80C186/C188
80C186XL/C188XL

USER'S MANUAL

intel®

Order Number: 272164-001

LITERATURE

To order Intel literature or obtain literature pricing information in the U.S. and Canada call or write Intel Literature Sales. In Europe and other international locations, please contact your ***local*** sales office or distributor.

INTEL LITERATURE SALES
P.O. Box 7641
Mt. Prospect, IL 60056-7641

In the U.S. and Canada
call toll free
(800) 548-4725
This 800 number is for external customers only.

CURRENT HANDBOOKS

Product line handbooks contain data sheets, application notes, article reprints and other design information. All handbooks can be ordered individually, and most are available in a pre-packaged set in the U.S. and Canada.

TITLE	INTEL ORDER NUMBER	ISBN
SET OF TEN HANDBOOKS (Available in U.S. and Canada)	**231003**	**N/A**
CONTENTS LISTED BELOW FOR INDIVIDUAL ORDERING:		
EMBEDDED CONTROLLERS & PROCESSORS (2 volume set)	270645	1-55512-140-3
MEMORY PRODUCTS	210830	1-55512-144-6
MICROCOMMUNICATIONS	231658	1-55512-148-9
MICROCOMPUTER PRODUCTS	280407	1-55512-143-8
MICROPROCESSORS	230843	1-55512-150-0
MULTIMEDIA & SUPERCOMPUTING PROCESSORS	272084	1-55512-149-7
PACKAGING	240800	1-55512-145-4
PERIPHERAL COMPONENTS	296467	1-55512-146-2
PRODUCT OVERVIEW (A guide to Intel Architectures and Applications)	210846	1-55512-142-x
PROGRAMMABLE LOGIC	296083	1-55512-147-0
ADDITIONAL LITERATURE: (Not included in handbook set)		
AUTOMOTIVE HANDBOOK	231792	1-55512-125-x
COMPONENTS QUALITY/RELIABILITY	210997	1-55512-132-2
CUSTOMER LITERATURE GUIDE	210620	N/A
EMBEDDED APPLICATIONS	270648	1-55512-123-3
INTERNATIONAL LITERATURE GUIDE (Available in Europe only)	E00029	N/A
MILITARY HANDBOOK (2 volume set)	210461	1-55512-126-8
SYSTEMS QUALITY/RELIABILITY	231762	1-55512-046-6
HANDBOOK DIRECTORY (Index of all data sheets contained in the handbooks)	241197	N/A

intel®

U.S. and CANADA LITERATURE ORDER FORM

NAME: ______________________________

COMPANY: ______________________________

ADDRESS: ______________________________

CITY: ____________________ STATE: __________ ZIP: __________

COUNTRY: ______________________________

PHONE NO.: (   ) ______________________________

ORDER NO.	TITLE	QTY.		PRICE		TOTAL
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	

Subtotal __________

Must Add Your
Local Sales Tax __________

Include postage:
Must add 15% of Subtotal to cover U.S. and Canada postage. (20% all other.) → Postage __________

Total __________

Pay by check, money order, or include company purchase order with this form ($200 minimum). We also accept VISA, MasterCard or American Express. Make payment to Intel Literature Sales. Allow 2-4 weeks for delivery.

☐ VISA ☐ MasterCard ☐ American Express Expiration Date ______________________________

Account No. ______________________________

Signature ______________________________

Mail To: Intel Literature Sales
P.O. Box 7641
Mt. Prospect, IL 60056-7641

International Customers outside the U.S. and Canada should use the International order form on the next page or contact their local Sales Office or Distributor.

For phone orders in the U.S. and Canada
Call Toll Free: (800) 548-4725

Prices good until 12/31/92.
Source HB

INTERNATIONAL LITERATURE ORDER FORM

NAME: ______________________________

COMPANY: ______________________________

ADDRESS: ______________________________

CITY: ____________________ STATE: ________ ZIP: ________

COUNTRY: ______________________________

PHONE NO.: (   ) ______________________________

ORDER NO.	TITLE	QTY.		PRICE		TOTAL
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	

Subtotal ________

Must Add Your Local Sales Tax ________

Total ________

PAYMENT

Cheques should be made payable to your ***local*** Intel Sales Office (see inside back cover).

Other forms of payment may be available in your country. Please contact the Literature Coordinator at your ***local*** Intel Sales Office for details.

The completed form should be marked to the attention of the LITERATURE COORDINATOR and returned to your ***local*** Intel Sales Office.

intel®

80C186/188, 80C186XL/C188XL USER'S MANUAL

1992

Order Number 272164-001

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

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

TABLE OF CONTENTS

Figures

Tables

Examples

Introduction 1

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 very high performance "x86" compatible CPUs that today includes the i386™ and i486™ 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 quarter 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 rapidly proliferate the 80186 family 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 operates down to 2.7 volts to directly support portable applications. This makes it the first high integration microprocessor to work directly off of two standard cell batteries. 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 extended again 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 plus 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 a total of 14 on-chip peripherals (see Figure 1.1).

The 80C186 family of products are the direct result of ten years of Intel development. They offer the designer the peace of mind of a well established architecture with the benefits of state of the art technology.

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			IMPROVED	IMPROVED
DMA UNIT	2 CHANNEL	2 CHANNEL		4 CHANNEL
SERIAL COMMUNICATIONS UNIT				
REFRESH CONTROL UNIT				
WATCHDOG TIMER UNIT				
I/O PORTS			16 TOTAL	22 TOTAL

Figure 1.1. Comparison of 80C186 Modular Core Family Products

1.1 DIFFERENCES BETWEEN THE 80C186 AND THE 80C186XL PRODUCT FAMILIES

As described earlier in this chapter, the 80C186XL and 80C188XL are functionally identical to the 80C186 and 80C188 respectively. Below is a list of the key differences:

1. The 80C186/C188 were developed on a 1.5 micron CMOS process, while the 80C186XL/C188XL were developed on a 1.0 micron CMOS process.

2. The 80C186/C188 are dynamic (i.e. requires a minimum operating frequency), while the 80C186XL/C188XL are static (i.e. minimum operating frequency is DC).
3. The maximum operating frequency of the 80C186/C188 is 16 MHz, while the 80C186XL/C188XL operate up to 20 MHz.
4. The 80C186XL/C188XL consume lower current than a 80C186/C188 operating at the same frequency.
5. The 80C186XL/C188XL have a differentiated set of A.C. and D.C. specifications over the 80C186/C188 due to its increased performance (see Appendix D for details).
6. The 80C186XL/C188XL fix all of the errata documented on the 80C186/C188.

In most applications, the 80C186XL/C188XL can replace the 80C186/C188 without any modifications to board layout, hardware design or device speed selection. However, since there are some A.C. and D.C. specification changes, it is recommended that a thorough design analysis be completed to ensure reliable system operation.

1.2 HOW TO USE THIS MANUAL

Throughout this manual you will come across phrases such as "*80C186 Modular Core Family*" or "*80C188 Modular Core*" as well as references to specific products such as "*80C188EA*". Each of these terms 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: 80C186EA/C188EA, 80C186EB/C188EB, 80C186EC/C188EC and 80C186XL/C188XL.

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

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

Specific Product References: For example the phrase "*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 of 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.

Overview of the 80C186 Family Modular Microprocessor Core Architecture

2

CHAPTER 2
OVERVIEW OF THE 80C186 FAMILY MODULAR MICROPROCESSOR CORE ARCHITECTURE

The 80C186 Modular Microprocessor Core shares a common base architecture with the 8086, 8088, 80186, 80188, 80286, i386™ and i486™ 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. There are several additional instructions which simplify programming and reduce code size (see *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 and 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) may 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) may 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 register containing flag bits. Figure 2.1 is a simplified CPU block diagram.

