because of the microprocessor's four-clock bus, clocking some logic elements on each CLKOUT phase is advantageous (see Figure 7-4).

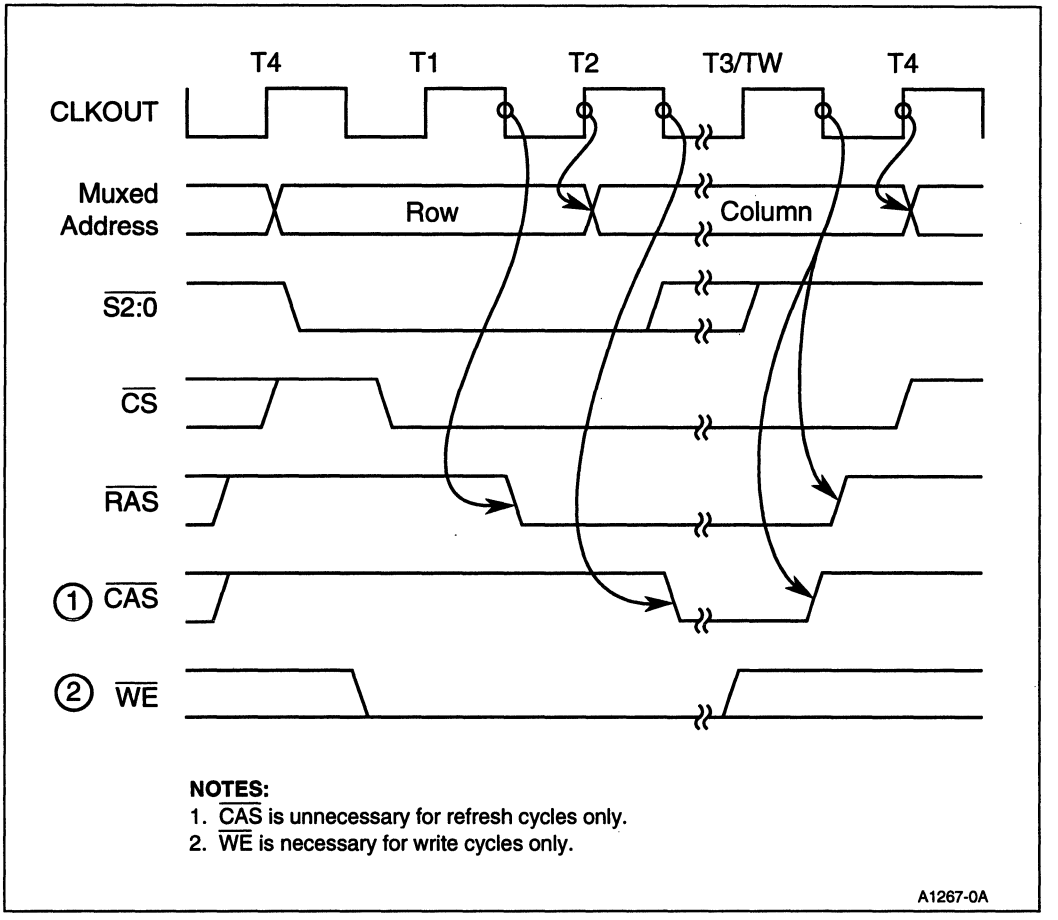


Figure 7-4. Suggested DRAM Control Signal Timing Relationships

The cycle begins with presentation of the row address. $\overline{RAS}$ should go active on the falling edge of T2. At the rising edge of T2, the address lines should switch to a column address. $\overline{CAS}$ goes active on the falling edge of T3. Refresh cycles do not require $\overline{CAS}$. When $\overline{CAS}$ is present, the "dummy read" cycle becomes a true read cycle (the DRAM drives the bus), and the DRAM row still gets refreshed.

Both $\overline{RAS}$ and $\overline{CAS}$ stay active during any wait states. They go inactive on the falling edge of T4. At the rising edge of T4, the address multiplexer shifts to its original selection (row addressing), preparing for the next DRAM access.

7.7 PROGRAMMING THE REFRESH CONTROL UNIT

Given a specific processor operating frequency and information about the DRAMs in the system, the user can program the Refresh Control Unit registers.

7.7.1 Calculating the Refresh Interval

DRAM data sheets show DRAM refresh requirements as a number of refresh cycles necessary and the maximum period to run the cycles. (The number of refresh cycles is the same as the number of rows.) You must compensate for bus latency — the time it takes for the Refresh Control Unit to gain control of the bus. This is typically 1–5%, but if an external bus master will be extremely slow to release the bus, increase the overhead percentage. At standard operating frequencies, DRAM refresh bus overhead totals 2–3% of the total bus bandwidth.

Given this information and the CPU operating frequency, use the formula in Figure 7-5 to determine the correct value for the RFTIME Register value.

$$\frac{R_{PERIOD} \times F_{CPU}}{Rows + (Rows \times Overhead\%)} = RFTIME\ RegisterValue$$

R_{PERIOD} = Maximum refresh period specified by DRAM manufacturer (in µs).
F_{CPU} = Operating frequency (in MHz).
Rows = Total number of rows to be refreshed.
Overhead % = Derating factor to compensate for missed refresh requests (typically 1 – 5 %).

Figure 7-5. Formula for Calculating Refresh Interval for RFTIME Register

7.7.2 Refresh Control Unit Registers

Three contiguous Peripheral Control Block registers operate the Refresh Control Unit: the Refresh Base Address Register, Refresh Clock Interval Register and the Refresh Control Register. A fourth register, the Refresh Address Register, permits examination of the refresh address bits generated by the Refresh Control Unit.

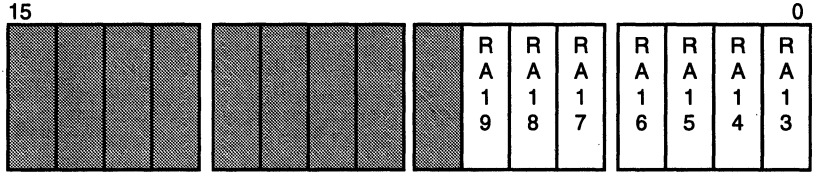
7.7.2.1 Refresh Base Address Register

The Refresh Base Address Register (Figure 7-6) programs the base (upper seven bits) of the refresh address. Seven-bit mapping places the refresh address at any 4 Kbyte boundary within the 1 Mbyte address space. When the partial refresh address from the 12-bit address counter (see Figure 7-1 and "Refresh Control Unit Capabilities" on page 7-2) passes FFFH, the Refresh Control Unit does **not** increment the refresh base address. Setting the base address ensures that the address driven during a refresh bus cycle activates the DRAM chip select.

Register Name: Refresh Base Address Register

Register Mnemonic: RFBASE

Register Function: Determines upper 7 bits of refresh address.



Bit Mnemonic	Bit Name	Reset State	Function
RA19:13	Refresh Base	00H	Uppermost address bits for DRAM refresh cycles.

NOTE: Reserved register bits are shown with gray shading. Reserved bits must be written to a logic zero to ensure compatibility with future Intel products.

Figure 7-6. Refresh Base Address Register

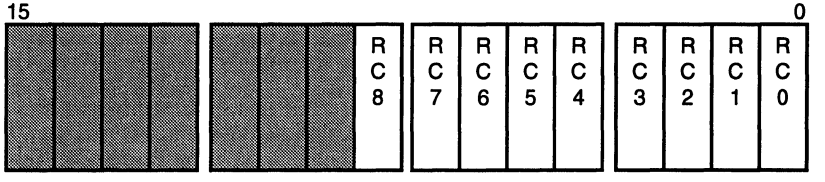
7.7.2.2 Refresh Clock Interval Register

The Refresh Clock Interval Register (Figure 7-7) defines the time between refresh requests. The higher the value, the longer the time between requests. The down-counter decrements every falling CLKOUT edge, regardless of core activity. When the counter reaches one, the Refresh Control Unit generates a refresh request, and the counter reloads the value from the register.

Register Name: Refresh Clock Interval Register

Register Mnemonic: RFTIME

Register Function: Sets refresh rate.



A1288-0A

Bit Mnemonic	Bit Name	Reset State	Function
RC8:0	Refresh Counter Reload Value	000H	Sets the desired clock count between refresh cycles.

NOTE: Reserved register bits are shown with gray shading. Reserved bits must be written to a logic zero to ensure compatibility with future Intel products.

Figure 7-7. Refresh Clock Interval Register

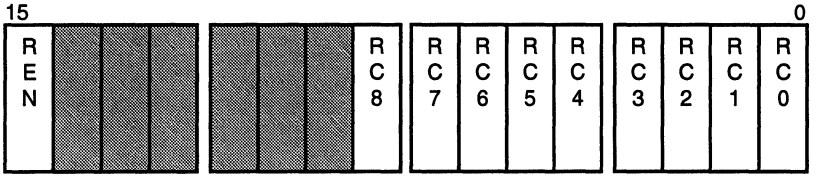
7.7.2.3 Refresh Control Register

Figure 7-8 shows the Refresh Control Register. The user may read or write the REN bit at any time to turn the Refresh Control Unit on or off. The lower nine bits contain the current nine-bit down-counter value. **The user cannot program these bits.** Disabling the Refresh Control Unit clears both the counter and the corresponding counter bits in the control register.

Register Name: Refresh Control Register

Register Mnemonic: RFCON

Register Function: Controls Refresh Unit operation.



A1311-0A

Bit Mnemonic	Bit Name	Reset State	Function
REN	Refresh Control Unit Enable	0	Setting REN enables the Refresh Unit. Clearing REN disables the Refresh Unit.
RC8:0	Refresh Counter	000H	These bits contain the present value of the down-counter that triggers refresh requests. **The user cannot program these bits.**

NOTE: Reserved register bits are shown with gray shading. Reserved bits must be written to a logic zero to ensure compatibility with future Intel products.

Figure 7-8. Refresh Control Register

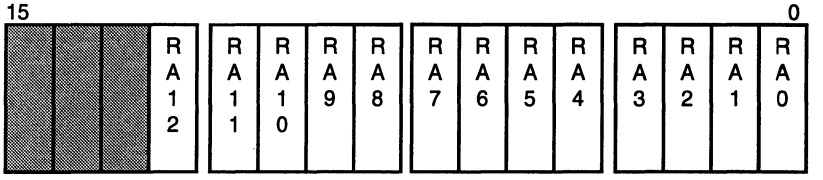
7.7.2.4 Refresh Address Register

The Refresh Address Register (Figure 7-9) contains address bits RA12:1, which will appear on the bus as A12:1 on the next refresh bus cycle. Bit 0 is fixed as a one in the register and in all refresh addresses.

Register Name: Refresh Address Register

Register Mnemonic: RFADDR

Register Function: Contains the generated refresh address bits.



A1501-0A

Bit Mnemonic	Bit Name	Reset State	Function
RA12:1	Refresh Address Bits	000H	These bits comprise A12:1 of the refresh address.
RA0	Refresh Bit 0	1	A0 of the refresh address. This bit is always 1 and is read-only.

NOTE: Reserved register bits are shown with gray shading. Reserved bits must be written to a logic zero to ensure compatibility with future Intel products.

Figure 7-9. Refresh Address Register

7.7.3 Programming Example

Example 7-1 contains sample code to initialize the Refresh Control Unit.

```
$mod186
name            example_80C186_RCU_code

; FUNCTION:  This function initializes the DRAM Refresh
; Control Unit to refresh the DRAM starting at dram_addr
; at clock_time intervals.

; SYNTAX:
; extern void far config_rcu(int dram_addr, int clock_time);

; INPUTS:       dram_addr  - Base address of DRAM to refresh
;               clock_time - DRAM refresh rate

; OUTPUTS:      None

;               NOTE:  Parameters are passed on the stack as
;               required by high-level languages.

RFBASE          equ  xxxxh                  ;substitute register offset
RFTIME          equ  xxxxh
RFCON           equ  xxxxh
Enable          equ  8000h                  ;enable bit

lib_80186       segment public 'code'
                assume cs:lib_80186

                public _config_rcu
_config_rcu     proc far

                push bp                     ;save caller's bp
                mov  bp, sp                 ;get current top of stack

_clock_time     equ  word ptr[bp+6]         ;get parameters off
_dram_addr      equ  word ptr[bp+8]         ;the stack

                push ax                     ;save registers that
                push cx                     ;will be modified
                push dx
                push di
```

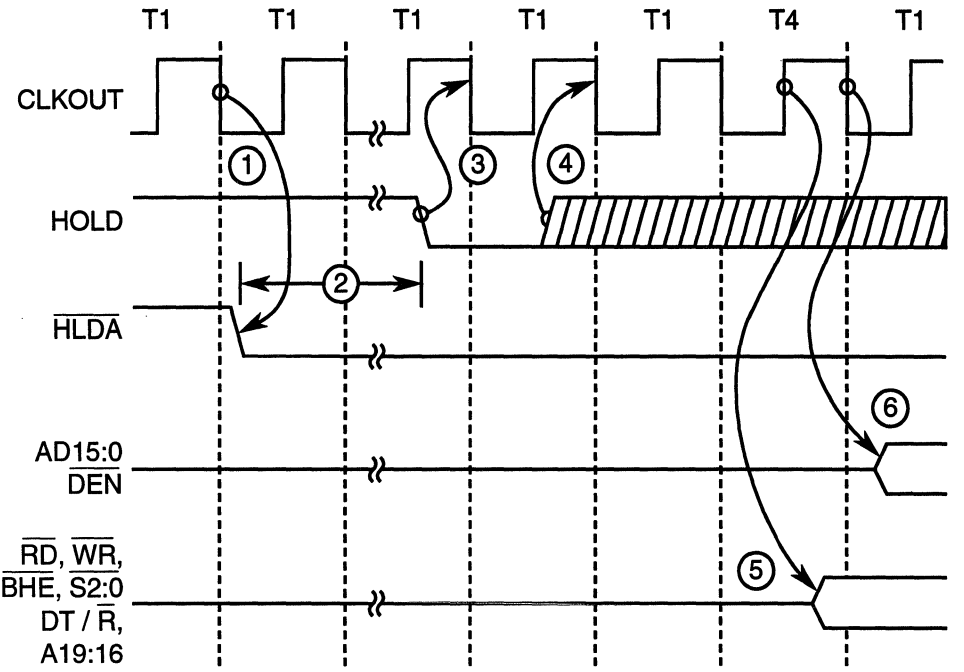
Example 7-1. Initializing the Refresh Control Unit

```
                mov  dx, RFBASE              ;set upper 7 address bits
                mov  ax, _dram_addr
                out  dx, al

                mov  dx, RFTIME              ;set clock pre_scaler
                mov  ax, _clock_time
                out  dx, al

                mov  dx, RFCON               ;Enable RCU
                mov  ax, Enable
                out  dx, al

                mov  cx, 8                   ;8 dummy cycles are
                                             ;required by DRAMs
                xor  di, di                  ;before actual use

_exercise_ram:
                mov  word ptr [di], 0
                loop _exercise_ram

                pop  di                      ;restore saved registers
                pop  dx
                pop  cx
                pop  ax
                pop  bp                      ;restore caller's bp

                ret
_config_rcu     endp
lib_80186       ends
                end
```

Example 7-1. Initializing the Refresh Control Unit (Continued)

7.8 REFRESH OPERATION AND BUS HOLD

When another bus master controls the bus, the processor keeps $\overline{\text{HLDA}}$ active as long as the HOLD input remains active. If the Refresh Control Unit generates a refresh request during bus hold, the processor drives the $\overline{\text{HLDA}}$ signal inactive, indicating to the current bus master that it wishes to regain bus control (see Figure 7-10). The BIU begins a refresh bus cycle only after the alternate master removes HOLD. The user must design the system so that the processor can regain bus control. If the alternate master asserts HOLD after the processor starts the refresh cycle, the CPU will relinquish control by asserting $\overline{\text{HLDA}}$ when the refresh cycle is complete.



NOTES:

1. HLDA is deasserted; signaling need to run DRAM refresh cycles less than T_{CLOV}.
2. External bus master terminates use of the bus.
3. HOLD deasserted; greater than T_{CLIS}.
4. Hold may be reasserted after one clock.
5. Lines come out of float in order to run DRAM refresh cycle.

A1269-0A

Figure 7-10. Regaining Bus Control to Run a DRAM Refresh Bus Cycle

intel®

8

Interrupt Control Unit

CHAPTER 8
INTERRUPT CONTROL UNIT

The 80C186 Modular Core has a single maskable interrupt input. (See "Interrupts and Exception Handling" on page 2-39.) The Interrupt Control Unit (ICU) expands the interrupt capabilities beyond a single input. It receives and processes maskable interrupts from multiple sources and presents them to the CPU through the maskable interrupt input. Interrupts can originate from the on-chip peripherals and from five external interrupt pins. The Interrupt Control Unit synchronizes and prioritizes the interrupts and provides the interrupt type vector to the CPU. (See Figure 8-1.)

The Interrupt Control Unit has the following features:

- Programmable priority of each interrupt source
- Individual masking of each interrupt source
- Nesting of interrupt sources
- Support for polled operation
- Support for cascading external 8259A modules to expand external interrupt sources

8.1 FUNCTIONAL OVERVIEW

All microcomputer systems must communicate in some way with the external world. A typical system might have a keyboard, a disk drive and a communications port, all requiring CPU attention at different times. There are two distinct ways to process peripheral I/O requests: polling and interrupts.

Polling requires that the CPU check each peripheral device in the system periodically to see whether it requires servicing. It would not be unusual to poll a low-speed peripheral (a serial port, for instance) thousands of times before it required servicing. In most cases, the use of polling has a detrimental effect on system throughput. Any time used to check the peripherals is time spent away from the main processing tasks.

Interrupts eliminate the need for polling by signalling the CPU that a peripheral device requires servicing. The CPU then stops executing the main task, saves its state and transfers execution to the peripheral-servicing code (the *interrupt handler*). At the end of the interrupt handler, the CPU's original state is restored and execution continues at the point of interruption in the main task.

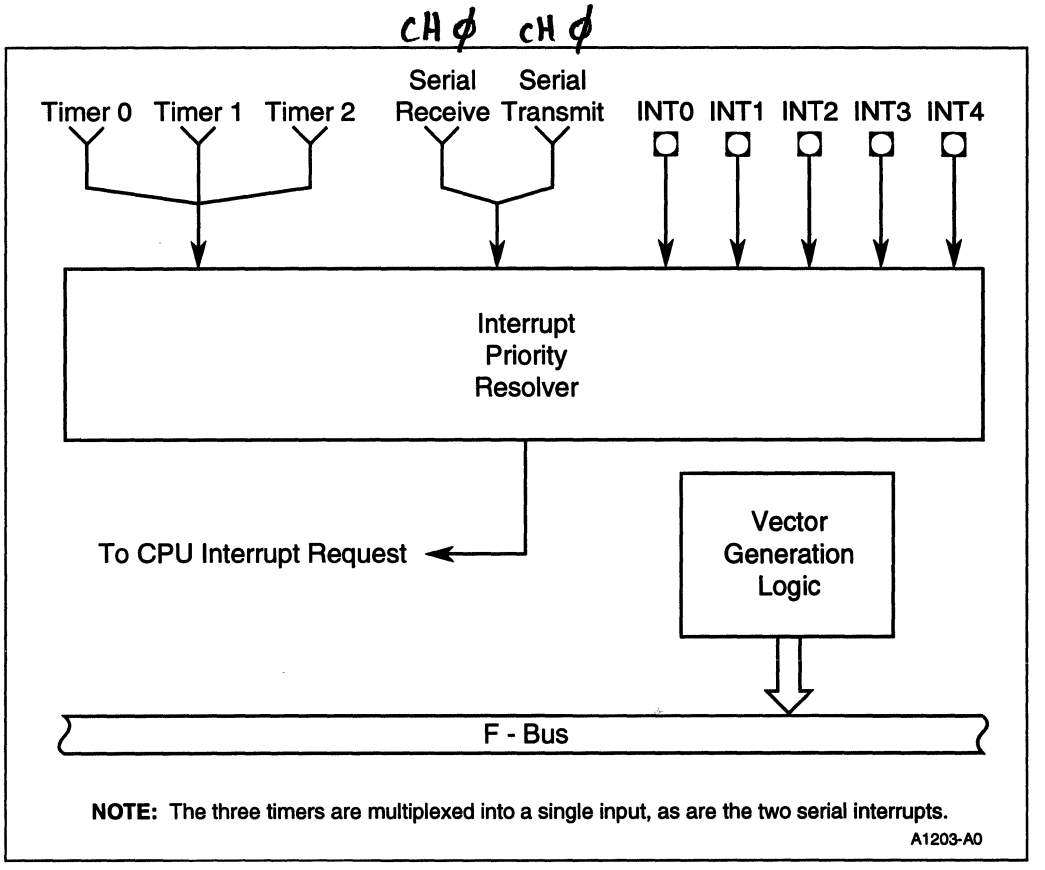


Figure 8-1. Interrupt Control Unit Block Diagram

(See 7/31/95 ERRATA)

8.1.1 Generic Functions

Several functions of the Interrupt Control Unit are common among most interrupt controllers. This section describes how those generic functions are implemented in the Interrupt Control Unit.

8.1.1.1 Interrupt Masking

There are circumstances in which a programmer may need to disable an interrupt source temporarily (for example, while executing a time-critical section of code or servicing a high-priority task). This temporary disabling is called interrupt masking. All interrupts from the Interrupt Control Unit can be masked either globally or individually.

The Interrupt Enable bit in the Processor Status Word globally enables or disables the maskable interrupt request from the Interrupt Control Unit. The programmer controls the Interrupt Enable bit with the STI (set interrupt) and CLI (clear interrupt) instructions.

Besides being globally enabled or disabled by the Interrupt Enable bit, each interrupt source can be individually enabled or disabled. The Interrupt Mask register has a single bit for each interrupt source. The programming can selectively mask (disable) or unmask (enable) each interrupt source by setting or clearing the corresponding bit in the Interrupt Mask register.

8.1.1.2 Interrupt Priority

One critical function of the Interrupt Control Unit is to prioritize interrupt requests. When multiple interrupts are pending, priority determines which interrupt request is serviced first. In many systems, an interrupt handler may itself be interrupted by another interrupt source. This is known as *interrupt nesting*. With interrupt nesting, priority determines whether an interrupt source can preempt an interrupt handler that is currently executing.

Each interrupt source is assigned a priority between zero (highest) and seven (lowest). After reset, the interrupts default to the priorities shown in Table 8-1. Because the timers share an interrupt source, they also share a priority. Within the assigned priority, each has a relative priority (Timer 0 has the highest relative priority and Timer 2 has the lowest). The serial channel 0 receive and transmit interrupts also share a priority. Within the assigned priority, the receive interrupt has the higher relative priority.

(ERRATA 12/14/95)

Table 8-1. Default Interrupt Priorities

Interrupt Name	Relative Priority
Timer 0	0 (a)
Timer 1	0 (b)
Timer 2	0 (c)
Serial Channel 0 Receive	1(a)
Serial Channel 0 Transmit	1(b)

NAME	PRIORITY
INT4	2
INT0	3
INT1	4
INT2	5
INT3	6

The priority of each source is programmable. The Interrupt Control register enables the programmer to assign each source a priority that differs from the default. The priority must still be between zero (highest) and seven (lowest). Interrupt sources can be programmed to share a priority. The Interrupt Control Unit uses the default priorities (see Table 8-1) within the shared priority level to determine which interrupt to service first. For example, assume that INT0 and INT1 are both programmed to priority seven. Because INT0 has the higher default priority, it is serviced first.

Interrupt sources can be masked on the basis of their priority. The Priority Mask register masks all interrupts with priorities lower than its programmed value. After reset, the Priority Mask register contains priority seven, which effectively enables all interrupts. The programmer can then program the register with any valid priority level.

8.1.1.3 Interrupt Nesting

When entering an interrupt handler, the CPU pushes the Processor Status Word onto the stack and clears the Interrupt Enable bit. The processor enters all interrupt handlers with maskable interrupts disabled. Maskable interrupts remain disabled until either the IRET instruction restores the Interrupt Enable bit or the programmer explicitly enables interrupts. Enabling maskable interrupts within an interrupt handler allows interrupts to be nested. Otherwise, interrupts are processed sequentially; one interrupt handler must finish before another executes.

The simplest way to use the Interrupt Control Unit is without nesting. The operation and servicing of all sources of maskable interrupts is straightforward. However, the application tradeoff is that an interrupt handler will finish executing even if a higher priority interrupt occurs. This can add considerable latency to the higher priority interrupt.

In the simplest terms, the Interrupt Control Unit asserts the maskable interrupt request to the CPU, waits for the interrupt acknowledge, then presents the interrupt type of the highest priority unmasked interrupt to the CPU. The CPU then executes the interrupt handler for that interrupt. Because the interrupt handler never sets the Interrupt Enable bit, it can never be interrupted.

The function of the Interrupt Control Unit is more complicated with interrupt nesting. In this case, an interrupt can occur during execution of an interrupt handler. That is, one interrupt can *preempt* another. Two rules apply for interrupt nesting:

- An interrupt source cannot preempt interrupts of higher priority.
- An interrupt source cannot preempt itself. The interrupt handler must finish executing before the interrupt is serviced again. (Special Fully Nested Mode is an exception. See "Special Fully Nested Mode" on page 8-8.)

8.2 FUNCTIONAL OPERATION

This section covers the process in which the Interrupt Control Unit receives interrupts and asserts the maskable interrupt request to the CPU.

8.2.1 Typical Interrupt Sequence

When the Interrupt Control Unit first detects an interrupt, it sets the corresponding bit in the Interrupt Request register to indicate that the interrupt is pending. The Interrupt Control Unit checks all pending interrupt sources. If the interrupt is unmasked and meets the priority criteria (see "Priority Resolution" on page 8-5), the Interrupt Control Unit asserts the maskable interrupt request to the CPU, then waits for the interrupt acknowledge.

When the Interrupt Control Unit receives the interrupt acknowledge, it passes the interrupt type to the CPU. At that point, the CPU begin the interrupt processing sequence.(See "Interrupt/Exception Processing" on page 2-39 for details.) The Interrupt Control Unit always passes the vector that has the highest priority at the time the acknowledge is received. If a higher priority interrupt occurs before the interrupt acknowledge, the higher priority interrupt has precedence.

When it receives the interrupt acknowledge, the Interrupt Control Unit clears the corresponding bit in the Interrupt Request register and sets the corresponding bit in the In-Service register. The In-Service register keeps track of which interrupt handlers are being processed. At the end of an interrupt handler, the programmer must issue an End-of-Interrupt (EOI) command to explicitly clear the In-Service register bit. If the bit remains set, the Interrupt Control Unit **cannot** process any additional interrupts from that source.

8.2.2 Priority Resolution

The decision to assert the maskable interrupt request to the CPU is somewhat complicated. The complexity is needed to support interrupt nesting. First, an interrupt occurs and the corresponding Interrupt Request register bit is set. The Interrupt Control Unit then asserts the maskable interrupt request to the CPU, if the pending interrupt satisfies these requirements:

1. its Interrupt Mask bit is cleared (it is unmasked)
2. its priority is higher than the value in the Priority Mask register
3. its In-Service bit is cleared
4. its priority is equal to or greater than that of any interrupt whose In-Service bit is set

The In-Service register keeps track of interrupt handler execution. The Interrupt Control Unit uses this information to decide whether another interrupt source has sufficient priority to preempt an interrupt handler that is executing.

8.2.2.1 Priority Resolution Example

This example illustrates priority resolution. Assume these initial conditions:

- the Interrupt Control Unit has been initialized
- no interrupts are pending
- no In-Service bits are set
- the Interrupt Enable bit is set
- all interrupts are unmasked
- the default priority scheme is being used
- the Priority Mask register is set to the lowest priority (seven)

The example uses two external interrupt sources, INT0 and INT3, to describe the process.

1. A low-to-high transition on INT0 sets its Interrupt Request bit. The interrupt is now pending.
2. Because INT0 is the only pending interrupt, it meets all the priority criteria. The Interrupt Control Unit asserts the interrupt request to the CPU and waits for an acknowledge.
3. The CPU acknowledges the interrupt.
4. The Interrupt Control Unit passes the interrupt type (in this case, type 12) to the CPU.
5. The Interrupt Control Unit clears the INT0 bit in the Interrupt Request register and sets the INT0 bit in the In-Service register.
6. The CPU executes the interrupt processing sequence and begins executing the interrupt handler for INT0.
7. During execution of the INT0 interrupt handler, a low-to-high transition on INT3 sets its Interrupt Request bit.
8. The Interrupt Control Unit determines that INT3 has a lower priority than INT0, which is currently executing (INT0's In-Service bit is set). INT3 does not meet the priority criteria, so no interrupt request is sent to the CPU. (If INT3were programmed with a higher priority than INT0, the request would be sent.) INT3 remains pending in the Interrupt Request register.
9. The INT0 interrupt handler completes and sends an EOI command to clear the INT0 bit in the In-Service register.
10. INT3 is still pending and now meets all the priority criteria. The Interrupt Control Unit asserts the interrupt request to the CPU and the process begins again.

8.2.2.2 Interrupts That Share a Single Source

Multiple interrupt requests can share a single interrupt input to the Interrupt Control Unit. (For example, the three timers share a single input.) Although these interrupts share an input, each has its own interrupt vector. (For example, when a Timer 0 interrupt occurs, the Timer 0 interrupt handler is executed.) This section uses the three timers as an example to describe how these interrupts are prioritized and serviced.

The Interrupt Status register acts as a second-level request register to process the timer interrupts. It contains a bit for each timer interrupt. When a timer interrupt occurs, both the individual Interrupt Status register bit and the shared Interrupt Request register bit are set. From this point, the interrupt is processed like any other interrupt source.

When the shared interrupt is acknowledged, the timer interrupt with the highest priority (see Table 8-1 on page 8-3) **at that time** is serviced first and that timer's Interrupt Status bit is cleared. If no other timer Interrupt Status bits are set, the shared Interrupt Request bit is also cleared. If other timer interrupts are pending, the Interrupt Request bit remains set.

When the timer interrupt is acknowledged, the shared In-Service bit is set. **No other timer interrupts can occur when the In-Service bit is set.** If a second timer interrupt occurs while another timer interrupt is being serviced, the second interrupt remains pending until the interrupt handler for the first interrupt finishes and clears the In-Service bit. (This is true even if the second interrupt has a higher priority than the first.)

8.2.3 Cascading with External 8259As

For applications that require more external interrupt pins than the number provided on the Interrupt Control Unit, external 8259A modules can be used to increase the number of external interrupt pins. The cascade mode of the Interrupt Control Unit supports the external 8259As. The INT2/$\overline{\text{INTA0}}$ and INT3/$\overline{\text{INTA1}}$ pins can serve either of two functions. Outside cascade mode, they serve as external interrupt inputs. In cascade mode, they serve as interrupt acknowledge outputs. $\overline{\text{INTA0}}$ is the acknowledge for INT0, and $\overline{\text{INTA1}}$ is the acknowledge for INT1. (See Figure 8-2.)

The INT2/$\overline{\text{INTA0}}$ and INT3/$\overline{\text{INTA1}}$ pins are inputs after reset until the pins are confiugred as outputs. The pullup resistors ensure that the $\overline{\text{INTA}}$ pins never float (which would cause a spurious interrupt acknowledge to the 8259A). The value of the resistors must be high enough to prevent excessive loading on the pins.

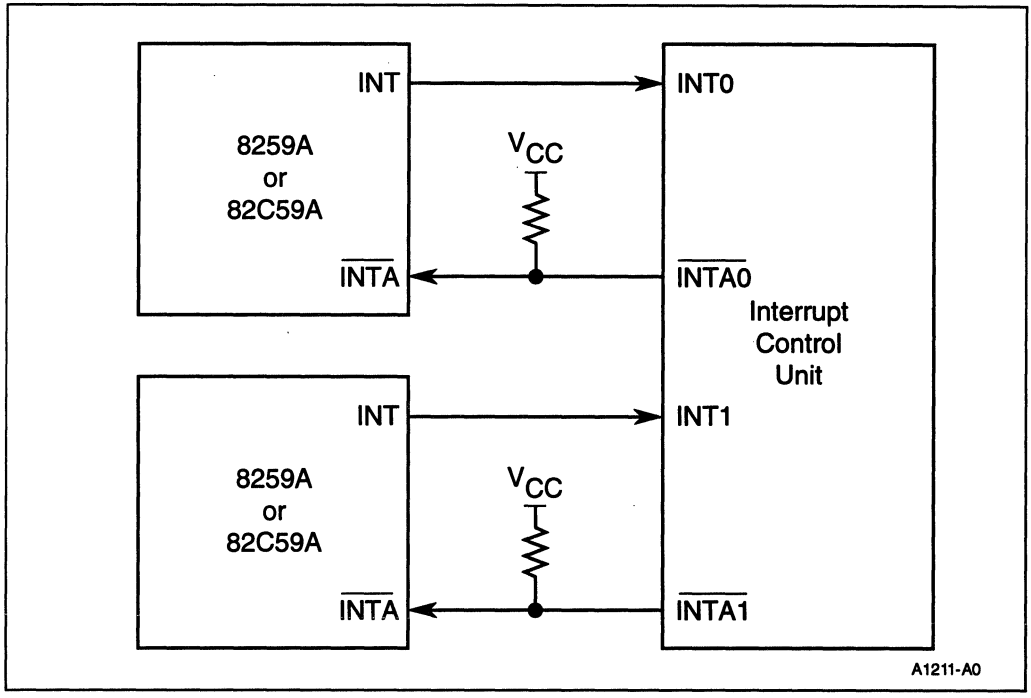


Figure 8-2. Using External 8259A Modules in Cascade Mode

8.2.3.1 Special Fully Nested Mode

Special fully nested mode is an optional feature normally used with cascade mode. It is applicable only to INT0 and INT1. In special fully nested mode, an interrupt request is serviced even if its In-Service bit is set.

In cascade mode, an 8259A controls up to eight external interrupts that share a single interrupt input pin. Special fully nested mode allows the 8259A's priority structure to be maintained. For example, assume that the CPU is servicing a low-priority interrupt from the 8259A. While the interrupt handler is executing, the 8259A receives a higher priority interrupt from one of its sources. The 8259A applies its own priority criteria to that interrupt and asserts its interrupt to the Interrupt Control Unit. Special fully nested mode allows the higher priority interrupt to be serviced even though the In-Service bit for that source is already set. A higher priority interrupt has preempted a lower priority interrupt, and interrupt nesting is fully maintained.

Special fully nested mode can also be used without cascade mode. In this case, it allows a single external interrupt pin (either INT0 or INT1) to preempt itself.

8.2.4 Interrupt Acknowledge Sequence

During the interrupt acknowledge sequence, the Interrupt Control Unit passes the interrupt type to the CPU. The CPU then multiplies the interrupt type by four to derive the interrupt vector address in the interrupt vector table. ("Interrupt/Exception Processing" on page 2-39 describes the interrupt acknowledge sequence and Figure 2-25 on page 2-40 illustrates the interrupt vector table.)

The interrupt types for all sources are fixed and unalterable (see Table 8-2). The Interrupt Control Unit passes these types to the CPU internally. The first external indication of the interrupt acknowledge sequence is the CPU fetch from the interrupt vector table.

In cascade mode, the external 8259A supplies the interrupt type. In this case, the CPU runs an external interrupt acknowledge cycle to fetch the interrupt type from the 8259A (see "Interrupt Acknowledge Bus Cycle" on page 3-25).

Table 8-2. Fixed Interrupt Types

Interrupt Name	Interrupt Type
Timer 0	8
Timer 1	18
Timer 2	19
Serial Channel 0 Receive	20
Serial Channel 0 Transmit	21
INT4	17
INT0	12
INT1	13
INT2	14
INT3	15

8.2.5 Polling

In some applications, it is desirable to poll the Interrupt Control Unit. The CPU polls the Interrupt Control Unit for any pending interrupts, and software can service interrupts whenever it is convenient. The Poll and Poll Status registers support polling.

Software reads the Poll register to get the type of the highest priority pending interrupt, then calls the corresponding interrupt handler. Reading the Poll register also acknowledges the interrupt. This clears the Interrupt Request bit and sets the In-Service bit for the interrupt. The Poll Status register has the same format as the Poll register, but reading the Poll Status register does **not** acknowledge the interrupt.

8.2.6 Edge and Level Triggering

The external interrupts (INT4:0) can be programmed for either edge or level triggering (see "Interrupt Control Registers" on page 8-12). Both types of triggering are active high. An edge-triggered interrupt is generated by a zero-to-one transition on an external interrupt pin. The pin must remain high until after the CPU acknowledges the interrupt, then must go low to reset the edge-detection circuitry. (See the current data sheet for timing requirements.) The edge-detection circuitry must be reset to enable further interrupts to occur.

A level-triggered interrupt is generated by a valid logic one on the external interrupt pin. The pin must remain high until after the CPU acknowledges the interrupt. Unlike edge-triggered interrupts, level-triggered interrupts will continue to occur if the pin remains high. A level-triggered external interrupt pin must go low before the EOI command to prevent another interrupt.

NOTE

When external 8259As are cascaded into the Interrupt Control Unit, INT0 and INT1 must be programmed for level-triggered interrupts.

8.2.7 Additional Latency and Response Time

The Interrupt Control Unit adds 5 clocks to the interrupt latency of the CPU. Cascade mode adds 13 clocks to the interrupt response time because the CPU must run the interrupt acknowledge bus cycles. (See Figure 8-3 on page 8-11 and Figure 2-27 on page 2-46.)

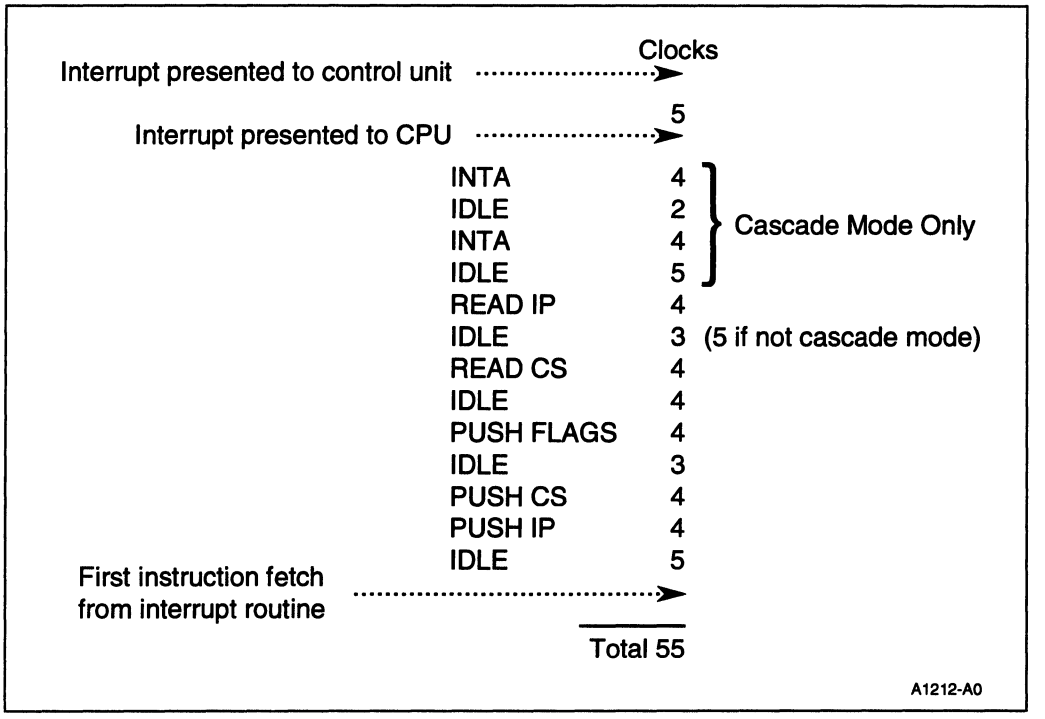


Figure 8-3. Interrupt Control Unit Latency and Response Time

8.3 PROGRAMMING THE INTERRUPT CONTROL UNIT

Table 8-3 lists the Interrupt Control Unit registers with their Peripheral Control Block offset addresses. The remainder of this section describes the functions of the registers.

Table 8-3. Interrupt Control Unit Registers

Register Name	Offset Address
INT3 Control	1EH
INT2 Control	1CH
INT1 Control	1AH
INT0 Control	18H
INT4 Control	16H
Serial Control	14H
Timer Control	12H
Interrupt Status	10H
Interrupt Request	0EH

Table 8-3. Interrupt Control Unit Registers (Continued)

Register Name	Offset Address
In-Service	0CH
Priority Mask	0AH
Interrupt Mask	08H
Poll Status	06H
Poll	04H
EOI	02H

8.3.1 Interrupt Control Registers

Each interrupt source has its own Interrupt Control register. The Interrupt Control register allows you to define the behavior of each interrupt source. Figure 8-4 shows the registers for the timers and serial channel, Figure 8-5 shows the registers for INT4:2, and Figure 8-6 shows the registers for INT0 and INT1.

All Interrupt Control registers have a three-bit field (PM2:0) that defines the priority level for the interrupt source and a mask bit (MSK) that enables or disables the interrupt source. The mask bit is the same as the one in the Interrupt Mask register. Modifying a bit in either register also modifies that same bit in the other register.

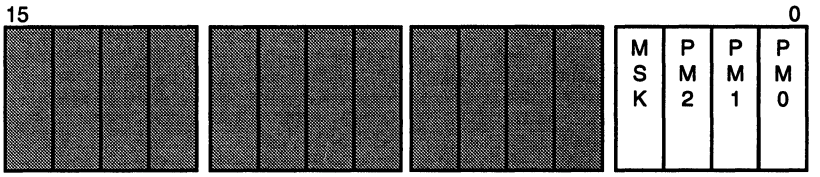
The Interrupt Control registers for the external interrupt pins also have a bit (LVL) that selects level-triggered or edge-triggered mode for that interrupt. (See "Edge and Level Triggering" on page 8-10.)

The Interrupt Control registers for the cascadable external interrupt pins (INT0 and INT1) have two additional bits to support the external 8259As. The CAS bit enables cascade mode, and the SFNM bit enables special fully nested mode.

Register Name: Interrupt Control Register (internal sources)

Register Mnemonic: TCUCON, SCUCON,

Register Function: Control register for the internal interrupt sources



Bit Mnemonic	Bit Name	Reset State	Function
MSK	Interrupt Mask	1	Clear to enable interrupts from this source.
PM2:0	Priority Level	111	Defines the priority level for this source.

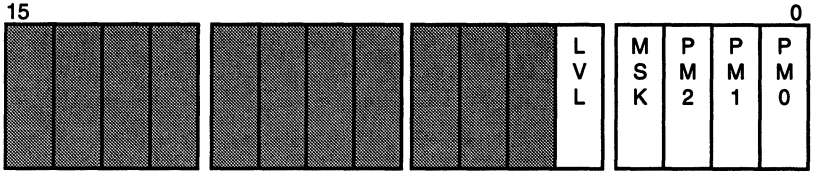
NOTE: Reserved register bits are shown with gray shading. Reserved bits must be written to a logic zero to ensure compatibility with future Intel products.

Figure 8-4. Interrupt Control Register for Internal Sources

Register Name: Interrupt Control Register (non-cascadable pins)

Register Mnemonic: I2CON, I3CON, I4CON

Register Function: Control register for the non-cascadable external internal interrupt pins



A1214-A0

Bit Mnemonic	Bit Name	Reset State	Function
LVL	Level-trigger	0	Selects the interrupt triggering mode: 0 = edge triggering 1 = level triggering.
MSK	Interrupt Mask	1	Clear to enable interrupts from this source.
PM2:0	Priority Level	111	Defines the priority level for this source.

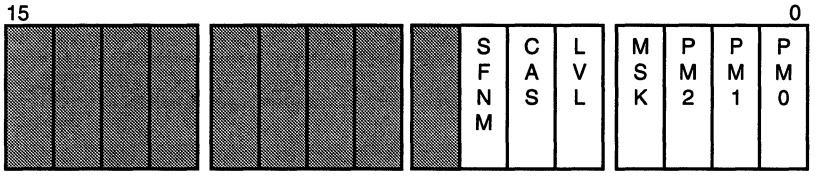
NOTE: Reserved register bits are shown with gray shading. Reserved bits must be written to a logic zero to ensure compatibility with future Intel products.

Figure 8-5. Interrupt Control Register for Noncascadable External Pins

Register Name: Interrupt Control Register (cascadable pins)

Register Mnemonic: I0CON, I1CON

Register Function: Control register for the cascadable external interrupt pins



A1215-A0

Bit Mnemonic	Bit Name	Reset State	Function
SFNM	Special Fully Nested Mode	0	Set to enable special fully nested mode.
CAS	Cascade Mode	0	Set to enable cascade mode.
LVL	Level-trigger	0	Selects the interrupt triggering mode: 0 = edge triggering 1 = level triggering. The LVL bit **must be set** when external 8259As are cascaded into the Interrupt Control Unit.
MSK	Interrupt Mask	1	Clear to enable interrupts from this source.
PM2:0	Priority Level	111	Defines the priority level for this source.