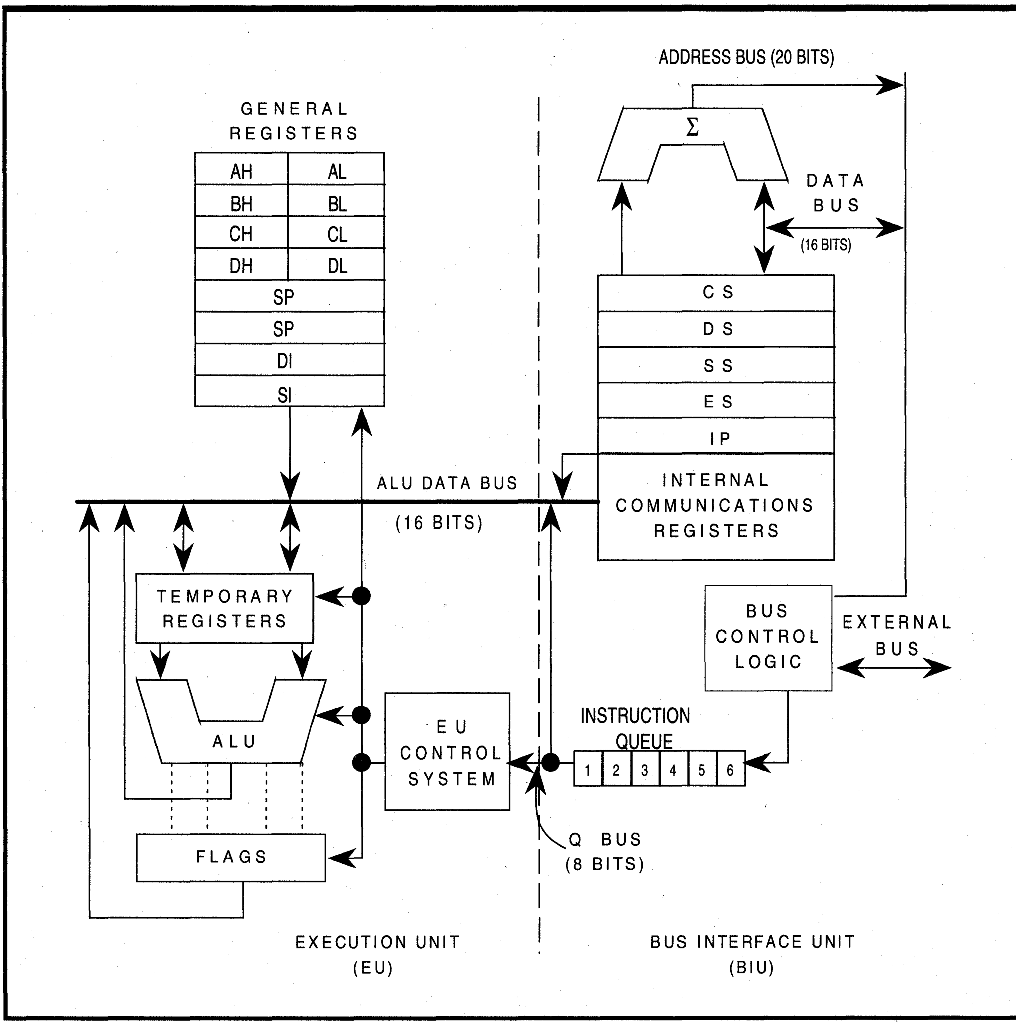


Figure 2.1. Simplified Functional Block Diagram of the 80C186 Modular Core Family CPU

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 which allows the Execution Unit to access the full megabyte of memory space.

For the Execution Unit to execute an instruction, it must 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 instruction byte to be fetched by the Bus Interface Unit.

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.

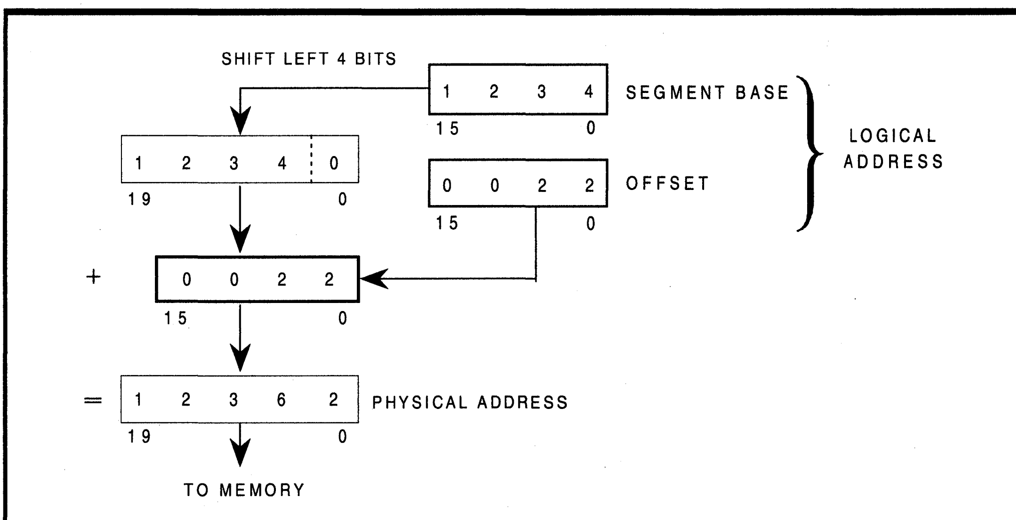


Figure 2.2. Physical Address Generation

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).

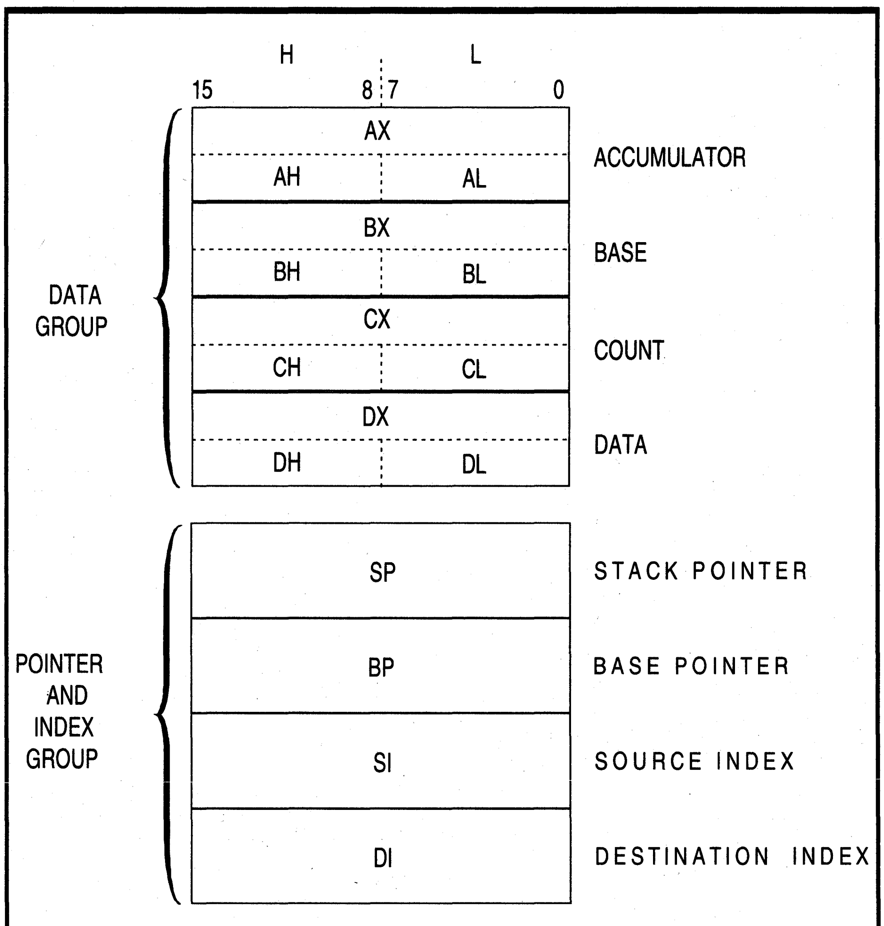


Figure 2.3. General Registers

The data registers may 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 one megabyte 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, typically used for data storage. Programs can access and manipulate the segment registers with several instructions.

The CS register initializes to 0FFFFH and the DS, ES and SS registers initialize to 0000H.

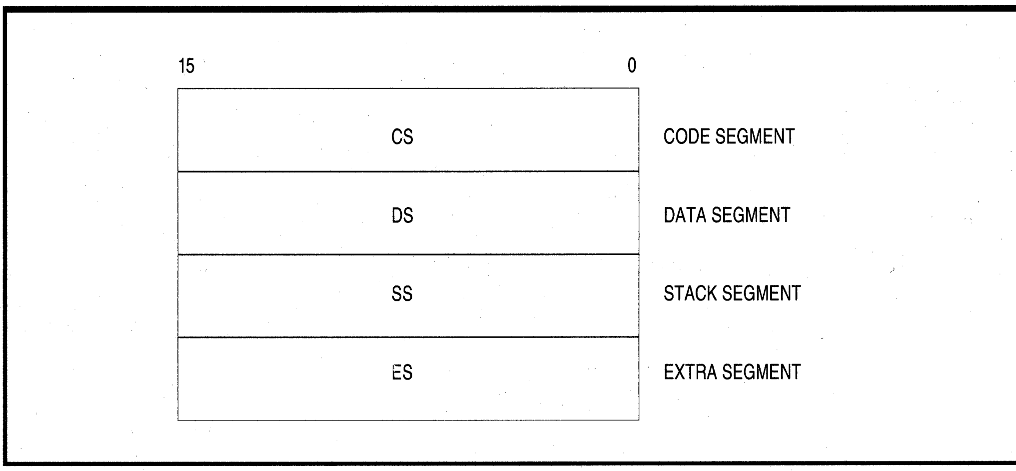


Figure 2.4. Segment Registers

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 may 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 Section 2.1.8 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 the high address to the low address or "right to left". Clearing DF causes string operations to auto-increment on process strings "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.

Both 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 OF000H.

2.1.7. MEMORY SEGMENTATION

Programs for the 80C186 Modular Core family view the one megabyte memory space as a group of user-defined segments. A segment is a logical unit of memory that may 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 may be adjacent, disjoint, partially overlapped or fully overlapped (see Figure 2.6). A physical memory location may 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.