NOTE: Reserved register bits are shown with gray shading. Reserved bits must be written to a logic zero to ensure compatibility with future Intel products.

Figure 8-6. Interrupt Control Register for Cascadable Interrupt Pins

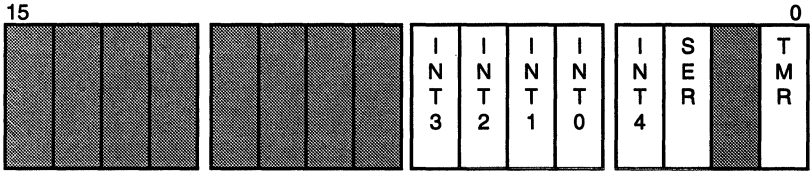
8.3.2 Interrupt Request Register

The Interrupt Request register (Figure 8-7) has one bit for each interrupt source. When a source requests an interrupt, its Interrupt Request bit is set (without regard to whether the interrupt is masked). The Interrupt Request bit is cleared when the interrupt is acknowledged. An external interrupt pin must remain asserted until its interrupt is acknowledged. Otherwise, the Interrupt Request bit will be cleared, but the interrupt will not be serviced.

Register Name: Interrupt Request Register

Register Mnemonic: REQST

Register Function: Stores pending interrupt requests



A1206-A0

Bit Mnemonic	Bit Name	Reset State	Function
INT3:0, INT4	External Interrupts	0000 0	A bit is set to indicate a pending interrupt from the corresponding external interrupt pin.
SER	Serial Channel 0 Interrupt	0	This bit is set to indicate a pending interrupt from serial channel 0 (either a receive or a transmit interrupt).
TMR	Timer Interrupt	0	This bit is set to indicate a pending interrupt from one of the timers.

NOTE: Reserved register bits are shown with gray shading. Reserved bits must be written to a logic zero to ensure compatibility with future Intel products.

Figure 8-7. Interrupt Request Register

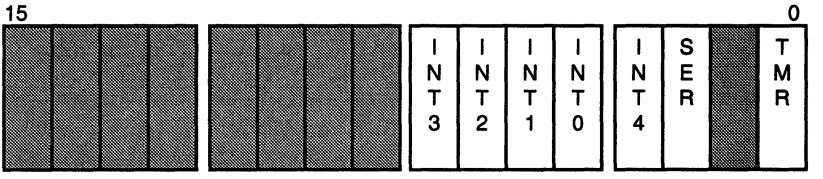
8.3.3 Interrupt Mask Register

The Interrupt Mask register (Figure 8-8) contains a mask bit for each interrupt source. This register allows you to mask (disable) individual interrupts. Set a mask bit to disable interrupts from the corresponding source. The mask bit is the same as the one in the Interrupt Control register. Modifying a bit in either register also modifies that same bit in the other register.

Register Name: Interrupt Mask Register

Register Mnemonic: IMASK

Register Function: Masks individual interrupt sources



A1207-A0

Bit Mnemonic	Bit Name	Reset State	Function
INT3:0, INT4	External Interrupt Mask	~~0000 0~~ Fh, 1h	Set a bit to mask (disable) interrupt requests from the corresponding external interrupt pin.
SER	Serial Channel 0 Interrupt Mask	~~0~~ 1h	Set to mask (disable) interrupt requests from serial channel 0.
TMR	Timer Interrupt Mask	~~0~~ 1h	Set to mask (disable) interrupt requests from the timers.

NOTE: Reserved register bits are shown with gray shading. Reserved bits must be written to a logic zero to ensure compatibility with future Intel products.

Figure 8-8. Interrupt Mask Register

(12/14/95 ERRATA)

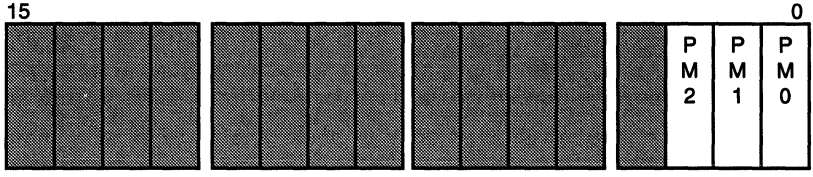
8.3.4 Priority Mask Register

The Priority Mask register (Figure 8-9) contains a three-level field that holds a priority value. This register allows you to mask interrupts based on their priority levels. Write a priority value to the PM2:0 field to specify the lowest priority interrupt to be serviced. This disables (masks) any interrupt source whose priority is lower than the PM2:0 value. After reset, the Priority Mask register is set to the lowest priority (seven), which enables all interrupts of any priority.

Register Name: Priority Mask Register

Register Mnemonic: PRIMSK

Register Function: Masks lower-priority interrupt sources



A1216-A0

Bit Mnemonic	Bit Name	Reset State	Function
PM2:0	Priority Mask	111	Defines a priority-based interrupt mask. Interrupts whose priority is lower than this value are masked.

NOTE: Reserved register bits are shown with gray shading. Reserved bits must be written to a logic zero to ensure compatibility with future Intel products.

Figure 8-9. Priority Mask Register

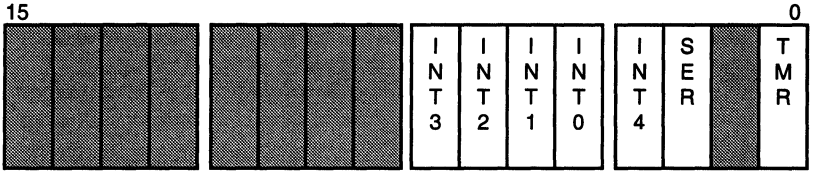
8.3.5 In-Service Register

The In-Service register has a bit for each interrupt source. The bits indicate which source's interrupt handlers are currently executing. The In-Service bit is set when an interrupt is acknowledged; the interrupt handler must clear it with an End-of-Interrupt (EOI) command. The Interrupt Control Unit uses the In-Service register to support interrupt nesting.

Register Name: In-Service Register

Register Mnemonic: INSERV

Register Function: Indicates which interrupt handlers are in process



A1204-A0

Bit Mnemonic	Bit Name	Reset State	Function
INT3:0, INT4	External Interrupt In-Service	0000 0	A bit is set to indicate that the corresponding external interrupt is being serviced.
SER	Serial Channel 0 Interrupt In-Service	0	This bit is set to indicate that a serial channel interrupt is being serviced.
TMR	Timer Interrupt In-Service	0	This bit is set to indicate that a timer interrupt is being serviced.

NOTE: Reserved register bits are shown with gray shading. Reserved bits must be written to a logic zero to ensure compatibility with future Intel products.

Figure 8-10. In-Service Register

8.3.6 Poll and Poll Status Registers

The Poll and Poll Status registers allow you to poll the Interrupt Control Unit and service interrupts through software. You can read these registers to determine whether an interrupt is pending and, if so, the interrupt type. The registers contain identical information, but reading them produces different results.

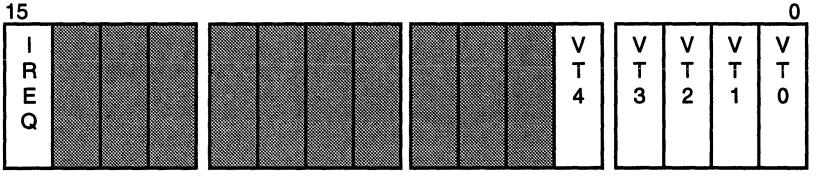
Reading the Poll register (Figure 8-11) acknowledges the pending interrupt, just as if the CPU had started the interrupt vectoring sequence. The Interrupt Control Unit updates the Interrupt Request, In-Service, Poll, and Poll Status registers, as it does in the normal interrupt acknowledge sequence. However, the processor does not run an interrupt acknowledge sequence or fetch the vector from the vector table. Instead, software must read the interrupt type and execute the proper routine to service the pending interrupt.

Reading the Poll Status register (Figure 8-12) will merely transmit the status of the polling bits without modifying any of the other Interrupt Controller registers.

Register Name: Poll Register

Register Mnemonic: POLL

Register Function: Read to check for and acknowledge pending interrupts when polling



A1208-A0

Bit Mnemonic	Bit Name	Reset State	Function
IREQ	Interrupt Request	0	This bit is set to indicate a pending interrupt.
VT4:0	Vector Type	0	Contains the interrupt type of the highest priority pending interrupt.

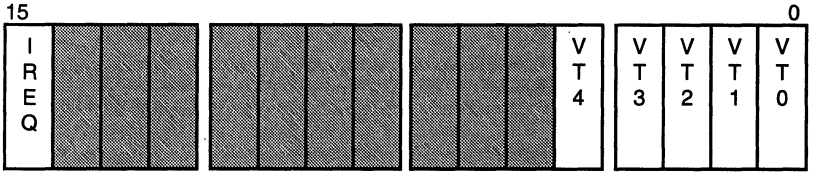
NOTE: Reserved register bits are shown with gray shading. Reserved bits must be written to a logic zero to ensure compatibility with future Intel products.

Figure 8-11. Poll Register

Register Name: Poll Status Register

Register Mnemonic: POLLSTS

Register Function: Read to check for pending interrupts when polling



A1209-A0

Bit Mnemonic	Bit Name	Reset State	Function
IREQ	Interrupt Request	0	This bit is set to indicate a pending interrupt.
VT4:0	Vector Type	0	Contains the interrupt type of the highest priority pending interrupt.

NOTE: Reserved register bits are shown with gray shading. Reserved bits must be written to a logic zero to ensure compatibility with future Intel products.

Figure 8-12. Poll Status Register

8.3.7 End-of-Interrupt (EOI) Register

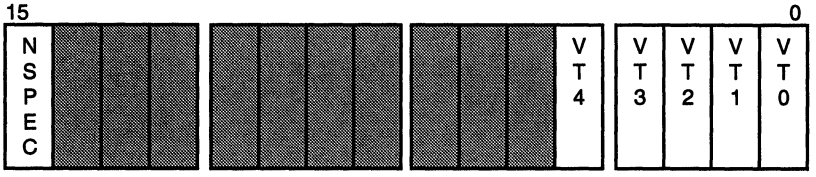
The End-of-Interrupt register (Figure 8-13) issues an End-of-Interrupt (EOI) command to the Interrupt Control Unit, which clears the In-Service bit for the associated interrupt. An interrupt handler typically ends with an EOI command. There are two types of EOI commands: nonspecific and specific. A nonspecific EOI simply clears the In-Service bit of the highest priority interrupt. To issue a nonspecific EOI command, set the NSPEC bit. (Write 8000H to the EOI register.)

A specific EOI clears a particular In-Service bit. To issue a specific EOI command, clear the NSPEC bit and write the VT4:0 bits with the interrupt type of the interrupt whose In-Service bit you wish to clear. For example, to clear the In-Service bit for INT2, write 000EH to the EOI register. The timer interrupts share an In-Service bit. To clear the In-Service bit for any timer interrupt with a specific EOI, write 0008H (interrupt type 8) to the EOI register.

Register Name: End-of-Interrupt Register

Register Mnemonic: EOI

Register Function: Used to issue an EOI command



A1210-A0

Bit Mnemonic	Bit Name	Reset State	Function
NSPEC	Nonspecific EOI	0	Set to issue a nonspecific EOI.
VT4:0	Interrupt Type	0 0000	Write with the interrupt type of the interrupt whose In-Service bit is to be cleared.

NOTE: Reserved register bits are shown with gray shading. Reserved bits must be written to a logic zero to ensure compatibility with future Intel products.

Figure 8-13. End-of-Interrupt Register

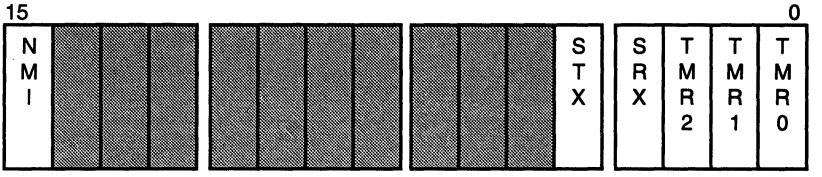
8.3.8 Interrupt Status Register

The Interrupt Status register (Figure 8-14) contains one bit for each interrupt that shares an interrupt source and one bit for the nonmaskable interrupt (NMI). A bit is set to indicate a pending interrupt and is cleared when the interrupt request is acknowledged. Any number of bits can be set at any one time.

Register Name: Interrupt Status Register

Register Mnemonic: INTSTS

Register Function: Indicates pending NMI or shared-source interrupts



A1205-A0

Bit Mnemonic	Bit Name	Reset State	Function
NMI	Non-maskable Interrupt	0	This bit is set to indicate a pending NMI.
STX	Serial Transmit	0	This bit is set to indicate a pending serial transmit interrupt.
SRX	Serial Receive	0	This bit is set to indicate a pending serial receive interrupt.
TMR2:0	Timer	0	A bit is set to indicate a pending interrupt from the corresponding timer.

NOTE: Reserved register bits are shown with gray shading. Reserved bits must be written to a logic zero to ensure compatibility with future Intel products.

Figure 8-14. Interrupt Status Register

8.3.9 Initializing the Interrupt Control Unit

Follow these steps to initialize the Interrupt Control Unit.

1. Determine which interrupt sources you want to use.
2. Determine whether to use the default priority scheme or devise your own.

3. Program the Interrupt Control register for each interrupt source.
   — For external interrupt pins, select edge or level triggering.
   — For INT0 or INT1, enable cascade mode, special fully nested mode, or both, if you wish to use them.
   — If you are using a custom priority scheme, program the priority level for each interrupt source.
4. Program the Priority Mask with a priority mask level, if you wish to mask interrupts based on priority. (The default is level seven, which enables all interrupt levels.)
5. Set the mask bit in the Interrupt Mask register for any interrupts that you wish to disable.

Example 8-1 shows sample code to initialize the Interrupt Control Unit.

```
$mod186
name          example_80C186_ICU_initialization
;
;This routine configures the interrupt controller to provide two cascaded
;interrupt inputs (through an external 8259A connected to INT0 and INTA0#)
;and two direct interrupt inputs connected to INT1 and INT3. The default
;priorities are used.
;
;The example assumes that the register addresses have been properly defined.
;
code          segment
              assume cs:code
set_int_      proc near
              push dx
              push ax
              mov  ax,0110111B              ;cascade mode, priority seven
              mov  dx,I0CON                 ;INT0 control register
              out  dx,ax
              mov  ax,01001101B             ;unmask INT1 and INT3
              mov  dx,IMASK
              out  dx,ax
              pop  ax
              pop  dx
              ret
set_int_      endp
code          ends
              end
```

Example 8-1. Initializing the Interrupt Control Unit

intel®

9

Timer/Counter Unit

CHAPTER 9
TIMER/COUNTER UNIT

The Timer/Counter Unit can be used in many applications. Some of these applications include a real-time clock, a square-wave generator and a digital one-shot. All of these can be implemented in a system design. A real-time clock can be used to update time-dependent memory variables. A square-wave generator can be used to provide a system clock tick for peripheral devices. (See "Timer/Counter Unit Application Examples" on page 9-17 for code examples that configure the Timer/Counter Unit for these applications.)

9.1 FUNCTIONAL OVERVIEW

The Timer/Counter Unit is composed of three independent 16-bit timers (see Figure 9-1). The operation of these timers is independent of the CPU. The internal Timer/Counter Unit can be modeled as a single counter element, time-multiplexed to three register banks. The register banks are dual-ported between the counter element and the CPU. During a given bus cycle, the counter element and CPU can both access the register banks; these accesses are synchronized.

The Timer/Counter Unit is serviced over four clock periods, one timer during each clock, with an idle clock at the end (see Figure 9-2). No connection exists between the counter element's sequencing through timer register banks and the Bus Interface Unit's sequencing through T-states. Timer operation and bus interface operation are asynchronous. This time-multiplexed scheme results in a delay of 2½ to 6½ CLKOUT periods from timer input to timer output.

Each timer keeps its own running count and has a user-defined maximum count value. Timers 0 and 1 can use one maximum count value (single maximum count mode) or two alternating maximum count values (dual maximum count mode). Timer 2 can use only one maximum count value. The control register for each timer determines the counting mode to be used. When a timer is serviced, its present count value is incremented and compared to the maximum count for that timer. If these two values match, the count value resets to zero. The timers can be configured either to stop after a single cycle or to run continuously.

Timers 0 and 1 are functionally identical. Figure 9-3 illustrates their operation. Each has a latched, synchronized input pin and a single output pin. Each timer can be clocked internally or externally. Internally, the timer can either increment at ¼ CLKOUT frequency or be prescaled by Timer 2. A timer that is prescaled by Timer 2 increments when Timer 2 reaches its maximum count value.

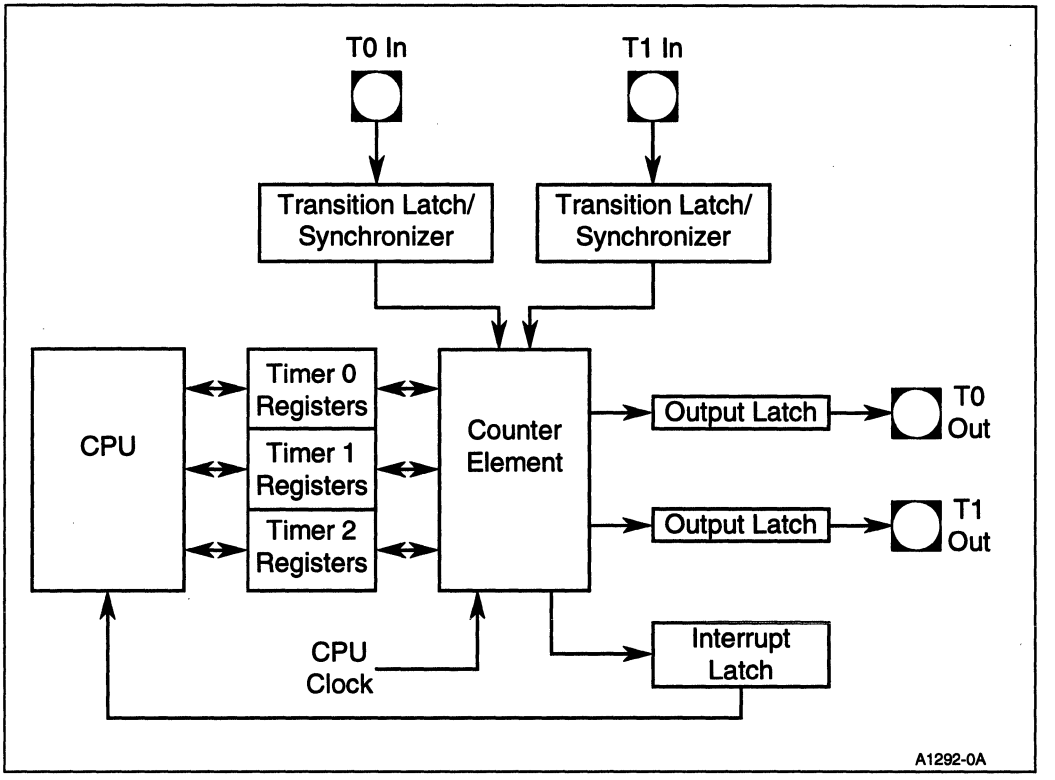
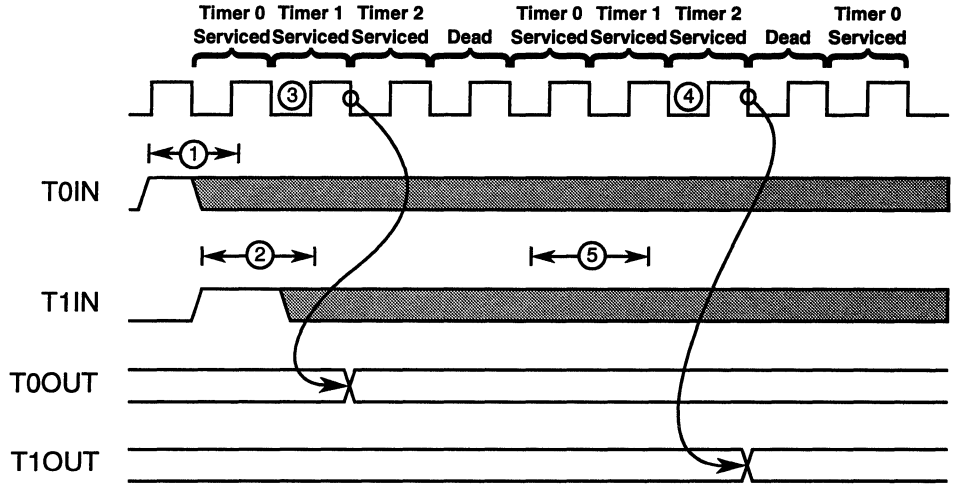


Figure 9-1. Timer/Counter Unit Block Diagram



NOTES:
1. TOIN resolution time (setup time met).
2. T1IN resolution time (setup time not met).
3. Modified count value written into Timer 0 count register.
4. T1IN resolution time, count value written into Timer 1 count register.
5. T1IN resolution time.

A1293-0A

Figure 9-2. Counter Element Multiplexing and Timer Input Synchronization

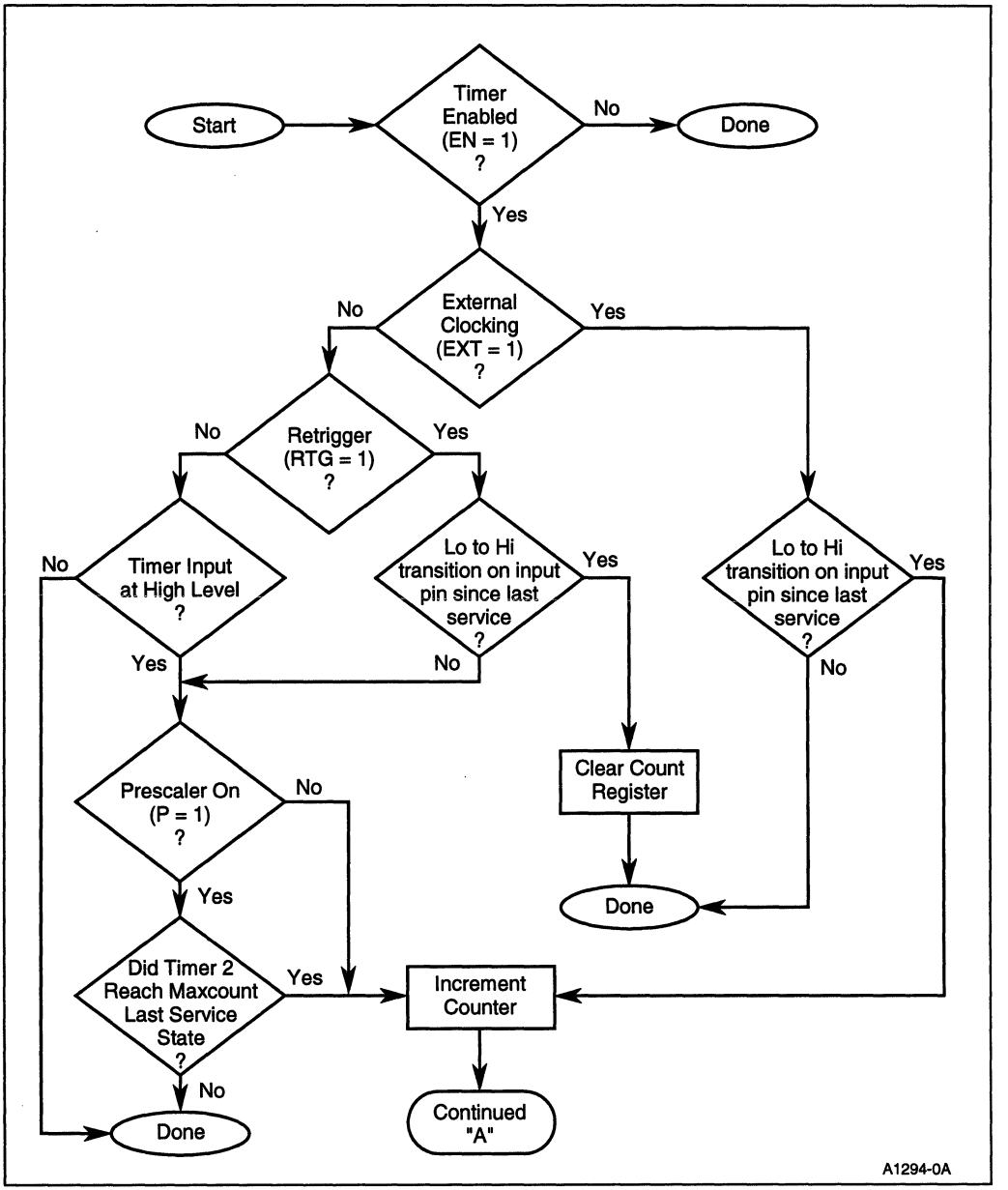


Figure 9-3. Timers 0 and 1 Flow Chart

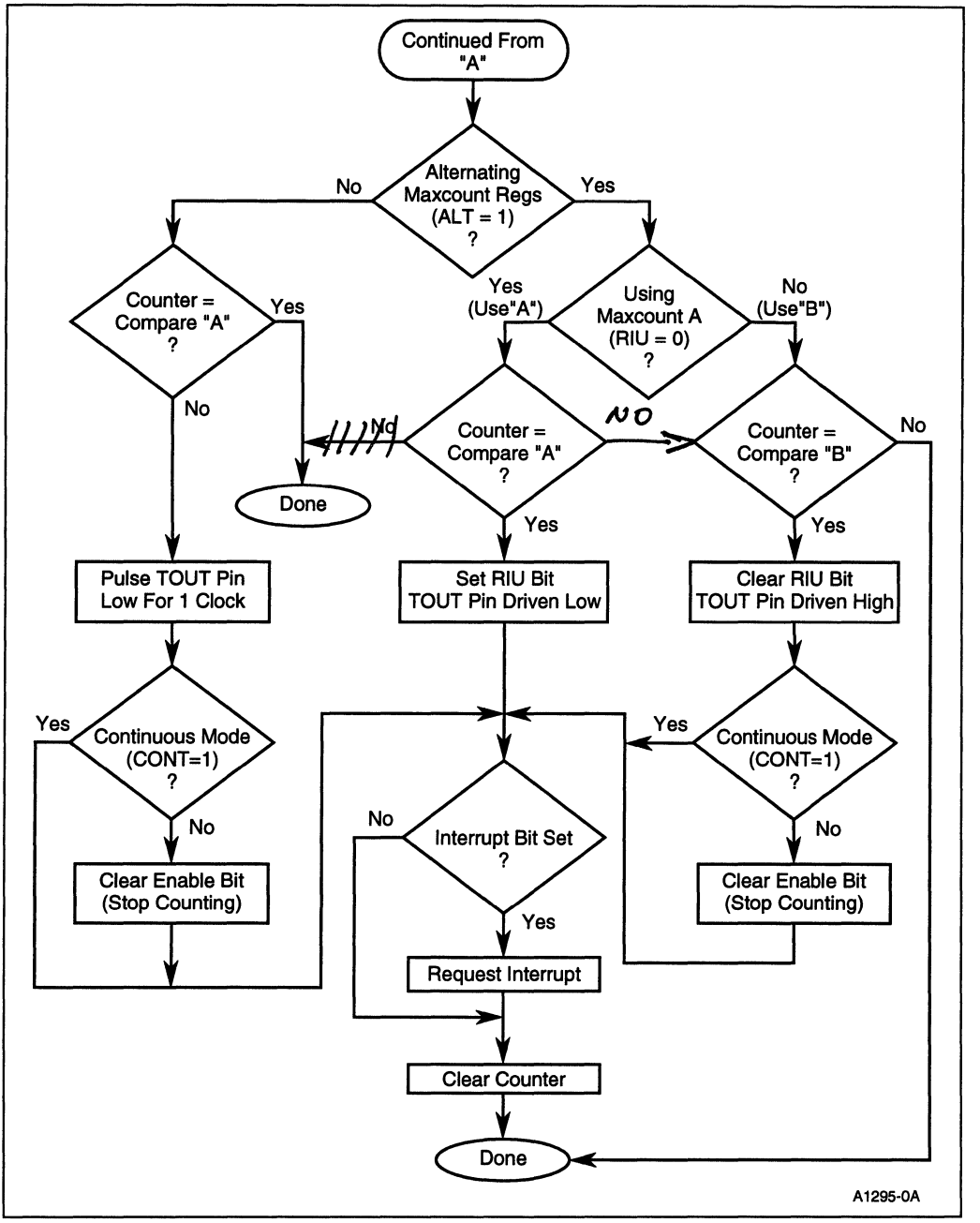


Figure 9-3. Timers 0 and 1 Flow Chart (Continued)

(12/14/95 ERRATA)

When configured for internal clocking, the Timer/Counter Unit uses the input pins either to enable timer counting or to retrigger the associated timer. Externally, a timer increments on low-to-high transitions on its input pin (up to ¼ CLKOUT frequency).

Timers 0 and 1 each have a single output pin. Timer output can be either a single pulse, indicating the end of a timing cycle, or a variable duty cycle wave. These two output options correspond to single maximum count mode and dual maximum count mode, respectively (Figure 9-4). Interrupts can be generated at the end of every timing cycle.

Timer 2 has no input or output pins and can be operated only in single maximum count mode (Figure 9-4). It can be used as a free-running clock and as a prescaler to Timers 0 and 1. Timer 2 can be clocked only internally, at ¼ CLKOUT frequency. Timer 2 can also generate interrupts at the end of every timing cycle.

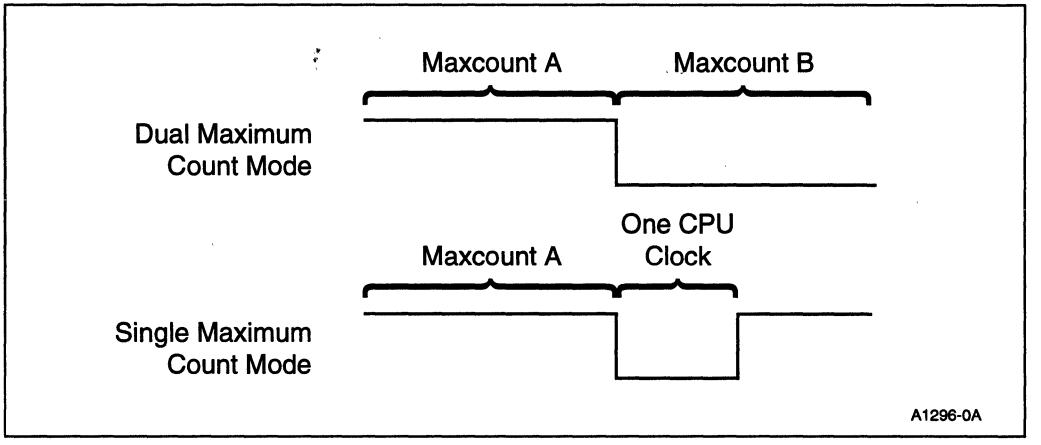


Figure 9-4. Timer/Counter Unit Output Modes

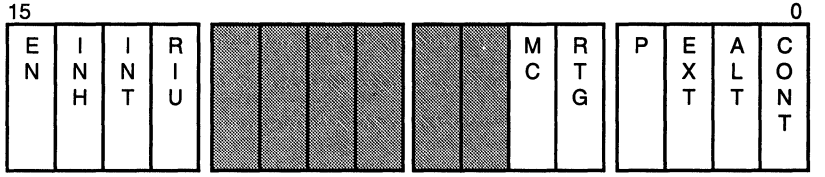
9.2 PROGRAMMING THE TIMER/COUNTER UNIT

Each timer has three registers: a Timer Control register (Figure 9-5 and Figure 9-6), a Timer Count register (Figure 9-7) and a Timer Maxcount Compare register (Figure 9-8). Timers 0 and 1 also have access to an additional Maxcount Compare register. The Timer Control register controls timer operation. The Timer Count register holds the current timer count value, and the Maxcount Compare register holds the maximum timer count value.

Register Name: Timer 0 and 1 Control Registers

Register Mnemonic: T0CON, T1CON

Register Function: Defines Timer 0 and 1 operation.



A1297-0A

Bit Mnemonic	Bit Name	Reset State	Function
EN	Enable	0	Set to enable the timer. This bit can be written only when the INH bit is set.
INH	Inhibit	X	Set to enable writes to the EN bit. Clear to ignore writes to the EN bit. The INH bit is not stored; it always reads as zero.
INT	Interrupt	X	Set to generate an interrupt request when the Count register equals a Maximum Count register. Clear to disable interrupt requests.
RIU	Register In Use	X	Indicates which compare register is in use. When set, the current compare register is Maxcount Compare B; when clear, it is Maxcount Compare A.
MC	Maximum Count	X	This bit is set when the counter reaches a maximum count. **The MC bit must be cleared by writing to the Timer Control register. This is not done automatically.** If MC is clear, the counter has not reached a maximum count.

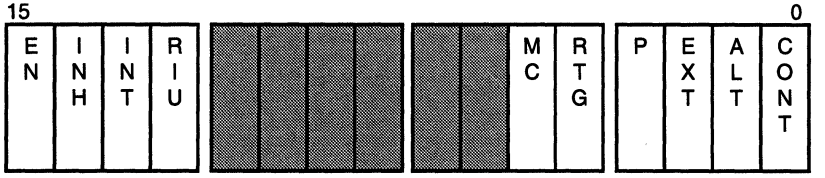
NOTE: Reserved register bits are shown with gray shading. Reserved bits must be written to a logic zero to ensure compatibility with future Intel products.

Figure 9-5. Timer 0 and Timer 1 Control Registers

Register Name: Timer 0 and 1 Control Registers

Register Mnemonic: T0CON, T1CON

Register Function: Defines Timer 0 and 1 operation.



A1297-0A

Bit Mnemonic	Bit Name	Reset State	Function
RTG	Retrigger	X	This bit specifies the action caused by a low-to-high transition on the TMR INx input. Set RTG to reset the count; clear RTG to enable counting. This bit is ignored with external clocking (EXT=1).
P	Prescaler	X	Set to increment the timer when Timer 2 reaches its maximum count. Clear to increment the timer at ¼ CLKOUT. This bit is ignored with external clocking (EXT=1).
EXT	External Clock	X	Set to use external clock; clear to use internal clock. The RTG and P bits are ignored with external clocking (EXT set).
ALT	Alternate Compare Register	X	This bit controls whether the timer runs in single or dual maximum count mode (see Figure 9-4 on page 9-6). Set to specify dual maximum count mode; clear to specify single maximum count mode.
CONT	Continuous Mode	X	Set to cause the timer to run continuously. Clear to disable the counter (clear the EN bit) after each counting sequence.

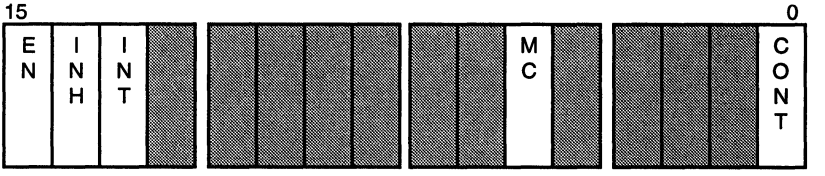
NOTE: Reserved register bits are shown with gray shading. Reserved bits must be written to a logic zero to ensure compatibility with future Intel products.

Figure 9-5. Timer 0 and Timer 1 Control Registers (Continued)

Register Name: Timer 2 Control Register

Register Mnemonic: T2CON

Register Function: Defines Timer 2 operation.



A1298-0A

Bit Mnemonic	Bit Name	Reset State	Function
EN	Enable	0	Set to enable the timer. This bit can be written only when the INH bit is set.
INH	Inhibit	X	Set to enable writes to the EN bit. Clear to ignore writes to the EN bit. The INH bit is not stored; it always reads as zero.
INT	Interrupt	X	Set to generate an interrupt request when the Count register equals a Maximum Count register. Clear to disable interrupt requests.
MC	Maximum Count	X	This bit is set when the counter reaches a maximum count. **The MC bit must be cleared by writing to the Timer Control register. This is not done automatically.** If MC is clear, the counter has not reached a maximum count.
CONT	Continuous Mode	X	Set to cause the timer to run continuously. Clear to disable the counter (clear the EN bit) after each counting sequence.

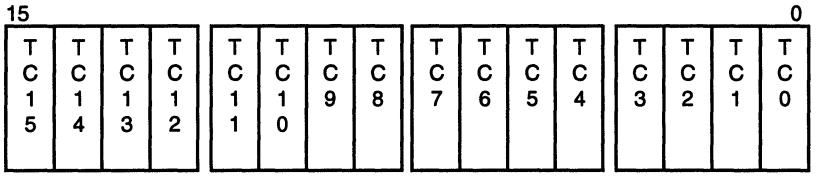
NOTE: Reserved register bits are shown with gray shading. Reserved bits must be written to a logic zero to ensure compatibility with future Intel products.

Figure 9-6. Timer 2 Control Register

Register Name: Timer Count Register

Register Mnemonic: T0CNT, T1CNT, T2CNT

Register Function: Contains the current timer count.



A1299-0A

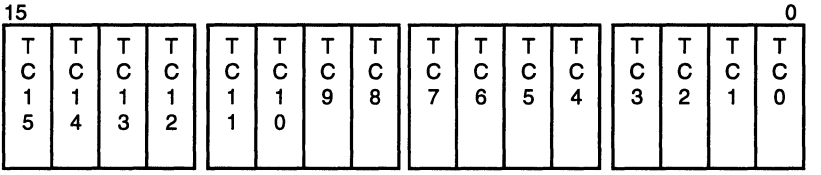
Bit Mnemonic	Bit Name	Reset State	Function
TC15:0	Timer Count Value	XXXXH	Contains the current count of the associated timer.

Figure 9-7. Timer Count Registers

Register Name: Timer Maxcount Compare Register

Register Mnemonic: T0CMPA, T0CMPB, T1CMPA, T1CMPB, T2CMPA

Register Function: Contains timer maximum count value.



Bit Mnemonic	Bit Name	Reset State	Function
TC15:0	Timer Compare Value	XXXXH	Contains the maximum value a timer will count to before resetting its Count register to zero.

Figure 9-8. Timer Maxcount Compare Registers

9.2.1 Initialization Sequence

When initializing the Timer/Counter Unit, the following sequence is suggested:

1. If timer interrupts will be used, program interrupt vectors into the Interrupt Vector Table.
2. **Clear the Timer Count register.** This must be done before the timer is enabled because the count register is undefined at reset. Clearing the count register ensures that counting begins at zero.
3. Write the desired maximum count value to the Timer Maxcount Compare register. For dual maximum count mode, write a value to both Maxcount Compare A and B.
4. Program the Timer Control register to enable the timer. When using Timer 2 to prescale another timer, enable Timer 2 last. If Timer 2 is enabled first, it will be at an unknown point in its timing cycle when the timer to be prescaled is enabled. This results in an unpredictable duration of the first timing cycle for the prescaled timer.

9.2.2 Clock Sources

The 16-bit Timer Count register increments once for each timer event. A timer event can be a low-to-high transition on a timer input pin (Timers 0 and 1), a pulse generated every fourth CPU clock (all timers) or a timeout of Timer 2 (Timers 0 and 1). Up to 65536 (2^{16}) events can be counted.

Timers 0 and 1 can be programmed to count low-to-high transitions on their input pins as timer events by setting the External (EXT) bit in their control registers. Transitions on the external pin are synchronized to the CPU clock before being presented to the timer circuitry. The timer counts **transitions** on this pin. The input signal must go low, then high, to cause the timer to increment. The maximum count-rate for the timers is ¼ the CPU clock rate (measured at CLKOUT) because the timers are serviced only once every four clocks.

All timers can use transitions of the CPU clock as timer events. For internal clocking, the timer increments every fourth CPU clock due to the counter element's time-multiplexed servicing scheme. Timer 2 can use only the internal clock as a timer event.

Timers 0 and 1 can also use Timer 2 reaching its maximum count as a timer event. In this configuration, Timer 0 or Timer 1 increments each time Timer 2 reaches its maximum count. See Table 9-1 for a summary of clock sources for Timers 0 and 1. Timer 2 must be initialized and running in order to increment values in other timer/counters.

Table 9-1. Timer 0 and 1 Clock Sources

EXT	P	Clock Source
0	0	Timer clocked internally at ¼ CLKOUT frequency.
0	1	Timer clocked internally, prescaled by Timer 2.
1	X	Timer clocked externally at up to ¼ CLKOUT frequency.

9.2.3 Counting Modes

All timers have a Timer Count register and a Maxcount Compare A register. Timers 0 and 1 also have access to a second Maxcount Compare B register. Whenever the contents of the Timer Count register equal the contents of the Maxcount Compare register, the count register resets to zero. The maximum count value will never be stored in the count register. This is because the counter element increments, compares and resets a timer in one clock cycle. Therefore, the maximum value is never written back to the count register. The Maxcount Compare register can be written at any time during timer operation.

The timer counting from its initial count (usually zero) to its maximum count (either Maxcount Compare A or B) and resetting to zero defines one timing cycle. A Maxcount Compare value of 0 implies a maximum count of 65536, a Maxcount Compare value of 1 implies a maximum count of 1, etc.

Only equivalence between the Timer Count and Maxcount Compare registers is checked. The count does not reset to zero if its value is greater than the maximum count. If the count value exceeds the Maxcount Compare value, the timer counts to 0FFFFH, increments to zero, then counts to the value in the Maxcount Compare register. Upon reaching a maximum count value, the Maximum Count (MC) bit in the Timer Control register sets. **The MC bit must be cleared by writing to the Timer Control register. This is not done automatically.**

The Timer/Counter Unit can be configured to execute different counting sequences. The timers can operate in single maximum count mode (all timers) or dual maximum count mode (Timers 0 and 1 only). They can also be programmed to run continuously in either of these modes. The Alternate (ALT) bit in the Timer Control register determines the counting modes used by Timers 0 and 1.

All timers can use single maximum count mode, where only Maxcount Compare A is used. The timer will count to the value contained in Maxcount Compare A and reset to zero. Timer 2 can operate only in this mode.

Timers 0 and 1 can also use dual maximum count mode. In this mode, Maxcount Compare A and Maxcount Compare B are both used. The timer counts to the value contained in Maxcount Compare A, resets to zero, counts to the value contained in Maxcount Compare B, and resets to zero again. The Register In Use (RIU) bit in the Timer Control register indicates which Maxcount Compare register is currently in use.

The timers can be programmed to run continuously in single maximum count and dual maximum count modes. The Continuous (CONT) bit in the Timer Control register determines whether a timer is disabled after a single counting sequence.

9.2.3.1 Retriggering

The timer input pins affect timer counting in three ways (see Table 9-2). The programming of the External (EXT) and Retrigger (RTG) bits in the Timer Control register determines how the input signals are used. When the timers are clocked internally, the RTG bit determines whether the input pin enables timer counting or retriggers the current timing cycle.

Table 9-2. Timer Retriggering

EXT	RTG	Timer Operation
0	0	Timer counts internal events, if input pin remains high.
0	1	Timer counts internal events; count resets to zero on every low-to-high transition on the input pin.
1	X	Timer input acts as clock source.

When the EXT and RTG bits are clear, the timer counts internal timer events. In this mode, the input is level-sensitive, not edge-sensitive. A low-to-high transition on the timer input is not required for operation. The input pin acts as an external enable. If the input is high, the timer will count through its sequence, provided the timer remains enabled.

When the EXT bit is clear and the RTG bit is set, every low-to-high transition on the timer input pin causes the Count register to reset to zero. After the timer is enabled, counting begins only after the first low-to-high transition on the input pin. If another low-to-high transition occurs before the end of the timer cycle, the timer count resets to zero and the timer cycle begins again. In dual maximum count mode, the Register In Use (RIU) bit does not clear when a low-to-high transition occurs. For example, if the timer retriggers while Maxcount Compare B is in use, the timer resets to zero and counts to maximum count B before the RIU bit clears. **In dual maximum count mode, the timer retriggering extends the use of the current Maxcount Compare register.**

9.2.4 Pulsed and Variable Duty Cycle Output

Timers 0 and 1 each have an output pin that can perform two functions. First, the output can be a single pulse, indicating the end of a timing cycle (single maximum count mode). Second, the output can be a level, indicating the Maxcount Compare register currently in use (dual maximum count mode). The output occurs one clock after the counter element services the timer when the maximum count is reached (see Figure 9-9).