15															0
				O F	D F	I F	T F	S F	Z F		A F		P F		C F

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 will recognize maskable interrupt requests. If IF is clear, maskable interrupts are ignored.
TF	*Trap Flag*	0	If TF is set, the processor will enter 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 ZP is set, the result of an operation is zero.
AF	*Auxiliary Carry 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.

Figure 2.5. Processor Status Word

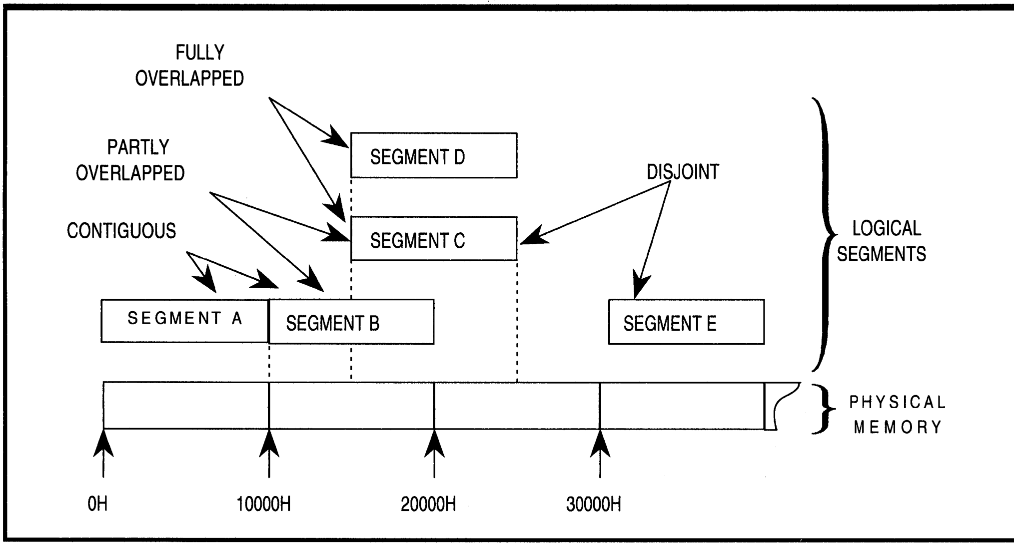


Figure 2.6. Segment Locations in Physical Memory

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.

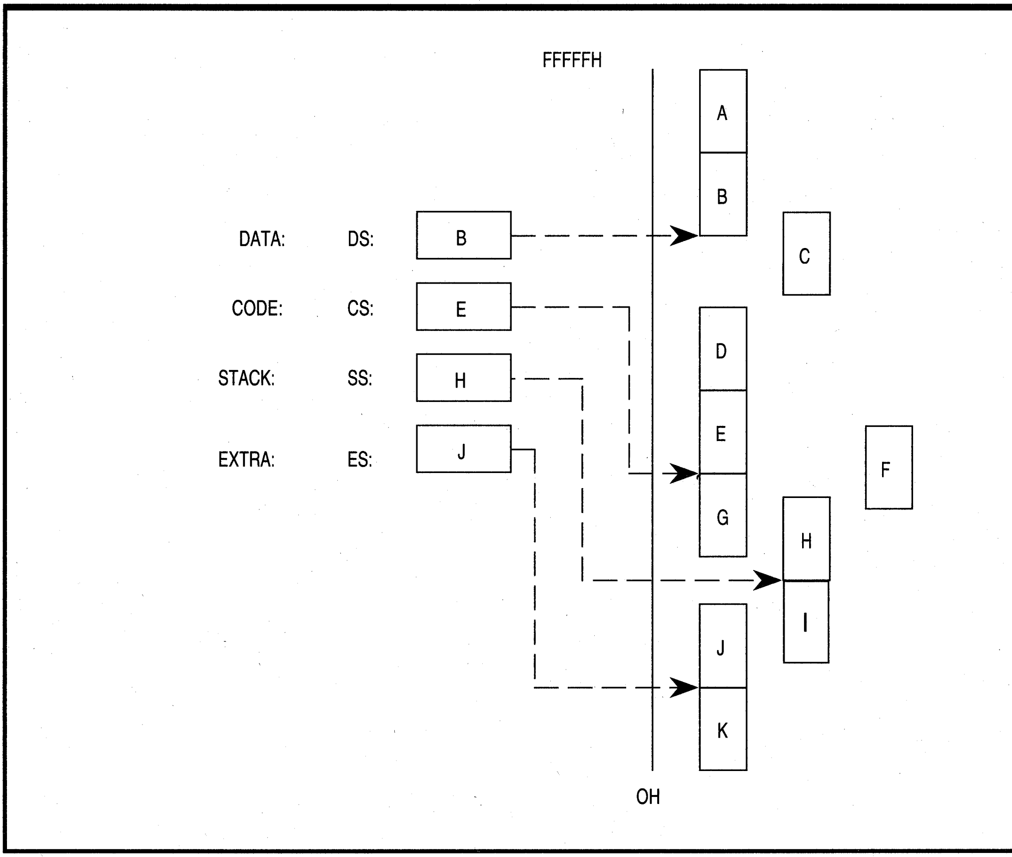


Figure 2.7. Currently Addressable Segments

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 which 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.

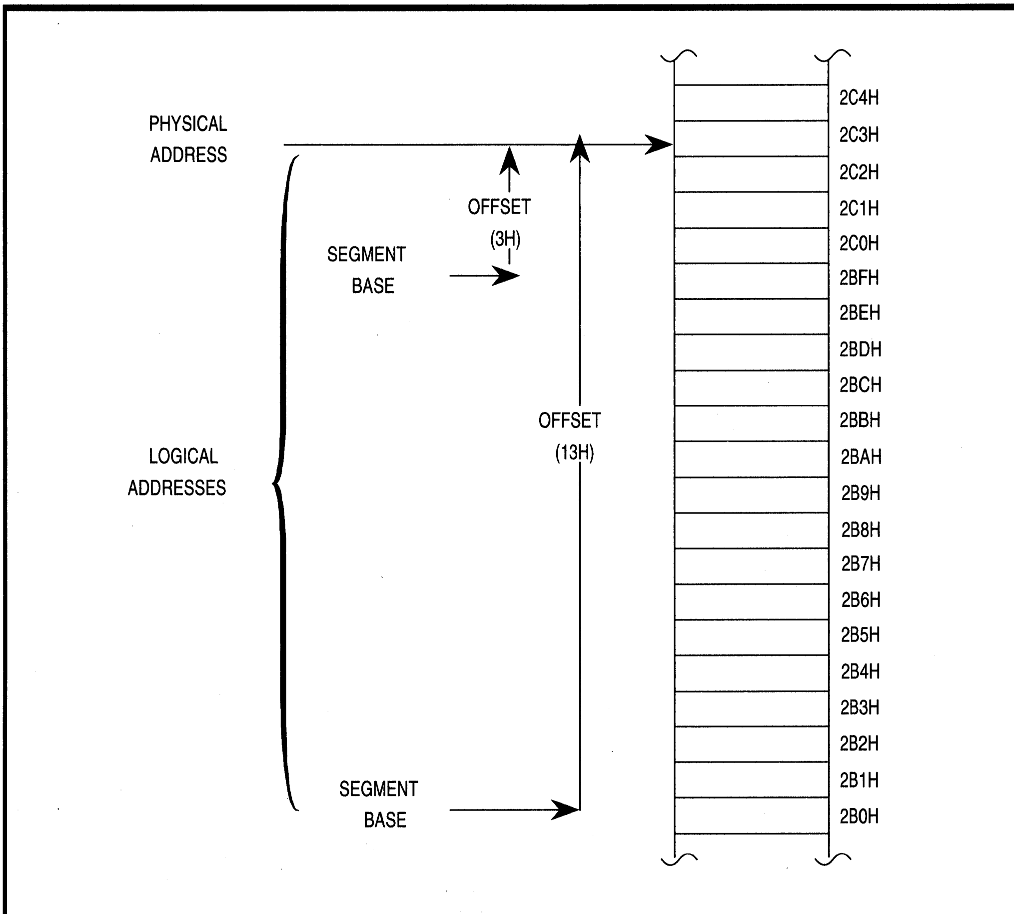


Figure 2.8. Logical and Physical Address

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 Section 2.2.2). 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 may 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 only available in non-adjacent fragments, other program segments can be compacted to free up enough continuous space. This process is illustrated graphically in Figure 2.9.

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 the segment registers are updated to point to the new base addresses.

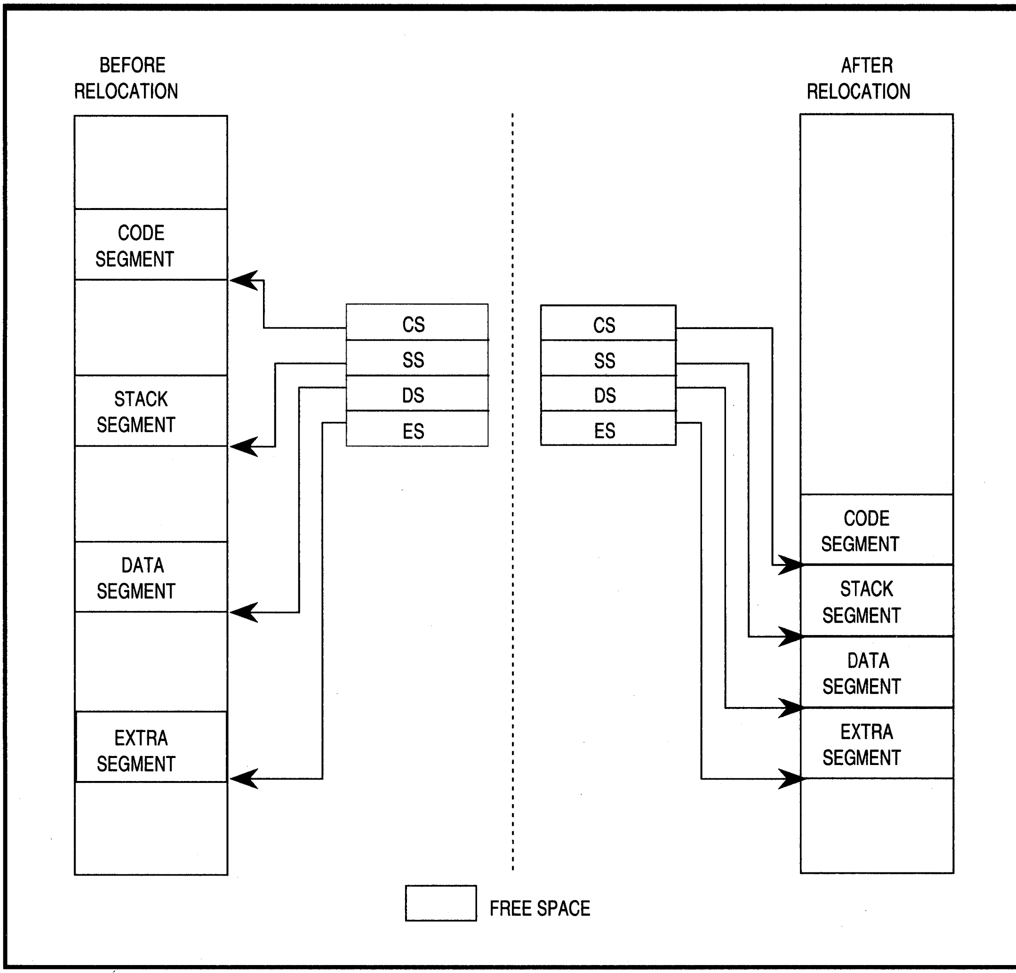


Figure 2.9. Dynamic Code Relocation

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 may have multiple stacks. A stack may 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. Only one stack is directly addressable at a time. 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 which 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 since the processor begins execution at 0FFFF0H.
- Locations 0F8H through 0FFH in I/O space are reserved for communication with other Intel hardware products and may not be used. On the 80C186 core, these addresses are used as I/O ports for the 80C187 numerics processor extension.

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 *80C186 Instruction Set Additions and Extensions*). The following sections provide a description of the instructions by category and a detailed discussion of the operand addressing modes.

Software for 80C186 core family systems does not need to 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 straight forward 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 may be written in assembly language and linked with high-level code.

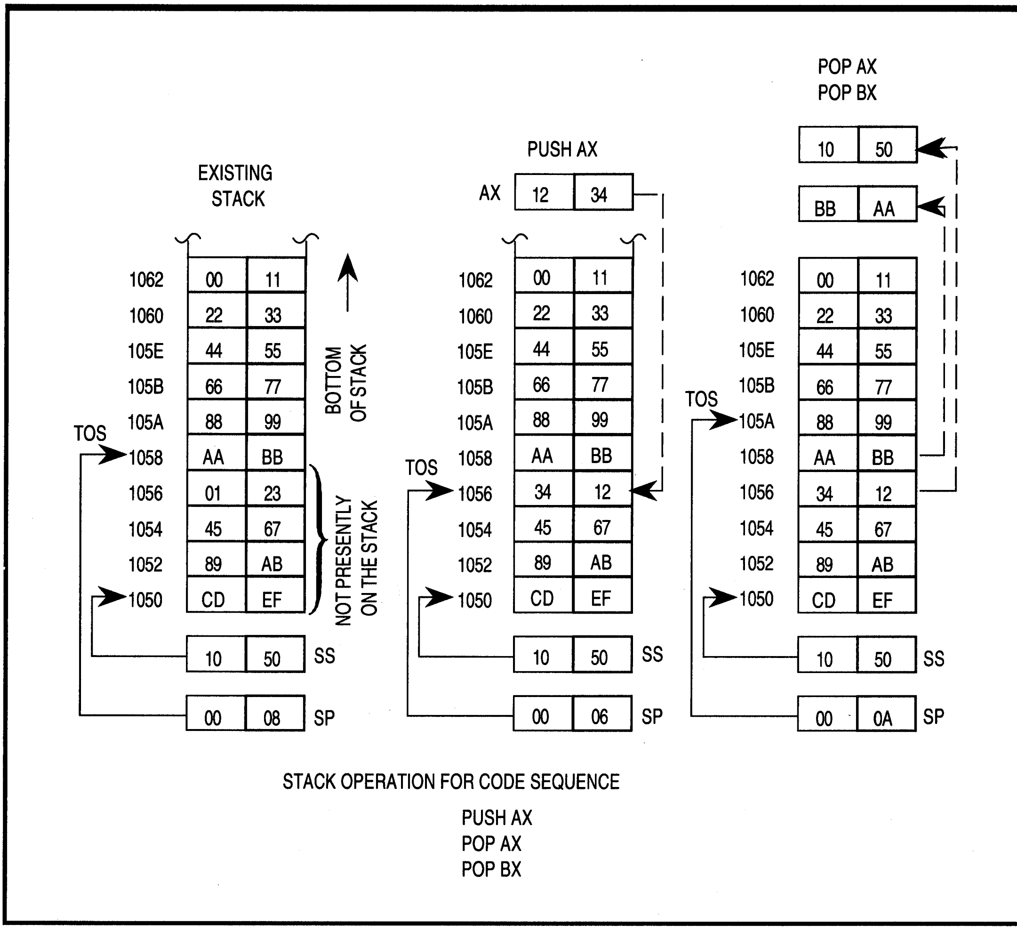


Figure 2.10. Stack Operation

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 may be specified interchangeably in most instructions. The exception to this is immediate values must serve as source operands and not destination operands. Memory variables may be added to, subtracted from, shifted, compared, etc., without moving them in and out of registers. This saves instructions, registers and execution time in assembly language programs. In high-level languages, where most variables are memory-based, compilers 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. To increment a register, however, does not require as much information, so the instruction can be shorter. The 80C186 Core family has eight one 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 registers and I/O ports. Table 2.3 lists the four types of data transfer instructions and their functions.

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.

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 onto stack
POPF	Pop flags off stack

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 doubleword

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

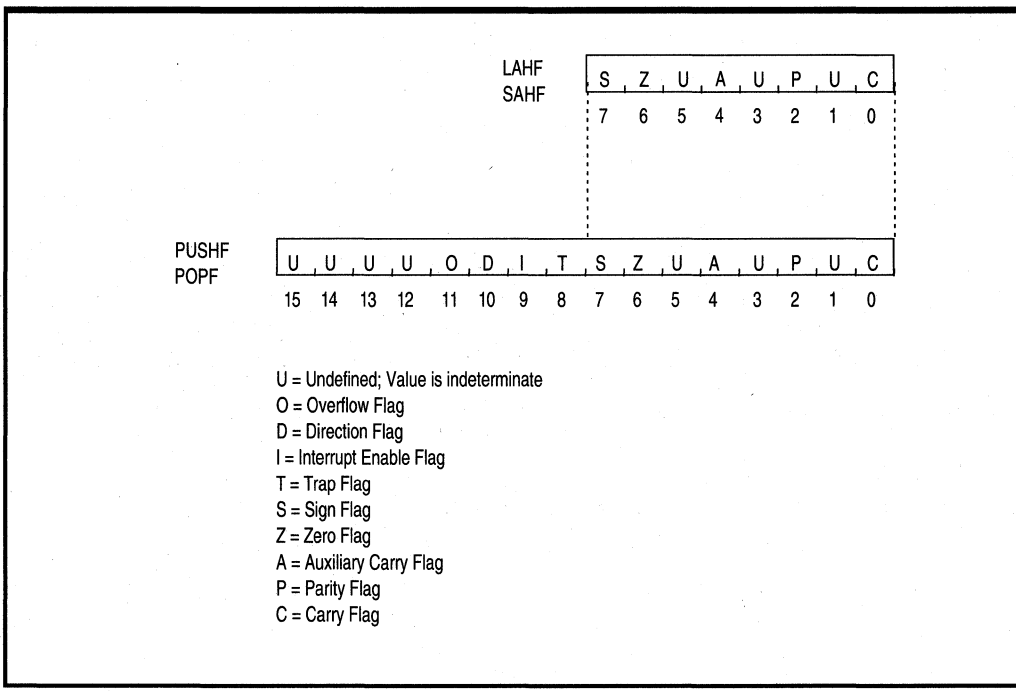


Figure 2.11. Flag Storage Format

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 may be 8 or 16 bits long. Decimal numbers are stored in bytes, two digits per byte for packed decimal and one digit per byte for unpacked decimal. The processor 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 arithmetic instruction's results and adjusts status flags accordingly.