With external clocking, the time between a transition on a timer input and the corresponding transition of the timer output varies from 2½ to 6½ clocks. This delay occurs due to the time-multiplexed servicing scheme of the Timer/Counter Unit. The exact timing depends on when the input occurs relative to the counter element's servicing of the timer. Figure 9-2 on page 9-3 shows the two extremes in timer output delay. Timer 0 demonstrates the best possible case, where the input occurs immediately before the timer is serviced. Timer 1 demonstrates the worst possible case, where the input is latched, but the setup time is not met and the input is not recognized until the counter element services the timer again.

In single maximum count mode, the timer output pin goes low for one CPU clock period (see Figure 9-4 on page 9-6). This occurs when the count value equals the Maxcount Compare A value. If programmed to run continuously, the timer generates periodic pulses.

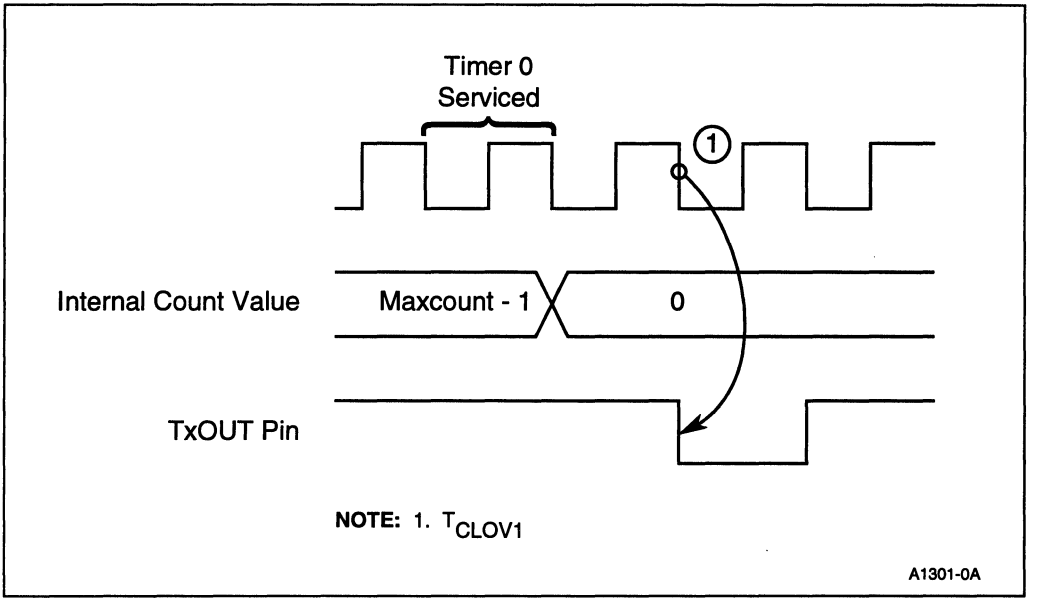


Figure 9-9. TxOUT Signal Timing

In dual maximum count mode, the timer output pin indicates which Maxcount Compare register is currently in use. A low output indicates Maxcount Compare B, and a high output indicates Maxcount Compare A (see Figure 9-4 on page 9-6). If programmed to run continuously, a repetitive waveform can be generated. For example, if Maxcount Compare A contains 10, Maxcount Compare B contains 20, and CLKOUT is 12.5 MHz, the timer generates a 33 percent duty cycle waveform at 104 KHz. The output pin always goes high at the end of the counting sequence (even if the timer is not programmed to run continuously).

9.2.5 Enabling/Disabling Counters

Each timer has an Enable (EN) bit in its Control register to allow or prevent timer counting. The Inhibit ($\overline{INH}$) bit controls write accesses to the EN bit. Timers 0 and 1 can be programmed to use their input pins as enable functions also. If a timer is disabled, the count register does not increment when the counter element services the timer.

The Enable bit can be altered by programming or the timers can be programmed to disable themselves at the end of a counting sequence with the Continuous (CONT) bit. If the timer is not programmed for continuous operation, the Enable bit automatically clears at the end of a counting sequence. In single maximum count mode, this occurs after Maxcount Compare A is reached. In dual maximum count mode, this occurs after Maxcount Compare B is reached (Timers 0 and 1 only).

The input pins for Timers 0 and 1 provide an alternate method for enabling and disabling timer counting. When using internal clocking, the input pin can be programmed either to enable the timer or to reset the timer count, depending on the state of the Retrigger (RTG) bit in the control register. When used as an enable function, the input pin either allows (input high) or prevents (input low) timer counting. To ensure recognition of an input level, it must be valid for four CPU clocks. This is due to the counter element's time-multiplexed servicing scheme for the timers.

9.2.6 Timer Interrupts

All timers can generate internal interrupt requests. Although all three timers share a single interrupt request to the CPU, each has its own vector location and internal priority. Timer 0 has the highest interrupt priority and Timer 2 has the lowest.

Timer Interrupts are enabled or disabled by the Interrupt (INT) bit in the Timer Control register. If enabled, an interrupt is generated every time a maximum count value is reached. In dual maximum count mode, an interrupt is generated each time the value in Maxcount Compare A or Maxcount Compare B is reached. If the interrupt is disabled after a request has been generated, but before a pending interrupt is serviced, the interrupt request remains active (the Interrupt Controller latches the request). If a timer generates a second interrupt request before the CPU services the first interrupt request, the first request is lost.

9.2.7 Programming Considerations

Timer registers can be read or written whether the timer is operating or not. Since processor accesses to timer registers are synchronized with counter element accesses, a half-modified count register will never be read.

When Timer 0 and Timer 1 use an internal clock source, the input pin must be high to enable counting.

9.3 TIMING

Certain timing considerations need to be made with the Timer/Counter Unit. These include input setup and hold times, synchronization and operating frequency.

9.3.1 Input Setup and Hold Timings

To ensure recognition, setup and hold times must be met with respect to CPU clock edges. The timer input signal must be valid T_{CHIS} before the rising edge of CLKOUT and must remain valid T_{CHIH} after the same rising edge. If these timing requirements are not met, the input will not be recognized until the next clock edge.

9.3.2 Synchronization and Maximum Frequency

All timer inputs are latched and synchronized with the CPU clock. Because of the internal logic required to synchronize the external signals, and the multiplexing of the counter element, the Timer/Counter Unit can operate only up to ¼ of the CLKOUT frequency. Clocking at greater frequencies will result in missed clocks.

9.3.2.1 Timer/Counter Unit Application Examples

The following examples are possible applications of the Timer/Counter Unit. They include a real-time clock, a square wave generator and a digital one-shot.

9.3.3 Real-Time Clock

Example 9-1 contains sample code to configure Timer 2 to generate an interrupt request every 10 milliseconds. The CPU then increments memory-based clock variables.

9.3.4 Square-Wave Generator

A square-wave generator can be useful to act as a system clock tick. Example 9-2 illustrates how to configure Timer 1 to operate this way.

9.3.5 Digital One-Shot

Example 9-3 configures Timer 1 to act as a digital one-shot.

```
$mod186
name example_80186_family_timer_code

;FUNCTION:   This function sets up the timer and interrupt controller
;            to cause the timer to generate an interrupt every
;            10 milliseconds and to service interrupts to
;            implement a real time clock.
;
;            Timer 2 is used in this example because no input or
;            output signals are required.
;
;SYNTAX:     extern void far set_time(hour, minute, second, T2Compare)
;
;INPUTS:     hour - hour to set time to.
;            minute - minute to set time to.
;            second - second to set time to.
;            T2Compare - T2CMPA value (see note below)
;
;OUTPUTS:    None

;NOTE:       Parameters are passed on the stack as required by
;            high-level languages
;
;            For a CLKOUT of 16Mhz,
;
;            f(timer2)              = 16Mhz/4
;                                   = 4Mhz
;                                   = 0.25us for T2CMPA = 1
;
;            T2CMPA(10ms)           = 10ms/0.25us
;                                   = 10e-3/0.25e-6
;                                   = 40000

;substitute register offsets

     T2CON   equ xxxxh             ;Timer 2 Control register
     T2CMPA  equ xxxxh             ;Timer 2 Compare register
     T2CNT   equ xxxxh             ;Timer 2 Counter register
     TCUCON  equ xxxxh             ;Int. Control register
     EOI     equ xxxxh             ;End Of Interrupt register
     INTSTS  equ xxxxh             ;Interrupt Status register
timer_2_int  equ 19                ;timer 2:vector type 19

data segment public 'data'

     public _hour, _minute, _second, _msec

     _hour       db ?
     _minute     db ?
     _second     db ?
     _msec       db ?

data ends
```

Example 9-1. Configuring a Real-Time Clock

```
lib_80186 segment public 'code'
     assume cs:lib_80186, ds:data

public _set_time
_set_time proc far

     push    bp                    ;save caller's bp
     mov     bp, sp                ;get current top of stack

     hour    equ word ptr[bp+6]    ;get parameters off stack
     minute  equ word ptr[bp+8]
     second  equ word ptr[bp+10]
     T2Compare equ word ptr[bp+12]

     push    ax                    ;save registers used
     push    dx
     push    si

     push    ds
     xor     ax, ax                ;set interrupt vector
     mov     ds, ax
     mov     si, 4*timer_2_int
     mov     word ptr ds:[si], offset

timer_2_interrupt_routine
     inc     si
     inc     si
     mov     ds:[si], cs
     pop     ds

     mov     ax, hour              ;set time
     mov     _hour, al
     mov     ax, minute
     mov     _minute, al
     mov     ax, second
     mov     _second, al
     mov     _msec, 0

     mov     dx, T2CNT             ;clear Count register
     xor     ax, ax
     out     dx, al

     mov     dx, T2CMPA            ;set maximum count value
     mov     ax, T2Compare         ;see note in header above
     out     dx, al
     mov     dx, T2CON             ;set up the control word:
     mov     ax, 0E001H            ;enable counting,
     out     dx, al                ;generate interrupt on MC,
                                   ;continuous counting

     mov     dx, TCUCON            ;set up interrupt controller
     xor     ax, ax                ;unmask highest priority interrupt
     out     dx, al
```

Example 9-1. Configuring a Real-Time Clock (Continued)

```
        sti                             ;enable interrupts

        pop     si                      ;restore saved registers
        pop     dx
        pop     ax
        pop     bp                      ;restore caller's bp
        ret
_set_time endp

timer_2_interrupt_routine proc far

        push    ax                      ;save registers used
        push    dx
        cmp     _msec, 99               ;has 1 sec passed?
        jae     bump_second             ;if above or equal...
        inc     _msec
        jmp     short reset_int_ctl

bump_second:
        mov     _msec, 0                ;reset millisecond
        cmp     _minute, 59             ;has 1 minute passed?
        jae     bump_minute
        inc     _second
        jmp     short reset_int_ctl

bump_minute:
        mov     _second, 0              ;reset second
        cmp     _minute, 59             ;has 1 hour passed?
        jae     bump_hour
        inc     _minute
        jmp     short reset_int_ctl

bump_hour:
        mov     _minute, 0              ;reset minute
        cmp     _hour, 12               ;have 12 hours passed?
        jae     reset_hour
        inc     _hour
        jmp     reset_int_ctl

reset_hour:
        mov     _hour, 1                ;reset hour

reset_int_ctl:
        mov     dx, EOI
        mov     ax, 8000h               ;non-specific end of interrupt
        out     dx, al
        pop     dx
        pop     ax
        iret

timer_2_interrupt_routine endp

lib_80186    ends
             end
```

Example 9-1. Configuring a Real-Time Clock (Continued)

```
$mod186
name          example_timer1_square_wave_code

;FUNCTION:  This function generates a square wave of given
;           frequency and duty cycle on Timer 1 output pin.
;
;  SYNTAX:  extern void far clock(int mark, int space)
;
;  INPUTS:  mark  - This is the mark (1) time.
;           space - This is the space (0) time.
;
;           The register compare value for a given time can be
;           easily calculated from the formula below.
;
;           CompareValue = (req_pulse_width*f)/4
;
; OUTPUTS:  None
;
;    NOTE:  Parameters are passed on the stack as required by
;           high-level Languages

     T1CMPA equ xxxxH              ;substitute register offsets
     T1CMPB equ xxxxH
     T1CNT  equ xxxxH
     T1CON  equ xxxxH

lib_80186   segment public 'code'
     assume cs:lib_80186

public      _clock
_clock      proc far

     push   bp                     ;save caller's bp
     mov    bp, sp                 ;get current top of stack
     _space equ word ptr[bp+6]     ;get parameters off the stack
     _mark  equ word ptr[bp+8]

     push   ax                     ;save registers that will be
     push   bx                     ;modified
     push   dx

     mov    dx, T1CMPA             ;set mark time
     mov    ax, _mark
     out    dx, al

     mov    dx, T1CMPB             ;set space time
     mov    ax, _space
     out    dx, al

     mov    dx, T1CNT              ;Clear Timer 1 Counter
     xor    ax, ax
     out    dx, al

     mov    dx, T1CON              ;start Timer 1
     mov    ax, C003H
     out    dx, al
```

Example 9-2. Configuring a Square-Wave Generator

```
     pop     dx                    ;restore saved registers
     pop     bx
     pop     ax

     pop     bp                    ;restore caller's bp
     ret
_clock       endp
lib_80186    ends
     end
```

Example 9-2. Configuring a Square-Wave Generator (Continued)

```
$mod186
name example_timer1_1_shot_code

; FUNCTION:  This function generates an active-low one-shot pulse
;            on Timer 1 output pin.
;
; SYNTAX:    extern void far one_shot(int CMPB);
;
; INPUTS:    CMPB - This is the T1CMPB value required to generate a
;            pulse of a given pulse width. This value is calculated
;            from the formula below.
;
;            CMPB = (req_pulse_width*f)/4
;
; OUTPUTS:   None
;
; NOTE:      Parameters are passed on the stack as required by
;            high-level languages

     T1CNT   equ xxxxH             ;substitute register offsets
     T1CMPA  equ xxxxH
     T1CMPB  equ xxxxH
     T1CON   equ xxxxH
     MaxCount equ 0020H

lib_80186    segment public 'code'
     assume  cs:lib_80186

public       _one_shot
_one_shot    proc far

     push    bp                    ;save caller's bp
     mov     bp, sp                ;get current top of stack
```

Example 9-3. Configuring a Digital One-Shot

```
     _CMPB   equ word ptr[bp+6]     ;get parameter off the stack

     push    ax                     ;save registers that will be
     push    dx                     ;modified
     mov     dx, T1CNT              ;Clear Timer 1 Counter
     xor     ax, ax
     out     dx, al
     mov     dx, T1CMPA             ;set time before t_shot to 0
     mov     ax, 1
     out     dx, al
     mov     dx, T1CMPB             ;set pulse time
     mov     ax, _CMPB
     out     dx, al
     mov     dx, T1CON
     mov     ax, C002H              ;start Timer 1
     out     dx, al

CountDown:  in ax, dx               ;read in T1CON
     test    ax, MaxCount           ;max count occurred?
     jz      CountDown              ;no: then wait
     and     ax, not MaxCount       ;clear max count bit
     out     dx, al                 ;update T1CON

     pop     dx                     ;restore saved registers
     pop     ax
     pop     bp                     ;restore caller's bp
     ret
_one_shot    endp
lib_80186    ends
     end
```

Example 9-3. Configuring a Digital One-Shot (Continued)

intel®

10

Serial Communications Unit

intel

CHAPTER 10
SERIAL COMMUNICATIONS UNIT

10.1 INTRODUCTION

The Serial Communications Unit is composed of two identical serial ports, or channels. Each serial port is independent of the other. This chapter describes the operation of a single serial port.

The serial port implements several industry-standard asynchronous communications protocols, and it readily interfaces to many different processors over a standard serial interface. Several processors and systems can be connected to a common serial bus using a multiprocessor protocol. The serial port also implements a simple synchronous protocol. The synchronous protocol is most commonly used to expand the number of I/O pins with shift registers.

Features:

- Full duplex operation
- Programmable seven, eight or nine data bits in asynchronous mode
- Independent baud rate generator
- Maximum baud rate of 1/16 the processor clock
- Double-buffered transmit and receive
- Clear-to-Send feature for transmission
- Break character transmission and detection
- Programmable even, odd or no parity
- Detects both framing and overrun errors
- Supports interrupt on transmit and receive

10.1.1 Asynchronous Communications

Asynchronous communications protocols allow different devices to communicate without a common reference clock. The devices communicate at a common baud rate, or bits per second. Data is transmitted and received in *frames*. A *frame* is a sequence of bits shifted serially onto or off the communications line.

Each asynchronous frame consists of a start bit (always a logic zero), followed by the data bits and a terminating stop bit. The serial port can transmit and receive seven, eight or nine data bits. The last data bit can optionally be replaced by an even or odd parity bit. Figure 10-1 shows a typical 10-bit frame.

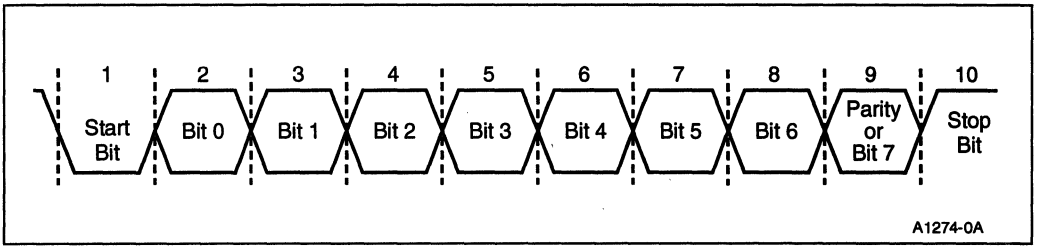


Figure 10-1. Typical 10-Bit Asynchronous Data Frame

When discussing asynchronous communications, it makes sense to talk about the receive machine (RX machine) and the transmit machine (TX machine) separately. Each is completely independent. Transmission and reception can occur simultaneously, making the asynchronous modes full-duplex.

10.1.1.1 RX Machine

The RX machine (Figure 10-2) shifts the received serial data into the receive shift register. When the reception has completed, the data is then moved into the Serial Receive Buffer (SxRBUF) Register. From there, the user can read the received data byte.

The RX machine samples the RXD pin, looking for a logical low (start bit) signifying the beginning of a reception. Once the logical low has been detected, the RX machine begins the receive process. Each expected bit-time is divided into eight samples by the 8X baud clock. The RX machine takes the three middle samples and, based on a two-out-of-three majority, determines the data bit value. This *oversampling* is common for asynchronous serial ports and improves noise immunity. This majority value is then shifted into the receive shift register.

Using this method, the RX machine can tolerate incoming baud rates that differ from its own internal baud rates by 2.5% overspeed and 5.5% underspeed. These limits exceed the CCITT extended signaling rate specifications.

A stop bit is expected by the RX machine after the proper number of data bits. When the stop bit has been validated, the data from the shift register is copied into SxRBUF and the Receive Interrupt (RI) bit is set. Note that the stop bit is actually validated right after its middle three samples are taken. Therefore, the data is moved into SxRBUF and the RI bit is set approximately in the middle of the stop bit time.

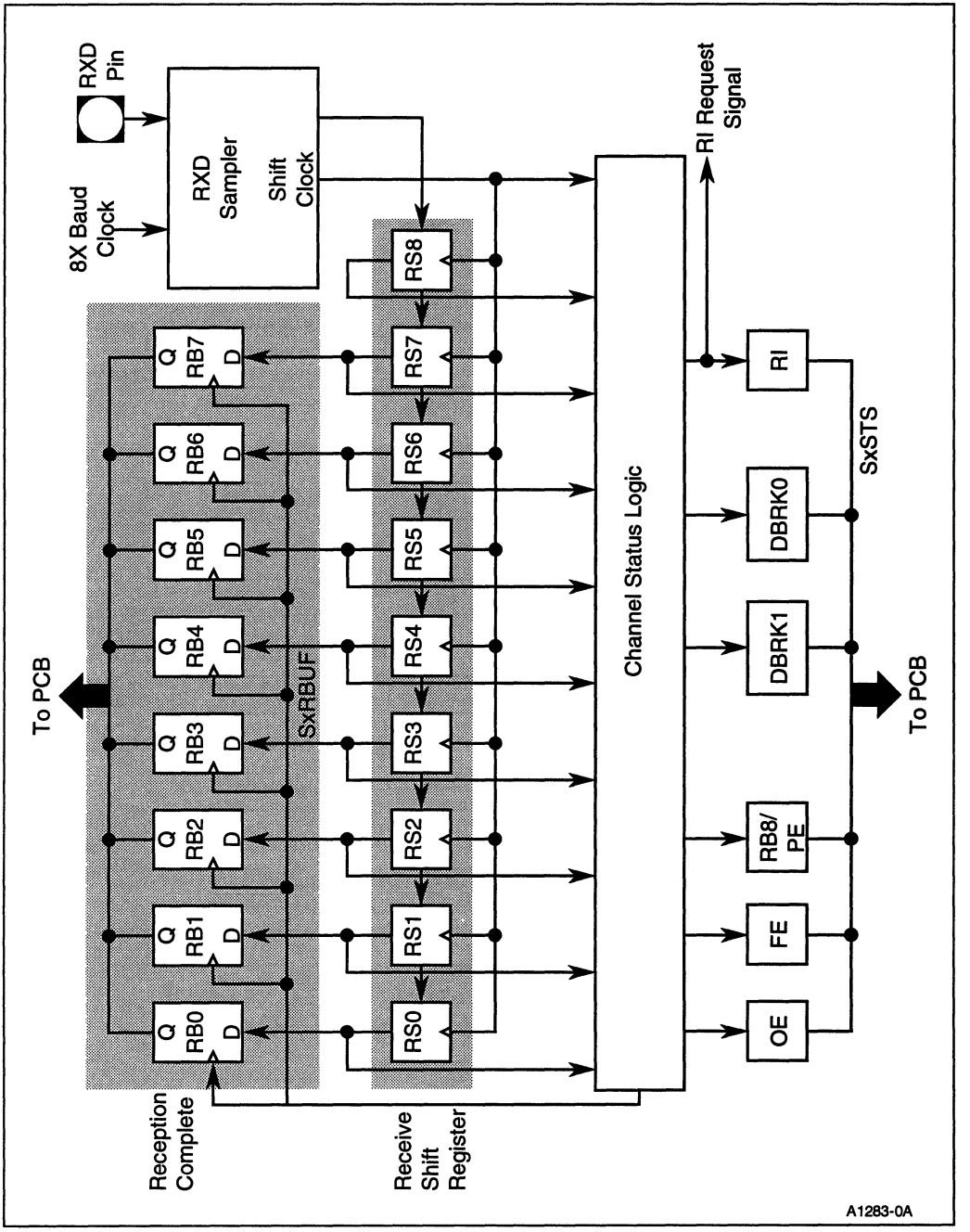


Figure 10-2. RX Machine

The RX machine can detect several error conditions that may occur during reception:

1. Parity errors — A parity error flag is set when the parity of the received data is incorrect.
2. Framing errors — If a valid stop bit is not received when expected by the RX machine, a framing error flag is set.
3. Overrun errors — If SxRBUF is not read before another reception completes, the old data in SxRBUF is overwritten and an overrun error flag is set. This indicates that data from an earlier reception has been lost.

The RX machine also recognizes two different break characters. The shorter break character is M bit times, where M is equal to the total number of bits (start + data + stop) in a frame. The longer break character is 2M + 3 bit times. A break character results in at least one null (all zero) character with a framing error being received. Other error flags could be set depending on the length of the break character and the mode of the serial port.

10.1.1.2 TX Machine

A block diagram of the TX machine is shown in Figure 10-3. The TX machine logic supports the following features:

- parity generation (even, odd or none)
- Clear-to-Send
- break character transmission
- double-buffered operation

A transmission begins by writing a byte to the Serial Transmit Buffer (SxTBUF) Register. SxTBUF is a holding register for the transmit shift register. The contents of SxTBUF are transferred to the transmit shift register as soon as it is empty. If no transmission is in progress (i.e., the transmit shift register is empty), SxTBUF is copied immediately to the transmit shift register. If parity is enabled, the parity bits are calculated and appended to the transmit shift register during the transfer. The start and stop bits are added when the data is transmitted. The Transmit Interrupt bit (TI) is set at the beginning of the stop bit time.

Double buffering is a useful feature of the TX machine. When the transmit shift register is empty, the user can write two sequential bytes to SxTBUF. The first byte is transmitted immediately and the second byte is held in SxTBUF until the first byte has been transmitted.

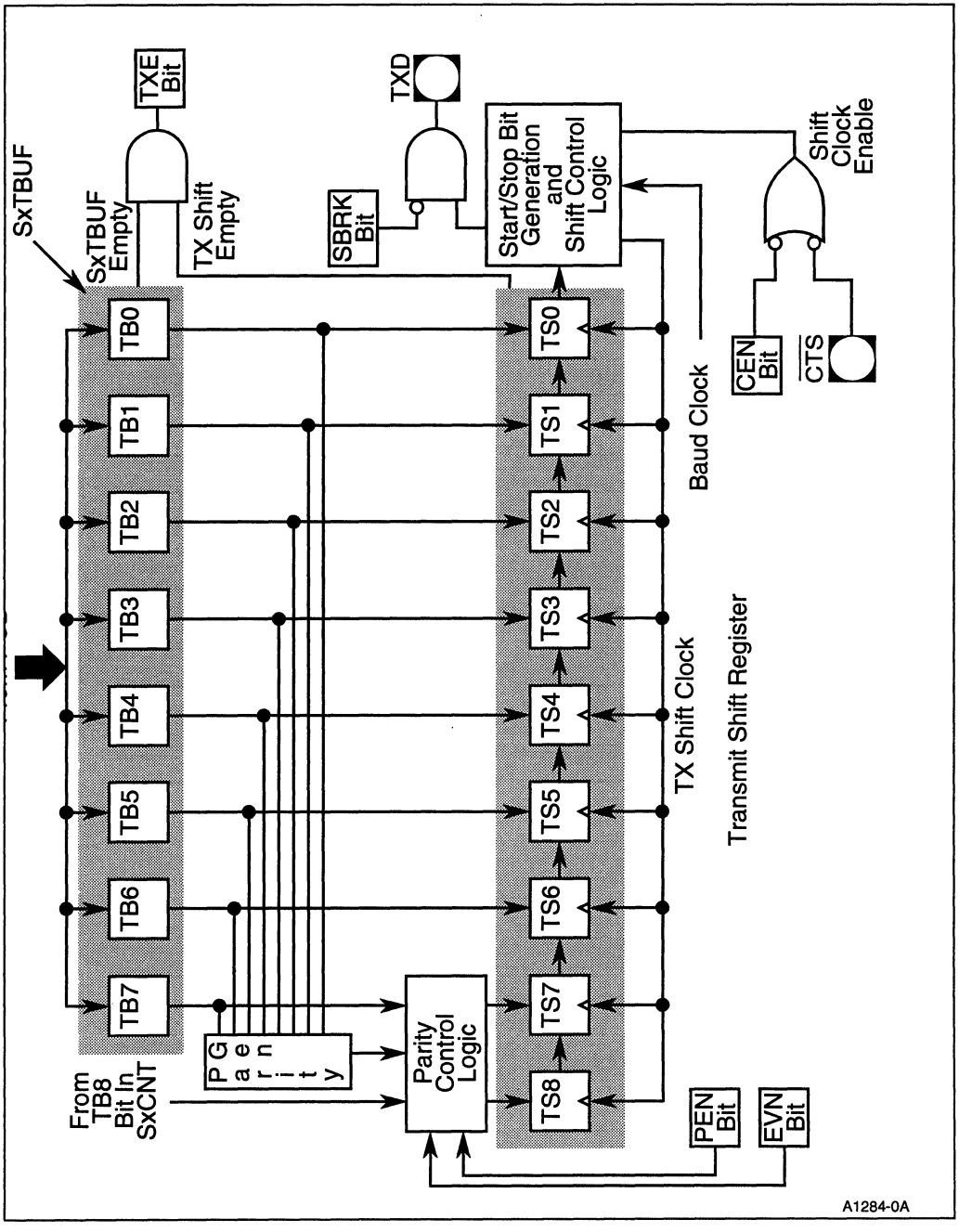


Figure 10-3. TX Machine

The Transmit machine can be disabled by an external source by using the Clear-to-Send feature. When the Clear-to-Send feature is enabled, the TX machine will not transmit until the $\overline{CTS}$ pin is asserted. The $\overline{CTS}$ pin is level sensitive. Asserting the $\overline{CTS}$ pin before a pending transmission for at least 1½ clock cycles ensures that the entire frame will be transmitted. See "$\overline{CTS}$ Pin Timings" on page 10-19 for details.

The TX machine can also transmit a break character. Setting the SBRK bit forces the TXD pin immediately low. The TXD pin remains low until the user clears SBRK. The TX machine will continue the transmission sequence even if SBRK is set. **Use caution** when setting SBRK or characters will be lost.

10.1.1.3 Modes 1, 3 and 4

The three asynchronous modes of the serial ports, Modes 1, 3 and 4, operate in approximately the same manner. Mode 1 is the 8-bit asynchronous communications mode. Each frame consists of a start bit, eight data bits and a stop bit, as shown in Figure 10-4. When parity is used, the eighth data bit becomes the parity bit. Both the RX and TX machines use this frame in Mode 1 with no exceptions.

Mode 3 is the 9-bit asynchronous communications mode (see Figure 10-5). Mode 3 is the same as Mode 1 except that a frame contains nine data bits. The ninth data bit becomes the parity bit when the parity feature is enabled. When parity is disabled, the ninth data bit is controlled by the user. (See "Modes 2 and 3 for Multiprocessor Communications" on page 10-14.) Mode 3 can be used with Mode 2 for multiprocessor communications or alone for "8 data bits + parity" frames.

Mode 4 is the 7-bit asynchronous communications mode. Each frame consists of a start bit, seven data bits and a stop bit, as shown in Figure 10-6. Parity is not available in Mode 4. Both the RX and TX machines use this frame in Mode 4 with no exceptions.

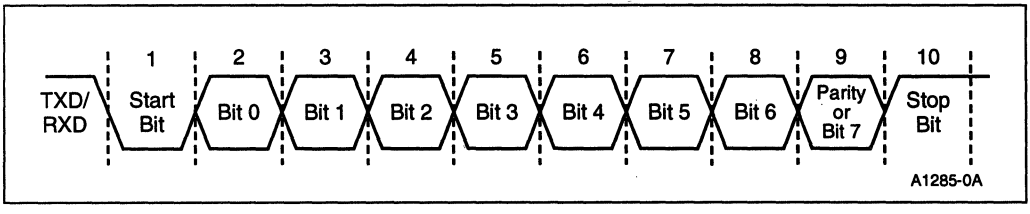


Figure 10-4. Mode 1 Waveform

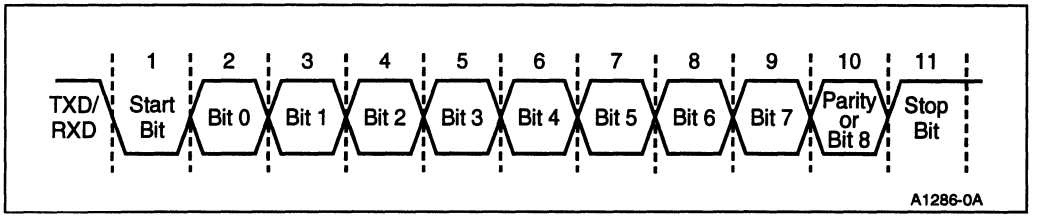


Figure 10-5. Mode 3 Waveform

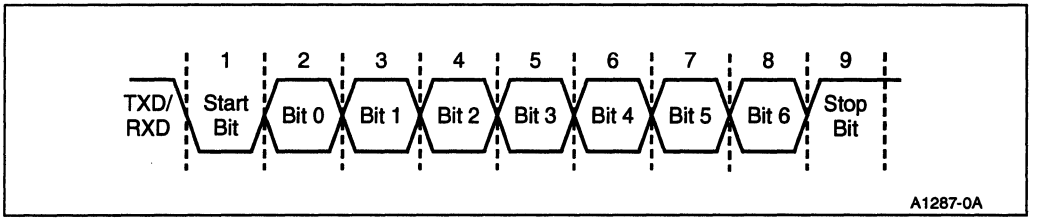


Figure 10-6. Mode 4 Waveform

10.1.1.4 Mode 2

Asynchronous Mode 2 is referred to as the "address recognition mode." Mode 2 is used together with Mode 3 for multiprocessor communications over a common serial link.

In Mode 2, the RX machine will not complete a reception unless the ninth data bit is a one. Any character received with the ninth bit equal to zero is ignored. No flags are set, no interrupts occur and no data is transferred to SxRBUF. In Mode 3, characters are received regardless of the state of the ninth data bit. The following is brief example of using Modes 2 and 3. See "Master/Slave Example" on page 10-28 for more information.

Assume one master serial port connects to multiple slave serial ports over a serial link. The slaves are initially in Mode 2, and the master is always in Mode 3. The master communicates with one slave at a time. The CPU overhead of the serial communications burdens only the master and the target slave device.

1. The master transmits the "address" of the target slave, with the ninth bit set, over the serial link.
2. All slaves receive the character and check whether that address is theirs.
3. The target slave switches to Mode 3; all other slaves remain in Mode 2.
4. The master and the target slave continue the communication with all ninth data bits equal to zero. The other slave devices ignore the activity on the serial link.

5. At the end of the communication, the target slave switches back to Mode 2 and waits for another address.

The parity feature cannot be used when implementing multiprocessor communications with Modes 2 and 3, as the ninth data bit is a control bit and cannot be used as the parity bit.

10.1.2 Synchronous Communications

The synchronous mode (Mode 0) is useful primarily with shift register-based peripheral devices. The device outputs a synchronizing clock on TXD and transmits and receives data on RXD in 8-bit frames (Figure 10-7). The serial port always provides the synchronizing clock; it can never receive a synchronous clock on TXD. Communication in the synchronous mode is half-duplex. The RXD pin cannot transmit and receive data at the same time. Because the serial port always acts as the master in Mode 0, all transmissions and receptions are controlled by the serial port. In Mode 0, the parity functions and break character detection functions are not available.

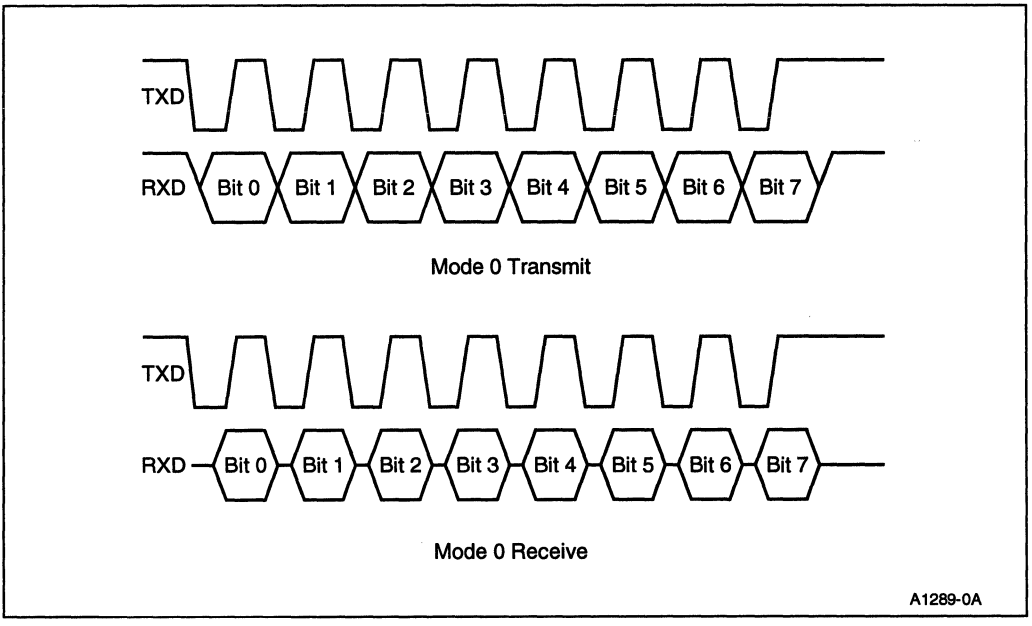


Figure 10-7. Mode 0 Waveforms

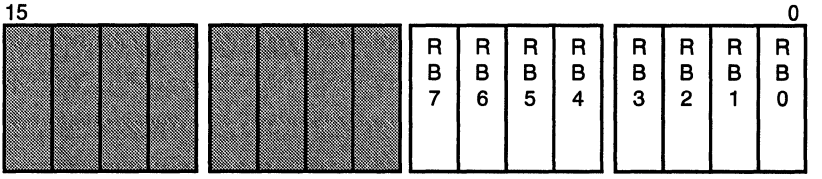
10.2 PROGRAMMING

This section describes how to program the serial port using the appropriate registers. The Serial Receive Buffer Register (SxRBUF) is shown in Figure 10-8 and the Serial Transmit Buffer Register (SxTBUF) is shown in Figure 10-9. These registers have the same functions in any serial port mode.

Register Name: Serial Receive Buffer Register

Register Mnemonic: SxRBUF

Register Function: Received data bytes are stored in SxRBUF.



Bit Mnemonic	Bit Name	Reset State	Function
RB7:0	Received Data	0	Received data byte.

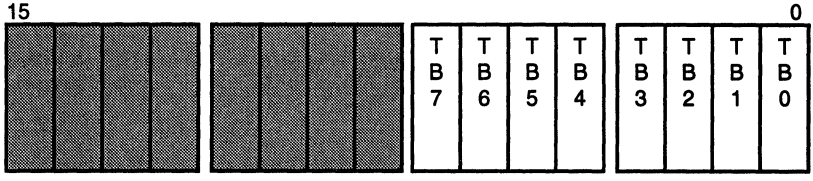
NOTE: Reserved register bits are shown with gray shading. Reserved bits must be written to a logic zero to ensure compatibility with future Intel products.

Figure 10-8. Serial Receive Buffer Register (SxRBUF)

Register Name: Serial Transmit Buffer Register

Register Mnemonic: SxTBUF

Register Function: Bytes are written to SxTBUF to be transmitted.



A1291-0A

Bit Mnemonic	Bit Name	Reset State	Function
TB7:0	Transmit Data Field	0	Data byte to be transmitted.

NOTE: Reserved register bits are shown with gray shading. Reserved bits must be written to a logic zero to ensure compatibility with future Intel products.

Figure 10-9. Serial Transmit Buffer Register (SxTBUF)

10.2.1 Baud Rates

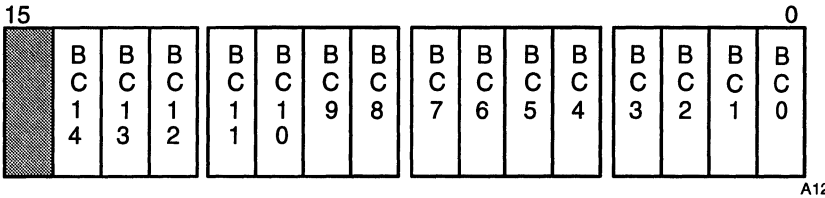
The baud rate generator is composed of a 15-bit counter register (BxCNT) and a 15-bit compare register (BxCMP). BxCNT (Figure 10-10) is a free-running counter that is incremented by the baud timebase clock. The baud timebase clock can be either the internal CPU clock or an external clock applied to the BCLK pin. BxCMP (Figure 10-11) is programmed by the user to determine the baud rate. The most-significant bit of BxCMP (ICLK) selects which source is used as the baud timebase clock.

BxCNT is incremented by the baud timebase clock and compared to BxCMP. When BxCNT and BxCMP are equal, the baud rate generator outputs a pulse and resets BxCNT. This pulse train is the actual baud clock used by the RX and TX machines. The baud clock is eight times the baud rate in the asynchronous modes because of the sampling requirements. The baud clock equals the baud rate in the synchronous mode.

Register Name: Baud Rate Counter Register

Register Mnemonic: BxCNT

Register Function: 15-bit baud rate counter value.



Bit Mnemonic	Bit Name	Reset State	Function
BC14:0	Baud rate counter field	0	Reflects current value of the baud rate counter. **NOTE:** Writing to this register while the serial port is transmitting causes indeterminate operation.

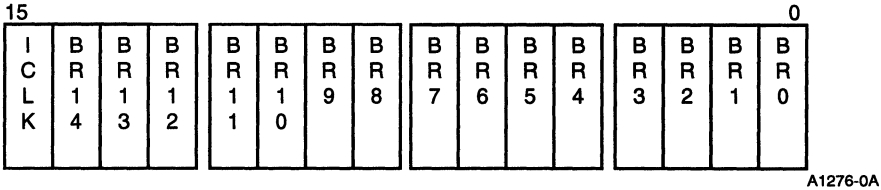
NOTE: Reserved register bits are shown with gray shading. Reserved bits must be written to a logic zero to ensure compatibility with future Intel products.

Figure 10-10. Baud Rate Counter Register (BxCNT)

Register Name: Baud Rate Compare Register

Register Mnemonic: BxCMP

Register Function: Determines baud rate for the serial port.



Bit Mnemonic	Bit Name	Reset State	Function
ICLK	Internal Clocking	0	Selects the input clock: 0 = BCLK is input to baud clock. 1 = CPU clock is input to baud clock.
BR14:0	Baud Rate Compare Field	0	Sets the compare value for the baud rate clock.

Figure 10-11. Baud Rate Compare Register (BxCMP)

The equations in Figure 10-12 show how to calculate the proper BxCMP value for a specific baud rate (where F_{CPU} = CPU operating frequency = ½ CLKIN frequency).

	Mode 0	Mode 1–4
If CPU clock is baud timebase clock:	$BxCMP = \frac{F_{CPU}}{baudrate} - 1$	$BxCMP = \frac{F_{CPU}}{baudrate \times 8} - 1$
If BCLK is baud timebase clock:	$BxCMP = \frac{BCLK}{baudrate}$	$BxCMP = \frac{BCLK}{baudrate \times 8}$

Figure 10-12. Calculating the BxCMP Value for a Specific Baud Rate

Due to internal synchronization requirements, the maximum input frequency to BCLK is one-half the CPU operating frequency. See "BCLK Pin Timings" on page 10-19 for more information. Table 10-1 shows the correct BxCMP values for common baud rates.

Table 10-1. BxCMP Values for Typical Baud Rates and CPU Frequencies

Baud Rate	CPU Frequency							
	25 MHz		20 MHz		16 MHz		8 MHz	
	BxCMP Value	% Error	BxCMP Value	% Error	BxCMP Value	% Error	BxCMP Value	% Error
19,200	80A2H	–0.14	8081H	0.16	8067H	0.16	8033H	0.16
9,600	8145H	–0.14	8103H	0.16	80CFH	0.16	8067H	0.16
4,800	828AH	0.00	8208H	–0.03	81A0H	–0.08	80CFH	0.16
2,400	8515H	0.00	8411H	–0.03	8340H	0.04	81A0H	–0.08
1,200	8A2BH	0.00	8822H	0.01	8682H	–0.02	8340H	0.04

NOTE

A zero or one value for BxCMP is illegal and results in unpredictable operation. Programming BxCMP during a transmission or reception causes indeterminate operation.

10.2.2 Asynchronous Mode Programming

The serial port operation is controlled by two registers. The Serial Port Control (SxCON) Register controls the mode of operation of the serial port (see Figure 10-13). The Serial Port Status (SxSTS) Register acts as the flags register, reporting on errors and the state of the RX and TX machines (see Figure 10-14). Depending on the serial port mode, these registers can have different functionality. This section outlines how to use SxCON and SxSTS to obtain the desired operation from the serial port.

10.2.2.1 Modes 1, 3 and 4 for Stand-alone Serial Communications

When using these modes for their respective seven, eight or nine bit data modes, operation is fairly straightforward. The serial port must be initialized correctly (through SxCON), then SxSTS needs to be interpreted.

To configure the serial port, first program the baud rate through the BxCMP register, then program SxCON (Figure 10-13 on page 10-16) as follows.

1. Determine the values for M2:0 for the desired serial port mode.
2. If parity is used, enable it with the PEN bit. Set the sense of parity (even or odd) with the EVN bit. Note that parity is not available in Mode 4 (seven bit data).

3. If the Clear-to-Send feature is used, set the CEN bit to enable it.
4. If receptions are desired, set the REN bit to enable the RX machine. Note the TX machine need **not** be explicitly enabled.

At this point, you will be able to transmit and receive in the mode specified. Now that the serial port is operating, you must correctly interpret its status. This is done by reading the SxSTS register (Figure 10-14 on page 10-17) and interpreting its contents. Reading SxSTS clears all bits except the CTS and TXE bits. SxSTS must first be saved in memory and then each bit can be interpreted individually.

The RI, TI and TXE bits indicate the condition of the transmit and receive buffers. RI and TI are also used with the Interrupt Control Unit for interrupt-based communications. The OE, FE and PE bits indicate any errors when a character is received. Once an error occurs, the appropriate bit remains set until SxSTS is read. For example, assume a character is received with a parity error (PE set) and a subsequent error-free character is received. If the SxSTS register was not read between the two receptions, the PE bit remains set.