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 these three groups of bit manipulation instructions with their functions.

Logical instructions include the Boolean operators NOT, AND, OR and exclusive OR (XOR). Logical instructions also include a TEST instruction that 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 arithmetically or logically. Up to 32 shifts may be performed, according to the value of the count operand coded in the instruction. The count may 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 may specify either an immediate value or the CL register. The carry flag may 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 may be manipulated with these instructions. Instructions are available to move, compare or scan for a value, as well as move string elements to and from the accumulator. Table 2.7 lists the string instructions. These basic operations may 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 with a software loop. The repetitions can be terminated by a variety of conditions. Repeated operations may be interrupted and resumed.

String instructions operate similarly in many respects (see Table 2.8). A string instruction may have a source operand, a destination operand or both. The hardware assumes that a source string resides in the current data segment. A segment prefix may 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/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, DI or both registers prior to 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 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.

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 may transfer control 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 will stack only the Instruction Pointer, while a FAR CALL will stack 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: 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 may be NEAR or FAR.

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

Table 2.7 String Instructions

REPE/ REPZ	Repeat while equal/zero
REPNE/ REPNZ	Repeat while not equal/not zero
MOVSB/ MOVSW	Move byte or 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

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	0 = auto-increment SI, DI 1 = auto-decrement SI, DI
ZF	Scan/compare terminator

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
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

Conditional transfer instructions are jumps that may or may not transfer control. This depends on the state of the CPU flags when the instruction is executed. These 18 instructions (see Table 2.10) each test a different combination of flags for a condition. 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 2.10. 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: "above" and "below" refer to the relationship of two unsigned values; "greater" and "less" refer to the relationship of two signed values.

Iteration control instructions can be used to regulate the repetition of software loops. These instructions use the CX register as a counter. Like the conditional transfers, the iteration control instructions are self-relative and may only transfer to targets that are within -128 to +127 bytes of themselves. They are SHORT transfers.

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

2.2.1.6. PROCESSOR CONTROL INSTRUCTIONS

Processor control instructions (see Table 2.11) allow programs to control various CPU functions. One group of instructions updates flags and another group is used primarily for synchronizing the microprocessor to external events. Another 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 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 may be contained in registers, the instruction itself, 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 may serve as source operands, destination operands or both.

Immediate operands are constant data contained in an instruction. Immediate data may be either 8 or 16 bits in length. Immediate operands are available directly from the instruction queue and can be accessed quickly. Like the register operand, no bus cycles need to be run to get an immediate operand. Immediate operands can only be 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 where 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 may be present in a given instruction. This allows a variety of memory addressing modes.

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 may be specified as the base register for an effective address calculation.

Similarly, either the SI or DI register may be specified as the index register. The displacement value is a constant. The contents of the base and index registers may 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 may 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. The programmer typically uses direct addressing to access scalar variables.

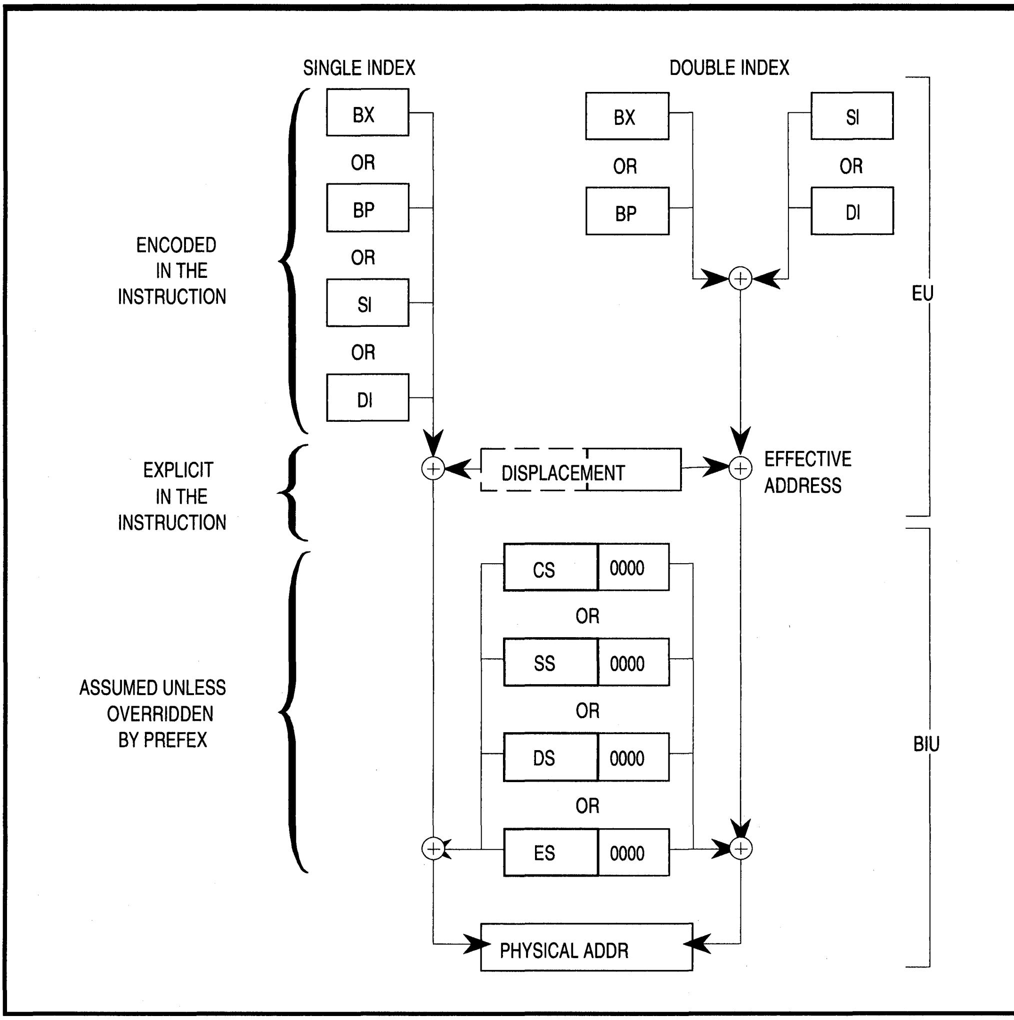


Figure 2.12. Memory Address Computation

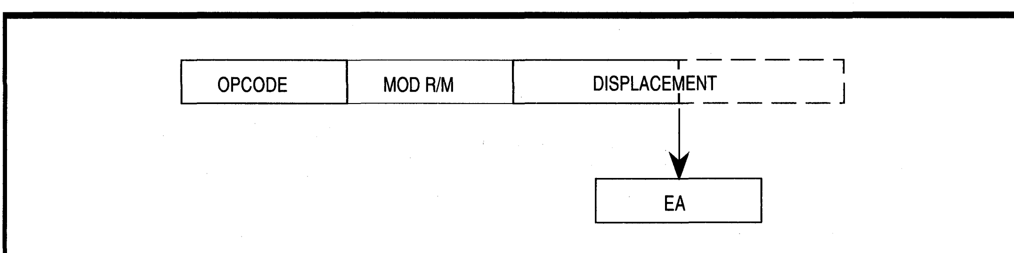


Figure 2.13. Direct Addressing

With register indirect addressing, the effective address of a memory operand may 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 may be used for register indirect addressing with the JMP or CALL instructions.

In based addressing (see Figure 2.15), the effective address is the sum of a displacement value and the contents of the BX or BP register. 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.

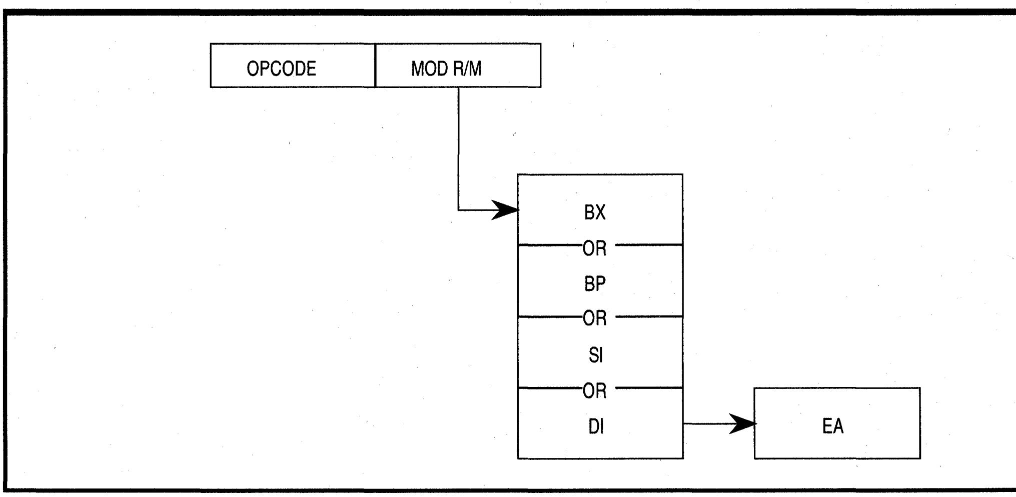


Figure 2.14. Register Indirect Addressing

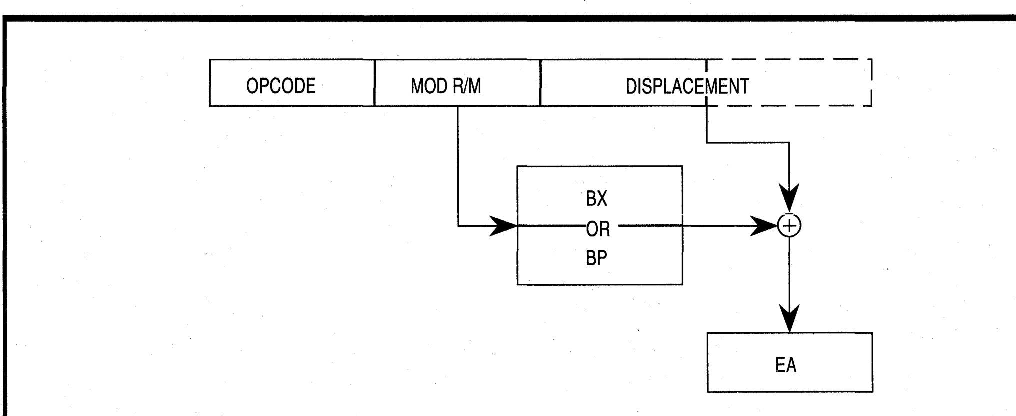


Figure 2.15. Based Addressing

Based addressing provides a simple way to address data structures which 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 displacement. Different copies of the same structure can be accessed by simply changing the base register.

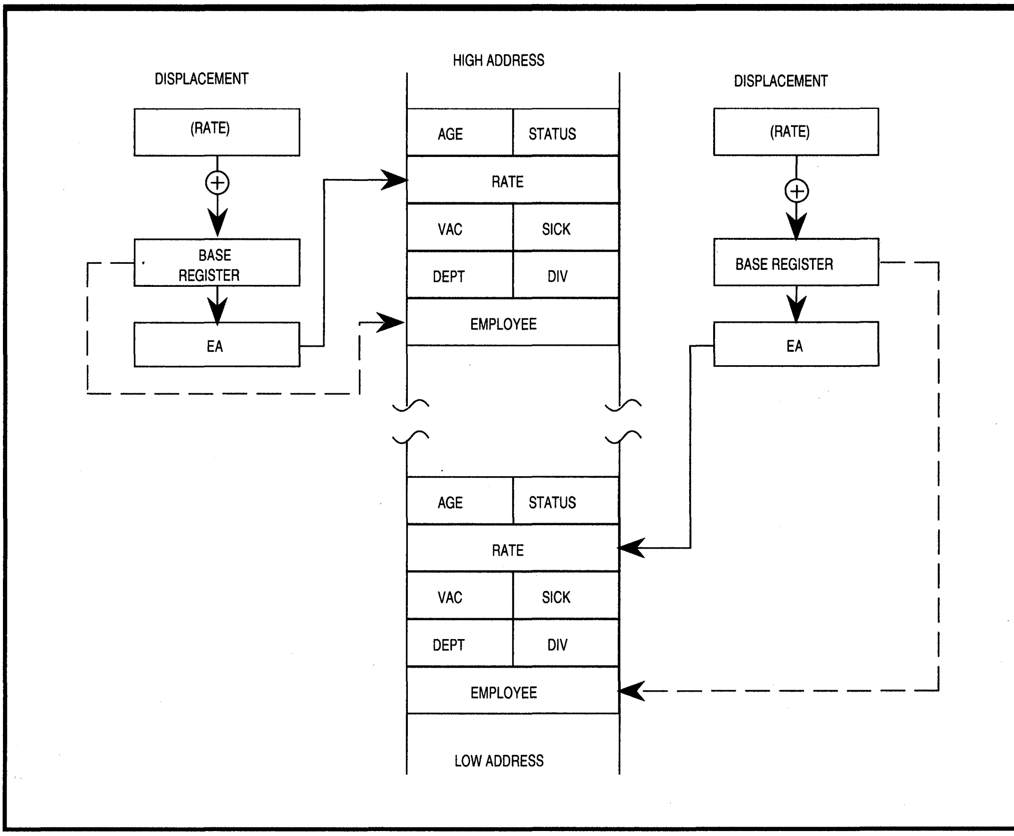


Figure 2.16. Accessing a Structure with Based Addressing

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 may select any element.

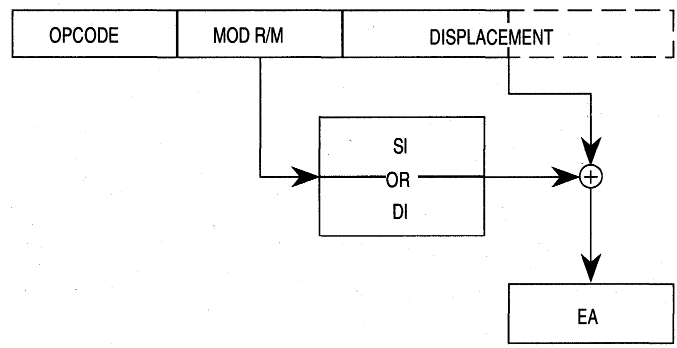


Figure 2.17. Indexed Addressing

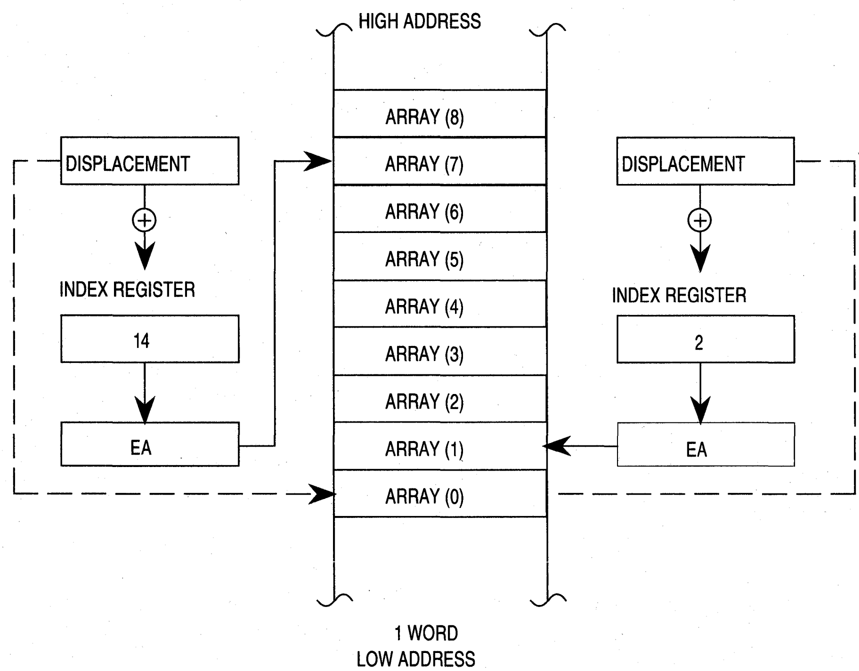


Figure 2.18. Accessing an Array with Indexed Addressing

Based index addressing generates an effective address which 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.

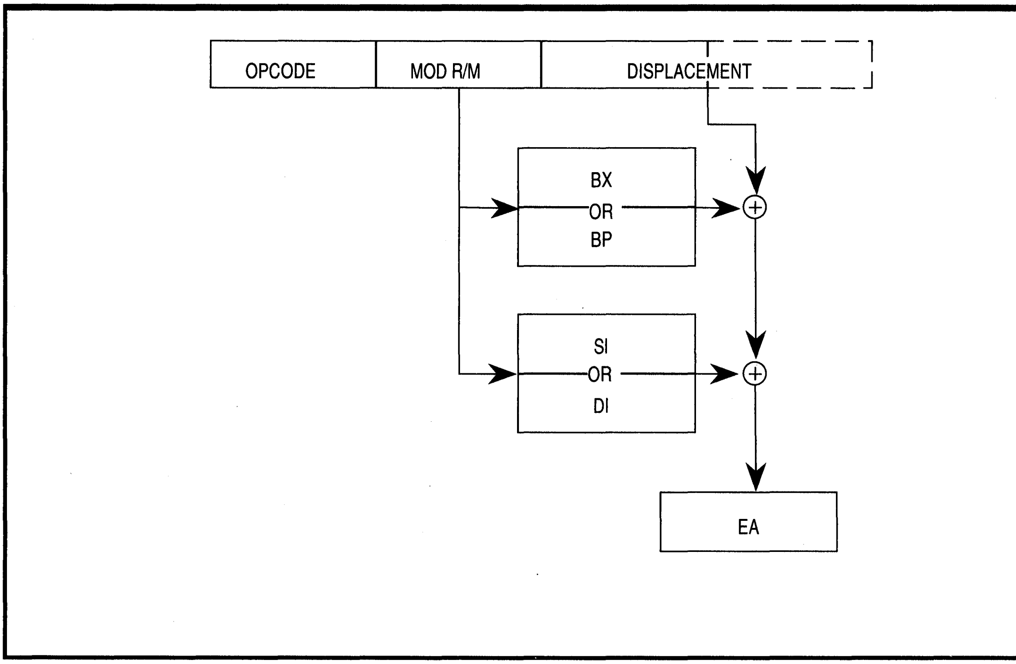


Figure 2.19. Based Index Addressing

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. 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 one megabyte address space.