10.2.2.2 Modes 2 and 3 for Multiprocessor Communications

Programming for multiprocessor communications is much the same as the stand-alone operation. The only added complexity is that the ninth data bit must be controlled and interpreted correctly.

The ninth data bit is set for transmissions by setting the TB8 bit in SxCON. TB8 is cleared after *every* transmission. TB8 is not double-buffered. This is usually not a problem, as very few bytes are actually transmitted with TB8 equal to one. When writing TB8, make sure that the other bits in SxCON are written with their appropriate value.

In Modes 2 and 3, the state of the ninth data bit can be determined by the RB8 bit in SxSTS. RB8 reflects the ninth bit for the character currently in SxRBUF. Note that the RB8 bit shares functionality with the PE bit in SxSTS. When parity is enabled, the PE bit has precedence over RB8.

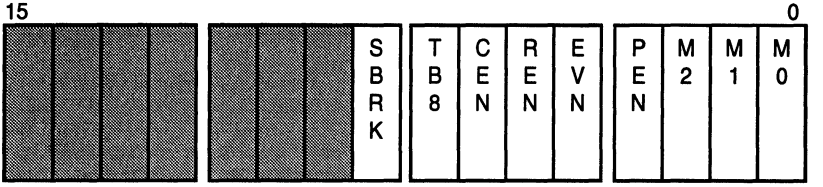
10.2.2.3 Sending and Receiving a Break Character

The serial port can send as well as receive BREAK characters. A BREAK character is a long string of zeros. To send a BREAK character, set the SBRK bit in SxCON. SBRK drives the TXD pin immediately low, regardless of the current serial port mode. The user controls the length of the BREAK character in software by controlling the length of time that SBRK remains set. When writing SBRK, make sure the other bits in SxCON retain their current states.

Register Name: Serial Port Control Register

Register Mnemonic: SxCON

Register Function: Controls serial port operating modes.



A1277-0A

Bit Mnemonic	Bit Name	Reset State	Function
SBRK	Send Break	0	Setting SBRK drives TXD low. TXD remains low until SBRK is cleared.
TB8	Transmitted Bit 8	0	TB8 is the eighth data bit transmitted in modes 2 and 3.
CEN	Clear-to-Send Enable	0	When CEN is set, no transmissions will occur until the $\overline{CTS}$ pin is asserted.
REN	Receive Enable	0	Set to enable the receive machine.
EVN	Even Parity Select	0	When parity is enabled, EVN selects between even and odd parity. Set for even, clear for odd parity.
PEN	Parity Enable	0	Setting PEN enables the parity generation/checking for all transmissions/receptions.
M2:0	Serial Port Mode Field	0	Operating mode for the serial port channel. **M2 M1 M0 Mode** 0 0 0 Synchronous Mode0 0 0 1 10-Bit Asynch Mode1 0 1 0 11-Bit Asynch Mode2 0 1 1 11-Bit Asynch Mode3 1 0 0 9-Bit Asynch Mode4 1 0 1 Reserved 1 1 0 Reserved 1 1 1 Reserved

NOTE: Reserved register bits are shown with gray shading. Reserved bits must be written to a logic zero to ensure compatibility with future Intel products.

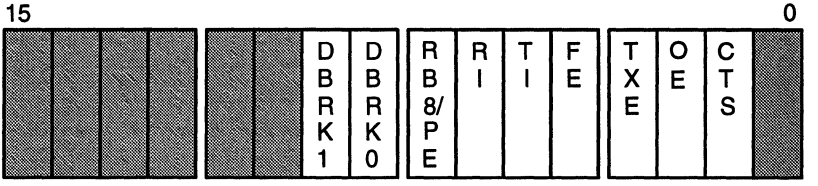
Figure 10-13. Serial Port Control Register (SxCON)

The serial port receives BREAK characters of two different lengths. If a BREAK character longer than M bit-times is detected, the DBRK0 bit in SxSTS is set. If the BREAK character is longer than 2M+3 bit-times, DBRK1 in SxSTS is set. M is equal to the total number of bits in a frame. For example, M is equal to 11 (decimal) in Mode 3.

Register Name: Serial Status Register

Register Mnemonic: SxSTS

Register Function: Indicates the status of the serial port.



A1278-0A

Bit Mnemonic	Bit Name	Reset State	Function
DBRK1	Detect Break 1	0	Set when a break longer than 2M+3 bits occurs.
DBRK0	Detect Break 0	0	Set when a break longer than M bits occurs.
RB8/PE	Received Bit8/Parity Error	0	Contains the 9th received data bit in modes 2 and 3. PE is set when a parity error occurs. PE is valid only when parity is enabled in Mode 1, 2 or 3.
RI	Receive Interrupt	0	RI is set when a character has been received and placed in SxRBUF. Note that RI need not be explicitly cleared to receive more characters. Writing a one to this bit will not cause an interrupt.
TI	Transmit Interrupt	0	TI is set when a character has finished transmitting. TI determines when one more character can be transmitted. Writing a one to this bit will not cause an interrupt.

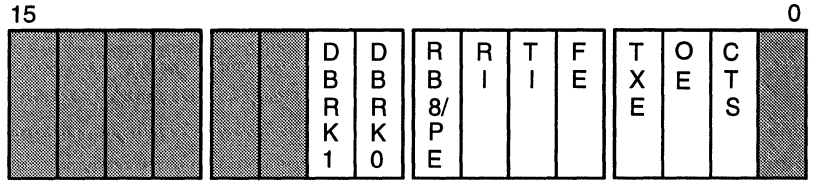
NOTE: Reserved register bits are shown with gray shading. Reserved bits must be written to a logic zero to ensure compatibility with future Intel products.

Figure 10-14. Serial Port Status Register (SxSTS)

Register Name: Serial Status Register

Register Mnemonic: SxSTS

Register Function: Indicates the status of the serial port.



A1278-0A

Bit Mnemonic	Bit Name	Reset State	Function
FE	Framing Error	0	FE is set when a framing error occurs. A framing error occurs when a valid stop bit is **not** detected.
TXE	Transmitter Empty	1	TXE is set when both SxTBUF and the transmit shift register are empty. TXE determines when two consecutive bytes can be written to SxTBUF for transmission. Accessing SxSTS does **not** clear TXE.
OE	Overrun Error	0	OE is set when an overrun error occurs. An overrun error occurs when the character in SxRBUF is not read before another complete character is received. SxRBUF always contains the most recent reception.
CTS	Clear To Send	0	CTS is the complement of the value on the CTF pin. Accessing SxSTS does **not** clear CTS.

NOTE: Reserved register bits are shown with gray shading. Reserved bits must be written to a logic zero to ensure compatibility with future Intel products.

Figure 10-14. Serial Port Status Register (Continued)

When either BREAK character is detected, an overrun error occurs (OE is set). SxRBUF will contain at least one null character.

10.2.3 Programming in Mode 0

Programming is much easier in Mode 0 than in the asynchronous modes. Configuring SxCON (Figure 10-13 on page 10-16) for Mode 0 requires only two steps:

1. Program M2:0 with the correct combination for Mode 0.
2. If the Clear-to-Send feature is desired, set the CEN bit.

The serial port is now configured for Mode 0. To transmit, write a character to SxTBUF. The TI and TXE bits reflect the status of SxTBUF and the transmit shift register. Note that the SBRK bit is independent of serial port mode functions in Mode 0.

Receptions in Mode 0 are controlled by software. To begin a reception, set the REN bit in SxCON. The RI bit must be zero or the reception will not begin. Data begins shifting in on RXD as soon as REN is set. The asynchronous error flags (OE, FE and PE) and break flags (DBRK0 and DBRK1) are invalid in Mode 0.

10.3 HARDWARE CONSIDERATIONS FOR THE SERIAL PORT

There are several interface considerations when using the serial port.

10.3.1 $\overline{\text{CTS}}$ Pin Timings

When the Clear-to-Send feature is enabled, transmissions will not begin until the $\overline{\text{CTS}}$ pin is asserted *while a transmission is pending.* Figure 10-15 shows the recognition of a valid $\overline{\text{CTS}}$.

The $\overline{\text{CTS}}$ pin is sampled by the rising edge of CLKOUT. The CLKOUT high time synchronizes the $\overline{\text{CTS}}$ signal. On the falling edge of CLKOUT, the synchronized $\overline{\text{CTS}}$ signal is presented to the serial port. $\overline{\text{CTS}}$ is an asynchronous signal. The setup and hold times are given only to ensure recognition at a specific clock edge. When $\overline{\text{CTS}}$ is asynchronously, it should be asserted for at least 1½ clock cycles to guarantee that the signal is recognized.

$\overline{\text{CTS}}$ is not latched internally. If $\overline{\text{CTS}}$ is asserted before a transmission starts, the subsequent transmission will not begin. A write to SxTBUF "arms" the $\overline{\text{CTS}}$ sense circuitry.

10.3.2 BCLK Pin Timings

The BCLK pin can be configured as the input to the baud timebase clock. The baud timebase clock increments the BxCNT register. However, the BCLK signal does not run directly into the baud timebase clock. BCLK is first synchronized to the CPU clock (Figure 10-16.) The internal synchronization logic uses a low-to-high level transition on BCLK to generate the baud timebase clock that increments the BxCNT register. The CPU recognizes a low-to-high transition by sampling the BCLK pin low, then high.

The CPU samples BCLK on the rising edge of CLKOUT. The CLKOUT high time synchronizes the BCLK signal. On the falling edge of CLKOUT, the synchronized BCLK signal is presented to the baud timebase clock.

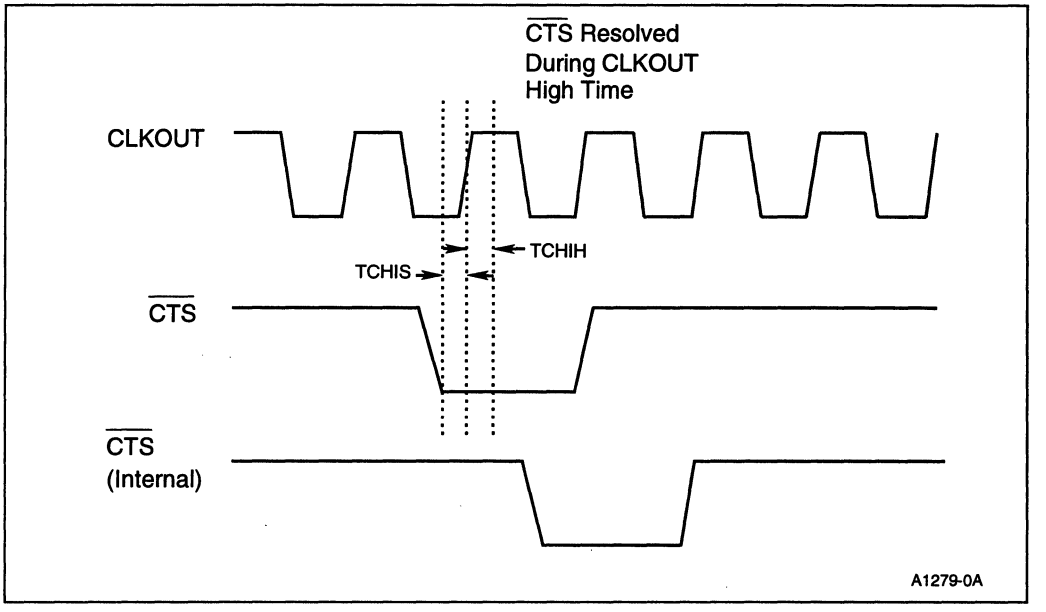


Figure 10-15. CTS Recognition Sequence

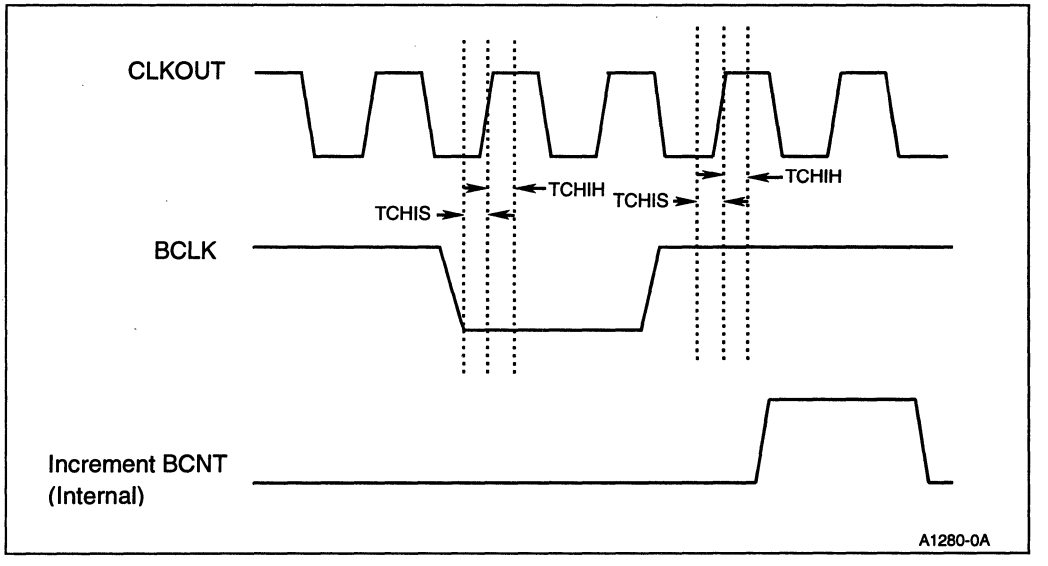


Figure 10-16. BCLK Synchronization

BCLK is an asynchronous input. However, the pin does have setup and hold times, which guarantee recognition at a specific CLKOUT. If the BCLK input signal has high and low times that are both at least 1½ CLKOUT periods, than synchronization to CLKOUT is not necessary. However, when the BCLK signal has a high or a low time of less than 1½ CLKOUT periods, meeting the setup and hold times to CLKOUT is necessary to avoid missing BCLK transitions. The maximum input frequency to BCLK is one-half the frequency of CLKOUT (CPU operating frequency).

10.3.3 Mode 0 Timings

This section shows the timings of the TXD and RXD pins in Mode 0. In Mode 0, TXD never floats. When not transmitting or receiving, TXD is high. RXD floats except when transmitting a character.

10.3.3.1 CLKOUT as Baud Timebase Clock

The behavior of the transmit/receive clock (on TXD) is governed by the value of BxCMP. When the BxCMP value is **greater than or equal to two**. The TXD pin is low for two CLKOUT periods and is high for (BxCMP – 1) CLKOUT periods (see Figure 10-17). BxCMP cannot be equal to a one, otherwise the serial port buffer registers (SxRBUF) will not receive the correct data.

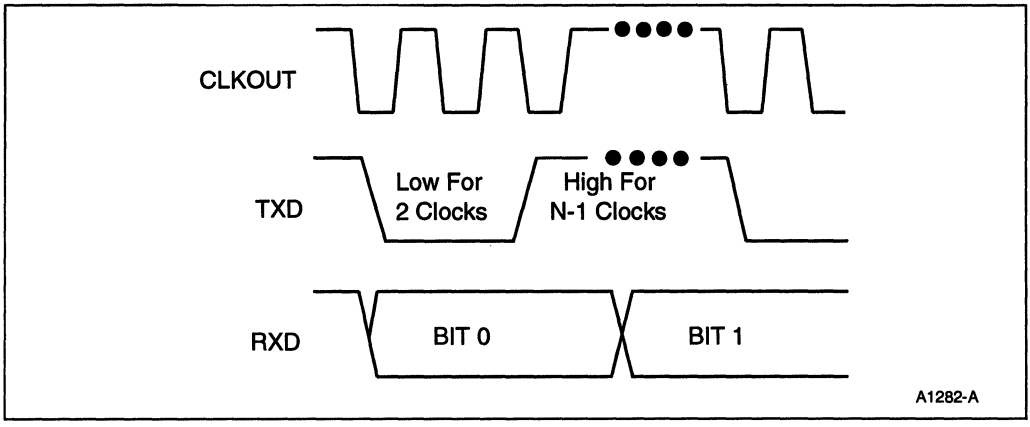


Figure 10-17. Mode 0, BxCMP > 2

For transmissions, the RXD pin changes on the next CLKOUT falling edge following a low-to-high transition on TXD. Therefore, the data on the RXD pin is guaranteed to be valid on the rising edges of TXD. Use the rising edge of TXD to latch the value on RXD. For receptions, the incoming serial data must meet the setup and hold timings with respect to the rising edge of TXD. These timings can be found in the AC timings section of the data sheet.

10.3.3.2 BCLK as Baud Timebase Clock

BCLK does not run directly into the baud timebase clock, but is first synchronized to the CPU clock. BCLK causes the baud timebase clock to increment, but transitions on TXD and RXD (for transmissions) still occur relative to CLKOUT.

A low-to-high transition on BCLK increments BxCNT. If BxCNT is equal to BxCMP, TXD goes low approximately 4½ CLKOUTs later. TXD will always remain low for two CLKOUT periods and then go high. TXD will go low again 4½ CLKOUTs after BxCNT equals BxCMP. Therefore, the output frequency on TXD is roughly equal to the input frequency on BCLK multiplied by BxCMP. There will be some clock jitter, as the output on TXD will always be some multiple of CLKOUTs. This is due to the internal synchronization.

10.4 SERIAL COMMUNICATIONS UNIT INTERRUPTS

Serial communication is usually interrupt-driven. An interrupt needs to occur on each reception and on each transmission of a character. The RI and TI flags in the SxSTS register (Figure 10-14 on page 10-17) provide the interrupt mechanism for the serial ports. The two serial ports, or channels, have different interrupt circuitry. Serial channel 0 is directly supported by the integrated Interrupt Control Unit. Serial channel 1 is supported by the SINT1 output.

10.4.1 Channel 0 Interrupts

Figure 10-18 illustrates the channel 0 interrupt circuitry. Channel 0 receptions assert an internal signal, Receive_Interrupt_Request_0. This signal is routed both to the Interrupt Control Unit and to the S0STS register, where it sets the RI bit. The RI bit has no effect on the internal interrupt request. Writing to RI does not cause an interrupt, and setting it does not prevent interrupts.

Channel 0 transmissions assert an internal signal, Transmit_Interrupt_Request_0. Like the Receive_Interrupt_Request_0 signal, this signal is routed to the Interrupt Control Unit and to the S0STS register. This signal sets the TI bit in S0STS. Like the RI bit, TI has no effect on the internal interrupt request. Writing to TI does not cause an interrupt, and setting it does not prevent interrupts.

10.4.2 Channel 1 Interrupts

Figure 10-18 illustrates the channel 1 interrupt circuitry. Channel 1 receptions assert an internal Receive_Interrupt_Request_1 signal and transmissions assert an internal Transmit_Interrupt_Request_1 signal. Serial channel 1 is supported by the SINT1 output. Each internal signal is routed to the S1STS register, where it sets the RI or TI bit. The RI and TI bits are ORed into the SINT1 signal, so setting either bit asserts SINT1. Reading S1STS clears the RI and TI bits and deasserts SINT1. (This is the only method available for deasserting SINT1.)

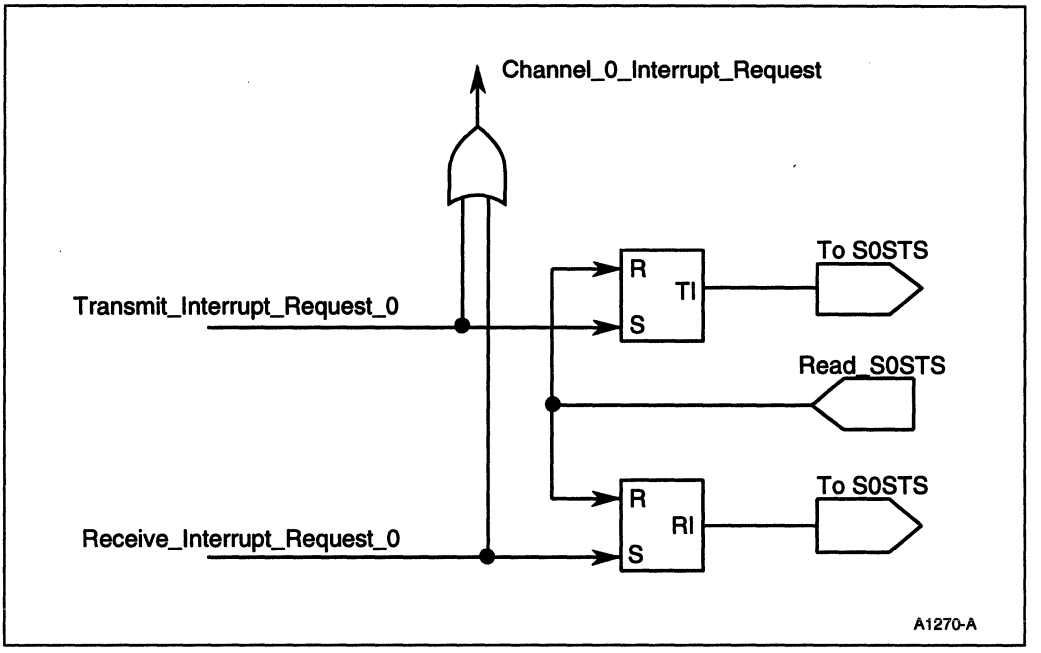


Figure 10-18. Channel 0 Interrupts

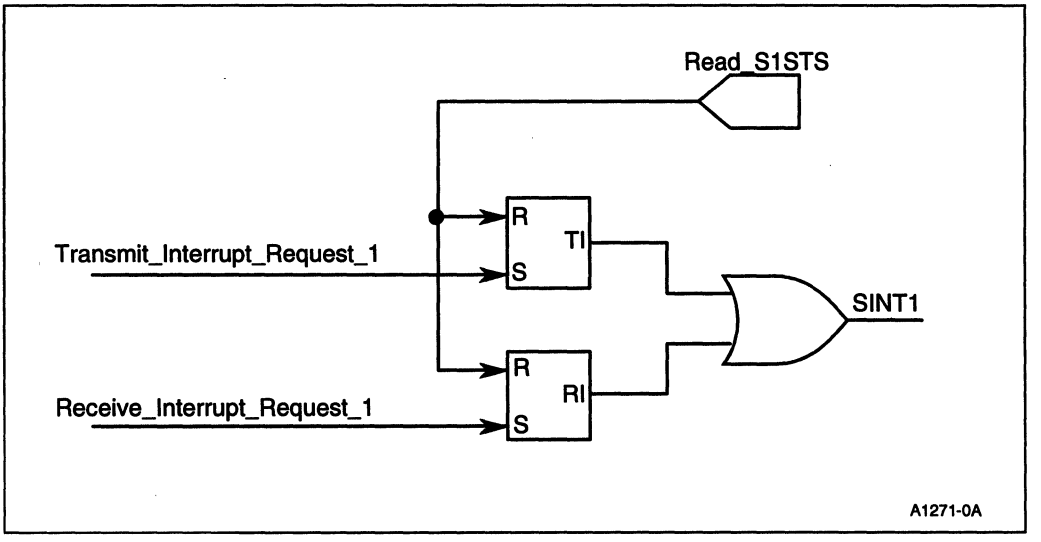


Figure 10-19. Channel 1 Interrupts

10.5 SERIAL PORT EXAMPLES

This section contains examples that show ways to use the serial port.

NOTE

The examples assume that the Peripheral Control Block is located in I/O space.

10.5.1 Asynchronous Mode Example

Example 10-1 contains sample code to initialize Serial Port 0 for 9600-baud operation in asynchronous Mode 4.

```
$mod186
name            scu_async_example
;
;This file contains an example of initialization code for the 80C186EB
;Serial Communications Unit. The example has three procedures:
;ASYNC_CHANNEL_SETUP sets up channel 0 as 9600 baud, full duplex,
;               7 data bits+parity, with CTS# control.
;ASYNC_REC_INT_PROC is an interrupt handler for a reception. The procedure
;               is nearly empty, since the code to perform error checking
;               and receive buffer handling is application dependent.
;ASYNC_XMIT_INT_PROC is an interrupt handler for a transmission. This
;               procedure, too, is nearly devoid of code. A typical
;               application would test the TXE bit, then copy data from
;               the transmit buffer in memory to the S0TBUF register.
;We assume serial port registers have been correctly defined and the PCB
;               is located in I/O space.
B0CMP           equ   0xxxx           ;channel 0 baud rate compare
S0CON           equ   0xxxx           ;channel 0 control
S0STS           equ   0xxxx           ;channel 0 status
S0RBUF          equ   0xxxx           ;channel 0 receive buffer
S0TBUF          equ   0xxxx           ;channel 0 transmit buffer
RI_TYPE         equ   xx              ;receive is interrupt type 20
TI_TYPE         equ   21              ;transmit is interrupt type 21
EOI             equ   0ff02h          ;end-of-interrupt register
SCUCON          equ   0ff14h          ;SCU interrupt control register

code_seg        segment public
assume          cs:code_seg

ASYNC_CHANNEL_SETUP proc near

;First, set up the interrupt handler vectors
                xor   ax,ax
                mov   ds,ax           ;need DS to point to interrupt vector
                                      ;table at 0h
                mov   bx,RI_TYPE*4
                mov   ax,offset ASYNC_REC_INT_PROC
                mov   [bx],ax
                mov   ax, seg ASYNC_REC_INT_PROC
                mov   [bx+2],ax
```

Example 10-1. Asynchronous Mode 4 Example

```
                mov   bx, TI_TYPE*4
                mov   ax,offset ASYNC_XMIT_INT_PROC
                mov   [bx],ax
                mov   ax, seg ASYNC_XMIT_INT_PROC
                mov   [bx+2],ax

;Now set up channel 0 options

                mov   ax,8067h        ;for 9600 baud from 16MHz CPU clock
                mov   dx,B0CMP
                out   dx,ax           ;set baud rate
                mov   ax,0059H        ;CEN=1 (CTS enabled)
                                      ;REN=0 (receiver not yet enabled)
                                      ;EVN=1 (even parity)
                                      ;PEN=1 (parity turned on)
                                      ;MODE=1 (10-bit frame)
                mov   dx,S0CON
                out   dx,ax           ;write to serial control register

;Clear any pending RI or TI, just for safety

                mov   dx,S0STS
                in    ax,dx           ;clear any old RI or TI

;Clear interrupt mask bit in interrupt unit to allow SCU interrupts

                mov   dx,SCUCON       ;SCU interrupt control
                in    ax,dx
                and   ax,0007h        ;clear mask bit to enable
                out   dx,ax

;Turn on the receiver
                mov   dx,S0CON
                in    ax,dx           ;read S0CON
                or    ax,0020         ;set REN bit (REN=1, receiver enabled)
                out   dx,ax           ;write S0CON

                ret
ASYNC_CHANNEL_SETUP endp

;Now the receiver is enabled and sampling of the RXD line begins.
;Any write to S0TBUF will initiate a transmission.

;The next procedure is executed every time a reception is completed.

ASYNC_REC_INT_PROC proc near

                mov   dx, S0STS
                in    ax, dx          ;get status info
                test  al, 10000000b   ;test for parity error
                jnz   parity_error
                test  al, 00010000b   ;test for framing error
                jnz   framing_error
                test  al, 00000100b   ;test for overrun error
                jnz   overrun_error

;At this point, we know the received data is OK.
```

Example 10-1. Asynchronous Mode 4 Example (Continued)

```
                mov   dx, S0RBUF
                in    ax, dx          ;read received data
                and   ax, 07fh        ;strip off parity bit

;Code to store the data in a receive buffer would go here. It has been omitted
;since this is heavily application dependent.

                jmp   eoi_rcv_int

parity_error:

;Code for parity error handling goes here.

                jmp   eoi_rcv_int

framing_error:

;Code for framing error handling goes here.

                jmp   eoi_rcv_int

overrun_error:

;Code for overrun error handling goes here.

                jmp   eoi_rcv_int

;Now we must issue the end-of-interrupt command to the interrupt unit.

eoi_rcv_int:
                mov   dx,EOI
                mov   ax,8000h        ;issue non-specific EOI
                out   dx, ax
                iret

ASYNC_REC_INT_PROCendp

ASYNC_XMIT_INT_PROCproc near

;This procedure is entered whenever a transmission completes. Typical code
;would be inserted here to transmit the next byte from a transmit buffer
;set up in memory. Since the configuration of such a buffer is application
;dependent, this section is omitted.

;Now we must issue the end-of-interrupt command to the interrupt unit.

eoi_xmit_int:
                mov   dx,EOI
                mov   ax,8000h        ;issue non-specific EOI
                out   dx, ax
                iret

ASYNC_XMIT_INT_PROCendp

code_seg        ends
end
```

Example 10-1. Asynchronous Mode 4 Example (Continued)

10.5.2 Mode 0 Example

Example 10-2 shows a sample Mode 0 application.

```
$mod186
name                    example_SCU_mode_0

;*************************************************************
; FUNCTION: This function transmits the user's data, user_data, serially
; over RXD1. TXD1 provides the transmit clock. The transmission frequency
; is calculated as follows:
;
;               tran_freq = (0.5*CLKIN/BAUDRATE)-1
;
; A 0-1-0 pulse on P1.0 indicates the end of transmission.
;
; SYNTAX:       extern void far parallel_serial(char user_data,int tran_freq)
;
; INPUTS:       user_data - byte to send out serially
;               tran_freq - baud rate compare value
; OUTPUTS:      None
; NOTE:         Parameters are passed on the stack as required by high-level
;               languages.
;*************************************************************

B1CMP           equ   xxxxH           ;Channel 1 Baud Rate Compare
S1CON           equ   xxxxH           ;Channel 1 Control
S1STS           equ   xxxxH           ;Channel 1 Status
S1TBUF          equ   xxxxH           ;Channel 1 Receive Buffer

;xxxx - substitute register offset

;Example assumes that all the port pins are configured correctly and
;PCB is located in I/O space.
lib_80186       segment public 'code'
                assume cs:lib_80186

public          _parallel_serial
_parallel_serialproc far

                push bp               ;save caller's bp
                mov  bp, sp           ;get current top of stack

user_data       equ word ptr [bp+6];get parameters off the stack
tran_freq       equ word ptr [bp+8]

                push ax               ;save registers that
                push dx               ;will be modifiled

                mov  dx, S1STS        ;clear any pending exceptions
```

Example 10-2. Mode 0 Example

```
                mov   dx, P1CON       ;Get state of port 1 controls
                in    ax, dx
                and   ax, 0feh        ;make sure P1.0 is port
                out   dx, al
                mov   dx, B1CMP
                mov   ax, tran_freq
                or    ax, 8000h       ;set internal clocking bit
                out   dx, ax          ;Mode 0, 1 million bps

                mov   dx, P2CON       ;set Port 2.1 for TXD
                mov   ax, 0ffh
                out   dx, al

                mov   dx, S1TBUF      ;send user's data
                mov   ax, user_data
                out   dx, al
                mov   dx, S1CON       ;Mode 0, No CTS, Transmit
                xor   ax, ax
                out   dx, ax
                mov   dx, S1STS

Check_4_TI:     in    ax, dx
                test  ax, 0020h       ;check for TI bit
                jz    Check_4_TI

                mov   dx, P1LTCH      ;pulse P1.0
                xor   ax, ax
                out   dx, al

                not   ax              ;set P1.0 high
                out   dx, al

                not   ax              ;set P1.0 low
                out   dx, al

                pop   dx              ;restore saved registers
                pop   ax
                pop   bp              ;restore user's bp
                ret

_parallel_serial endp
lib_80186       ends
                end
```

Example 10-2. Mode 0 Example (Continued)

10.5.3 Master/Slave Example

This section shows an example of a Mode 2 and 3 master/slave network. Figure 10-20 shows the proper connection of the master to the slaves. The buffer is necessary to avoid contention on the receive line. Alternatively, an open-collector buffer could be used and the port pin function could be deleted.

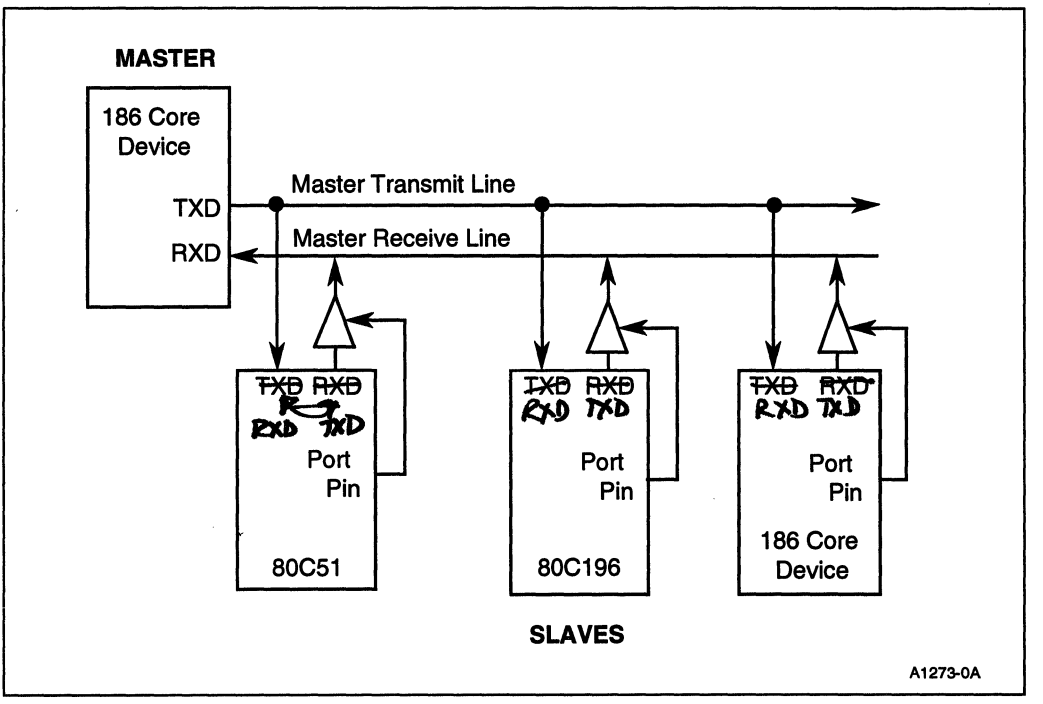


Figure 10-20. Master/Slave Example
(See 7/31/95 CHANGES)

Example 10-3 demonstrates how to implement a master/slave network in a typical system. The remaining three examples show the routines used in the implementation. Example 10-4 is a master routine that addresses a slave and waits for it to respond. Example 10-5 is a slave routine that responds to commands sent by the master. Equation 10-6 is the master routine that sends commands to the slave.

```
$mod186
name                    example_master_slave

;*****************************************************************
; FUNCTION:     This function demonstrates how to implement the three
; master/slave routines (_slave_1, _select_slave, and _send_slave_command)
; in a typical setup.
;
; NOTE:         It is assumed that the network is set up as shown in
;               Figure 10-20, that the slave unit is running the
;               _slave_1 code, and that the PCB is located in I/O space.
;
;*****************************************************************
Slave1          equ   01h             ;address assigned to slave unit 1
Flash           equ   01h             ;command to flash EVAL board LEDs
Disc            equ   0fh             ;command to disconnect from network
False           equ   00h

lib_80186       segment public 'code'   ;declare external routines
extrn           _select_slave:far
extrn           _send_slave_cmd:far
lib_80186       ends

code            segment public 'code'
                assume cs:code

public          _main
_main           proc near
                push Slave1           ;get slave unit 1 address
;send the address over the network
                call far ptr _select_slave
                add  sp, 2            ;adjust sp
                cmp  ax, false        ;was slave 1 properly selected ?
                je   SlaveExit        ;no: then exit

                push Flash            ;yes: then send Flash command

;send it
                call far ptr _send_slave_cmd
                add  sp, 2            ;adjust sp

;insert a delay routine to allow completion of last command

                push Disc             ;prepare to disconnect slave
;send it
                call far ptr _send_slave_cmd
                add  sp, 2            ;adjust sp

SlaveExit:      ret
_main           endp
code            ends
                end _main
```

Example 10-3. Master/Slave — Implementing the Master/Slave Routines

```
$mod186
name                    example_master_select_slave
;*************************************************************;
;                       select_slave
;
; FUNCTION: This function transmits a slave address, _slave_addr, over the
; serial network, with data bit 9 set to one. It then waits for the addressed
; slave to respond with its (slave) address. If this address does not match
; the originally transmitted slave address, or if there is no response within
; a set time, the function will return false (ax = 0). Otherwise, the function
; will return true (ax <> 0).
;
; SYNTAX:       extern int far select_slave(int slave_addr);
;
; INPUTS:       _slave_addr - address of the slave on the network
;
; OUTPUTS:      True/False
;
; NOTE:         Parameters are passed on the stack as required by high-level
;               languages. Example assumes that PCB is located in I/O space.
;*************************************************************
; substitute register offset in place of xxxxh

P1CON           equ   xxxxh           ;Port 1 Control register
P2CON           equ   xxxxh           ;Port 2 Control register
S1CON           equ   xxxxh           ;Serial Port 1 Control register
S1STS           equ   xxxxh           ;Serial Port 1 Status register
S1TBUF          equ   xxxxh           ;Serial Port 1 Transmit Buffer
S1RBUF          equ   xxxxh           ;Serial Port 1 Receive Buffer

lib_80186       segment public 'code'
                assume cs:lib_80186

public          _select_slave
_select_slave   proc far

                push bp               ;save caller's bp
                mov  bp, sp           ;get current top of stack

;get slave address off the stack
_slave_addr equ word ptr [bp+6]
                push cx               ;save registers that will be
                push dx               ;modified

                mov  dx, P1CON        ;Get state of port 1 controls
                in   ax, dx
                and  ax, 0f0h         ;make sure P1.0:3 is port
                out  dx, al
                mov  dx, P2CON        ;set Port 2.1 for TXD1, P2.0 RXD1
                mov  ax, 0ffh
                out  dx, al
```

Example 10-4. Master/Slave — The _select_slave Routine

```
                mov   dx, S1STS       ;clear any pending exceptions
                in    ax, dx
                mov   dx, S1CON       ;prepare to send address
                mov   ax, 0083h       ;d9=1, mode 3
                out   dx, ax
                mov   dx, S1TBUF      ;select slave
                mov   ax, _slave_addr
                                      ;get slave address
                out   dx, al          ;send it


                mov   dx, S1CON
                mov   ax, 0023h       ;set REN
                out   dx, ax          ;enable receiver

                xor   cx, cx          ;reset time-out counter

                mov   dx, S1STS       ;check to see if data is waiting

Check_4_RI:     dec   cx              ;decrement time-out counter
                jnz   NoTimeOut       ;time-out=false:then continue

                xor   ax, ax          ;time-out=true:set return
                                      ;value false (0)
                jmp   short SlaveExit

NoTimeOut:      in    ax, dx
                test  ax, 0040h       ;test for RI bit

                jz    Check_4_RI      ;keep checking till data received

                mov   dx, S1RBUF      ;get slave response
                in    ax, dx
                and   ax, 0ffh        ;mask off unwanted bits

                xor   ax, _slave_addr;did addressed slave respond?
                                      ;ax=0:true else false
                not   ax              ;invert state of ax to be consistent
                                      ;with false(0) and true(non zero)
SlaveExit:      pop   dx              ;restore saved registers
                pop   cx
                pop   bp              ;restore caller's bp
                ret

_select_slave   endp

lib_80186       ends
                end
```

Example 10-4. Master/Slave — The _select_slave Routine (Continued)

```
$mod186
name                    example_slave_1_routine

;*************************************************************;
;                       slave_1
;
; FUNCTION:     This function represents a slave unit connected to a multi-
;               processor master/slave network. This slave responds to two
;               commands:
;               Flash the LEDs on the EVAL Board, and
;               Disconnect from the Network.
;               Other commands are easily added.
;
; SYNTAX:       extern void far slave_1(void);
;
; INPUTS:       None
;
; OUTPUTS:      None
;
; NOTE:         Parameters are passed on the stack as required by high-level
;               languages. The slave should be running this code before the
;               master calls the slave. Example assumes PCB is in I/O space.
;*************************************************************
;substitute register offsets in place of xxxxh

P1CON           equ   xxxxh           ;Port 1 Control register
P1LTCH          equ   xxxxh           ;Port 1 Latch register
P2CON           equ   xxxxh           ;Port 2 Control register
S1CON           equ   xxxxh           ;Serial Port 1 Control register
S1STS           equ   xxxxh           ;Serial Port 1 Status register
S1TBUF          equ   xxxxh           ;Serial Port 1 Transmit Buffer
S1RBUF          equ   xxxxh           ;Serial Port 1 Receive Buffer

lib_80186       segment public 'code'
                assume cs:lib_80186

My_Address      equ   01h             ;slave 1 network address
TriStateEna     equ   08h             ;Tri-state buffer enable
TriStateDis     equ   00h             ;Tri-state buffer disable
FlashLEDs       equ   01h             ;list of commands unit 1 responds to
Disconnect      equ   0fh
public          _slave_1
_slave_1        proc far

                push  ax              ;save registers that will be modified
                push  bx
                push  cx
                push  dx
```

Example 10-5. Master/Slave — The slave_1 Routine

```
DisconnectMode:
                mov   dx, S1STS       ; clear any pending exceptions
                in    ax,dx
                mov   dx, P1CON       ; get state of port 1 controls
                in    ax, dx
                and   ax, 0f0h        ; make sure P1.0:P1.3 is port
                out   dx, ax
                mov   dx, P2CON       ; set P2.1 for TXD1, P2.0for RXD1
                mov   ax, 0ffh
                out   dx, ax
                mov   dx, P1LTCH      ; make sure TXD latch is tristated
                mov   ax, TriStateDis
                out   dx, ax          ; set P1.7 to zero

                mov   dx, S1CON       ; select control register
                mov   ax, 0022h       ; receive, Mode 2
                out   dx, ax

SelStatus:      mov   dx, S1STS       ; select status register
Check_4_RI:     in    ax, dx          ; get status
                test  ax, 0040h       ; data waiting?
                jz    Check_4_RI      ; no: then keep checking

                mov   dx, S1SRUF      ; yes: then get data
                in    ax, dx
                cmp   al, My_Address  ; is slave_1 being addressed?
                jne   SelStatus       ; no: then ignore

                mov   dx, S1CON       ; yes: then switch to Mode 3, transmit
                mov   ax, 0003h       ; Mode 3
                out   dx, ax

                mov   dx, P1LTCH      ; enable tristate buffer
                mov   ax, TriStateEna
                out   dx, ax          ; gate TXD onto master's RXD
                mov   dx, S1TBUF      ; echo My_Address to the master
                mov   ax, My_Address
                out   dx, ax

                mov   dx, S1CON       ; switch to receive mode
                mov   ax, 0023h       ; Mode 3, receive
                out   dx, ax

Wait_4_Cmd:     mov   dx, S1STS       ; select status register
                in    ax, dx          ; get status
                test  ax, 0040h       ; command waiting?
                jz    Wait_4_Cmd      ; no: then keep checking

                mov   dx, S1RBUF      ; yes: then get command
                in    ax, dx

                cmp   al, Disconnect  ; Disconnect command?
                je    DisconnectMode  ; yes: then disconnect RXD from network
```

Example 10-5. Master/Slave — The slave_1 Routine (Continued)

```
                cmp   al, FlashLEDs  ; Flash LEDs command
                jne   Wait_4_Cmd     ; no: then ignore

                mov   dx, P1LTCH     ; yes: then flash LEDs 10 times
                mov   cx, 20
                xor   ax, ax

Send:           not   ax
                out   dx, ax
                mov bx, 0ffffh

Dly1:           dec   bx
                jnz   Dly1

                dec   cx
                jnz   Send

                jmp   short Wait_4_Cmd

                pop   dx
                pop   cx
                pop   bx
                pop   ax

                ret
_slave_1        endp

lib_80186       ends
                end
```

Example 10-5. Master/Slave — The slave_1 Routine (Continued)

```
$mod186
name            example_master_send_slave_command
;*************************************************************************
;
;               send_slave_cmd
; FUNCTION:     This function transmits a slave command, _slave_cmd, over
;               the serial network to a previously addressed slave.
; SYNTAX:       extern void far send_slave_cmd (int slave_cmd)
;
; INPUTS:       _slave_cmd (command to send to addressed slave)
;
; OUTPUTS:      None
;
; NOTE:         Parameters are passed on the stack as required by
;               high-level languages. Example assumes PCB is in I/O space.
;*************************************************************************;
; substitute register offsets in place of xxxxh

S1STS           equ  xxxxh          ; Serial Port 1 Status register
S1CON           equ  xxxxh          ; Serial Port 1 Control register
S1TBUF          equ  xxxxh          ; Serial Port 1 Transmit Buffer register
;
lib_80186       segment public 'code'
                assume cs:lib_80186

public          _send_slave_cmd
                _send_slave_cmd proc far
                push bp             ; save caller's bp
                mov  bp, sp         ; get current top of stack

; get slave command off the stack
_slave_cmd      equ  word ptr [bp+6]
                push ax             ; save registers that are modified
                push dx

                mov  dx, S1STS      ; clear any pending exceptions
                in   ax, dx
                mov  dx, S1CON      ; prepare to send command
                mov  ax, 0003h      ; Mode 3
                out  dx, ax

                mov  dx, S1TBUF     ; select slave
                mov  ax, _slave_cmd ; get command to send to slave
                out  dx, al         ; send it

                pop  dx             ; restore saved registers
                pop  ax
                pop  bx             ; restore caller's bp
                ret

_send_slave_cmdendp

lib_80186       ends
                end
```

Example 10-6. Master/Slave — The _send_slave_command Routine

intel®

11

Input/Output Ports

CHAPTER 11
INPUT/OUTPUT PORTS

Many applications do not require full use of all the on-chip peripheral functions. For example, the Chip-Select Unit provides a total of ten chip-select lines; only a large design would require all ten. For smaller designs that require fewer than ten chip-selects, these pins would be wasted.