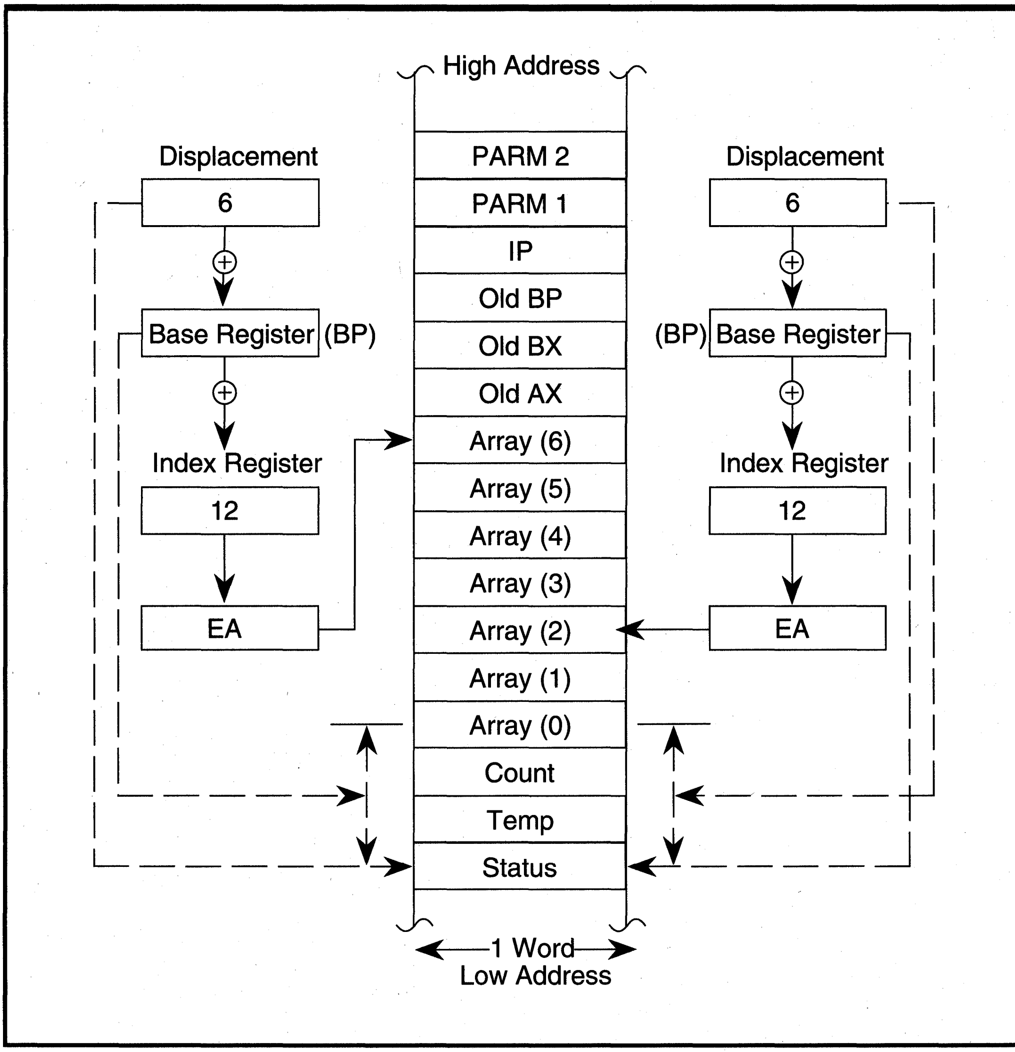


Figure 2.20. Accessing a Stacked Array with Based Index Addressing

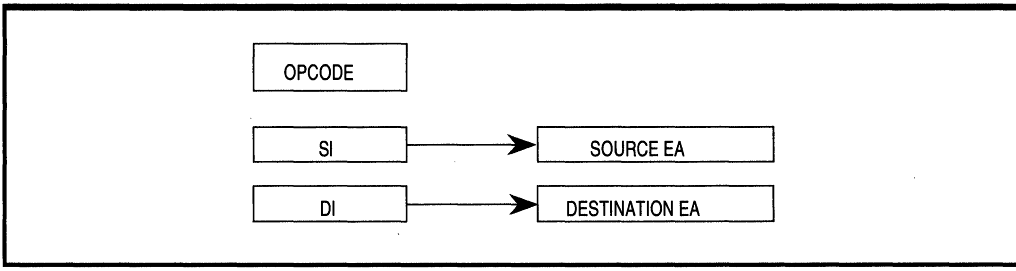


Figure 2.21. String Operand

2.2.2.3. I/O PORT ADDRESSING

Any memory operand addressing modes may 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). The port number is an 8-bit immediate operand for direct addressing. 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. By adjusting the contents of the DX register, one instruction can access any port in the I/O space. A group of adjacent ports can be accessed using a simple software loop that adjusts the value of the DX register.

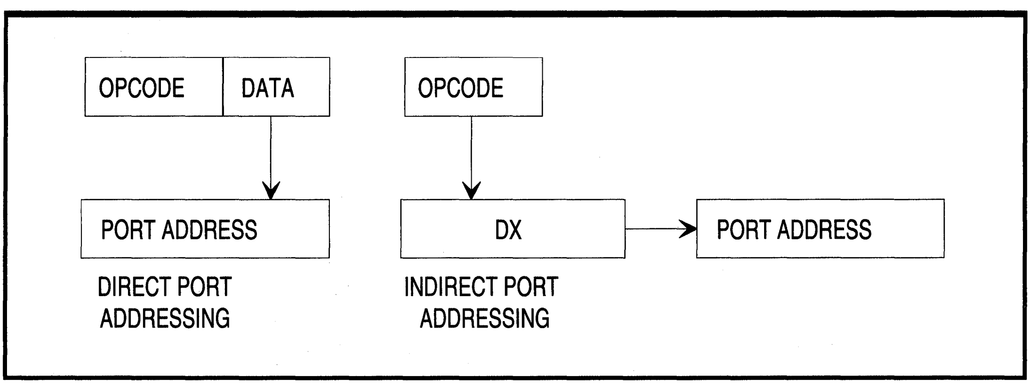


Figure 2.22. I/O Port Addressing

2.2.2.4. DATA TYPES USED IN THE 80C186 MODULAR CORE FAMILY

The 80C186 Modular Core family supports the following data types:

- Integer - A signed 8- or 16-bit binary numeric value. All operations assume a 2's complement representation. Signed 32- and 64-bit integers are directly supported with the addition of an 80C187 Numerics Processor Extension to an 80C186 Modular Core system. The 80C188 Modular Core does not support the 80C187.
- Ordinal - An unsigned 8- or 16-bit binary numeric value.
- Pointer - A 16- or 32-bit quantity, composed of a 16-bit offset component or a 16-bit segment base component in addition to a 16-bit offset component.
- String - A contiguous sequence of bytes or words. A string may contain from one to 64 Kbytes.
- ASCII - A byte representation of alphanumeric and control characters using the ASCII standard.
- BCD - A byte (unpacked) representation of the decimal digits 0-9.
- Packed BCD - A byte (packed) representation of two decimal digits (0-9). One digit is stored in each nibble (4 bits) of the byte.
- 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.

In general, individual data elements must fit within defined segment limits. Figure 2.23 graphically represents the data types supported by the 80C186 Modular Core family.

2.3. INTERRUPTS AND EXCEPTION HANDLING

Interrupts and exceptions alter the 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 the same as 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 section covers only areas of interrupts and exceptions common to the 80C186 Modular Core Architecture. The Interrupt Control Unit is proliferation dependent and is covered in another section.