The input/output ports give system designers the flexibility to replace the functions of unused peripheral pins with general-purpose I/O ports. Many of the on-chip peripheral pin functions are multiplexed with an I/O port. If a particular peripheral pin function is unnecessary in an application, that pin can be used for I/O. The 80C186EB/80C188EB has four types of ports: bidirectional, input-only, output-only, and open-drain bidirectional.

11.1 FUNCTIONAL OVERVIEW

All port pin types are derived from a common bidirectional port logic module. Unidirectional and open-drain ports are a subset of the bidirectional module. The following sections describe each port type. The bidirectional port is described in detail, as it is the basis for all of the other port types. The descriptions for the unidirectional and open-drain ports only highlight their specific differences from the common bidirectional module.

11.1.1 Bidirectional Port

Figure 11-1 shows a simplified schematic of a bidirectional port pin. The overall function of a bidirectional port pin is controlled by the state of the Port Control Latch. The output of the Port Control Latch selects the source of output data and the source of the control signal for the three-state output driver. When the port is programmed to act as a peripheral pin, both the data for the pin and the directional control signal for the pin come from the associated integrated peripheral. When a bidirectional port pin is programmed as an I/O port, all port parameters are under software control.

The output of the Port Direction latch enables (or disables) the three-state output driver when the pin is programmed as an I/O port. The three-state output driver is enabled by clearing the Port Direction latch. The data driven on an output port pin is held in the Port Data latch. Setting the Port Direction latch disables the three-state output driver, making the pin an input.

The signal present on the device pin is routed through a synchronizer to a three-state latch that connects to the internal data bus. The state of the pin can be read at any time, regardless of whether the pin is used as an I/O port or for a peripheral function.

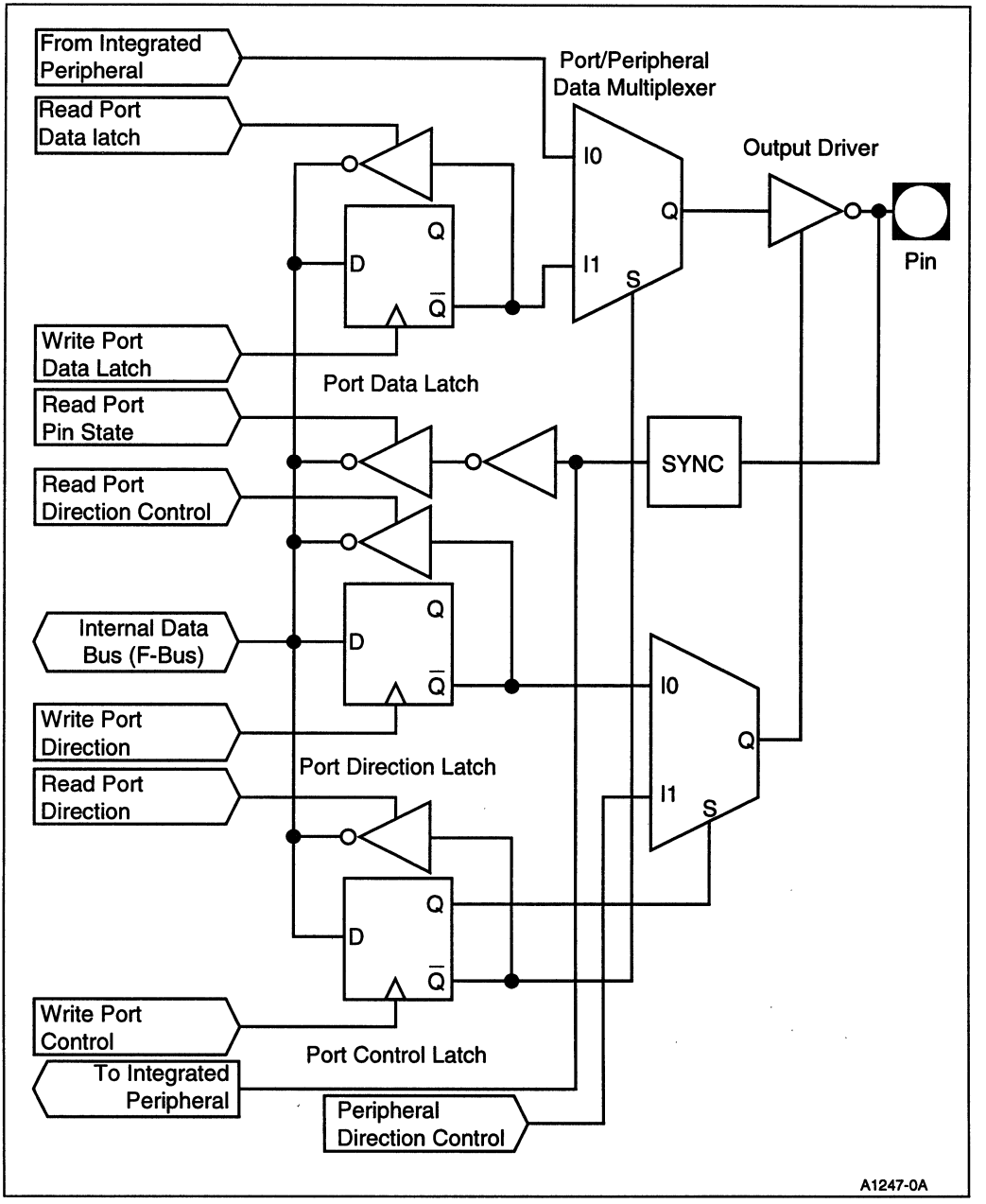


Figure 11-1. Simplified Logic Diagram of a Bidirectional Port Pin

11.1.2 Input Port

Figure 11-3 shows the internal construction of an input port pin. An internal connection permanently disables the three-state output driver. The Port Pin register holds the current state (synchronized to the CPU clock) of the input pin. The Port Direction and Port Data bits are not used for an input-only port pin; they can be used for storage.

11.1.3 Output Port

Figure 11-3 shows the internal construction of an output port pin. An internal connection permanently enables the three-state output driver. The Port Control latch selects the source of data for the pin, which can be either the on-chip peripheral or the Port Data latch. The Port Direction bit has no effect on an output-only pin; it can be used for storage.

11.1.4 Open-Drain Bidirectional Port

Figure 11-4 shows the internal control logic for the open-drain bidirectional port pin. The logic is slightly different from that for the other port types. When the open-drain port pin is configured as an output, clearing the Port Data latch turns on the N-channel driver, resulting in a "hard zero" being present at the pin. A one value in the Port Data Latch shuts off the driver, resulting in a high impedance (input) state at the pin. The open-drain pin can be floated directly by setting its Port Direction bit.

The open-drain ports are not multiplexed with on-board peripherals. The port/peripheral data multiplexer exists for open-drain ports, even though the pins are not shared with peripheral functions. The open-drain port pin floats if the Port Control latch is programmed to select the nonexistent peripheral function.

11.1.5 Port Pin Organization

The port pins are organized as two functional groups, Port 1 and Port 2. Port 1 consists of eight output-only pins. Port 2 has one bidirectional, two output-only, three input-only, and two open-drain bidirectional pins. Most of the port pins are multiplexed with peripheral functions.

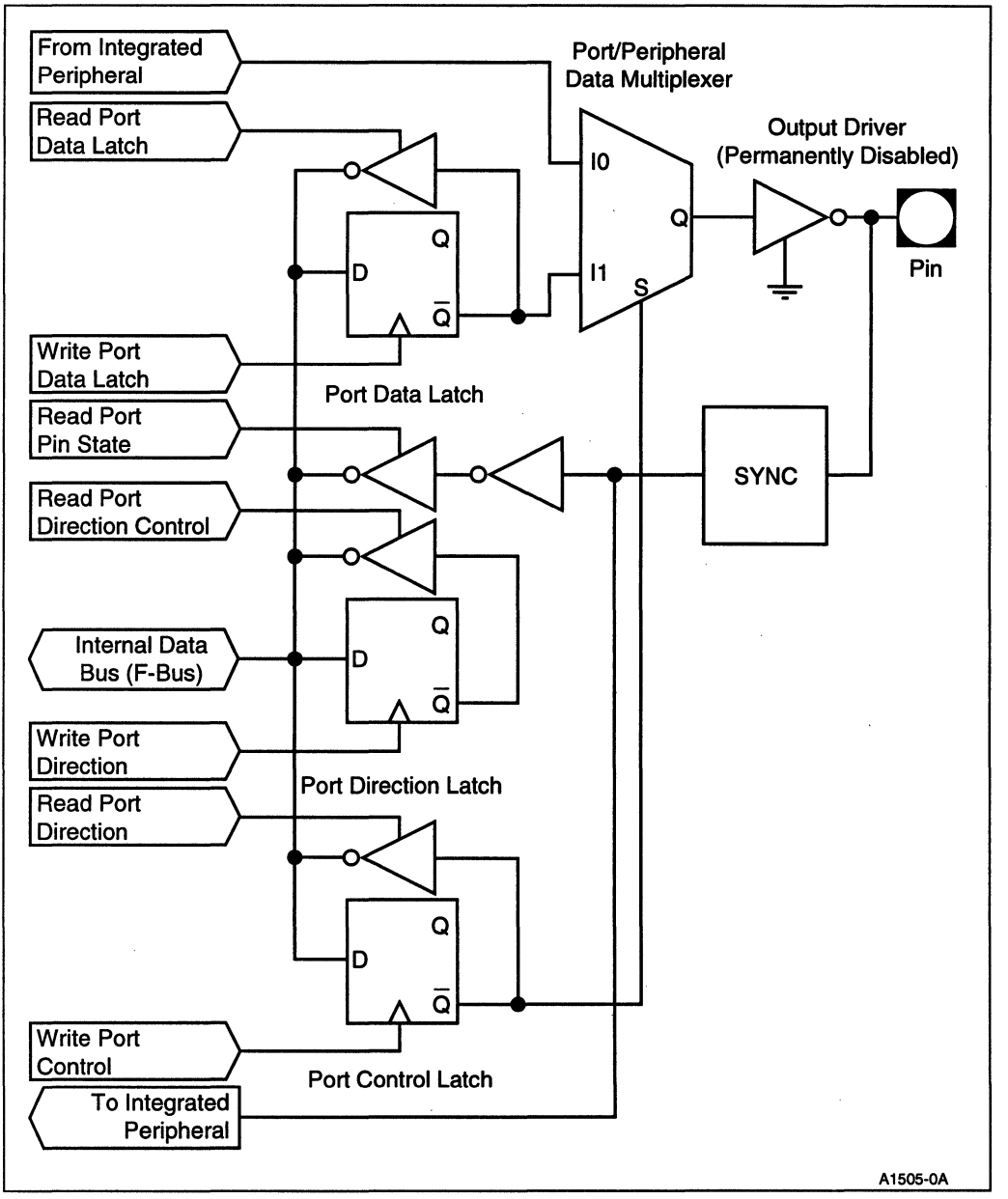


Figure 11-2. Simplified Logic Diagram of an Input Port Pin

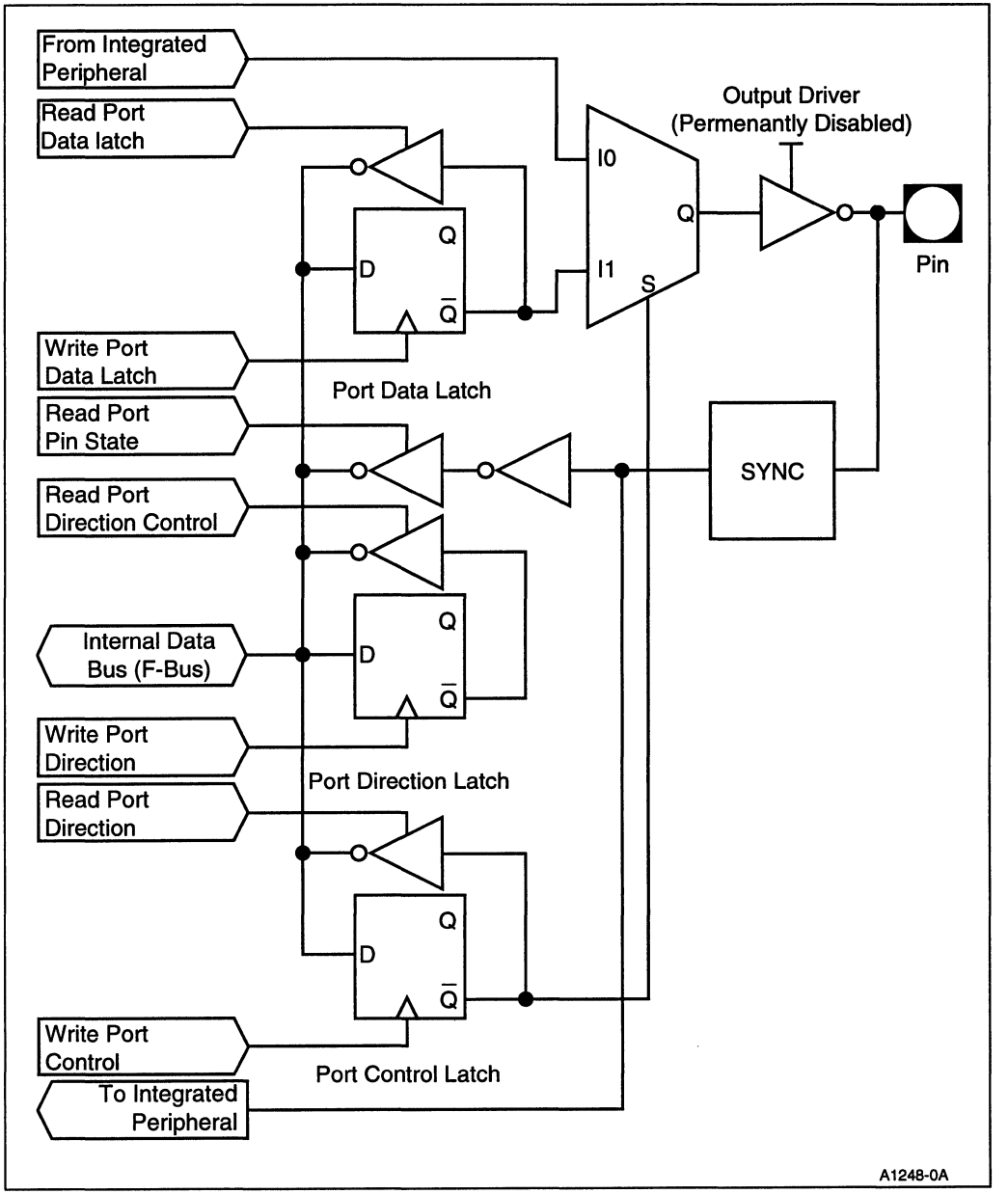


Figure 11-3. Simplified Logic Diagram of an Output Port Pin

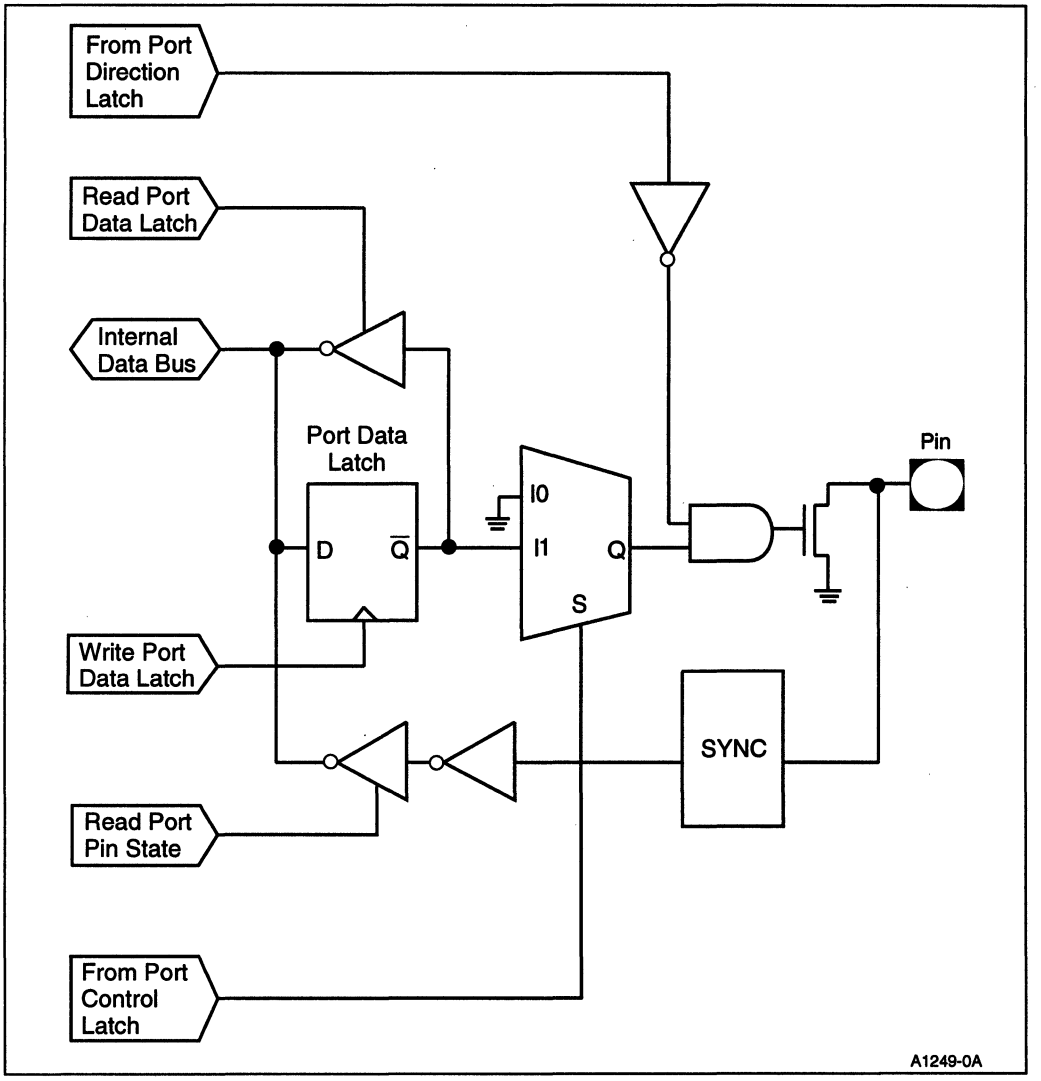


Figure 11-4. Simplified Logic Diagram of an Open-Drain Bidirectional Port

11.1.5.1 Port 1 Organization

Port 1 consists of eight **output-only** port pins. The Port 1 pins are multiplexed with the general-purpose chip-selects ($\overline{GCS7:0}$). Table 11-1 shows the multiplexing options for Port 1.

Table 11-1. Port 1 Multiplexing Options

Pin Name	Peripheral Function	Port Function
P1.7/$\overline{GCS7}$	GCS7	P1.7
P1.6/$\overline{GCS6}$	GCS6	P1.6
P1.5/$\overline{GCS5}$	GCS5	P1.5
P1.4/$\overline{GCS4}$	GCS4	P1.4
P1.3/$\overline{GCS3}$	GCS3	P1.3
P1.2/$\overline{GCS2}$	GCS2	P1.2
P1.1/$\overline{GCS1}$	GCS1	P1.1
P1.0/$\overline{GCS0}$	GCS0	P1.0

11.1.5.2 Port 2 Organization

Six of the Port 2 pins are multiplexed with serial channel functions; the other two provide open-drain bidirectional port pin functions. Table 11-2 shows the Port 2 multiplexing options.

Table 11-2. Port 2 Multiplexing Options

Pin Name	Peripheral Function		Port Function
P2.7	None		P2.7 (Open-drain)
P2.6	None		P2.6 (Open-drain)
P2.5/BCLK0	BCLK0	(Input)	P2.5
P2.4/$\overline{CTS1}$	CTS1	(Input)	P2.4
P2.3/SINT1	SINT1	(Output)	P2.3
P2.2/BCLK1	BCLK1	(Input)	P2.2
P2.1/TXD1	TXD1	(Output)	P2.1
P2.0/RXD1	RXD1	(I/O)	P2.0

11.2 PROGRAMMING THE I/O PORT UNIT

Each port is controlled by a set of four Peripheral Control Block registers: the Port Control Register (PxCON), the Port Direction Register (PxDIR), the Port Data Latch Register (PxLTCH) and the Port Pin State Register (PxPIN).

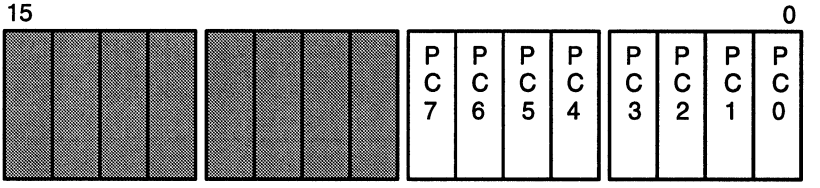
11.2.1 Port Control Register

The Port Control Register (Figure 11-5) selects the overall function for each port pin: peripheral or port. For I/O ports, the Port Control Register is used to assign the pin to either the associated on-chip peripheral or to a general-purpose I/O port. For output-only ports, the Port Control Register selects the source of data for the pin: either an on-chip peripheral or the Port Data latch.

Register Name: Port Control Register

Register Mnemonic: PxCON (P1CON, P2CON)

Register Function: Selects port or peripheral function for a port pin.



Bit Mnemonic	Bit Name	Reset State	Function
PC7:0	Port Control 7:0	FFH	When the PC bit for a specific pin is set, the associated integrated peripheral controls both pin direction and pin data. Clearing the PC bit makes the pin a general-purpose I/O port.

NOTE: Reserved register bits are shown with gray shading. Reserved bits must be written to a logic zero to ensure compatibility with future Intel products.

Figure 11-5. Port Control Register (PxCON)

11.2.2 Port Direction Register

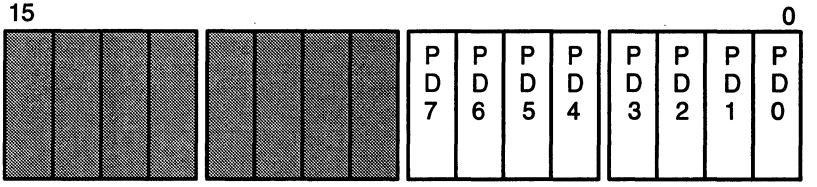
The Port Direction Register (Figure 11-6) controls the direction (input or output) for each pin programmed as a general-purpose I/O port. The Port Direction bit has no effect on output-only port pins. These unused direction control bits can be used for bit storage.

The Port Direction Register is read/write. When read, the register returns the value written to it previously. Pins with their direction fixed return the value in this register, **not** a value indicating their true direction. The direction of a port pin assigned to a peripheral function is controlled by the peripheral; the Port Direction value is ignored.

Register Name: Port Direction Register

Register Mnemonic: PxDIR (P1DIR, P2DIR)

Register Function: Controls the direction of pins programmed as I/O ports.



A1313-0A

Bit Mnemonic	Bit Name	Reset State	Function
PD7:0	Port Direction 7:0	FFH	Setting the PD bit for a pin programmed as a general-purpose I/O port selects the pin as an input. Clearing the PD bit selects the pin as an output.

NOTE: Reserved register bits are shown with gray shading. Reserved bits must be written to a logic zero to ensure compatibility with future Intel products.

Figure 11-6. Port Direction Register (PxDIR)

11.2.3 Port Data Latch Register

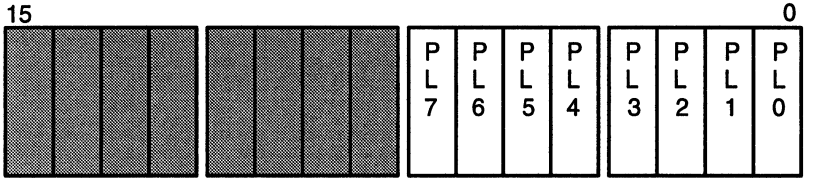
The Port Data Latch Register (Figure 11-7) holds the value to be driven on an output or bidirectional pin. This value appears at the pin only if it is programmed as a port.

The Port Data Latch Register is read/write. Reading a Port Data Latch Register returns the value of the latch itself and **not** that of the associated port pin.

Register Name: Port Data Latch Register

Register Mnemonic: PxLTCH (P1LTCH, P2LTCH)

Register Function: Contains the data driven on pins programmed as output ports.



A1314-0A

Bit Mnemonic	Bit Name	Reset State	Function
PL7:0	Port Data Latch 7:0	FFH	The data written to a PL bit appears on pins programmed as general-purpose output ports.

NOTE: Reserved register bits are shown with gray shading. Reserved bits must be written to a logic zero to ensure compatibility with future Intel products.

Figure 11-7. Port Data Latch Register (PxLTCH)

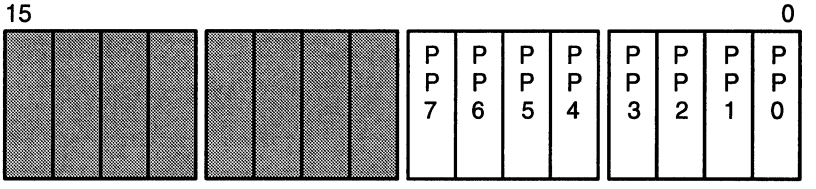
11.2.4 Port Pin State Register

The Port Pin State Register (Figure 11-8) is a read-only register that is used to determine the state of a port pin. When the Port Pin State Register is read, the current state of the port pins is gated to the internal data bus.

Register Name: Port Pin State Register

Register Mnemonic: PxPIN (P1PIN, P2PIN)

Register Function: Reads the logic state at a port pin.



Bit Mnemonic	Bit Name	Reset State	Function
PP7:0	Port Pin State 7:0	XXXXH	Reading the Port Pin State register returns the logic state present on the associated pin.

NOTE: Reserved register bits are shown with gray shading. Reserved bits must be written to a logic zero to ensure compatibility with future Intel products.

Figure 11-8. Port Pin State Register (PxPIN)

11.2.5 Initializing the I/O Ports

The state of the I/O ports following a reset is as follows:

- Port 1 is configured for peripheral function (general-purpose chip-selects, $\overline{\text{GCS7:0}}$).
- Port 2 is configured for peripheral function. The direction of each pin is the default direction for the peripheral function (e.g., P2.1/TXD1 is an output, P2.5/BCLK0 is an input). See Table 11-2 on page 11-7 for details.

There are no set rules for initializing the I/O ports. The Port Data Latch should be programmed before selecting a pin as an output port (to prevent unknown Port Data Latch values from reaching the pins).

11.3 PROGRAMMING EXAMPLE

Example 11-1 shows a typical ASM86 routine to configure the I/O ports. $\overline{GCS7}$ through $\overline{GCS4}$ are routed to the pins, while P1.0 through P1.4 are used as output ports. The binary value 0101 is written to P1.0 through P1.3. The states of pins P2.6 and P2.7 are read and stored in the AL register.

```
$mod186
name              io_port_unit_example
;
;This file contains sample programming code for the 80C186EB I/O Port Unit.
;
;PCB EQUates in an include file.
#include PCBMAP.inc

code_seg          segment public
                  assume cs:code_seg

IO_UNIT_EXAMPL    proc near
;write 0101B to data latch for pins P1.3 through P1.0

                  mov     dx,P1LTCH
                  mov     al,0101b
                  out     dx,al

;Gate data latch to output pins. P1.3 to P1.0 are port pins.

                  mov     dx,P1CON
                  mov     al,0F0h
                  out     dx,al

;Read P2.6 and P2.7.We assume they have not been changed to output pins since
;reset.

                  mov     dx,P2PIN
                  in      al,dx
                  and     al,0C0h;              strip unused bits

;AL now holds the states of the P2.6 and P2.7 pins.

IO_UNIT_EXMPL     endp

code_seg          ends
end
```

Example 11-1. I/O Port Programming Example

intel®

12

Math Coprocessing

intel

CHAPTER 12
MATH COPROCESSING

The 80C186 Modular Core Family meets the need for a general-purpose embedded microprocessor. In most data control applications, efficient data movement and control instructions are foremost and arithmetic performed on the data is simple. However, some applications do require more powerful arithmetic instructions and more complex data types than those provided by the 80C186 Modular Core.

12.1 OVERVIEW OF MATH COPROCESSING

Applications needing advanced mathematics capabilities have the following characteristics.

- Numeric data values are non-integral or vary over a wide range
- Algorithms produce very large or very small intermediate results
- Computations must be precise (i.e., calculations must retain several significant digits)
- Computations must be reliable without dependence on programmed algorithms
- Overall math performance exceeds that afforded by a general-purpose processor and software alone

For the 80C186 Modular Core family, the 80C187 math coprocessor satisfies the need for powerful mathematics. The 80C187 can increase the math performance of the microprocessor system by 50 to 100 times.

12.2 AVAILABILITY OF MATH COPROCESSING

The 80C186 Modular Core supports the 80C187 with a hardware interface under microcode control. However, not all proliferations support the 80C187. Some package types have insufficient leads to support the required external handshaking requirements. The 3-volt versions of the processor do not specify math coprocessing because the 80C187 has only a 5-volt rating. Please refer to the current data sheets for details.

The core has an Escape Trap (ET) bit in the PCB Relocation Register (Figure 4-1 on page 4-2) to control the availability of math coprocessing. If the ET bit is set, an attempted numerics execution results in a Type 7 interrupt. The 80C187 will not work with the 8-bit bus version of the processor because all 80C187 accesses must be 16-bit. The 80C188 Modular Core automatically traps ESC (numerics) opcodes to the Type 7 interrupt, regardless of Relocation Register programming.

12.3 THE 80C187 MATH COPROCESSOR

The 80C187's high performance is due to its 80-bit internal architecture. It contains three units: a Floating Point Unit, a Data Interface and Control Unit and a Bus Control Logic Unit. The foundation of the Floating Point Unit is an 8-element register file, which can be used either as individually addressable registers or as a register stack. The register file allows storage of intermediate results in the 80-bit format. The Floating Point Unit operates under supervision of the Data Interface and Control Unit. The Bus Control Logic Unit maintains handshaking and communications with the host microprocessor. The 80C187 has built-in exception handling.

The 80C187 executes code written for the Intel387™ DX and Intel387 SX math coprocessors. The 80C187 conforms to ANSI/IEEE Standard 754-1985.

12.3.1 80C187 Instruction Set

80C187 instructions fall into six functional groups: data transfer, arithmetic, comparison, transcendental, constant and processor control. Typical 80C187 instructions accept one or two operands and produce a single result. Operands are usually located in memory or the 80C187 stack. Some operands are predefined; for example, FSQRT always takes the square root of the number in the top stack element. Other instructions allow or require the programmer to specify the operand(s) explicitly along with the instruction mnemonic. Still other instructions accept one explicit operand and one implicit operand (usually the top stack element).

As with the basic (non-numerics) instruction set, there are two types of operands for coprocessor instructions, source and destination. Instruction execution does not alter a source operand. Even when an instruction converts the source operand from one format to another (for example, real to integer), the coprocessor performs the conversion in a work area to preserve the source operand. A destination operand differs from a source operand because the 80C187 can alter the register when it receives the result of the operation. For most destination operands, the coprocessor usually replaces the destinations with results.

12.3.1.1 Data Transfer Instructions

Data transfer instructions move operands between elements of the 80C187 register stack or between stack top and memory. Instructions can convert any data type to temporary real and load it onto the stack in a single operation. Conversely, instructions can convert a temporary real operand on the stack to any data type and store it to memory in a single operation. Table 12-1 summarizes the data transfer instructions.

Table 12-1. 80C187 Data Transfer Instructions

Real Transfers	
FLD	Load real
FST	Store real
FSTP	Store real and pop
FXCH	Exchange registers
Integer Transfers	
FILD	Integer load
FIST	Integer store
FISTP	Integer store and pop
Packed Decimal Transfers	
FBLD	Packed decimal (BCD) load
FBSTP	Packed decimal (BCD) store and pop

12.3.1.2 Arithmetic Instructions

The 80C187's arithmetic instruction set includes many variations of add, subtract, multiply, and divide operations and several other useful functions. Examples include a simple absolute value and a square root instruction that executes faster than ordinary division. Other arithmetic instructions perform exact modulo division, round real numbers to integers and scale values by powers of two.

Table 12-2 summarizes the available operation and operand forms for basic arithmetic. In addition to the four normal operations, "reversed" instructions make subtraction and division "symmetrical" like addition and multiplication. In summary, the arithmetic instructions are highly flexible for these reasons:

- the 80C187 uses register or memory operands
- the 80C187 can save results in a choice of registers

Available data types include temporary real, long real, short real, short integer and word integer. The 80C187 performs automatic type conversion to temporary real.

Table 12-2. 80C187 Arithmetic Instructions

Addition	
FADD	Add real
FADDP	Add real and pop
FIADD	Integer add
Subtraction	
FSUB	Subtract real
FSUBP	Subtract real and pop
FISUB	Integer subtract
FSUBR	Subtract real reversed
FSUBRP	Subtract real reversed and pop
FISUBR	Integer subtract reversed
Multiplication	
FMUL	Multiply real
FMULP	Multiply real and pop
FIMUL	Integer multiply

Division	
FDIV	Divide real
FDIVP	Divide real and pop
FIDIV	Integer divide
FDIVR	Divide real reversed
FDIVRP	Divide real reversed and pop
FIDIVR	Integer divide reversed
Other Operations	
FSQRT	Square root
FSCALE	Scale
FPREM	Partial remainder
FRNDINT	Round to integer
FXTRACT	Extract exponent and significand
FABS	Absolute value
FCHS	Change sign
FPREMI	Partial remainder (IEEE)

12.3.1.3 Comparison Instructions

Each comparison instruction (see Table 12-3) analyzes the stack top element, often in relationship to another operand. Then it reports the result in the Status Word condition code. The basic operations are compare, test (compare with zero) and examine (report tag, sign and normalization).

Table 12-3. 80C187 Comparison Instructions

FCOM	Compare real
FCOMP	Compare real and pop
FCOMPP	Compare real and pop twice
FICOM	Integer compare
FICOMP	Integer compare and pop
FTST	Test
FXAM	Examine
FUCOM	Unordered compare
FUCOMP	Unordered compare and pop
FUCOMPP	Unordered compare and pop twice

12.3.1.4 Transcendental Instructions

Transcendental instructions (see Table 12-4) perform the core calculations for common trigonometric, hyperbolic, inverse hyperbolic, logarithmic and exponential functions. Use prologue code to reduce arguments to a range accepted by the instruction. Use epilogue code to adjust the result to the range of the original arguments. The transcendentals operate on the top one or two stack elements and return their results to the stack.

Table 12-4. 80C187 Transcendental Instructions

FPTAN	Partial tangent
FPATAN	Partial arctangent
F2XM1	$2^X - 1$
FYL2X	$Y \log_2 X$
FYL2XP1	$Y \log_2 (X+1)$
FCOS	Cosine
FSIN	Sine
FSINCOS	Sine and Cosine

12.3.1.5 Constant Instructions

Each constant instruction (see Table 12-5) loads a commonly used constant onto the stack. The values have full 80-bit precision and are accurate to about 19 decimal digits. Since a temporary real constant occupies 10 memory bytes, the constant instructions, only 2 bytes long, save memory space.

Table 12-5. 80C187 Constant Instructions

FLDZ	Load + 0.1
FLD1	Load +1.0
FLDPI	Load
FLDL2T	Load $\log_2 10$
FLDL2E	Load $\log_2 e$
FLDLG2	Load $\log_{10} 2$
FLDLN2	Load $\log_e 2$

12.3.1.6 Processor Control Instructions

Computations do not use the processor control instructions; these instructions are available for activities at the operating system level. This group (see Table 12-6) includes initialization, exception handling and task switching instructions.

Table 12-6. 80C187 Processor Control Instructions

FINIT/FNINIT	Initialize processor	FLDENV	Load environment
FDISI/FNDISI	Disable interrupts	FSAVE/FNSAVE	Save state
FENI/FNENI	Enable interrupts	FRSTOR	Restore state
FLDCW	Load control word	FINCSTP	Increment stack pointer
FSTCW/FNSTCW	Store control word	FDECSTP	Decrement stack pointer
FSTSW/FNSTSW	Store status word	FFREE	Free register
FCLEX/FNCLEX	Clear exceptions	FNOP	No operation
FSTENV/FNSTENV	Store environment	FWAIT	CPU wait

12.3.2 80C187 Data Types

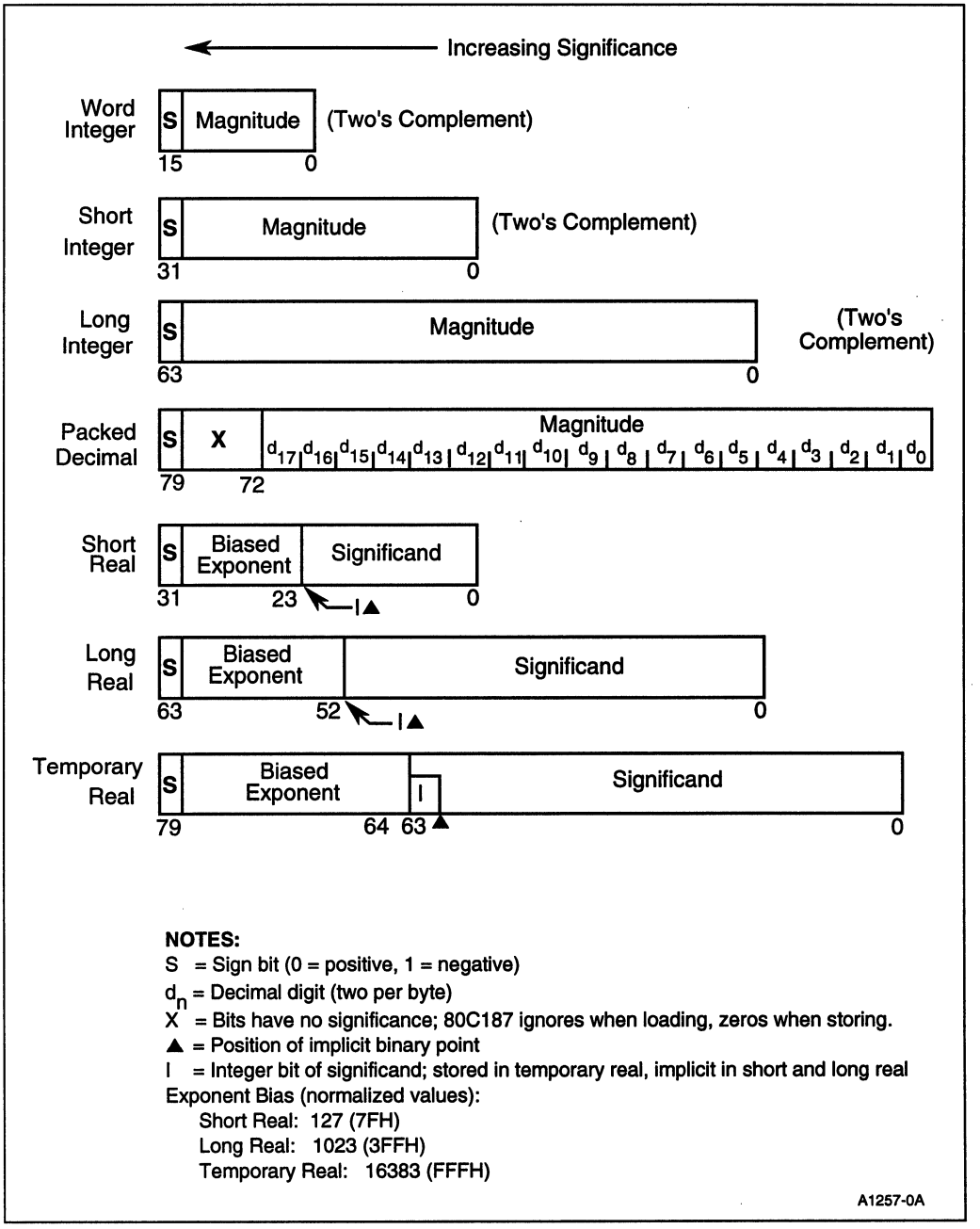
The microprocessor/math coprocessor combination supports seven data types:

- Word Integer — A signed 16-bit numeric value. All operations assume a 2's complement representation.
- Short Integer — A signed 32-bit numeric value (double word). All operations assume a 2's complement representation.
- Long Integer — A signed 64-bit numeric value (quad word). All operations assume a 2's complement representation.
- Packed Decimal — A signed numeric value contained in an 80-bit BCD format.
- Short Real — A signed 32-bit floating point numeric value.
- Long Real — A signed 64-bit floating point numeric value.
- Temporary Real — A signed 80-bit floating point numeric value. Temporary real is the native 80C187 format.

Figure 12-1 graphically represents these data types.

12.4 MICROPROCESSOR AND COPROCESSOR OPERATION

The 80C187 interfaces directly to the microprocessor (as shown in Figure 12-2) and operates as an I/O-mapped slave peripheral device. Hardware handshaking requires connections between the 80C187 and four special pins on the processor: $\overline{\text{NCS}}$, BUSY, PEREQ and $\overline{\text{ERROR}}$. These pins are not available in some package types. Refer to the data sheet for details.



NOTES:

S = Sign bit (0 = positive, 1 = negative)

d_n = Decimal digit (two per byte)

X = Bits have no significance; 80C187 ignores when loading, zeros when storing.

▲ = Position of implicit binary point

I = Integer bit of significand; stored in temporary real, implicit in short and long real

Exponent Bias (normalized values):

Short Real: 127 (7FH)

Long Real: 1023 (3FFH)

Temporary Real: 16383 (FFFH)

A1257-0A

Figure 12-1. 80C187-Supported Data Types

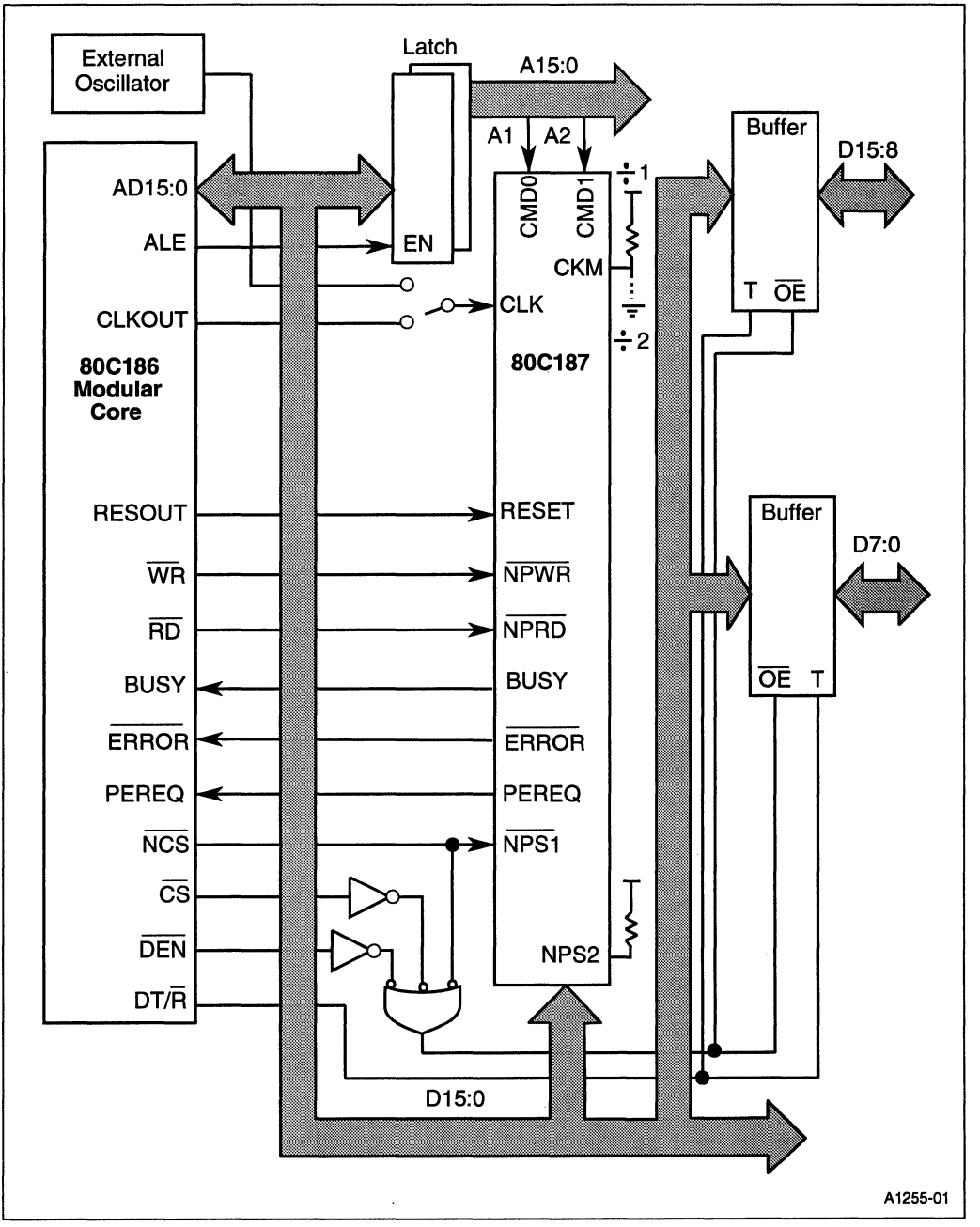


Figure 12-2. 80C186 Modular Core Family/80C187 System Configuration

12.4.1 Clocking the 80C187

The microprocessor and math coprocessor operate asynchronously, and their clock rates may differ. The 80C187 has a CKM pin that determines whether it uses the input clock directly or divided by two. Direct clocking works up to 12.5 MHz, which makes it convenient to feed the clock input from the microprocessor's CLKOUT pin. Beyond 12.5 MHz, the 80C187 must use a multiply-by-two clock input up to a maximum of 32 MHz. The microprocessor and the math coprocessor have correct timing relationships, even with operation at different frequencies.

12.4.2 Processor Bus Cycles Accessing the 80C187

Data transfers between the microprocessor and the 80C187 occur through the dedicated, 16-bit I/O ports shown in Table 12-7. When the processor encounters a numerics opcode, it first writes the opcode to the 80C187. The 80C187 decodes the instruction and passes elementary instruction information (Opcode Status Word) back to the processor. Since the 80C187 is a slave processor, the Modular Core processor performs all loads and stores to memory. Including the overhead in the microprocessor's microcode, each data transfer between memory and the 80C187 (via the microprocessor) takes at least 17 processor clocks.

Table 12-7. 80C187 I/O Port Assignments

I/O Address	Read Definition	Write Definition
00F8H	Status/Control	Opcode
00FAH	Data	Data
00FCH	Reserved	CS:IP, DS:EA
00FEH	Opcode Status	Reserved

The microprocessor cannot process any numerics (ESC) opcodes alone. If the CPU encounters a numerics opcode when the Escape Trap (ET) bit in the Relocation Register is a zero and the 80C187 is not present, its operation is indeterminate. Even the FINIT/FNINIT initialization instruction (used in the past to test the presence of a coprocessor) fails without the 80C187. If an application offers the 80C187 as an option, problems can be prevented in one of three ways:

- Remove all numerics (ESC) instructions, including code that checks for the presence of the 80C187.
- Use a jumper or switch setting to indicate the presence of the 80C187. The program can interrogate the jumper or switch setting and branch away from numerics instructions when the 80C187 socket is empty.
- Trick the microprocessor into predictable operation when the 80C187 socket is empty. The fix is placing pull-up or pull-down resistors on certain data and handshaking lines so the CPU reads a recognizable Opcode Status Word. This solution requires a detailed knowledge of the interface.

Bus cycles involving the 80C187 Math Coprocessor behave exactly like other I/O bus cycles with respect to the processor's control pins. See "System Design Tips" for information on integrating the 80C187 into the overall system.

12.4.3 System Design Tips

All 80C187 operations require that bus ready be asserted. The simplest way to return the ready indication is through hardware connected to the processor's external ready pin. If you program a chip-select to cover the math coprocessor port addresses, its ready programming is in force and can provide bus ready for coprocessor accesses. The user must verify that there are no conflicts from other hardware connected to that chip-select pin.

A chip-select pin goes active on 80C187 accesses if you program it for a range including the math coprocessor I/O ports. The converse is not true — a non-80C187 access cannot activate $\overline{\text{NCS}}$ (numerics coprocessor select), regardless of programming.

In a buffered system, it is customary to place the 80C187 on the local bus. Since $\overline{\text{DTR}}$ and $\overline{\text{DEN}}$ function normally during 80C187 transfers, you must qualify $\overline{\text{DEN}}$ with $\overline{\text{NCS}}$ (see Figure 12-3). Otherwise, contention between the 80C187 and the transceivers occurs on read cycles to the 80C187.

The microprocessor's local bus is available to the integrated peripherals during numerics execution whenever the CPU is not communicating with the 80C187. The idle bus allows the processor to intersperse DRAM refresh cycles with accesses to the 80C187.

The microprocessor's local bus is available to alternate bus masters during execution of numerics instructions when the CPU does not need it. Bus cycles driven by alternate masters (via the HOLD/$\overline{\text{HLDA}}$ protocol) can suspend coprocessor bus cycles for an indefinite period.

The programmer can lock 80C187 instructions. The CPU asserts the $\overline{\text{LOCK}}$ pin for the entire duration of a numerics instruction, monopolizing the bus for a very long time.

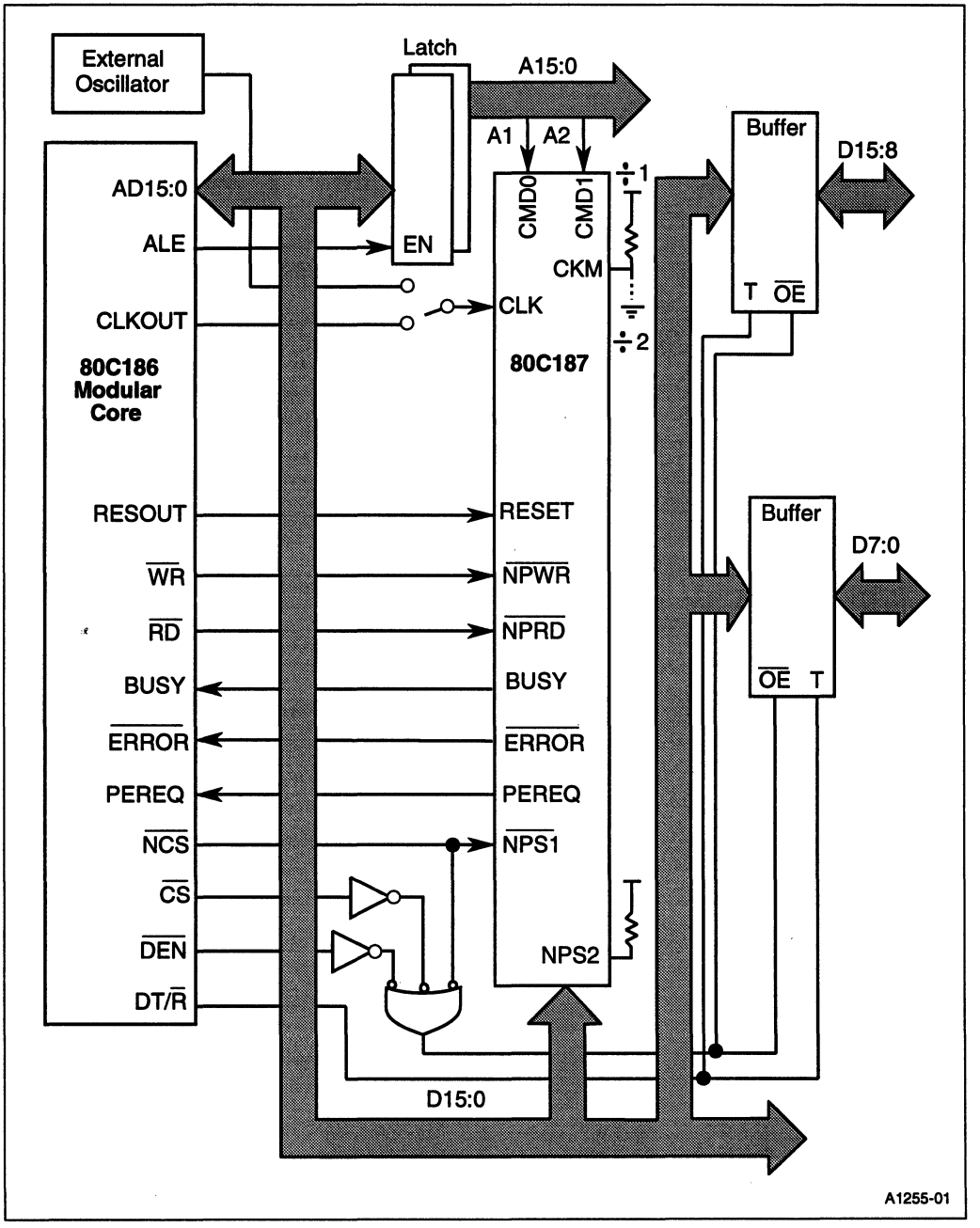


Figure 12-3. 80C187 Configuration with a Partially Buffered Bus

12.4.4 Exception Trapping

The 80C187 detects six error conditions that can occur during instruction execution. The 80C187 can apply default fix-ups or signal exceptions to the microprocessor's $\overline{\text{ERROR}}$ pin. The processor tests $\overline{\text{ERROR}}$ at the beginning of numerics instructions, so it traps an exception on the **next** attempted numerics instruction after it occurs. When $\overline{\text{ERROR}}$ tests active, the processor executes a Type 16 interrupt.

There is no automatic exception-trapping on the last numerics instruction of a series. If the last numerics instruction writes an invalid result to memory, subsequent non-numerics instructions can use that result as if it is valid, further compounding the original error. Insert the FNOP instruction at the end of the 80C187 routine to force an $\overline{\text{ERROR}}$ check. If the program is written in a high-level language, it is impossible to insert FNOP. In this case, route the error signal through an inverter to an interrupt pin on the microprocessor (see Figure 12-4). With this arrangement, use a flip-flop to latch BUSY upon assertion of $\overline{\text{ERROR}}$. The latch gets cleared during the exception-handler routine. Use an additional flip-flop to latch PEREQ to maintain the correct handshaking sequence with the microprocessor.

12.5 Example Math Coprocessor Routines

Example 12-1 shows the initialization sequence for the 80C187. Example 12-2 is an example of a floating point routine using the 80C187. The FSINCOS instruction yields both sine and cosine in one operation.

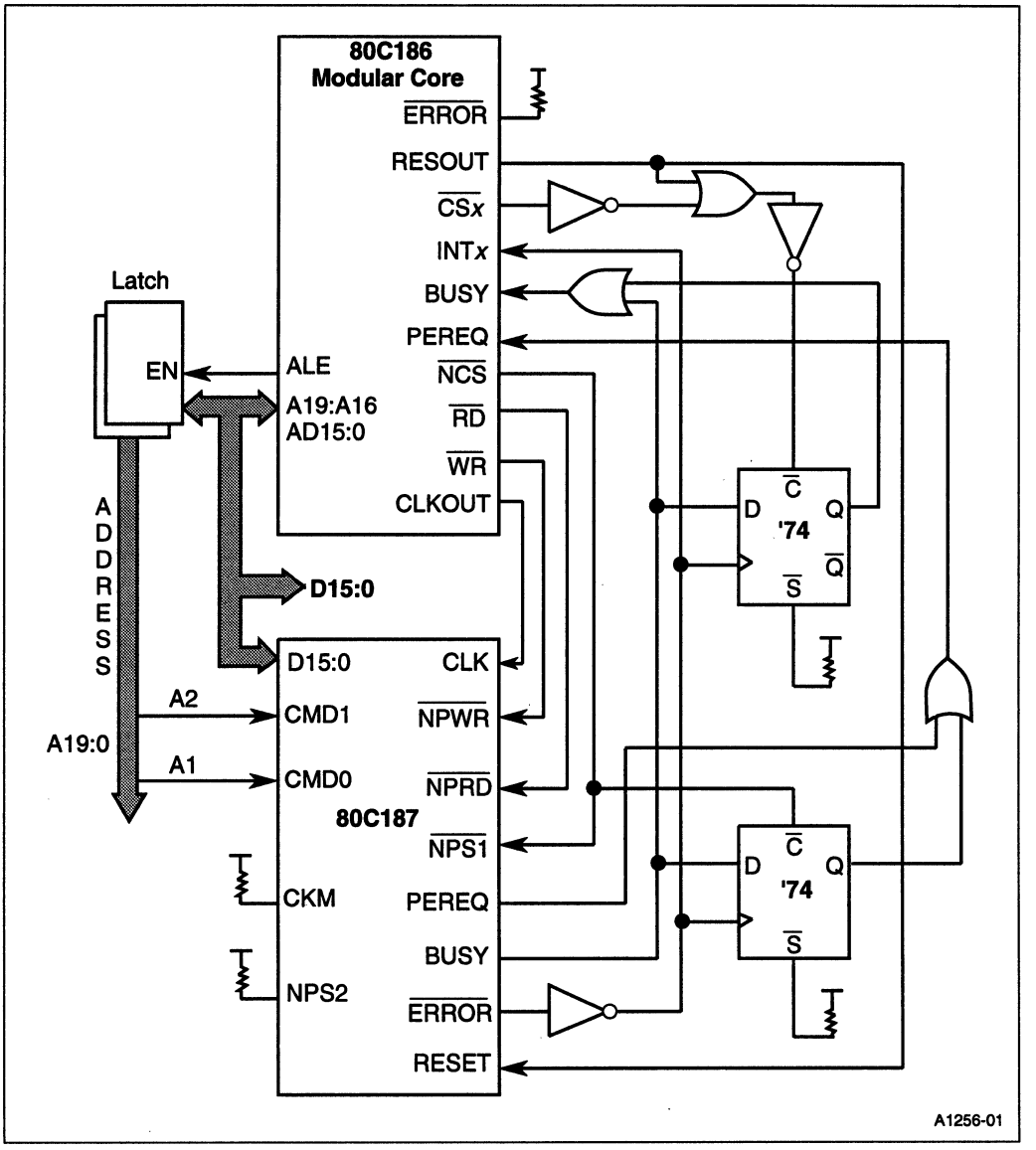


Figure 12-4. 80C187 Exception Trapping via Processor Interrupt Pin

```
$mod186
name    example_80C187_init
;
;FUNCTION:      This function initializes the 80C187 numerics coprocessor.
;
;SYNTAX:        extern unsigned char far 187_init(void);
;
;INPUTS:        None
;
;OUTPUTS:       unsigned char - 0000h -> False -> coprocessor not initialized
;                               ffffh -> True  -> coprocessor initialized
;
;NOTE:          Parameters are passed on the stack as required by
;               high-level languages.
;
lib_80186       segment public 'code'
                assume cs:lib_80186

                public  _187_init

_187_initproc far

        push    bp                      ;save caller's bp
        mov     bp, sp                  ;get current top of stack

        cli                             ;disable maskable interrupts

        fninit                          ;init 80C187 processor
        fnstcw  [bp-2]                  ;get current control word

        sti                             ;enable interrupts

        mov     ax, [bp-2]
        and     ax, 0300h               ;mask off unwanted control bits
        cmp     ax, 0300h               ;PC bits = 11
        je      Ok                      ;yes: processor ok
        xor     ax, ax                  ;return false (80C187 not ok)
        pop     bp                      ;restore caller's bp
        ret

    Ok: and     [bp-2], 0fffeh          ;unmask possible exceptions
        fldcw   [bp-2]

        mov     ax,0ffffh               ;return true (80C187 ok)
        pop     bp                      ;restore caller's bp
        ret

_187_initendp

lib_80186ends
        end
```

Example 12-1. Initialization Sequence for 80C187 Math Coprocessor

```
$mod186
$modc187

name    example_80C187_proc

;DESCRIPTION:   This code section uses the 80C187 FSINCOS transcendental
;               instruction to convert the locus of a point from polar
;               to Cartesian coordinates.
;
;VARIABLES:     The variables consist of the radius, r, and the angle, theta.
;               Both are expressed as 32-bit reals and 0 <= theta <= pi/4.
;
;RESULTS:       The results of the computation are the coordinates x and y
;               expressed as 32-bit reals.
;
;NOTES:         This routine is coded for Intel ASM86. It is not set up as an
;               HLL-callable routine.
;
;               This code assumes that the 80C187 has already been initialized.
;
        assume cs:code, ds:data

        data    segment at 0100h
                r       dd x.xxxx       ;substitute real operand
                theta   dd x.xxxx       ;substitute real operand
                x       dd ?
                y       dd ?
        data    ends

        code    segment at 0080h

convert         proc far
        mov     ax, data
        mov     ds, ax

        fld     r                       ;load radius
        fld     theta                   ;load angle
        fsincos                         ;st=cos, st(1)=sin
        fmul    st, st(2)               ;compute x
        fstp    x                       ;store to memory and pop
        fmul                            ;compute y
        fstp    y                       ;store to memory and pop

convert endp

code    ends
        end
```

Example 12-2. Floating Point Math Routine Using FSINCOS

intel®

13

ONCE Mode

CHAPTER 13
ONCE MODE

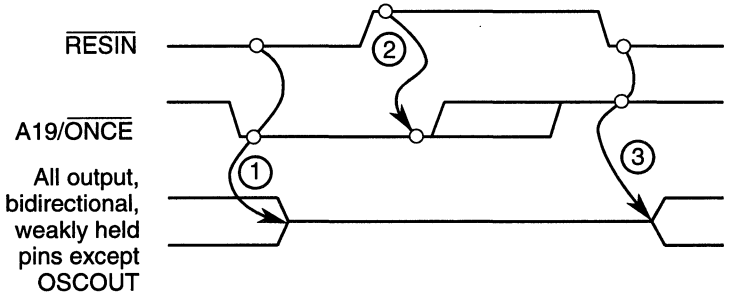
ONCE (pronounced "ahnce") Mode provides the ability to three-state all output, bidirectional, or weakly held high/low pins except OSCOUT. To allow device operation with a crystal network, OSCOUT does not three-state.

ONCE Mode electrically isolates the device from the rest of the board logic. This isolation allows a bed-of-nails tester to drive the device pins directly for more accurate and thorough testing. An in-circuit emulation probe uses ONCE Mode to isolate a surface-mounted device from board logic and essentially "take over" operation of the board (without removing the soldered device from the board).

13.1 ENTERING/LEAVING ONCE MODE

Forcing A19/$\overline{\text{ONCE}}$ low while $\overline{\text{RESIN}}$ is asserted (low) enables ONCE Mode (see Figure 13-1). Maintaining A19/$\overline{\text{ONCE}}$ and $\overline{\text{RESIN}}$ low continues to keep ONCE Mode active. Returning A19/$\overline{\text{ONCE}}$ high exits ONCE Mode.

However, it is possible to keep ONCE Mode always active by deasserting $\overline{\text{RESIN}}$ while keeping A19/$\overline{\text{ONCE}}$ low. Removing $\overline{\text{RESIN}}$ "latches" ONCE Mode and allows A19/$\overline{\text{ONCE}}$ to be driven to any level. A19/$\overline{\text{ONCE}}$ must remain low for at least one clock beyond the time $\overline{\text{RESIN}}$ is driven high. Asserting $\overline{\text{RESIN}}$ exits ONCE Mode, assuming A19/$\overline{\text{ONCE}}$ does not also remain low (see Figure 13-1).



NOTES: 1. Entering ONCE Mode.
2. Latching ONCE Mode.
3. Leaving ONCE Mode (assuming 2 occurred).

A1260-0A

Figure 13-1. Entering/Leaving ONCE Mode

intel®

A

80C186 Instruction Set Additions and Extensions

intel®

APPENDIX A
80C186 INSTRUCTION SET ADDITIONS AND EXTENSIONS

The 80C186 Modular Core family instruction set differs from the original 8086/8088 instruction set in two ways. First, several instructions that were not available in the 8086/8088 instruction set have been added. Second, several 8086/8088 instructions have been enhanced for the 80C186 Modular Core family instruction set.

A.1 80C186 INSTRUCTION SET ADDITIONS

This section describes the seven instructions that were added to the base 8086/8088 instruction set to make the instruction set for the 80C186 Modular Core family. These instructions did not exist in the 8086/8088 instruction set.

- Data transfer instructions
  - — PUSHA
  - — POPA
- String instructions
  - — INS
  - — OUTS
- High-level instructions
  - — ENTER
  - — LEAVE
  - — BOUND

A.1.1 Data Transfer Instructions

PUSHA/POPA

PUSHA (push all) and POPA (pop all) allow all general-purpose registers to be stacked and unstacked. The PUSHA instruction pushes all CPU registers (except as noted below) onto the stack. The POPA instruction pops all registers pushed by PUSHA off of the stack. The registers are pushed onto the stack in the following order: AX, CX, DX, BX, SP, BP, SI, DI. The Stack Pointer (SP) value pushed is the Stack Pointer value before the AX register was pushed. When POPA is executed, the Stack Pointer value is popped, but ignored. Note that this instruction does not save segment registers (CS, DS, SS, ES), the Instruction Pointer (IP), the Processor Status Word or any integrated peripheral registers.

A.1.2 String Instructions

INS *source_string, port*

INS (in string) performs block input from an I/O port to memory. The port address is placed in the DX register. The memory address is placed in the DI register. This instruction uses the ES segment register (which cannot be overridden). After the data transfer takes place, the pointer register (DI) increments or decrements, depending on the value of the Direction Flag (DF). The pointer register changes by one for byte transfers or by two for word transfers.

OUTS *port, destination_string*

OUTS (out string) performs block output from memory to an I/O port. The port address is placed in the DX register. The memory address is placed in the SI register. This instruction uses the DS segment register, but this may be changed with a segment override instruction. After the data transfer takes place, the pointer register (SI) increments or decrements, depending on the value of the Direction Flag (DF). The pointer register changes by one for byte transfers or by two for word transfers.

A.1.3 High-Level Instructions

ENTER *size, level*

ENTER creates the stack frame required by most block-structured high-level languages. The first parameter, *size*, specifies the number of bytes of dynamic storage to be allocated for the procedure being entered (16-bit value). The second parameter, *level*, is the lexical nesting level of the procedure (8-bit value). Note that the higher the lexical nesting level, the lower the procedure is in the nesting hierarchy.

The lexical nesting level determines the number of pointers to higher level stack frames copied into the current stack frame. This list of pointers is called the *display*. The first word of the display points to the previous stack frame. The display allows access to variables of higher level (lower lexical nesting level) procedures.

After ENTER creates a display for the current procedure, it allocates dynamic storage space. The Stack Pointer decrements by the number of bytes specified by *size*. All PUSH and POP operations in the procedure use this value of the Stack Pointer as a base.

Two forms of ENTER exist: non-nested and nested. A lexical nesting level of 0 specifies the non-nested form. In this situation, BP is pushed, then the Stack Pointer is copied to BP and decremented by the size of the frame. If the lexical nesting level is greater than 0, the nested form is used. Figure A-1 gives the formal definition of ENTER.

```
The following listing gives the formal definition of the
ENTER instruction for all cases.
LEVEL denotes the value of the second operand.

Push BP
Set a temporary value FRAME_PTR: = SP
If LEVEL > 0 then
                Repeat (LEVEL - 1) times:
                BP:=BP - 2
                Push the word pointed to by BP
                End Repeat
                Push FRAME_PTR
End if
BP:=FRAME_PTR
SP:=SP - first operand
```

Figure A-1. Formal Definition of ENTER

ENTER treats a reentrant procedure as a procedure calling another procedure at the same lexical level. A reentrant procedure can address only its own variables and variables of higher-level calling procedures. ENTER ensures this by copying only stack frame pointers from higher-level procedures.

Block-structured high-level languages use lexical nesting levels to control access to variables of previously nested procedures. For example, assume for Figure A-2 that Procedure A calls Procedure B, which calls Procedure C, which calls Procedure D. Procedure C will have access to the variables of Main and Procedure A, but not to those of Procedure B because Procedures C and B operate at the same lexical nesting level.

The following is a summary of the variable access for Figure A-2.

1. Main has variables at fixed locations.
2. Procedure A can access only the fixed variables of Main.
3. Procedure B can access only the variables of Procedure A and Main.
   Procedure B cannot access the variables of Procedure C or Procedure D.
4. Procedure C can access only the variables of Procedure A and Main.
   Procedure C cannot access the variables of Procedure B or Procedure D.
5. Procedure D can access the variables of Procedure C, Procedure A and Main.
   Procedure D cannot access the variables of Procedure B.

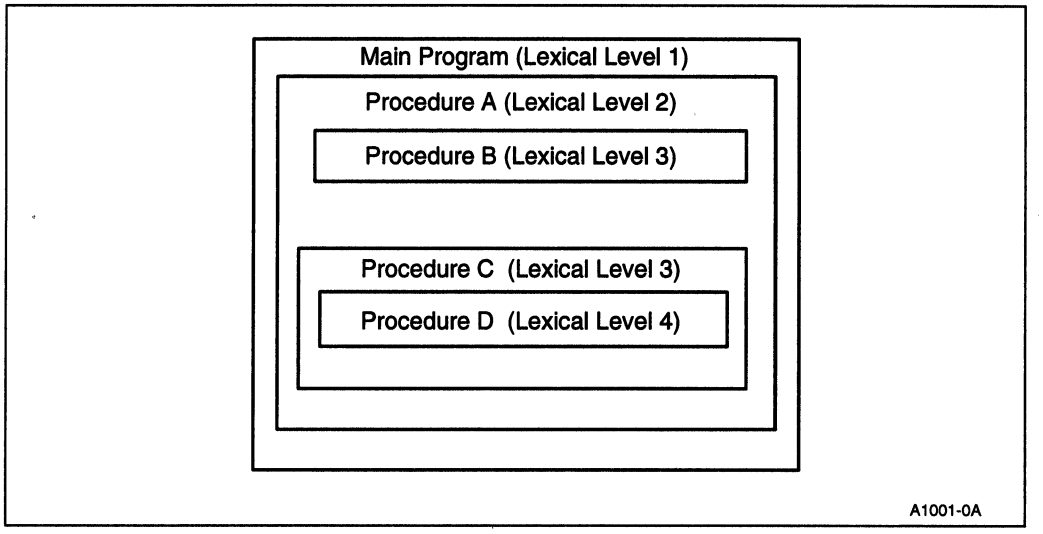


Figure A-2. Variable Access in Nested Procedures

The first ENTER, executed in the Main Program, allocates dynamic storage space for Main, but no pointers are copied. The only word in the display points to itself because no previous value exists to return to after LEAVE is executed (see Figure A-3).

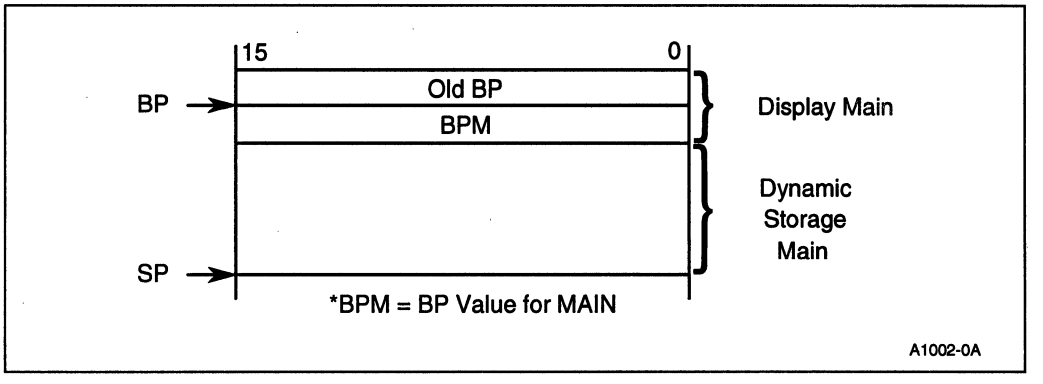


Figure A-3. Stack Frame for Main at Level 1

After Main calls Procedure A, ENTER creates a new display for Procedure A. The first word points to the previous value of BP (BPM). The second word points to the current value of BP (BPA). BPM contains the base for dynamic storage in Main. All dynamic variables for Main will be at a fixed offset from this value (see Figure A-4).

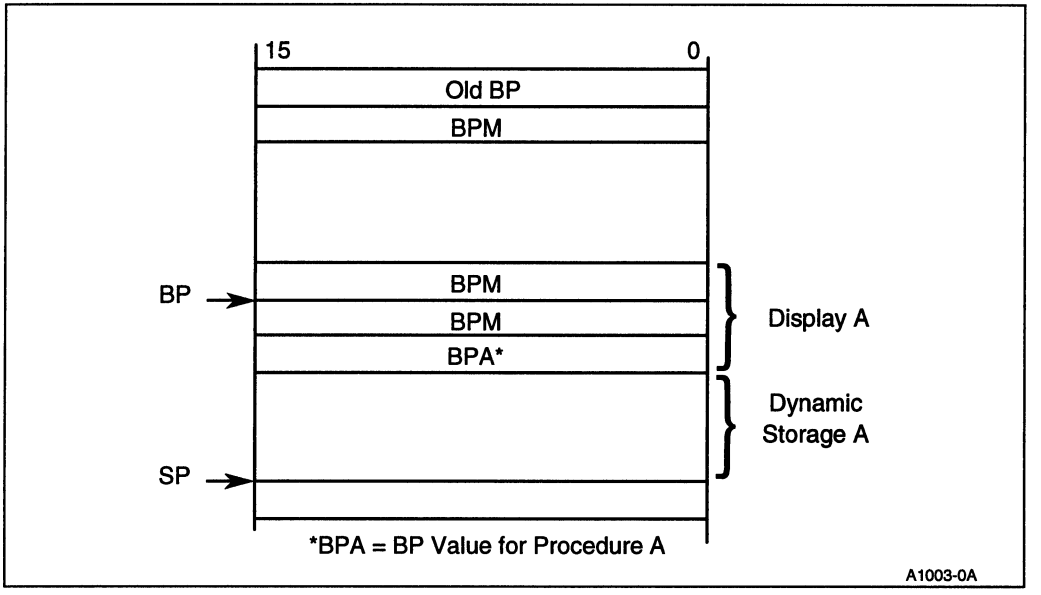


Figure A-4. Stack Frame for Procedure A at Level 2

After Procedure A calls Procedure B, ENTER creates the display for Procedure B. The first word of the display points to the previous value of BP (BPA). The second word points to the value of BP for MAIN (BPM). The third word points to the BP for Procedure A (BPA). The last word points to the current BP (BPB). Procedure B can access variables in Procedure A or Main via the appropriate BP in the display (see Figure A-5).

After Procedure B calls Procedure C, ENTER creates the display for Procedure C. The first word of the display points to the previous value of BP (BPB). The second word points to the value of BP for MAIN (BPM). The third word points to the value of BP for Procedure A (BPA). The fourth word points to the current BP (BPC). Because Procedure B and Procedure C have the same lexical nesting level, Procedure C cannot access variables in Procedure B. The only pointer to Procedure B in the display of Procedure C exists to allow the LEAVE instruction to collapse the Procedure C stack frame (see Figure A-6).

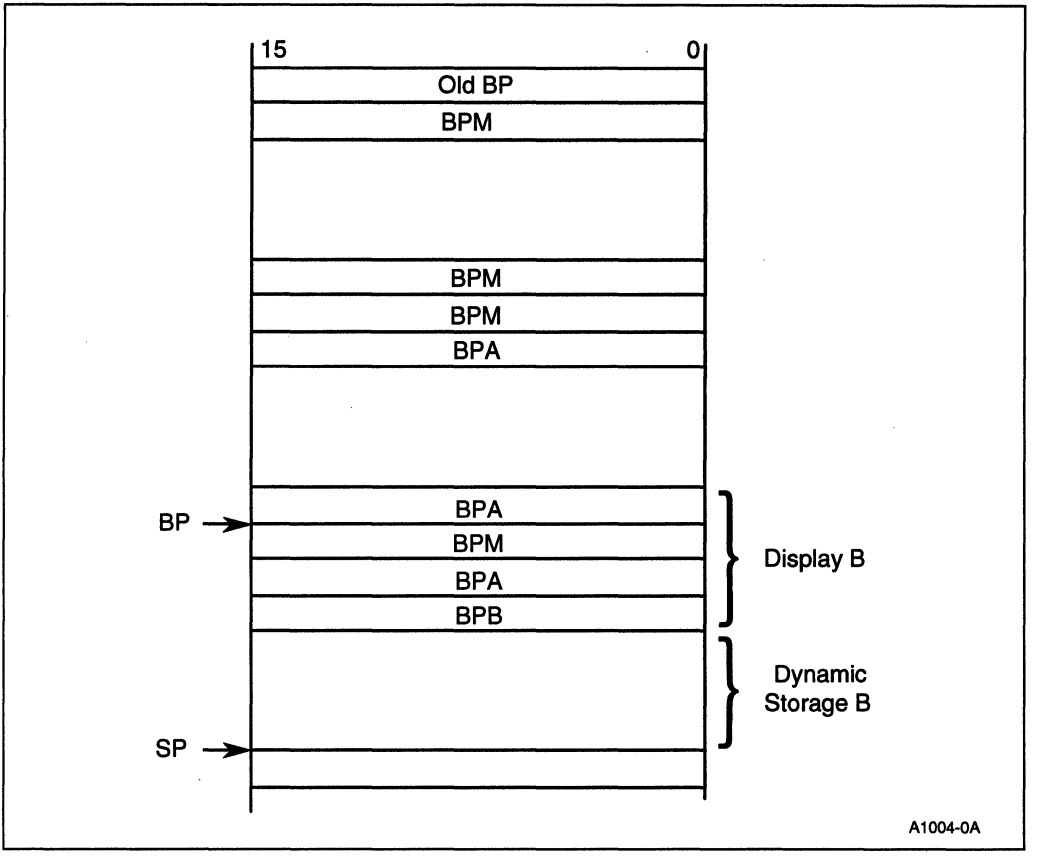


Figure A-5. Stack Frame for Procedure B at Level 3 Called from A

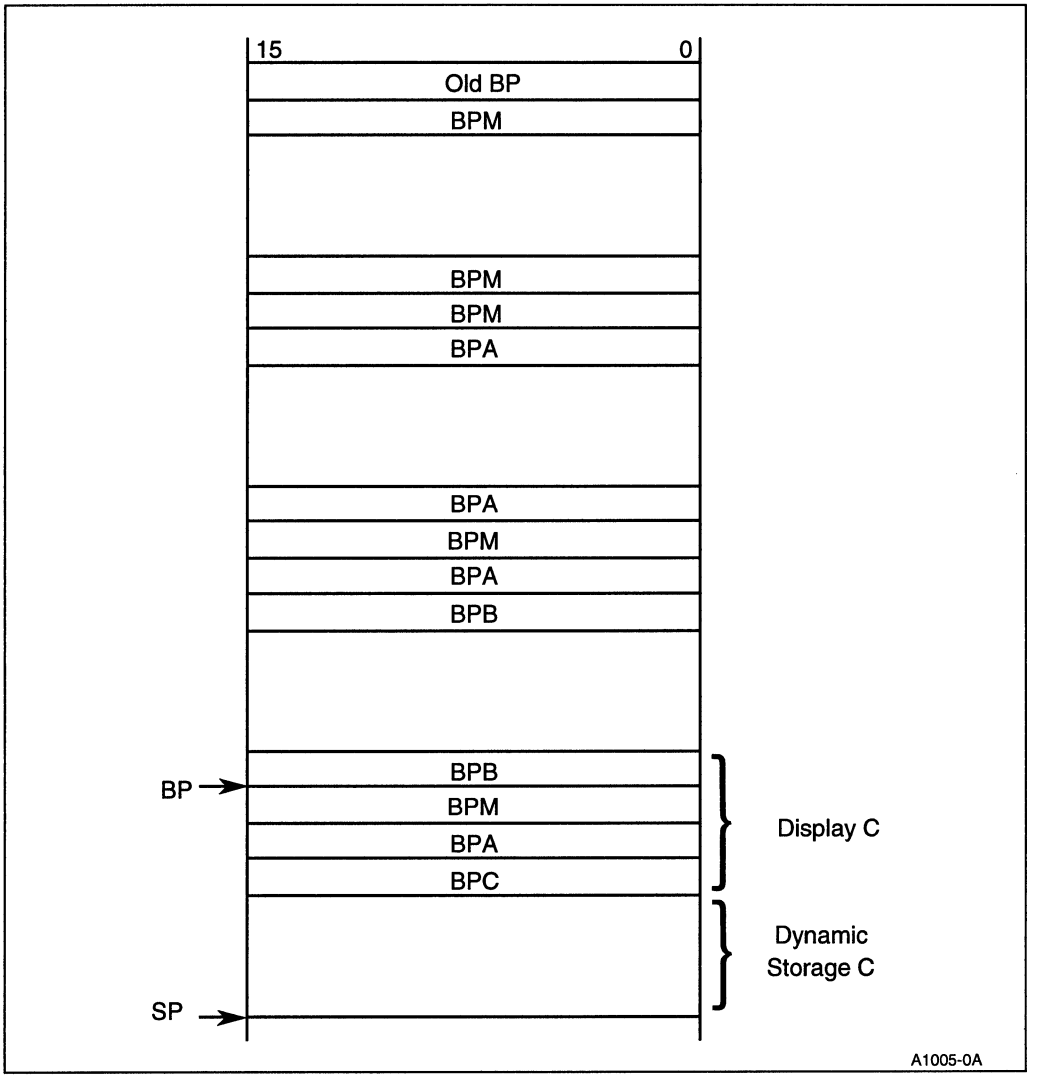


Figure A-6. Stack Frame for Procedure C at Level 3 Called from B

LEAVE

LEAVE reverses the action of the most recent ENTER instruction. It collapses the last stack frame created. First, LEAVE copies the current BP to the Stack Pointer, releasing the stack space allocated to the current procedure. Second, LEAVE pops the old value of BP from the stack, to return to the calling procedure's stack frame. A RET instruction will remove arguments stacked by the calling procedure for use by the called procedure.

BOUND *register, address*

BOUND verifies that the signed value in the specified register lies within specified limits. If the value does not lie within the bounds, an array bounds exception (type 5) occurs. BOUND is useful for checking array bounds before attempting to access an array element. This prevents the program from overwriting information outside the limits of the array.

BOUND has two operands. The first, *register*, specifies the register being tested. The second, *address*, contains the effective relative address of the two signed boundary values. The lower limit word is at this address and the upper limit word immediately follows. The limit values cannot be register operands (if they are, an invalid opcode exception occurs).

A.2 80C186 INSTRUCTION SET ENHANCEMENTS

This section describes ten instructions that were available with the 8086/8088 but have been enhanced for the 80C186 Modular Core family.

- Data transfer instructions
  - — PUSH
- Arithmetic instructions
  - — IMUL
- Bit manipulation instructions (shifts and rotates)
  - — SAL
  - — SHL
  - — SAR
  - — SHR
  - — ROL
  - — ROR
  - — RCL
  - — RCR

A.2.1 Data Transfer Instructions

PUSH *data*

PUSH (push immediate) allows an immediate argument, *data*, to be pushed onto the stack. The value can be either a byte or a word. Byte values are sign extended to word size before being pushed.

A.2.2 Arithmetic Instructions

IMUL *destination, source, data*

IMUL (integer immediate multiply, signed) allows a value to be multiplied by an immediate operand. IMUL requires three operands. The first, *destination*, is the register where the result will be placed. The second, *source*, is the effective address of the multiplier. The source may be the same register as the destination, another register or a memory location. The third, *data*, is an immediate value used as the multiplicand. The *data* operand may be a byte or word. If *data* is a byte, it is sign extended to 16 bits. Only the lower 16 bits of the result are saved. The result must be placed in a general-purpose register.

A.2.3 Bit Manipulation Instructions

This section describes the eight enhanced bit-manipulation instructions.

A.2.3.1 Shift Instructions

SAL *destination, count*

SAL (immediate shift arithmetic left) shifts the destination operand left by an immediate value. SAL has two operands. The first, *destination*, is the effective address to be shifted. The second, *count*, is an immediate byte value representing the number of shifts to be made. The CPU will AND *count* with 1FH before shifting, to allow no more than 32 shifts. Zeros shift in on the right.

SHL *destination, count*

SHL (immediate shift logical left) is physically the same instruction as SAL (immediate shift arithmetic left).

SAR *destination, count*

SAR (immediate shift arithmetic right) shifts the destination operand right by an immediate value. SAL has two operands. The first, *destination*, is the effective address to be shifted. The second, *count*, is an immediate byte value representing the number of shifts to be made. The CPU will AND *count* with 1FH before shifting, to allow no more than 32 shifts. The value of the original sign bit shifts into the most-significant bit to preserve the initial sign.

SHR *destination, count*

SHR (immediate shift logical right) is physically the same instruction as SAR (immediate shift arithmetic right).

A.2.3.2 Rotate Instructions

ROL *destination, count*

ROL (immediate rotate left) rotates the destination byte or word left by an immediate value. ROL has two operands. The first, *destination*, is the effective address to be rotated. The second, *count*, is an immediate byte value representing the number of rotations to be made. The most-significant bit of *destination* rotates into the least-significant bit.

ROR *destination, count*

ROR (immediate rotate right) rotates the destination byte or word right by an immediate value. ROR has two operands. The first, *destination*, is the effective address to be rotated. The second, *count*, is an immediate byte value representing the number of rotations to be made. The least-significant bit of *destination* rotates into the most-significant bit.

RCL *destination, count*

RCL (immediate rotate through carry left) rotates the destination byte or word left by an immediate value. RCL has two operands. The first, *destination*, is the effective address to be rotated. The second, *count*, is an immediate byte value representing the number of rotations to be made. The Carry Flag (CF) rotates into the least-significant bit of *destination*. The most-significant bit of *destination* rotates into the Carry Flag.

RCR *destination, count*

RCR (immediate rotate through carry right) rotates the destination byte or word right by an immediate value. RCR has two operands. The first, *destination*, is the effective address to be rotated. The second, *count*, is an immediate byte value representing the number of rotations to be made. The Carry Flag (CF) rotates into the most-significant bit of *destination*. The least-significant bit of *destination* rotates into the Carry Flag.

B

Input Synchronization

APPENDIX B
INPUT SYNCHRONIZATION

Many input signals to an embedded processor are asynchronous. Asynchronous signals do not **require** a specified setup or hold time to ensure the device does not incur a failure. However, asynchronous setup and hold times are specified in the data sheet to ensure **recognition**. Associated with each of these inputs is a synchronizing circuit (see Figure B-1) that samples the asynchronous signal and synchronizes it to the internal operating clock. The output of the synchronizing circuit is then safely routed to the logic units.

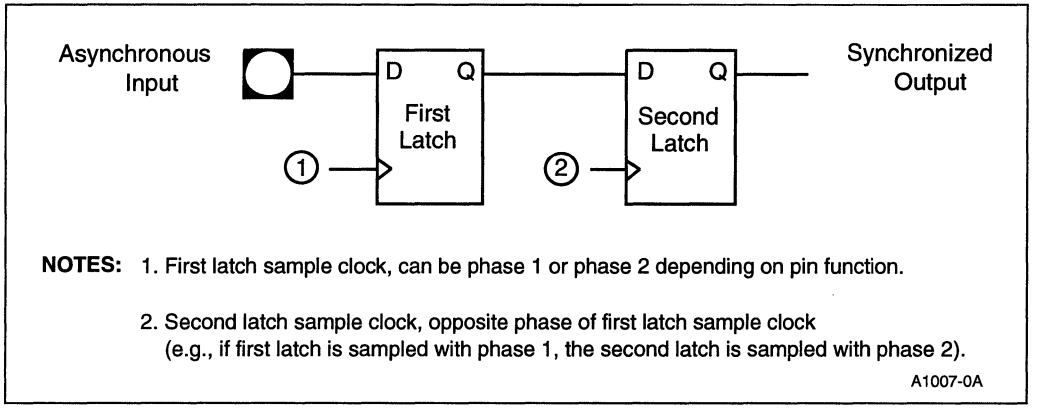


Figure B-1. Input Synchronization Circuit

B.1 WHY SYNCHRONIZERS ARE REQUIRED

Every data latch requires a specific setup and hold time to operate properly. The duration of the setup and hold time defines a *window* during which the device attempts to latch the data. If the input makes a transition within this window, the output may not attain a stable state. The data sheet specifies a setup and hold window larger than is actually required. However, variations in device operation (e.g., temperature, voltage) require that a larger window be specified to cover all conditions.

Should the input to the data latch make a transition during the sample and hold window, the output of the latch eventually attains a stable state. This stable state must be attained before the second stage of synchronization requires a valid input. To synchronize an asynchronous signal, the circuit in Figure B-1 samples the input into the first latch, allows the output to stabilize, then samples the stabilized value into a second latch. With the asynchronous signal resolved in this way, the input signal cannot cause an internal device failure.

A synchronization failure can occur when the output of the first latch does not meet the setup and hold requirements of the input of the second latch. The rate of failure is determined by the actual size of the sampling window of the data latch and by the amount of time between the strobe signals of the two latches. As the sampling window gets smaller, the number of times an asynchronous transition occurs during the sampling window drops.

B.2 ASYNCHRONOUS PINS

The 80C186EB/80C188EB inputs that use the two-stage synchronization circuit are T0IN, T1IN, NMI, $\overline{\text{TEST}}$/BUSY, INT4:0, HOLD, and all Port 2 input functions (P2.5/BCLK0, P2.2/BCLK1 and P2.4/$\overline{\text{CTS1}}$).

C

Instruction Set Descriptions

APPENDIX C
INSTRUCTION SET DESCRIPTIONS

This appendix provides reference information for the 80C186 Modular Core family instruction set. Tables C-1 through C-3 define the variables used in Table C-4, which lists the instructions with their descriptions and operations.

Table C-1. Instruction Format Variables

Variable	Description
dest	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.
src	A register, memory location or immediate value that is used in the operation, but is not altered by the instruction
target	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.
disp8	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.
accum	Register AX for word transfers, AL for bytes.
port	An I/O port number; specified as an immediate value of 0–255, or register DX (which contains port number in range 0–64K).
src-string	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	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	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	Immediate value of 0–255 identifying interrupt pointer number.
optional-pop-value	Number of bytes (0–64K, ordinarily an even number) to discard from the stack.
external-opcode	Immediate value (0–63) that is encoded in the instruction for use by an external processor.

Table C-2. Instruction Operands

Operand	Description
reg	An 8- or 16-bit general register.
reg16	An 16-bit general register.
seg-reg	A segment register.
accum	Register AX or AL
immed	A constant in the range 0–FFFFH.
immed8	A constant in the range 0–FFH.
mem	An 8- or 16-bit memory location.
mem16	A 16-bit memory location.
mem32	A 32-bit memory location.
src-table	Name of 256-byte translate table.
src-string	Name of string addressed by register SI.
dest-string	Name of string addressed by register DI.
short-label	A label within the –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.
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.
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.

Table C-3. Flag Bit Functions

Name	Function
AF	Auxiliary Flag: Set on carry from or borrow to the low order four bits of AL; cleared otherwise.
CF	Carry Flag: Set on high-order bit carry or borrow; cleared otherwise.
DF	Direction Flag: Causes string instructions to auto decrement the appropriate index register when set. Clearing DF causes auto increment.
IF	Interrupt-enable Flag: When set, maskable interrupts will cause the CPU to transfer control to an interrupt vector specified location.
OF	Overflow Flag: Set if the signed result cannot be expressed within the number of bits in the destination operand; cleared otherwise.
PF	Parity Flag: Set if low-order 8 bits of result contain an even number of 1 bits; cleared otherwise.
SF	Sign Flag: Set equal to high-order bit of result (0 if positive, 1 if negative).
TF	Single Step Flag: Once set, a single step interrupt occurs after the next instruction executes. TF is cleared by the single step interrupt.
ZF	Zero Flag: Set if result is zero; cleared otherwise.

Table C-4. Instruction Set

Name	Description	Operation	Flags Affected
AAA	**ASCII Adjust for Addition:** AAA Changes the contents of register AL to a valid unpacked decimal number; the high-order half-byte is zeroed. **Instruction Operands:** none	if ((AL) and 0FH) > 9 or (AF) = 1 then (AL) ← (AL) + 6 (AH) ← (AH) + 1 (AF) ← 1 (CF) ← (AF) (AL) ← (AL) and 0FH	AF ✓ CF ✓ DF – IF – OF ? PF ? SF ? TF – ZF ?
AAD	**ASCII Adjust for Division:** AAD Modifies the numerator in AL before dividing two valid unpacked decimal operands so that the quotient produced by the division will be a valid unpacked decimal number. AH must be zero for the subsequent DIV to produce the correct result. The quotient is returned in AL, and the remainder is returned in AH; both high-order half-bytes are zeroed. **Instruction Operands:** none	(AL) ← (AH) × 0AH + (AL) (AH) ← 0	AF ? CF ? DF – IF – OF ? PF ✓ SF ✓ TF – ZF ✓
AAM	**ASCII Adjust for Multiply:** AAM 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. **Instruction Operands:** none	(AH) ← (AL) / 0AH (AL) ← (AL) % 0AH	AF ? CF ? DF – IF – OF ? PF ✓ SF ✓ TF – ZF ✓

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected
AAS	**ASCII Adjust for Subtraction:** AAS Corrects the result of a previous subtraction of two valid unpacked decimal operands (the destination operand must have been specified as register AL). Changes the content of AL to a valid unpacked decimal number; the high-order half-byte is zeroed. **Instruction Operands:** none	if ((AL) and 0FH) > 9 or (AF) = 1 then (AL) ← (AL) – 6 (AH) ← (AH) – 1 (AF) ← 1 (CF) ← (AF) (AL) ← (AL) and 0FH	AF ✓ CF ✓ DF – IF – OF ? PF ? SF ? TF – ZF ?
ADC	**Add with Carry:** ADC *dest*, *src* 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). Since ADC incorporates a carry from a previous operation, it can be used to write routines to add numbers longer than 16 bits. **Instruction Operands:** ADC reg, reg ADC reg, mem ADC mem, reg ADC reg, immed ADC mem, immed ADC accum, immed	if (CF) = 1 then (dest) ← (dest) + (src) + 1 else (dest) ← (dest) + (src)	AF ✓ CF ✓ DF – IF – OF ✓ PF ✓ SF ✓ TF – ZF ✓

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected

ADD	**Addition:**
	ADD *dest, src*
	Sums 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).
	Instruction Operands:
	ADD reg, reg
	ADD reg, mem
	ADD mem, reg
	ADD reg, immed
	ADD mem, immed
	ADD accum, immed
		(dest) ← (dest) + (src)	AF ✓
			CF ✓
			DF –
			IF –
			OF ✓
			PF ✓
			SF ✓
			TF –
			ZF ✓

AND	**And Logical:** AND *dest, src* 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. **Instruction Operands:** AND reg, reg AND reg, mem AND mem, reg AND reg, immed AND mem, immed AND accum, immed	(dest) ← (dest) and (src) (CF) ← 0 (OF) ← 0	AF ? CF ✓ DF – IF – OF ✓ PF ✓ SF ✓ TF – ZF ✓

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected
BOUND	**Detect Value Out of Range:** BOUND *dest, src* Provides array bounds checking in hardware. The calculated array index is placed in one of the general purpose registers, and the upper and lower bounds of the array are placed in two consecutive memory locations. The contents of the register are compared with the memory location values, and if the register value is less than the first location or greater than the second memory location, a trap type 5 is generated. **Instruction Operands:** BOUND reg, mem	if ((dest) < (src) or (dest) > ((src) + 2) then (SP) ← (SP) – 2 ((SP) + 1 : (SP)) ← FLAGS (IF) ← 0 (TF) ← 0 (SP) ← (SP) – 2 ((SP) + 1 : (SP)) ← (CS) (CS) ← (1EH) (SP) ← (SP) – 2 ((SP) + 1 : (SP)) ← (IP) (IP) ← (1CH)	AF – CF – DF – IF – OF – PF – SF – TF – ZF –
CALL	**Call Procedure:** CALL *procedure-name* 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 assembler generates a different type of CALL instruction depending on whether the programmer has defined the procedure name as NEAR or FAR. **Instruction Operands:** CALL near-proc CALL far-proc CALL memptr16 CALL regptr16 CALL memptr32	if Inter-segment then (SP) ← (SP) – 2 ((SP) +1:(SP)) ← (CS) (CS) ← SEG (SP) ← (SP) – 2 ((SP) +1:(SP)) ← (IP) (IP) ← dest	AF – CF – DF – IF – OF – PF – SF – TF – ZF –

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected
CBW	**Convert Byte to Word:** CBW Extends the sign of the byte in register AL throughout register AH. Use to produce a double-length (word) dividend from a byte prior to performing byte division. **Instruction Operands:** none	if (AL) < 80H then (AH) ← 0 else (AH) ← FFH	AF – CF – DF – IF – OF – PF – SF – TF – ZF –
CLC	**Clear Carry flag:** CLC Zeroes the carry flag (CF) and affects no other flags. Useful in conjunction with the rotate through carry left (RCL) and the rotate through carry right (RCR) instructions. **Instruction Operands:** none	(CF) ← 0	AF – CF ✓ DF – IF – OF – PF – SF – TF – ZF –
CLD	**Clear Direction flag:** CLD Zeroes the direction flag (DF) causing the string instructions to auto-increment the source index (SI) and/or destination index (DI) registers. **Instruction Operands:** none	(DF) ← 0	AF – CF – DF ✓ IF – OF – PF – SF – TF – ZF –

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected
CLI	**Clear Interrupt-enable Flag:** CLI Zeroes the interrupt-enable flag (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 NMI line, however, is honored, as is a software interrupt. **Instruction Operands:** none	(IF) ← 0	AF – CF – DF – IF ✓ OF – PF – SF – TF – ZF –
CMC	**Complement Carry Flag:** CMC Toggles complement carry flag (CF) to its opposite state and affects no other flags. **Instruction Operands:** none	if (CF) = 0 then (CF) ← 1 else (CF) ← 0	AF – CF ✓ DF – IF – OF – PF – SF – TF – ZF –

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected

CMP	**Compare:**
	CMP *dest, src*
	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. 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.
	Instruction Operands:
	CMP reg, reg
	CMP reg, mem
	CMP mem, reg
	CMP reg, immed
	CMP mem, immed
	CMP accum, immed
		(dest) – (src)	AF ✓
			CF ✓
			DF –
			IF –
			OF ✓
			PF ✓
			SF ✓
			TF –
			ZF ✓

CMPS	**Compare String:**
	CMPS *dest-string, src-string*
	Subtracts the destination byte or word from the source byte or word. The destination byte or word is addressed by the destination index (DI) register and the source byte or word is addresses by the source index (SI) register. CMPS updates the flags to reflect the relationship of the destination element to the source element but does not alter either operand and updates SI and DI to point to the next string element.
	Instruction Operands:
	CMP dest-string, src-string
	CMP (repeat) dest-string, src-string
		(dest-string) – (src-string)
		if
		(DF) = 0
		then
		(SI) ← (SI) + DELTA
		(DI) ← (DI) + DELTA
		else
		(SI) ← (SI) – DELTA
		(DI) ← (DI) – DELTA
			AF ✓
			CF ✓
			DF –
			IF –
			OF ✓
			PF ✓
			SF ✓
			TF –
			ZF ✓

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected
CWD	**Convert Word to Doubleword:** CWD Extends the sign of the word in register AX throughout register DX. Use to produce a double-length (doubleword) dividend from a word prior to performing word division. **Instruction Operands:** none	if (AX) < 8000H then (DX) ← 0 else (DX) ← FFFFH	AF – CF – DF – IF – OF – PF – SF – TF – ZF –
DAA	**Decimal Adjust for Addition:** DAA Corrects the result of previously adding two valid packed decimal operands (the destination operand must have been register AL). Changes the content of AL to a pair of valid packed decimal digits. **Instruction Operands:** none	if ((AL) and 0FH) > 9 or (AF) = 1 then (AL) ← (AL) + 6 (AF) ← 1 if (AL) > 9FH or (CF) = 1 then (AL) ← (AL) + 60H (CF) ← 1	AF ✓ CF ✓ DF – IF – OF ? PF ✓ SF ✓ TF – ZF ✓
DAS	**Decimal Adjust for Subtraction:** DAS Corrects the result of a previous subtraction of two valid packed decimal operands (the destination operand must have been specified as register AL). Changes the content of AL to a pair of valid packed decimal digits. **Instruction Operands:** none	if ((AL) and 0FH) > 9 or (AF) = 1 then (AL) ← (AL) – 6 (AF) ← 1 if (AL) > 9FH or (CF) = 1 then (AL) ← (AL) – 60H (CF) ← 1	AF ✓ CF ✓ DF – IF – OF ? PF ✓ SF ✓ TF – ZF ✓

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected
DEC	**Decrement:** DEC *dest* Subtracts one from the destination operand. The operand may be a byte or a word and is treated as an unsigned binary number (see AAA and DAA). **Instruction Operands:** DEC reg DEC mem	(dest) ← (dest) – 1	AF ✓ CF – DF – IF – OF ✓ PF ✓ SF ✓ TF – ZF ✓

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected
DIV	**Divide:** DIV *src* 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 two-byte dividend assumed to be in registers AL and AH. The byte quotient is returned in AL, and the byte remainder is returned in AH. If the source operand is a word, it is divided into the two-word dividend in registers AX and DX. The word quotient is returned in AX, and the word remainder is returned in DX. If the quotient exceeds the capacity of its destination register (FFH for byte source, FFFFH for word source), as when division by zero is attempted, a type 0 interrupt is generated, and the quotient and remainder are undefined. Nonintegral quotients are truncated to integers. **Instruction Operands:** DIV reg DIV mem	**When Source Operand is a Byte:** (temp) ← (byte-src) if (temp) / (AX) > FFH then (type 0 interrupt is generated) (SP) ← (SP) – 2 ((SP) + 1:(SP)) ← FLAGS (IF) ← 0 (TF) ← 0 (SP) ← (SP) – 2 ((SP) + 1:(SP)) ← (CS) (CS) ← (2) (SP) ← (SP) – 2 ((SP) + 1:(SP)) ← (IP) (IP) ← (0) else (AL) ← (temp) / (AX) (AH) ← (temp) % (AX) **When Source Operand is a Word:** (temp) ← (word-src) if (temp) / (DX:AX) > FFFFH then (type 0 interrupt is generated) (SP) ← (SP) – 2 ((SP) + 1:(SP)) ← FLAGS (IF) ← 0 (TF) ← 0 (SP) ← (SP) – 2 ((SP) + 1:(SP)) ← (CS) (CS) ← (2) (SP) ← (SP) – 2 ((SP) + 1:(SP)) ← (IP) (IP) ← (0) else (AX) ← (temp) / (DX:AX) (DX) ← (temp) % (DX:AX)	AF ? CF ? DF – IF – OF ? PF ? SF ? TF – ZF ?

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected
ENTER	**Procedure Entry:** ENTER *locals, levels* Executes the calling sequence for a high-level language. It saves the current frame pointer in BP, copies the frame pointers from procedures below the current call (to allow access to local variables in these procedures) and allocates space on the stack for the local variables of the current procedure invocation. **Instruction Operands:** ENTER locals, level	(SP) ← (SP) – 2 ((SP) + 1:(SP)) ← (BP) (FP) ← (SP) if level > 0 then repeat (level – 1) times (BP) ← (BP) – 2 (SP) ← (SP) – 2 ((SP) + 1:(SP)) ← (BP) end repeat (SP) ← (SP) – 2 ((SP) + 1:(SP)) ← (FP) end if (BP) ← (FP) (SP) ← (SP) – (locals)	AF – CF – DF – IF – OF – PF – SF – TF – ZF –
ESC	**Escape:** ESC Provides a mechanism by which other processors (coprocessors) may receive their instructions from the 8086 or 8088 instruction stream and make use of the 8086 or 8088 addressing modes. The CPU (8086 or 8088) does a no operation (NOP) for the ESC instruction other than to access a memory operand and place it on the bus. **Instruction Operands:** ESC immed, mem ESC immed, reg	if mod ≠ 11 then data bus ← (EA)	AF – CF – DF – IF – OF – PF – SF – TF – ZF –

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected
HLT	**Halt:** HLT Causes the CPU 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 upon receipt of a maskable interrupt request on INTR (if interrupts are enabled). **Instruction Operands:** none	None	AF – CF – DF – IF – OF – PF – SF – TF – ZF –

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected
IDIV	**Integer Divide:** IDIV *src* 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. **Instruction Operands:** IDIV reg IDIV mem	**When Source Operand is a Byte:** (temp) ← (byte-src) if (temp) / (AX) > 0 and (temp) / (AX) > 7FH or (temp) / (AX) < 0 and (temp) / (AX) < 0 – 7FH – 1 then (type 0 interrupt is generated) (SP) ← (SP) – 2 ((SP) + 1:(SP)) ← FLAGS (IF) ← 0 (TF) ← 0 (SP) ← (SP) – 2 ((SP) + 1:(SP)) ← (CS) (CS) ← (2) (SP) ← (SP) – 2 ((SP) + 1:(SP)) ← (IP) (IP) ← (0) else (AL) ← (temp) / (AX) (AH) ← (temp) % (AX) **When Source Operand is a Word:** (temp) ← (word-src) if (temp) / (DX:AX) > 0 and (temp) / (DX:AX) > 7FFFH or (temp) / (DX:AX) < 0 and (temp) / (DX:AX) < 0 – 7FFFH – 1 then (type 0 interrupt is generated) (SP) ← (SP) – 2 ((SP) + 1:(SP)) ← FLAGS (IF) ← 0 (TF) ← 0 (SP) ← (SP) – 2 ((SP) + 1:(SP)) ← (CS) (CS) ← (2) (SP) ← (SP) – 2 ((SP) + 1:(SP)) ← (IP) (IP) ← (0) else (AX) ← (temp) / (DX:AX) (DX) ← (temp) % (DX:AX)	AF ? CF ? DF – IF – OF ? PF ? SF ? TF – ZF ?

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected
IMUL	**Integer Multiply:** IMUL *src* 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. **Instruction Operands:** IMUL reg IMUL mem IMUL immed	**When Source Operand is a Byte:** (AX) ← (byte-src) × (AL) if (AH) = sign-extension of (AL) then (CF) ← 0 else (CF) ← 1 (OF) ← (CF) **When Source Operand is a Word:** (DX:AX) ← (word-src) × (AX) if (DX) = sign-extension of (AX) then (CF) ← 0 else (CF) ← 1 (OF) ← (CF)	AF ? CF ✓ DF – IF – OF ✓ PF ? SF ? TF – ZF ?
IN	**Input Byte or Word:** IN *accum, port* 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. **Instruction Operands:** IN AL, immed8 IN AX, DX	**When Source Operand is a Byte:** (AL) ← (port) **When Source Operand is a Word:** (AX) ← (port)	AF – CF – DF – IF – OF – PF – SF – TF – ZF –

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected
INC	**Increment:** INC *dest* Adds one to the destination operand. The operand may be byte or a word and is treated as an unsigned binary number (see AAA and DAA). **Instruction Operands:** INC reg INC mem	(dest) ← (dest) + 1	AF ✓ CF – DF – IF – OF ✓ PF ✓ SF ✓ TF – ZF ✓
INS	**In String:** INS *dest-string, port* Performs block input from an I/O port to memory. The port address is placed in the DX register. The memory address is placed in the DI register. This instruction uses the ES register (which cannot be overridden). After the data transfer takes place, the DI register increments or decrements, depending on the value of the direction flag (DF). The DI register changes by 1 for byte transfers or 2 for word transfers. **Instruction Operands:** INS dest-string, port INS (repeat) dest-string, port	(dest) ← (src)	AF – CF – DF – IF – OF – PF – SF – TF – ZF –

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected
INT	**Interrupt:** INT *interrupt-type* Activates the interrupt procedure specified by the interrupt-type operand. 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. **Instruction Operands:** INT immed8	(SP) ← (SP) – 2 ((SP) + 1:(SP)) ← FLAGS (IF) ← 0 (TF) ← 0 (SP) ← (SP) – 2 ((SP) + 1:(SP)) ← (CS) (CS) ← (interrupt-type x 4 + 2) (SP) ← (SP) – 2 ((SP) + 1:(SP)) ← (IP) (IP) ← (interrupt-type x 4)	AF – CF – DF – IF ✓ OF – PF – SF – TF ✓ ZF –

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected
INTO	**Interrupt on Overflow:** INTO 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. **Instruction Operands:** none	if (OF) = 1 then (SP) ← (SP) – 2 ((SP) + 1:(SP)) ← FLAGS (IF) ← 0 (TF) ← 0 (SP) ← (SP) – 2 ((SP) + 1:(SP)) ← (CS) (CS) ← (12H) (SP) ← (SP) – 2 ((SP) + 1:(SP)) ← (IP) (IP) ← (10H)	AF – CF – DF – IF – OF – PF – SF – TF – ZF –
IRET	**Interrupt Return:** IRET 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. **Instruction Operands:** none	(IP) ← ((SP) + 1:(SP)) (SP) ← (SP) + 2 (CS) ← ((SP) + 1:(SP)) (SP) ← (SP) + 2 FLAGS ← ((SP) + 1:(SP)) (SP) ← (SP) + 2	AF ✓ CF ✓ DF ✓ IF ✓ OF ✓ PF ✓ SF ✓ TF ✓ ZF ✓
JA JNBE	**Jump on Above:** **Jump on Not Below or Equal:** JA *disp8* JNBE *disp8* Transfers control to the target location if the tested condition ((CF=0) or (ZF=0)) is true. **Instruction Operands:** JA short-label JNBE short-label	if ((CF) = 0) or ((ZF) = 0) then (IP) ← (IP) + disp8 (sign-ext to 16 bits)	AF – CF – DF – IF – OF – PF – SF – TF – ZF –

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected
JAE JNB	**Jump on Above or Equal:** **Jump on Not Below:** JAE *disp8* JNB *disp8* Transfers control to the target location if the tested condition (CF = 0) is true. **Instruction Operands:** JAE short-label JNB short-label	if (CF) = 0 then (IP) ← (IP) + disp8 (sign-ext to 16 bits)	AF – CF – DF – IF – OF – PF – SF – TF – ZF –
JB JNAE	**Jump on Below:** **Jump on Not Above or Equal:** JB *disp8* JNAE *disp8* Transfers control to the target location if the tested condition (CF = 1) is true. **Instruction Operands:** JB short-label JNAE short-label	if (CF) = 1 then (IP) ← (IP) + disp8 (sign-ext to 16 bits)	AF – CF – DF – IF – OF – PF – SF – TF – ZF –
JBE JNA	**Jump on Below or Equal:** **Jump on Not Above:** JBE *disp8* JNA *disp8* Transfers control to the target location if the tested condition ((C =1) or (ZF=1)) is true. **Instruction Operands:** JBE short-label JNA short-label	if ((CF) = 1) or ((ZF) = 1) then (IP) ← (IP) + disp8 (sign-ext to 16 bits)	AF – CF – DF – IF – OF – PF – SF – TF – ZF –
JC	**Jump on Carry:** JC *disp8* Transfers control to the target location if the tested condition (CF=1) is true. **Instruction Operands:** JC short-label	if (CF) = 1 then (IP) ← (IP) + disp8 (sign-ext to 16 bits)	AF – CF – DF – IF – OF – PF – SF – TF – ZF –

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected
JCXZ	**Jump if CX Zero:** JCXZ *disp8* Transfers control to the target location if CX is 0. 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. **Instruction Operands:** JCXZ short-label	if (CX) = 0 then (IP) ← (IP) + disp8 (sign-ext to 16 bits)	AF – CF – DF – IF – OF – PF – SF – TF – ZF –
JE JZ	**Jump on Equal:** **Jump on Zero:** JE *disp8* JZ *disp8* Transfers control to the target location if the condition tested (ZF = 1) is true. **Instruction Operands:** JE short-label JZ short-label	if (ZF) = 1 then (IP) ← (IP) + disp8 (sign-ext to 16 bits)	AF – CF – DF – IF – OF – PF – SF – TF – ZF –
JG JNLE	**Jump on Greater Than:** **Jump on Not Less Than or Equal:** JG *disp8* JNLE *disp8* Transfers control to the target location if the condition tested (SF = OF) and (ZF=0) is true. **Instruction Operands:** JG short-label JNLE short-label	if ((SF) = (OF)) and ((ZF) = 0) then (IP) ← (IP) + disp8 (sign-ext to 16 bits)	AF – CF – DF – IF – OF – PF – SF – TF – ZF –
JGE JNL	**Jump on Greater Than or Equal:** **Jump on Not Less Than:** JGE *disp8* JNL *disp8* Transfers control to the target location if the condition tested (SF=OF) is true. **Instruction Operands:** JGE short-label JNL short-label	if (SF) = (OF) then (IP) ← (IP) + disp8 (sign-ext to 16 bits)	AF – CF – DF – IF – OF – PF – SF – TF – ZF –

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected
JL JNGE	**Jump on Less Than:** **Jump on Not Greater Than or Equal:** JL *disp8* JNGE *disp8* Transfers control to the target location if the condition tested (SF≠OF) is true. **Instruction Operands:** JL short-label JNGE short-label	if (SF) ≠ (OF) then (IP) ← (IP) + disp8 (sign-ext to 16 bits)	AF – CF – DF – IF – OF – PF – SF – TF – ZF –
JLE JNG	**Jump on Less Than or Equal:** **Jump on Not Greater Than:** JGE *disp8* JNL *disp8* Transfers control to the target location If the condition tested ((SF≠OF) or (ZF=0)) is true. **Instruction Operands:** JGE short-label JNL short-label	if ((SF) ≠ (OF)) or ((ZF) = 1) then (IP) ← (IP) + disp8 (sign-ext to 16 bits)	AF – CF – DF – IF – OF – PF – SF – TF – ZF –
JMP	**Jump Unconditionally:** JMP *target* Transfers control to the target location. **Instruction Operands:** JMP short-label JMP near-label JMP far-label JMP memptr JMP regptr	if Inter-segment then (CS) ← SEG (IP) ← dest	AF – CF – DF – IF – OF – PF – SF – TF – ZF –
JNC	**Jump on Not Carry:** JNC *disp8* Transfers control to the target location if the tested condition (CF=0) is true. **Instruction Operands:** JNC short-label	if (CF) = 0 then (IP) ← (IP) + disp8 (sign-ext to 16 bits)	AF – CF – DF – IF – OF – PF – SF – TF – ZF –

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected
JNE JNZ	**Jump on Not Equal:** **Jump on Not Zero:** JNE *disp8* JNZ *disp8* Transfers control to the target location if the tested condition (ZF = 0) is true. **Instruction Operands:** JNE short-label JNZ short-label	if (ZF) = 0 then (IP) ← (IP) + disp8 (sign-ext to 16 bits)	AF – CF – DF – IF – OF – PF – SF – TF – ZF –
JNO	**Jump on Not Overflow:** JNO *disp8* Transfers control to the target location if the tested condition (OF = 0) is true. **Instruction Operands:** JNO short-label	if (OF) = 0 then (IP) ← (IP) + disp8 (sign-ext to 16 bits)	AF – CF – DF – IF – OF – PF – SF – TF – ZF –
JNS	**Jump on Not Sign:** JNS *disp8* Transfers control to the target location if the tested condition (SF = 0) is true. **Instruction Operands:** JNS short-label	if (SF) = 0 then (IP) ← (IP) + disp8 (sign-ext to 16 bits)	AF – CF – DF – IF – OF – PF – SF – TF – ZF –
JNP JPO	**Jump on Not Parity:** **Jump on Parity Odd:** JNO *disp8* JPO *disp8* Transfers control to the target location if the tested condition (PF=0) is true. **Instruction Operands:** JNO short-label JPO short-label	if (PF) = 0 then (IP) ← (IP) + disp8 (sign-ext to 16 bits)	AF – CF – DF – IF – OF – PF – SF – TF – ZF –

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected
JO	**Jump on Overflow:** JO *disp8* Transfers control to the target location if the tested condition (OF = 1) is true. **Instruction Operands:** JO short-label	if (OF) = 1 then (IP) ← (IP) + disp8 (sign-ext to 16 bits)	AF – CF – DF – IF – OF – PF – SF – TF – ZF –
JP JPE	**Jump on Parity:** **Jump on Parity Equal:** JP *disp8* JPE *disp8* Transfers control to the target location if the tested condition (PF = 1) is true. **Instruction Format:** JP short-label JPE short-label	if (PF) = 1 then (IP) ← (IP) + disp8 (sign-ext to 16 bits)	AF – CF – DF – IF – OF – PF – SF – TF – ZF –
JS	**Jump on Sign:** JS *disp8* Transfers control to the target location if the tested condition (SF = 1) is true. **Instruction Format:** JS short-label	if (SF) = 1 then (IP) ← (IP) + disp8 (sign-ext to 16 bits)	AF – CF – DF – IF – OF – PF – SF – TF – ZF –
LAHF	**Load Register AH From Flags:** LAHF Copies SF, ZF, AF, PF and CF (the 8080/8085 flags) into bits 7, 6, 4, 2 and 0, respectively, of register AH. The content of bits 5, 3, and 1 are undefined. LAHF is provided primarily for converting 8080/8085 assembly language programs to run on an 8086 or 8088. **Instruction Operands:** none	(AH) ← (SF):(ZF):X:(AF):X:(PF):X:(CF)	AF – CF – DF – IF – OF – PF – SF – TF – ZF –

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected

LDS	**Load Pointer Using DS:**
	LDS *dest, src*
	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.
	Instruction Operands:
	LDS reg16, mem32
		(dest) ← (EA)
		(DS) ← (EA + 2)
			AF –
			CF –
			DF –
			IF –
			OF –
			PF –
			SF –
			TF –
			ZF –

LEA	**Load Effective Address:** LEA *dest, src* Transfers the offset of the source operand (rather than its value) to the destination operand. **Instruction Operands:** LEA reg16, mem16	(dest) ← EA	AF – CF – DF – IF – OF – PF – SF – TF – ZF –
LEAVE	**Leave:** LEAVE Reverses the action of the most recent ENTER instruction. Collapses the last stack frame created. First, LEAVE copies the current BP to the stack pointer releasing the stack space allocated to the current procedure. Second, LEAVE pops the old value of BP from the stack, to return to the calling procedure's stack frame. A return (RET) instruction will remove arguments stacked by the calling procedure for use by the called procedure. **Instruction Operands:** none	(SP) ← (BP) (BP) ← ((SP) + 1:(SP)) (SP) ← (SP) + 2	AF – CF – DF – IF – OF – PF – SF – TF – ZF –

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected
LES	**Load Pointer Using ES:** LES *dest, src* Transfers a 32-bit pointer variable from the source operand to the destination operand and register ES. The offset word of the pointer is transferred to the destination operand. The segment word of the pointer is transferred to register ES. **Instruction Operands:** LES reg16, mem32	(dest) ← (EA) (ES) ← (EA + 2)	AF – CF – DF – IF – OF – PF – SF – TF – ZF –
LOCK	**Lock the Bus:** LOCK Causes the 8088 (configured in maximum mode) to assert its bus LOCK signal while the following instruction executes. The instruction most useful in this context is an exchange register with memory. The LOCK prefix may be combined with the segment override and/or REP prefixes. **Instruction Operands:** none	none	AF – CF – DF – IF – OF – PF – SF – TF – ZF –

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected
LODS	**Load String (Byte or Word):** LODS *src-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. **Instruction Operands:** LODS src-string LODS (repeat) src-string	**When Source Operand is a Byte:** (AL) ← (src-string) if (DF) = 0 then (SI) ← (SI) + DELTA else (SI) ← (SI) – DELTA **When Source Operand is a Word:** (AX) ← (src-string) if (DF) = 0 then (SI) ← (SI) + DELTA else (SI) ← (SI) – DELTA	AF – CF – DF – IF – OF – PF – SF – TF – ZF –
LOOP	**Loop:** LOOP *disp8* Decrements CX by 1 and transfers control to the target location if CX is not 0; otherwise the instruction following LOOP is executed. **Instruction Operands:** LOOP short-label	(CX) ← (CX) – 1 if (CX) ≠ 0 then (IP) ← (IP) + disp8 (sign-ext to 16 bits)	AF – CF – DF – IF – OF – PF – SF – TF – ZF –
LOOPE LOOPZ	**Loop While Equal:** **Loop While Zero:** LOOPE *disp8* LOOPZ *disp8* Decrements CX by 1 and transfers control is to the target location if CX is not 0 and if ZF is set; otherwise the next sequential instruction is executed. **Instruction Operands:** LOOPE short-label LOOPZ short-label	(CX) ← (CX) – 1 if (ZF) = 1 and (CX) ≠ 0 then (IP)←(IP) + disp8 (sign-ext to 16 bits)	AF – CF – DF – IF – OF – PF – SF – TF – ZF –

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected
LOOPNE LOOPNZ	**Loop While Not Equal:** **Loop While Not Zero:** LOOPNE *disp8* LOOPNZ *disp8* Decrements CX by 1 and transfers control to the target location if CX is not 0 and if ZF is clear; otherwise the next sequential instruction is executed. **Instruction Operands:** LOOPNE short-label LOOPNZ short-label	(CX) ← (CX) – 1 if (ZF) = 0 and (CX) ≠ 0 then (IP) ← (IP) + disp8 (sign-ext to 16 bits)	AF – CF – DF – IF – OF – PF – SF – TF – ZF –

MOV	**Move (Byte or Word):**
	MOV *dest, src*
	Transfers a byte or a word from the source operand to the destination operand.
	Instruction Operands:
	MOV mem, accum
	MOV accum, mem
	MOV reg, reg
	MOV reg, mem
	MOV mem, reg
	MOV reg, immed
	MOV mem, immed
	MOV seg-reg, reg16
	MOV seg-reg, mem16
	MOV reg16, seg-reg
	MOV mem16, seg-reg
		(dest)←(src)	AF –
			CF –
			DF –
			IF –
			OF –
			PF –
			SF –
			TF –
			ZF –

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected

MOVS	**Move String:**
	MOVS *dest-string, src-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.
	Instruction Operands:
	MOVS dest-string, src-string
	MOVS (repeat) dest-string, src-string
		(dest-string) ← (src-string)	AF –
			CF –
			DF –
			IF –
			OF –
			PF –
			SF –
			TF –
			ZF –

MUL	**Multiply:** MUL *src* 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 non-zero, CF and OF are set; otherwise they are cleared. **Instruction Operands:** MUL reg MUL mem	**When Source Operand is a Byte:** (AX) ← (AL) × (src) if (AH) = 0 then (CF) ← 0 else (CF) ← 1 (OF) ← (CF) **When Source Operand is a Word:** (DX:AX) ← (AX) × (src) if (DX) = 0 then (CF) ← 0 else (CF) ← 1 (OF) ← (CF)	AF ? CF ✓ DF – IF – OF ✓ PF ? SF ? TF – ZF ?

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected
NEG	**Negate:** NEG *dest* 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. **Instruction Operands:** NEG reg NEG mem	**When Source Operand is a Byte:** (dest) ← FFH – (dest) (dest) ← (dest) + 1 (affecting flags) **When Source Operand is a Word:** (dest) ← FFFFH – (dest) (dest) ← (dest) + 1 (affecting flags)	AF ✓ CF ✓ DF – IF – OF ✓ PF ✓ SF ✓ TF – ZF ✓
NOP	**No Operation:** NOP Causes the CPU to do nothing. **Instruction Operands:** none	None	AF – CF – DF – IF – OF – PF – SF – TF – ZF –
NOT	**Logical Not:** NOT *dest* Inverts the bits (forms the one's complement) of the byte or word operand. **Instruction Operands:** NOT reg NOT mem	**When Source Operand is a Byte:** (dest) ← FFH – (dest) **When Source Operand is a Word:** (dest) ← FFFFH – (dest)	AF – CF – DF – IF – OF – PF – SF – TF – ZF –

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected
OR	**Logical OR:** OR *dest,src* Performs the logical "inclusive or" of the two operands (bytes or words) 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. **Instruction Operands:** OR reg, reg OR reg, mem OR mem, reg OR accum, immed OR reg, immed OR mem, immed	(dest) ← (dest) or (src) (CF) ← 0 (OF) ← 0	AF ? CF ✓ DF – IF – OF ✓ PF ✓ SF ✓ TF – ZF ✓
OUT	**Output:** OUT *port, accumulator* 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. **Instruction Operands:** OUT immed8, AL OUT DX, AX	(dest) ← (src)	AF – CF – DF – IF – OF – PF – SF – TF – ZF –

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected
OUTS	**Out String:** OUTS *port, src_string* Performs block output from memory to an I/O port. The port address is placed in the DX register. The memory address is placed in the SI register. This instruction uses the DS segment register, but this may be changed with a segment override instruction. After the data transfer takes place, the pointer register (SI) increments or decrements, depending on the value of the direction flag (DF). The pointer register changes by 1 for byte transfers or 2 for word transfers. **Instruction Operands:** OUTS port, src_string OUTS (repeat) port, src_string	(dst) ← (src)	AF – CF – DF – IF – OF – PF – SF – TF – ZF –
POP	**Pop:** POP *dest* 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. **Instruction Operands:** POP reg POP seg-reg (CS illegal) POP mem	(dest) ← ((SP) + 1:(SP)) (SP) ← (SP) + 2	AF – CF – DF – IF – OF – PF – SF – TF – ZF –

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected
POPA	**Pop All:** POPA Pops all data, pointer, and index registers off of the stack. The SP value popped is discarded. **Instruction Operands:** none	(DI) ← ((SP) + 1:(SP)) (SP) ← (SP) + 2 (SI) ← ((SP) + 1:(SP)) (SP) ← (SP) + 2 (BP) ← ((SP) + 1:(SP)) (SP) ← (SP) + 2 (BX) ← ((SP) + 1:(SP)) (SP) ← (SP) + 2 (DX) ← ((SP) + 1:(SP)) (SP) ← (SP) + 2 (CX) ← ((SP) + 1:(SP)) (SP) ← (SP) + 2 (AX) ← ((SP) + 1:(SP)) (SP) ← (SP) + 2	AF – CF – DF – IF – OF – PF – SF – TF – ZF –
POPF	**Pop Flags:** 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. SP is then incremented by two to point to the new top of stack. **Instruction Operands:** none	Flags ← ((SP) + 1:(SP)) (SP) ← (SP) + 2	AF ✓ CF ✓ DF ✓ IF ✓ OF ✓ PF ✓ SF ✓ TF ✓ ZF ✓
PUSH	**Push:** PUSH *src* Decrements SP by two and then transfers a word from the source operand to the top of stack now pointed to by SP. **Instruction Operands:** PUSH reg PUSH seg-reg (CS legal) PUSH mem	(SP) ← (SP) – 2 ((SP) + 1:(SP)) ← (src)	AF – CF – DF – IF – OF – PF – SF – TF – ZF –

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected
PUSHA	**Push All:** PUSHA Pushes all data, pointer, and index registers onto the stack . The order in which the registers are saved is: AX, CX, DX, BX, SP, BP, SI, and DI. The SP value pushed is the SP value before the first register (AX) is pushed. **Instruction Operands:** none	temp ← (SP) (SP) ← (SP) – 2 ((SP) + 1:(SP)) ← (AX) (SP) ← (SP) – 2 ((SP) + 1:(SP)) ← (CX) (SP) ← (SP) – 2 ((SP) + 1:(SP)) ← (DX) (SP) ← (SP) – 2 ((SP) + 1:(SP)) ← (BX) (SP) ← (SP) – 2 ((SP) + 1:(SP)) ← (temp) (SP) ← (SP) – 2 ((SP) + 1:(SP)) ← (BP) (SP) ← (SP) – 2 ((SP) + 1:(SP)) ← (SI) (SP) ← (SP) – 2 ((SP) + 1:(SP)) ← (DI)	AF – CF – DF – IF – OF – PF – SF – TF – ZF –
PUSHF	**Push Flags:** PUSHF Decrements SP by two and then transfers all flags to the word at the top of stack pointed to by SP. **Instruction Operands:** none	(SP) ← (SP) – 2 ((SP) + 1:(SP)) ← Flags	AF – CF – DF – IF – OF – PF – SF – TF – ZF –

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected
RCL	**Rotate Through Carry Left:** RCL *dest, count* 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. **Instruction Operands:** RCL reg, n RCL mem, n RCL reg, CL RCL mem, CL	(temp) ← count do while (temp) ≠ 0 (tmpcf) ← (CF) (CF) ← high-order bit of (dest) (dest) ← (dest) x 2 + (tmpcf) (temp) ← (temp) – 1 if count = 1 then if high-order bit of (dest) ≠ (CF) then (OF) ← 1 else (OF) ← 0 else (OF) undefined	AF – CF ✓ DF – IF – OF ✓ PF – SF – TF – ZF –
RCR	**Rotate Through Carry Right:** RCR *dest, count* Operates exactly like RCL except that the bits are rotated right instead of left. **Instruction Operands:** RCR reg, n RCR mem, n RCR reg, CL RCR mem, CL	(temp) ← count do while (temp) ≠ 0 (tmpcf) ← (CF) (CF) ← low-order bit of (dest) (dest) ← (dest) / 2 high-order bit of (dest) ← (tmpcf) (temp) ← (temp) – 1 if count = 1 then if high-order bit of (dest) ≠ next-to-high-order bit of (dest) then (OF) ← 1 else (OF) ← 0 else (OF) undefined	AF – CF ✓ DF – IF – OF ✓ PF – SF – TF – ZF –

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected
REP REPE REPZ REPNE REPNZ	**Repeat:** **Repeat While Equal:** **Repeat While Zero:** **Repeat While Not Equal:** **Repeat While Not Zero:** Controls subsequent string instruction repetition. The different mnemonics are provided to improve program clarity. 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 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. ZF does not need to be initialized before executing the repeated string instruction. **Instruction Operands:** none	do while (CX) ≠ 0 service pending interrupts (if any) execute primitive string Operation in succeeding byte (CX) ← (CX) – 1 if primitive operation is CMPB, CMPW, SCAB, or SCAW and (ZF) ≠ 0 then exit from while loop	AF – CF – DF – IF – OF – PF – SF – TF – ZF –

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected
RET	**Return:** RET *optional-pop-value* Transfers control from a procedure back to the instruction following the CALL that activated the procedure. The assembler generates an intra-segment 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. **Instruction Operands:** RET immed8	(IP) ← ((SP) = 1:(SP)) (SP) ← (SP) + 2 if inter-segment then (CS) ← ((SP) + 1:(SP)) (SP) ← (SP) + 2 if add immed8 to SP then (SP) ← (SP) + data	AF – CF – DF – IF – OF – PF – SF – TF – ZF –
ROL	**Rotate Left:** ROL *dest, count* Rotates the destination byte or word left by the number of bits specified in the count operand. **Instruction Operands:** ROL reg, n ROL mem, n ROL reg, CL ROL mem CL	(temp) ← count do while (temp) ≠ 0 (CF) ← high-order bit of (dest) (dest) ← (dest) x 2 + (CF) (temp) ← (temp) – 1 if count = 1 then if high-order bit of (dest) ≠ (CF) then (OF) ← 1 else (OF) ← 0 else (OF) undefined	AF – CF ✓ DF – IF – OF ✓ PF – SF – TF – ZF –

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected
ROR	**Rotate Right:** ROR *dest, count* Operates similar to ROL except that the bits in the destination byte or word are rotated right instead of left. **Instruction Operands:** ROR reg, n ROR mem, n ROR reg, CL ROR mem, CL	(temp) ← count do while (temp) ≠ 0 (CF) ← low-order bit of (dest) (dest) ← (dest) / 2 high-order bit of (dest) ← (CF) (temp) ← (temp) – 1 if count = 1 then if high-order bit of (dest) ≠ next-to-high-order bit of (dest) then (OF) ← 1 else (OF) ← 0 else (OF) undefined	AF – CF ✓ DF – IF – OF ✓ PF – SF – TF – ZF –
SAHF	**Store Register AH Into Flags:** SAHF 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. **Instruction Operands:** none	(SF):(ZF):X:(AF):X:(PF):X:(CF) ← (AH)	AF ✓ CF ✓ DF – IF – OF – PF ✓ SF ✓ TF – ZF ✓

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected
SHL SAL	**Shift Logical Left:** **Shift Arithmetic Left:** SHL *dest, count* SAL *dest, count* Shifts the destination byte or word 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. **Instruction Operands:** SHL reg, n SAL reg, n SHL mem, n SAL mem, n SHL reg, CL SAL reg, CL SHL mem, CL SAL mem, CL	(temp) ← count do while (temp) ≠ 0 (CF) ← high-order bit of (dest) (dest) ← (dest) x 2 (temp) ← (temp) – 1 if count = 1 then if high-order bit of (dest) ≠ (CE) then (OF) ← 1 else (OF) ← 0 else (OF) undefined	AF ? CF ✓ DF – IF – OF ✓ PF ✓ SF ✓ TF – ZF ✓
SAR	**Shift Arithmetic Right:** SAR *dest, count* 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 –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. **Instruction Operands:** SAR reg, n SAR mem, n SAR reg, CL SAR mem, CL	(temp) ← count do while (temp) ≠ 0 (CF) ← low-order bit of (dest) (dest) ← (dest) / 2 (temp) ← (temp) – 1 if count = 1 then if high-order bit of (dest) ≠ next-to-high-order bit of (dest) then (OF) ← 1 else (OF) ← 0 else (OF) ← 0	AF ? CF ✓ DF – IF – OF ✓ PF ✓ SF ✓ TF – ZF ✓

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected
SBB	**Subtract With Borrow:** SBB *dest, src* 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) **Instruction Operands:** SBB reg, reg SBB reg, mem SBB mem, reg SBB accum, immed SBB reg, immed SBB mem, immed	if (CF) = 1 then (dest) = (dest) – (src) – 1 else (dest) ← (dest) – (src)	AF ✓ CF ✓ DF – IF – OF ✓ PF ✓ SF ✓ TF – ZF ✓

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected
SCAS	**Scan String:** SCAS *dest-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)." **Instruction Operands:** SCAS dest-string SCAS (repeat) dest-string	**When Source Operand is a Byte:** (AL) – (byte-string) if (DF) = 0 then (DI) ← (DI) + DELTA else (DI) ← (DI) – DELTA **When Source Operand is a Word:** (AX) – (word-string) if (DF) = 0 then (DI) ← (DI) + DELTA else (DI) ← (DI) – DELTA	AF ✓ CF ✓ DF – IF – OF ✓ PF ✓ SF ✓ TF – ZF ✓

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected
SHR	**Shift Logical Right:** SHR *dest, src* 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. **Instruction Operands:** SHR reg, n SHR mem, n SHR reg, CL SHR mem, CL	(temp) ← count do while (temp) ≠ 0 (CF) ← low-order bit of (dest) (dest) ← (dest) / 2 (temp) ← (temp) – 1 if count = 1 then if high-order bit of (dest) ≠ next-to-high-order bit of (dest) then (OF) ← 1 else (OF) ← 0 else (OF) undefined	AF ? CF ✓ DF – IF – OF ✓ PF ✓ SF ✓ TF – ZF ✓
STC	**Set Carry Flag:** STC Sets CF to 1. **Instruction Operands:** none	(CF) ← 1	AF – CF ✓ DF – IF – OF – PF – SF – TF – ZF –
STD	**Set Direction Flag:** STD Sets DF to 1 causing the string instructions to auto-decrement the SI and/or DI index registers. **Instruction Operands:** none	(DF) ← 1	AF – CF – DF ✓ IF – OF – PF – SF – TF – ZF –

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected
STI	**Set Interrupt-enable Flag:** STI 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. **Instruction Operands:** none	(IF) ← 1	AF – CF – DF – IF ✓ OF – PF – SF – TF – ZF –
STOS	**Store (Byte or Word) String:** STOS *dest-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. **Instruction Operands:** STOS dest-string STOS (repeat) dest-string	**When Source Operand is a Byte:** (DEST) ← (AL) if (DF) = 0 then (DI) ← (DI) + DELTA else (DI) ← (DI) – DELTA **When Source Operand is a Word:** (DEST) ← (AX) if (DF) = 0 then (DI) ← (DI) + DELTA else (DI) ← (DI) – DELTA	AF – CF – DF – IF – OF – PF – SF – TF – ZF –

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected

SUB	**Subtract:**
	SUB *dest, src*
	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).
	Instruction Operands:
	SUB reg, reg
	SUB reg, mem
	SUB mem, reg
	SUB accum, immed
	SUB reg, immed
	SUB mem, immed
		(dest) ← (dest) – (src)	AF ✓
			CF ✓
			DF –
			IF –
			OF ✓
			PF ✓
			SF ✓
			TF –
			ZF ✓

TEST	**Test:** TEST dest, src Performs the logical "and" of the two operands (bytes or words), 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 one bits in both operands. **Instruction Operands:** TEST reg, reg TEST reg, mem TEST accum, immed TEST reg, immed TEST mem, immed	(dest) and (src) (CF) ← 0 (OF) ← 0	AF ? CF ✓ DF – IF – OF ✓ PF ✓ SF ✓ TF – ZF ✓

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected
WAIT	**Wait:** WAIT Causes the CPU to enter the wait state while its test line is not active. **Instruction Operands:** none	None	AF – CF – DF – IF – OF – PF – SF – TF – ZF –
XCHG	**Exchange:** XCHG *dest, src* Switches the contents of the source and destination operands (bytes or words). 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. **Instruction Operands:** XCHG accum, reg XCHG mem, reg XCHG reg, reg	(temp) ← (dest) (dest) ← (src) (src) ← (temp)	AF – CF – DF – IF – OF – PF – SF – TF – ZF –

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

Name	Description	Operation	Flags Affected
XLAT	**Translate:** XLAT *translate-table* 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. **Instruction Operands:** XLAT src-table	AL ← ((BX) + (AL))	AF – CF – DF – IF – OF – PF – SF – TF – ZF –
XOR	**Exclusive Or:** XOR *dest, src* 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. **Instruction Operands:** XOR reg, reg XOR reg, mem XOR mem, reg XOR accum, immed XOR reg, immed XOR mem, immed	(dest) ← (dest) xor (src) (CF) ← 0 (OF) ← 0	AF ? CF ✓ DF – IF – OF ✓ PF ✓ SF ✓ TF – ZF ✓

NOTE: The three symbols used in the Flags Affected column are defined as follows:
– the contents of the flag remain unchanged after the instruction is executed
? the contents of the flag is undefined after the instruction is executed
✓the flag is updated after the instruction is executed

intel®

D

Instruction Set Opcodes and Clock Cycles

intel

APPENDIX D
INSTRUCTION SET OPCODES AND CLOCK CYCLES

This appendix provides reference information for the 80C186 Modular Core family instruction set. Table D-1 defines the variables used in Table D-2, which lists the instructions with their formats and execution times. Table D-3 is a guide for decoding machine instructions. Table D-4 is a guide for encoding instruction mnemonics, and Table D-5 defines Table D-4 abbreviations.

Table D-1. Operand Variables

Variable	Description
mod	*mod* and *r/m* determine the Effective Address (EA).
r/m	*r/m* and *mod* determine the Effective Address (EA).
reg	*reg* represents a register.
MMM	*MMM* and *PPP* are opcodes to the math coprocessor.
PPP	*PPP* and *MMM* are opcodes to the math coprocessor.
TTT	*TTT* defines which shift or rotate instruction is executed.

r/m	EA Calculation
0 0 0	(BX) + (SI) + DISP
0 0 1	(BX) + (DI) + DISP
0 1 0	(BP) + (SI) + DISP
0 1 1	(BP) + (DI) + DISP
1 0 0	(SI) + DISP
1 0 1	(DI) + DISP
1 1 0	(BP) + DISP, if mod ≠ 00 disp-high:disp-low, if mod =00
1 1 1	(BX) + DISP

mod	Effect on EA Calculation
0 0	if r/m ≠ 110, DISP = 0; disp-low and disp-high are absent
0 0	if r/m = 110, EA = disp-high:disp-low
0 1	DISP = disp-low, sign-extended to 16 bits; disp-high is absent
1 0	DISP = disp-high:disp-low
1 1	r/m is treated as a reg field

DISP follows the second byte of the instruction (before any required data).

Physical addresses of operands addressed by the BP register are computed using the SS segment register. Physical addresses of destination operands of string primitives (addressed by the DI register) are computed using the ES segment register, which cannot be overridden.

reg	16-bit (w=1)	8-bit (w=0)
0 0 0	AX	AL
0 0 1	CX	CL
0 1 0	DX	DL
0 1 1	BP	BL
1 0 0	SP	AH
1 0 1	BP	CH
1 1 0	SI	DH
1 1 1	DI	BH

TTT	Instruction
0 0 0	ROL
0 0 1	ROR
0 1 0	RCL
0 1 1	RCR
1 0 0	SHL/SAL
1 0 1	SHR
1 1 0	—
1 1 1	SAR

Table D-2. Instruction Set Summary

Function	Format				Clocks	Notes
DATA TRANSFER INSTRUCTIONS						
MOV = Move						
register to register/memory	1 0 0 0 1 0 0 w	mod reg r/m			2/12	
register/memory to register	1 0 0 0 1 0 1 w	mod reg r/m			2/9	
immediate to register/memory	1 1 0 0 0 1 1 w	mod 000 r/m	data	data if w=1	12/13	(1)
immediate to register	1 0 1 1 w reg	data	data if w=1		3/4	(1)
memory to accumulator	1 0 1 0 0 0 0 w	addr-low	addr-high		9	
accumulator to memory	1 0 1 0 0 0 1 w	addr-low	addr-high		8	
register/memory to segment register	1 0 0 0 1 1 1 0	mod 0 reg r/m			2/9	
segment register to register/memory	1 0 0 0 1 1 0 0	mod 0 reg r/m			2/11	
PUSH = Push						
memory	1 1 1 1 1 1 1 1	mod 110 r/m			16	
register	0 1 0 1 0 reg				10	
segment register	0 0 0 reg 1 1 0				9	
immediate	0 1 1 0 1 0 s 0	data	data if s=0		10	
POP = Pop						
memory	1 0 0 0 1 1 1 1	mod 000 r/m			20	
register	0 1 0 1 1 reg				10	
segment register	0 0 0 reg 1 1 1	(reg ?01)			8	
PUSHA = Push all	0 1 1 0 0 0 0 0				36	
POPA = Pop all	0 1 1 0 0 0 0 1				51	
XCHG = Exchange						
register/memory with register	1 0 0 0 0 1 1 w	mod reg r/m			4/17	
register with accumulator	1 0 0 1 0 reg				3	
DATA TRANSFER INSTRUCTIONS (Continued)						
XLAT = Translate byte to AL	1 1 0 1 0 1 1 1				11	
IN = Input from						
fixed port	1 1 1 0 0 1 0 w	port			10	
variable port	1 1 1 0 1 1 0 w				8	
OUT = Output from						
fixed port	1 1 1 0 0 1 0 w	port			9	

NOTES:
1. Clock cycles are given for 8-bit/16-bit operations.
2. Clock cycles are given for jump not taken/jump taken.
3. Clock cycles are given for interrupt taken/interrupt not taken.
4. If $\overline{TEST}$ = 0

Shading indicates additions and enhancements to the 8086/8088 instruction set. See Appendix A, "80C186 Instruction Set Additions and Extensions," for details.

Table D-2. Instruction Set Summary (Continued)

Function	Format				Clocks	Notes
variable port	1 1 1 0 1 1 0 w				7	
LEA = Load EA to register	1 0 0 0 1 1 0 1	mod reg r/m			6	
LDS = Load pointer to DS	1 1 0 0 0 1 0 1	mod reg r/m	(mod ?11)		18	
LES = Load pointer to ES	1 1 0 0 0 1 0 0	mod reg r/m	(mod ?11)		18	
ENTER = Build stack frame	1 1 0 0 1 0 0 0	data-low	data-high	L		
L = 0					15	
L = 1					25	
L > 1					22+16(n-1)	
LEAVE = Tear down stack frame	1 1 0 0 1 0 0 1				8	
LAHF = Load AH with flags	1 0 0 1 1 1 1 1				2	
SAHF = Store AH into flags	1 0 0 1 1 1 1 0				3	
PUSHF = Push flags	1 0 0 1 1 1 0 0				9	
POPF = Pop flags	1 0 0 1 1 1 0 1				8	
ARITHMETIC INSTRUCTIONS						
ADD = Add						
reg/memory with register to either	0 0 0 0 0 0 d w	mod reg r/m			3/10	
immediate to register/memory	1 0 0 0 0 0 s w	mod 000 r/m	data	data if sw=01	4/16	
immediate to accumulator	0 0 0 0 0 1 0 w	data	data if w=1		3/4	(1)
ADC = Add with carry						
reg/memory with register to either	0 0 0 1 0 0 d w	mod reg r/m			3/10	
immediate to register/memory	1 0 0 0 0 0 s w	mod 010 r/m	data	data if sw=01	4/16	
immediate to accumulator	0 0 0 1 0 1 0 w	data	data if w=1		3/4	(1)
INC = Increment						
register/memory	1 1 1 1 1 1 1 w	mod 000 r/m			3/15	
register	0 1 0 0 0 reg				3	
AAA = ASCII adjust for addition	0 0 1 1 0 1 1 1				8	
DAA = Decimal adjust for addition	0 0 1 0 0 1 1 1				4	

NOTES:
1. Clock cycles are given for 8-bit/16-bit operations.
2. Clock cycles are given for jump not taken/jump taken.
3. Clock cycles are given for interrupt taken/interrupt not taken.
4. If $\overline{TEST}$ = 0

Shading indicates additions and enhancements to the 8086/8088 instruction set. See Appendix A, "80C186 Instruction Set Additions and Extensions," for details.

Table D-2. Instruction Set Summary (Continued)

Function	Format				Clocks	Notes
SUB = Subtract						
reg/memory with register to either	0 0 1 0 1 0 d w	mod reg r/m			3/10	
immediate from register/memory	1 0 0 0 0 0 s w	mod 101 r/m	data	data if sw=01	4/16	
immediate from accumulator	0 0 0 1 1 1 0 w	data	data if w=1		3/4	(1)
SBB = Subtract with borrow						
reg/memory with register to either	0 0 0 1 1 0 d w	mod reg r/m			3/10	
immediate from register/memory	1 0 0 0 0 0 s w	mod 011 r/m	data	data if sw=01	4/16	
immediate from accumulator	0 0 0 1 1 1 0 w	data	data if w=1		3/4	(1)
ARITHMETIC INSTRUCTIONS (Continued)						
DEC = Decrement						
register/memory	1 1 1 1 1 1 1 w	mod 001 r/m			3/15	
register	0 1 0 0 1 reg				3	
NEG = Change sign	1 1 1 1 0 1 1 w	mod reg r/m			3	
CMP = Compare						
register/memory with register	0 0 1 1 1 0 1 w	mod reg r/m			3/10	
register with register/memory	0 0 1 1 1 0 0 w	mod reg r/m			3/10	
immediate with register/memory	1 0 0 0 0 0 s w	mod 111 r/m	data	data if sw=01	3/10	
immediate with accumulator	0 0 1 1 1 1 0 w	data	data if w=1		3/4	(1)
AAS = ASCII adjust for subtraction	0 0 1 1 1 1 1 1				7	
DAS = Decimal adjust for subtraction	0 0 1 0 1 1 1 1				4	
MUL = multiply (unsigned)	1 1 1 1 0 1 1 w	mod 100 r/m				
register-byte					26-28	
register-word					35-37	
memory-byte					32-34	
memory-word					41-43	
IMUL = Integer multiply (signed)	1 1 1 1 0 1 1 w	mod 101 r/m				
register-byte					25-28	
register-word					34-37	
memory-byte					31-34	

NOTES:
1. Clock cycles are given for 8-bit/16-bit operations.
2. Clock cycles are given for jump not taken/jump taken.
3. Clock cycles are given for interrupt taken/interrupt not taken.
4. If $\overline{TEST}$ = 0

Shading indicates additions and enhancements to the 8086/8088 instruction set. See Appendix A, "80C186 Instruction Set Additions and Extensions," for details.

Table D-2. Instruction Set Summary (Continued)

Function	Format				Clocks	Notes
memory-word					40-43	
integer immediate multiply (signed)	0 1 1 0 1 0 s 1	mod reg r/m	data	data if s=0	22-25 38-39	
ARITHMETIC INSTRUCTIONS (Continued)						
AAM = ASCII adjust for multiply	1 1 0 1 0 1 0 0	0 0 0 0 1 0 1 0			19	
DIV = Divide (unsigned)	1 1 1 1 0 1 1 w	mod 110 r/m				
register-byte					29	
register-word					38	
memory-byte					35	
memory-word					44	
IDIV = Integer divide (signed)	1 1 1 1 0 1 1 w	mod 111 r/m				
register-byte					29	
register-word					38	
memory-byte					35	
memory-word					44	
AAD = ASCII adjust for divide	1 1 0 1 0 1 0 1	0 0 0 0 1 0 1 0			15	
CBW = Convert byte to word	1 0 0 1 1 0 0 0				2	
CWD = Convert word to double-word	1 0 0 1 1 0 0 1				4	
BIT MANIPULATION INSTRUCTIONS						
NOT= Invert register/memory	1 1 1 1 0 1 1 w	mod 010 r/m			3	
AND = And						
reg/memory and register to either	0 0 1 0 0 0 d w	mod reg r/m			3/10	
immediate to register/memory	1 0 0 0 0 0 0 w	mod 100 r/m	data	data if w=1	4/16	
immediate to accumulator	0 0 1 0 0 1 0 w	data	data if w=1		3/4	(1)
BIT MANIPULATION INSTRUCTIONS (Continued)						
OR = Or						
reg/memory and register to either	0 0 0 0 1 0 d w	mod reg r/m			3/10	
immediate to register/memory	1 0 0 0 0 0 0 w	mod 001 r/m	data	data if w=1	4/10	

NOTES:
1. Clock cycles are given for 8-bit/16-bit operations.
2. Clock cycles are given for jump not taken/jump taken.
3. Clock cycles are given for interrupt taken/interrupt not taken.
4. If $\overline{TEST}$ = 0

Shading indicates additions and enhancements to the 8086/8088 instruction set. See Appendix A, "80C186 Instruction Set Additions and Extensions," for details.

Table D-2. Instruction Set Summary (Continued)

Function	Format				Clocks	Notes
immediate to accumulator	0000110w	data	data if w=1		3/4	(1)
XOR = Exclusive or						
reg/memory and register to either	001100dw	mod reg r/m			3/10	
immediate to register/memory	1000000w	mod 110 r/m	data	data if w=1	4/10	
immediate to accumulator	0011010w	data	data if w=1		3/4	(1)
TEST= And function to flags, no result						
register/memory and register	1000010w	mod reg r/m			3/10	
immediate data and register/memory	1111011w	mod 000 r/m	data	data if w=1	4/10	
immediate data and accumulator	1010100w	data	data if w=1		3/4	(1)
Shifts/Rotates						
register/memory by 1	1101000w	mod TTT r/m			2/15	
register/memory by CL	1101001w	mod TTT r/m			5+n/17+n	
register/memory by Count	1100000w	mod TTT r/m	count		5+n/17+n	
STRING MANIPULATION INSTRUCTIONS						
MOVS = Move byte/word	1010010w				14	
INS = Input byte/word from DX port	0110110w				14	
OUTS = Output byte/word to DX port	0110111w				14	
CMPS = Compare byte/word	1010011w				22	
SCAS = Scan byte/word	1010111w				15	
STRING MANIPULATION INSTRUCTIONS (Continued)						
LODS = Load byte/word to AL/AX	1010110w				12	
STOS = Store byte/word from AL/AX	1010101w				10	
Repeated by count in CX:						
MOVS = Move byte/word	11110010	1010010w			8+8n	
INS = Input byte/word from DX port	11110010	0110110w			8+8n	
OUTS = Output byte/word to DX port	11110010	0110111w			8+8n	
CMPS = Compare byte/word	1111001z	1010011w			5+22n	
SCAS = Scan byte/word	1111001z	1010111w			5+15n	
LODS = Load byte/word to AL/AX	11110010	0101001w			6+11n	
STOS = Store byte/word from AL/AX	11110100	0101001w			6+9n	

NOTES:
1. Clock cycles are given for 8-bit/16-bit operations.
2. Clock cycles are given for jump not taken/jump taken.
3. Clock cycles are given for interrupt taken/interrupt not taken.
4. If $\overline{TEST}$ = 0

Shading indicates additions and enhancements to the 8086/8088 instruction set. See Appendix A, "80C186 Instruction Set Additions and Extensions," for details.

Table D-2. Instruction Set Summary (Continued)

Function	Format			Clocks	Notes
PROGRAM TRANSFER INSTRUCTIONS					
Conditional Transfers — jump if:					
JE/JZ= equal/zero	01110100	disp		4/13	(2)
JL/JNGE = less/not greater or equal	01111100	disp		4/13	(2)
JLE/JNG = less or equal/not greater	01111110	disp		4/13	(2)
JB/JNAE = below/not above or equal	01110010	disp		4/13	(2)
JC = carry	01110010	disp		4/13	(2)
JBE/JNA = below or equal/not above	01110110	disp		4/13	(2)
JP/JPE = parity/parity even	01111010	disp		4/13	(2)
JO = overflow	01110000	disp		4/13	(2)
JS = sign	01111000	disp		4/13	(2)
JNE/JNZ = not equal/not zero	01110101	disp		4/13	(2)
PROGRAM TRANSFER INSTRUCTIONS (Continued)					
JNL/JGE = not less/greater or equal	01111101	disp		4/13	(2)
JNLE/JG = not less or equal/greater	01111111	disp		4/13	(2)
JNB/JAE = not below/above or equal	01110011	disp		4/13	(2)
JNC = not carry	01110011	disp		4/13	(2)
JNBE/JA = not below or equal/above	01110111	disp		4/13	(2)
JNP/JPO = not parity/parity odd	01111011	disp		4/13	(2)
JNO = not overflow	01110001	disp		4/13	(2)
JNS = not sign	01111001	disp		5/15	(2)
Unconditional Transfers					
CALL = Call procedure					
direct within segment	11101000	disp-low	disp-high	15	
reg/memory indirect within segment	11111111	mod 010 r/m		13/19	
indirect intersegment	11111111	mod 011 r/m	(mod ?11)	38	
direct intersegment	10011010	segment offset		23	
		selector			
RET = Return from procedure					
within segment	11000011			16	

NOTES:
1. Clock cycles are given for 8-bit/16-bit operations.
2. Clock cycles are given for jump not taken/jump taken.
3. Clock cycles are given for interrupt taken/interrupt not taken.
4. If $\overline{TEST}$ = 0

Shading indicates additions and enhancements to the 8086/8088 instruction set. See Appendix A, "80C186 Instruction Set Additions and Extensions," for details.

Table D-2. Instruction Set Summary (Continued)

Function	Format			Clocks	Notes
within segment adding immed to SP	11000010	data-low	data-high	18	
intersegment	11001011			22	
intersegment adding immed to SP	11001010	data-low	data-high	25	
PROGRAM TRANSFER INSTRUCTIONS (Continued)					
JMP = Unconditional jump					
short/long	11101011	disp-low		14	
direct within segment	11101001	disp-low	disp-high	14	
reg/memory indirect within segment	11111111	mod 100 r/m		26	
indirect intersegment	11111111	mod 101 r/m	(mod ?11)	11/17	
direct intersegment	11101010	segment offset		14	
		selector			
Iteration Control					
LOOP = Loop CX times	11100010	disp		6/16	(2)
LOOPZ/LOOPE =Loop while zero/equal	11100001	disp		5/16	(2)
LOOPNZ/LOOPNE = Loop while not zero/not equal	11100000	disp		5/16	(2)
JCXZ = Jump if CX = zero	11100011	disp		6/16	(2)
Interrupts					
INT = Interrupt					
Type specified	11001101	type		47	
Type 3	11001100			45	
INTO = Interrupt on overflow	11001110			48/4	(3)
BOUND = Detect value out of range	01100010	mod reg r/m		33-35	
IRET = Interrupt return	11001111			28	
PROCESSOR CONTROL INSTRUCTIONS					
CLC = Clear carry	11111000			2	

NOTES:
1. Clock cycles are given for 8-bit/16-bit operations.
2. Clock cycles are given for jump not taken/jump taken.
3. Clock cycles are given for interrupt taken/interrupt not taken.
4. If $\overline{TEST}$ = 0

Shading indicates additions and enhancements to the 8086/8088 instruction set. See Appendix A, "80C186 Instruction Set Additions and Extensions," for details.

Table D-2. Instruction Set Summary (Continued)

Function	Format		Clocks	Notes
CMC = Complement carry	1 1 1 1 0 1 0 1		2	
STC = Set carry	1 1 1 1 1 0 0 1		2	
CLD = Clear direction	1 1 1 1 1 1 0 0		2	
STD = Set direction	1 1 1 1 1 1 0 1		2	
CLI = Clear interrupt	1 1 1 1 1 0 1 0		2	
STI = Set interrupt	1 1 1 1 1 0 1 1		2	
HLT = Halt	1 1 1 1 0 1 0 0		2	
WAIT = Wait	1 0 0 1 1 0 1 1		6	(4)
LOCK = Bus lock prefix	1 1 1 1 0 0 0 0		2	
ESC = Math coprocessor escape	1 1 0 1 1 M M M	mod PPP r/m	6	
NOP = No operation	1 0 0 1 0 0 0 0		3	
SEGMENT OVERRIDE PREFIX				
CS	0 0 1 0 1 1 1 0		2	
SS	0 0 1 1 0 1 1 0		2	
DS	0 0 1 1 1 1 1 0		2	
ES	0 0 1 0 0 1 1 0		2	

NOTES:
1. Clock cycles are given for 8-bit/16-bit operations.
2. Clock cycles are given for jump not taken/jump taken.
3. Clock cycles are given for interrupt taken/interrupt not taken.
4. If $\overline{TEST}$ = 0

Shading indicates additions and enhancements to the 8086/8088 instruction set. See Appendix A, "80C186 Instruction Set Additions and Extensions," for details.

Table D-3. Machine Instruction Decoding Guide

Byte 1		Byte 2	Bytes 3–6	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
08	0000 0100	mod reg r/m	(disp-lo),(disp-hi)	or	reg8/mem8,reg8

Table D-3. Machine Instruction Decoding Guide (Continued)

Byte 1		Byte 2	Bytes 3–6	ASM-86 Instruction Format	
Hex	Binary				
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			—	
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

Table D-3. Machine Instruction Decoding Guide (Continued)

Byte 1		Byte 2	Bytes 3–6	ASM-86 Instruction Format	
Hex	Binary				
2E	0010 1110			DS:	(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)
37	0011 0111			aaa	
38	0011 1000	mod reg r/m	(disp-lo),(disp-hi)	xor	reg8/mem8,reg8
39	0011 1001	mod reg r/m	(disp-lo),(disp-hi)	xor	reg16/mem16,reg16
3A	0011 1010	mod reg r/m	(disp-lo),(disp-hi)	xor	reg8,reg8/mem8
3B	0011 1011	mod reg r/m	(disp-lo),(disp-hi)	xor	reg16,reg16/mem16
3C	0011 1100	data-8		xor	AL,immed8
3D	0011 1101	data-lo	data-hi	xor	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

Table D-3. Machine Instruction Decoding Guide (Continued)

Byte 1		Byte 2	Bytes 3–6	ASM-86 Instruction Format	
Hex	Binary				
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			pusha	
61	0110 0001			popa	
62	0110 0010	mod reg r/m		bound	reg16,mem16
63	0110 0011			—	
64	0110 0100			—	
65	0110 0101			—	
66	0110 0110			—	
67	0110 0111			—	
68	0110 1000	data-lo	data-hi	push	immed16
69	0110 1001	mod reg r/m	data-lo, data-hi	imul	immed16
70	0111 0000	IP-inc-8		jo	short-label
71	0111 0001	IP-inc-8		jno	short-label
72	0111 0010	IP-inc-8		jb/jnae/jc	short-label
73	0111 0011	IP-inc-8		jnb/jae/jnc	short-label
74	0111 0100	IP-inc-8		je/jz	short-label
75	0111 0101	IP-inc-8		jne/jnz	short-label
76	0111 0110	IP-inc-8		jbe/jna	short-label
77	0111 0111	IP-inc-8		jnbe/ja	short-label
78	0111 1000	IP-inc-8		js	short-label
79	0111 1001	IP-inc-8		jns	short-label
7A	0111 1010	IP-inc-8		jp/jpe	short-label
7B	0111 1011	IP-inc-8		jnp/jpo	short-label
7C	0111 1100	IP-inc-8		jl/jnge	short-label
7D	0111 1101	IP-inc-8		jnl/jge	short-label

Table D-3. Machine Instruction Decoding Guide (Continued)

Byte 1		Byte 2	Bytes 3–6	ASM-86 Instruction Format	
Hex	Binary				
7E	0111 1110	IP-inc-8		jle/jng	short-label
7F	0111 1111	IP-inc-8		jnle/jg	short-label
80	1000 0000	mod 000 r/m	(disp-lo),(disp-hi), data-8	add	reg8/mem8,immed8
		mod 001 r/m	(disp-lo),(disp-hi), data-8	or	reg8/mem8,immed8
		mod 010 r/m	(disp-lo),(disp-hi), data-8	adc	reg8/mem8,immed8
		mod 011 r/m	(disp-lo),(disp-hi), data-8	sbb	reg8/mem8,immed8
		mod 100 r/m	(disp-lo),(disp-hi), data-8	and	reg8/mem8,immed8
		mod 101 r/m	(disp-lo),(disp-hi), data-8	sub	reg8/mem8,immed8
		mod 110 r/m	(disp-lo),(disp-hi), data-8	xor	reg8/mem8,immed8
		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
		mod 001 r/m	(disp-lo),(disp-hi), data-lo,data-hi	or	reg16/mem16,immed16
		mod 010 r/m	(disp-lo),(disp-hi), data-lo,data-hi	adc	reg16/mem16,immed16
		mod 011 r/m	(disp-lo),(disp-hi), data-lo,data-hi	sbb	reg16/mem16,immed16
		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
		mod 110 r/m	(disp-lo),(disp-hi), data-lo,data-hi	xor	reg16/mem16,immed16
		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
		mod 001 r/m		—	
		mod 010 r/m	(disp-lo),(disp-hi), data-8	adc	reg8/mem8,immed8
		mod 011 r/m	(disp-lo),(disp-hi), data-8	sbb	reg8/mem8,immed8
		mod 100 r/m		—	
		mod 101 r/m	(disp-lo),(disp-hi), data-8	sub	reg8/mem8,immed8
		mod 110 r/m		—	
		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
		mod 001 r/m		—	
		mod 010 r/m	(disp-lo),(disp-hi), data-SX	adc	reg16/mem16,immed8
		mod 011 r/m	(disp-lo),(disp-hi), data-SX	sbb	reg16/mem16,immed8
		mod 100 r/m		—	
		mod 101 r/m	(disp-lo),(disp-hi), data-SX	sub	reg16/mem16,immed8
		mod 110 r/m		—	
		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

Table D-3. Machine Instruction Decoding Guide (Continued)

Byte 1		Byte 2	Bytes 3–6	ASM-86 Instruction Format	
Hex	Binary				
87	1000 0111	mod reg r/m	(disp-lo),(disp-hi)	xchg	reg16,reg16/mem16
88	1000 0100	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 OSR r/m	(disp-lo),(disp-hi)	mov	reg16/mem16,SEGREG
		mod 1 - r/m		—	
8D	1000 1101	mod reg r/m	(disp-lo),(disp-hi)	lea	reg16,mem16
8E	1000 1110	mod OSR r/m	(disp-lo),(disp-hi)	mov	SEGREG,reg16/mem16
		mod 1 - r/m		—	
8F	1000 1111			pop	mem16
90	1001 0000			nop	(xchg 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

Table D-3. Machine Instruction Decoding Guide (Continued)

Byte 1		Byte 2	Bytes 3–6	ASM-86 Instruction Format	
Hex	Binary				
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 0011	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
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	mod 000 r/m	data-8	rol	reg8/mem8, immed8
		mod 001 r/m	data-8	ror	reg8/mem8, immed8
		mod 010 r/m	data-8	rcl	reg8/mem8, immed8
		mod 011 r/m	data-8	rcr	reg8/mem8, immed8
		mod 100 r/m	data-8	shl/sal	reg8/mem8, immed8
		mod 101 r/m	data-8	shr	reg8/mem8, immed8
		mod 110 r/m		—	
		mod 111 r/m	data-8	sar	reg8/mem8, immed8
C1	1100 0001	mod 000 r/m	data-8	rol	reg16/mem16, immed8
		mod 001 r/m	data-8	ror	reg16/mem16, immed8
		mod 010 r/m	data-8	rcl	reg16/mem16, immed8
		mod 011 r/m	data-8	rcr	reg16/mem16, immed8
		mod 100 r/m	data-8	shl/sal	reg16/mem16, immed8
		mod 101 r/m	data-8	shr	reg16/mem16, immed8
		mod 110 r/m		—	

Table D-3. Machine Instruction Decoding Guide (Continued)

Byte 1		Byte 2	Bytes 3–6	ASM-86 Instruction Format	
Hex	Binary				
		mod 111 r/m	data-8	sar	reg16/mem16, immed8
C2	1100 0010	data-lo	data-hi	ret	immed16 (intrasegment)
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
		mod 001 r/m		—	
		mod 010 r/m		—	
		mod 011 r/m		—	
		mod 100 r/m		—	
		mod 101 r/m		—	
		mod 110 r/m		—	
C6	1100 0110	mod 111 r/m		—	
C7	1100 0111	mod 000 r/m	(disp-lo),(disp-hi),data-lo,data-hi	mov	mem16,immed16
		mod 001 r/m		—	
		mod 010 r/m		—	
		mod 011 r/m		—	
		mod 100 r/m		—	
		mod 101 r/m		—	
		mod 110 r/m		—	
		mod 111 r/m		—	
C8	1100 1000	data-lo	data-hi, level	enter	immed16, immed8
C9	1100 1001			leave	
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
		mod 001 r/m	(disp-lo),(disp-hi)	ror	reg8/mem8,1
		mod 010 r/m	(disp-lo),(disp-hi)	rcl	reg8/mem8,1
		mod 011 r/m	(disp-lo),(disp-hi)	rcr	reg8/mem8,1
		mod 100 r/m	(disp-lo),(disp-hi)	sal/shl	reg8/mem8,1
		mod 101 r/m	(disp-lo),(disp-hi)	shr	reg8/mem8,1
		mod 110 r/m		—	
		mod 111 r/m	(disp-lo),(disp-hi)	sar	reg8/mem8,1

Table D-3. Machine Instruction Decoding Guide (Continued)

Byte 1		Byte 2	Bytes 3–6	ASM-86 Instruction Format	
Hex	Binary				
D1	1101 0001	mod 000 r/m	(disp-lo),(disp-hi)	rol	reg16/mem16,1
		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
		mod 011 r/m	(disp-lo),(disp-hi)	rcr	reg16/mem16,1
		mod 100 r/m	(disp-lo),(disp-hi)	sal/shl	reg16/mem16,1
		mod 101 r/m	(disp-lo),(disp-hi)	shr	reg16/mem16,1
		mod 110 r/m		—	
		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
		mod 001 r/m	(disp-lo),(disp-hi)	ror	reg8/mem8,CL
		mod 010 r/m	(disp-lo),(disp-hi)	rcl	reg8/mem8,CL
		mod 011 r/m	(disp-lo),(disp-hi)	rcr	reg8/mem8,CL
		mod 100 r/m	(disp-lo),(disp-hi)	sal/shl	reg8/mem8,CL
		mod 101 r/m	(disp-lo),(disp-hi)	shr	reg8/mem8,CL
		mod 110 r/m		—	
		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
		mod 001 r/m	(disp-lo),(disp-hi)	ror	reg16/mem16,CL
		mod 010 r/m	(disp-lo),(disp-hi)	rcl	reg16/mem16,CL
		mod 011 r/m	(disp-lo),(disp-hi)	rcr	reg16/mem16,CL
		mod 100 r/m	(disp-lo),(disp-hi)	sal/shl	reg16/mem16,CL
		mod 101 r/m	(disp-lo),(disp-hi)	shr	reg16/mem16,CL
		mod 110 r/m		—	
		mod 111 r/m	(disp-lo),(disp-hi)	sar	reg16/mem16,CL
D4	1101 0100	0000 1010		aam	
D5	1101 0101	0000 1010		aad	
D6	1101 0110			—	
D7	1101 0111			xlat	source-table
D8	1101 1000	mod 000 r/m	(disp-lo),(disp-hi)	esc	opcode,source
D9	1101 1001	mod 001 r/m	(disp-lo),(disp-hi)	esc	opcode,source
DA	1101 1010	mod 010 r/m	(disp-lo),(disp-hi)	esc	opcode,source
DB	1101 1011	mod 011 r/m	(disp-lo),(disp-hi)	esc	opcode,source
DC	1101 1100	mod 100 r/m	(disp-lo),(disp-hi)	esc	opcode,source
DD	1101 1101	mod 101 r/m	(disp-lo),(disp-hi)	esc	opcode,source
DE	1101 1110	mod 110 r/m	(disp-lo),(disp-hi)	esc	opcode,source
DF	1101 1111	mod 111 r/m	(disp-lo),(disp-hi)	esc	opcode,source
E0	1110 0000	IP-inc-8		loopne/loopnz	short-label

Table D-3. Machine Instruction Decoding Guide (Continued)

Byte 1		Byte 2	Bytes 3–6	ASM-86 Instruction Format	
Hex	Binary				
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-inc-8		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			—	
F2	1111 0010			repne/repnz	
F3	1111 0011			rep/repe/repz	
F4	1111 0100			hlt	
F5	1111 0101			cmc	
F6	1111 0110	mod 000 r/m	(disp-lo),(disp-hi),data-8	test	reg8/mem8,immed8
		mod 001 r/m		—	
		mod 010 r/m	(disp-lo),(disp-hi)	not	reg8/mem8
		mod 011 r/m	(disp-lo),(disp-hi)	neg	reg8/mem8
		mod 100 r/m	(disp-lo),(disp-hi)	mul	reg8/mem8
		mod 101 r/m	(disp-lo),(disp-hi)	imul	reg8/mem8
		mod 110 r/m	(disp-lo),(disp-hi)	div	reg8/mem8
		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
		mod 001 r/m		—	
		mod 010 r/m	(disp-lo),(disp-hi)	not	reg16/mem16
		mod 011 r/m	(disp-lo),(disp-hi)	neg	reg16/mem16
		mod 100 r/m	(disp-lo),(disp-hi)	mul	reg16/mem16
		mod 101 r/m	(disp-lo),(disp-hi)	imul	reg16/mem16
		mod 110 r/m	(disp-lo),(disp-hi)	div	reg16/mem16
		mod 111 r/m	(disp-lo),(disp-hi)	idiv	reg16/mem16

Table D-3. Machine Instruction Decoding Guide (Continued)

Byte 1		Byte 2	Bytes 3–6	ASM-86 Instruction Format	
Hex	Binary				
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	mem16
		mod 001 r/m	(disp-lo),(disp-hi)	dec	mem16
		mod 010 r/m		—	
FE	1111 1110	mod 011 r/m		—	
		mod 100 r/m		—	
		mod 101 r/m		—	
		mod 110 r/m		—	
		mod 111 r/m		—	
FF	1111 1111	mod 000 r/m	(disp-lo),(disp-hi)	inc	mem16
		mod 001 r/m	(disp-lo),(disp-hi)	dec	mem16
		mod 010 r/m	(disp-lo),(disp-hi)	call	reg16/mem16 (intrasegment)
		mod 011 r/m	(disp-lo),(disp-hi)	call	mem16 (intersegment)
		mod 100 r/m	(disp-lo),(disp-hi)	jmp	reg16/mem16 (intrasegment)
		mod 101 r/m	(disp-lo),(disp-hi)	jmp	mem16 (intersegment)
		mod 110 r/m	(disp-lo),(disp-hi)	push	mem16
		mod 111 r/m		—	

Table D-4. Mnemonic Encoding Matrix (Left Half)

	x0	x1	x2	x3	x4	x5	x6	x7
0x	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
1x	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
2x	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
3x	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
4x	INC AX	INC CX	INC DX	INC BX	INC SP	INC BP	INC SI	INC DI
5x	PUSH AX	PUSH CX	PUSH DX	PUSH BX	PUSH SP	PUSH BP	PUSH SI	PUSH DI
6x	PUSHA	POPA	BOUND w,f,r/m					
7x	JO	JNO	JB/ JNAE/ JC	JNB/ JAE/ JNC	JE/ JZ	JNE/ JNZ	JBE/ JNA	JNBE/ JA
8x	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
9x	NOP (XCHG) AX	XCHG CX	XCHG DX	XCHG BX	XCHG SP	XCHG BP	XCHG SI	XCHG DI
Ax	MOV m→AL	MOV m→AX	MOV AL→m	MOV AX→m	MOVS	MOVS	CMPS	CMPS
Bx	MOV i→AL	MOV i→CL	MOV i→DL	MOV i→BL	MOV i→AH	MOV i→CH	MOV i→DH	MOV i→BH
Cx	Shift b,i	Shift w,i	RET (i+SP)	RET	LES	LDS	MOV b,i,r/m	MOV w,i,r/m
Dx	Shift b	Shift w	Shift b,v	Shift w,v	AAM	AAD		XLAT
Ex	LOOPNZ/ LOOPNE	LOOPZ/ LOOPE	LOOP	JCXZ	IN	IN	OUT	OUT
Fx	LOCK		REP	REP z	HLT	CMC	Grp1 b,r/m	Grp1 w,r/m

NOTE: Table D-5 defines abbreviations used in this matrix. Shading indicates reserved opcodes.

Table D-4. Mnemonic Encoding Matrix (Right Half)

x8	x9	xA	xB	xC	xD	xE	xF	
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		0x
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	1x
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	2x
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	3x
DEC AX	DEC CX	DEC DX	DEC BX	DEC SP	DEC BP	DEC SI	DEC DI	4x
POP AX	POP CX	POP DX	POP BX	POP SP	POP BP	POP SI	POP DI	5x
PUSH w,i	IMUL w,i	PUSH b,i	IMUL w,i	INS b	INS w	OUTS b	OUTS w	6x
JS	JNS	JP/ JPE	JNP/ JPO	JL/ JNGE	JNL/ JGE	JLE/ JNG	JNLE/ JG	7x
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	8x
CBW	CWD	CALL L,D	WAIT	PUSHF	POPF	SAHF	LAHF	9x
TEST b,ia	TEST w,ia	STOS	STOS	LODS	LODS	SCAS	SCAS	Ax
MOV i→AX	MOV i→CX	MOV i→DX	MOV i→BX	MOV i→SP	MOV i→BP	MOV i→SI	MOV i→DI	Bx
ENTER	LEAVE	RET I(i+SP)	RET I	INT type 3	INT (any)	INTO	IRET	Cx
ESC 0	ESC 1	ESC 2	ESC 3	ESC 4	ESC 5	ESC 6	ESC 7	Dx
CALL	JMP	JMP	JMP	IN	IN	OUT	OUT	Ex
CLC	STC	CLI	STI	CLS	STD	Grp2 b,r/m	Grp2 w,r/m	Fx

NOTE: Table D-5 defines abbreviations used in this matrix. Shading indicates reserved opcodes.

Table D-5. Abbreviations for Mnemonic Encoding Matrix

Abbr	Definition	Abbr	Definition	Abbr	Definition	Abbr	Definition
b	byte operation	ia	immediate to accumulator	m	memory	t	to CPU register
d	direct	id	indirect	r/m	EA is second byte	v	variable
f	from CPU register	is	immediate byte, sign extended	si	short intrasegment	w	word operation
i	immediate	l	long (intersegment)	sr	segment register	z	zero

Byte 2	Immed	Shift	Grp1	Grp2
mod 000 r/m	ADD	ROL	TEST	INC
mod 001 r/m	OR	ROR	—	DEC
mod 010 r/m	ADC	RCL	NOT	CALL id
mod 011 r/m	SBB	RCR	NEG	CALL l, id
mod 100 r/m	AND	SHL/SAL	MUL	JMP id
mod 101 r/m	SUB	SHR	IMUL	JMP i, id
mod 110 r/m	XOR	—	DIV	PUSH
mod 111 r/m	CMP	SAR	IDIV	—

mod and *r/m* determine the Effective Address (EA) calculation. See Table D-1 for definitions.

Index

INDEX

C

D

L

M

S

T

W

Z

UNITED STATES
Intel Corporation
2200 Mission College Boulevard
P.O. Box 58119
Santa Clara, CA 95052-8119

JAPAN
Intel Japan K.K.
5-6 Tokodai
Tsukuba-shi
Ibaraki-ken, 300-26

FRANCE
Intel Corporation S.A.R.L.
1, Rue Edison - BP 303
78054 Saint-Quentin-en-Yvelines, Cedex

UNITED KINGDOM
Intel Corporation (U.K.) Ltd.
Pipers Way
Swindon, SN3 1RJ Wiltshire
England

GERMANY
Intel GmbH
Dornacher Strasse 1
D-85622 Feldkirchen bei Muenchen

HONG KONG
Intel Semiconductor Ltd.
32/F, Two Pacific Place
88 Queensway, Central

CANADA
Intel Semiconductor of Canada, Ltd.
190 Attwell Drive, Suite 500
Rexdale, Ontario M9W 6H8

Order Number: 270830-003
Printed in USA/EAO95-116/395 2K/BC DS