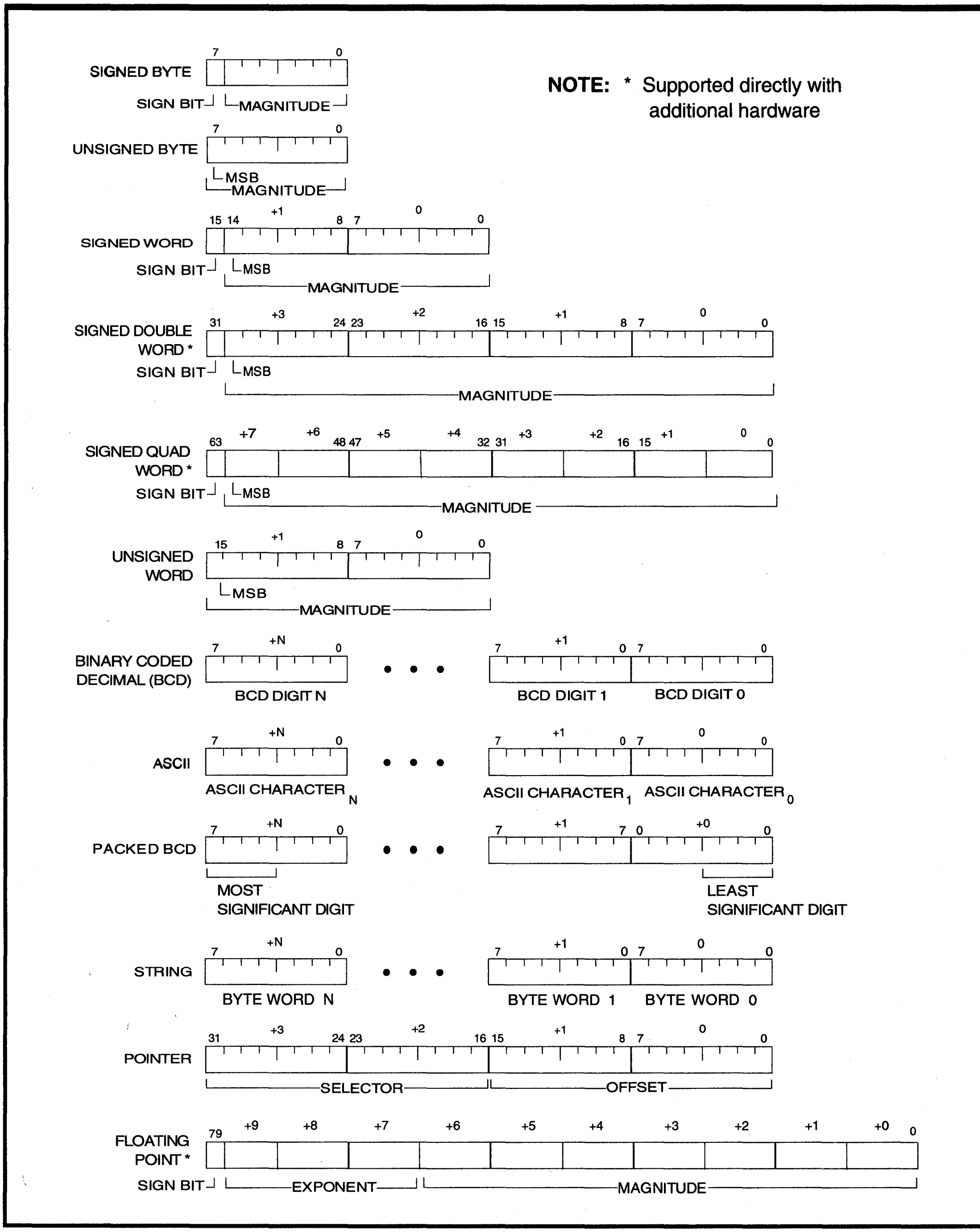


Figure 2.23. 80C186 Modular Core Family Supported Data Types

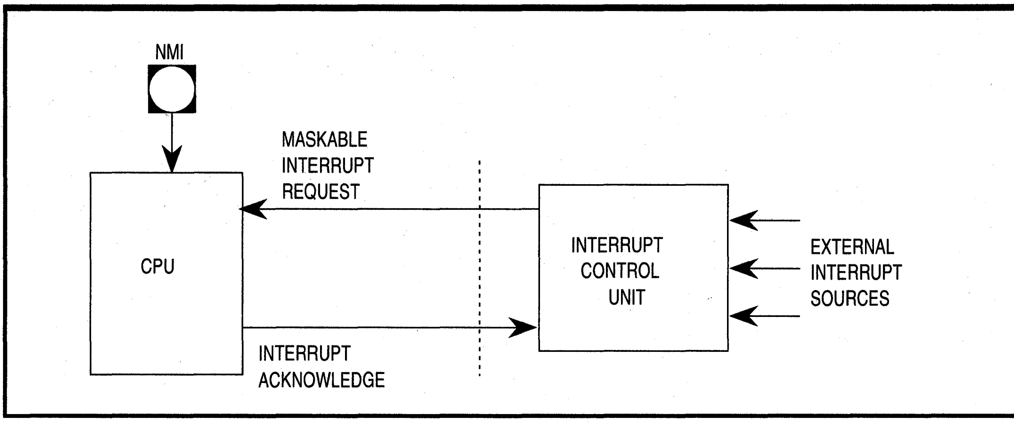


Figure 2.24. Interrupt Control Unit

2.3.1. INTERRUPT/EXCEPTION PROCESSING

The 80C186 Modular Core can service up to 256 different interrupts/exceptions. A 256 entry Interrupt Vector Table contains the pointers to interrupt service routines. Each interrupt/exception is given a type number, 0 through 255 corresponding to its position in the Interrupt Vector Table. See Figure 2.25. Each entry is 4 bytes long. An entry contains the Code Segment (CS) and Instruction Pointer (IP) of the first instruction in the interrupt service routine.

Interrupt types 0-31 are reserved for Intel and should not be used by an application program.

When an interrupt is acknowledged, a common sequence of events occur allowing the processor to execute the interrupt service routine (See Figure 2.26).

1. The processor saves a partial machine status by pushing the Program Status Word onto the stack.
2. The Trap Flag bit and Interrupt Enable bit are then cleared in the Program 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.

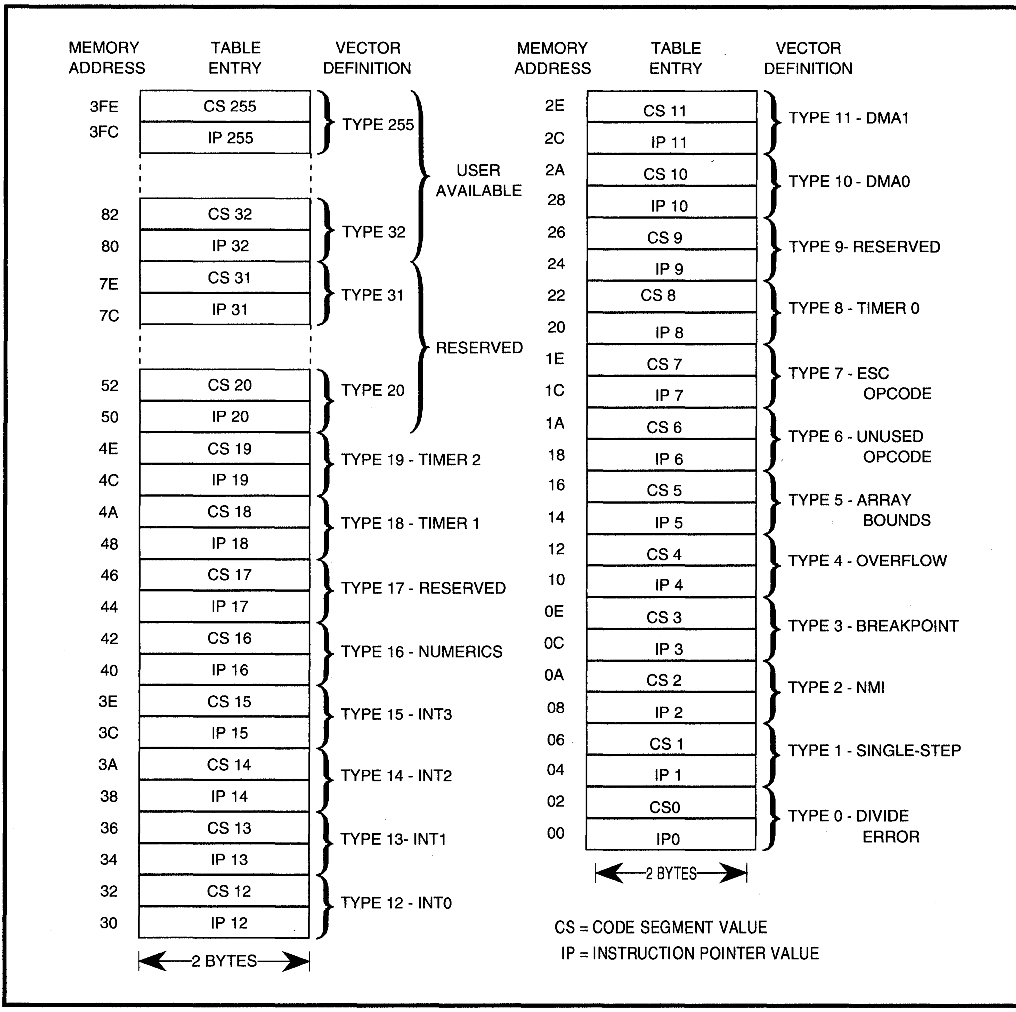


Figure 2.25. Interrupt Vector Table

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 or their contents will be lost. To allow nesting of maskable interrupts, the programmer must set the Interrupt Enable bit in the Program 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. An IRET instruction:

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

The CPU now executes from where it was before the interrupt/exception occurred.

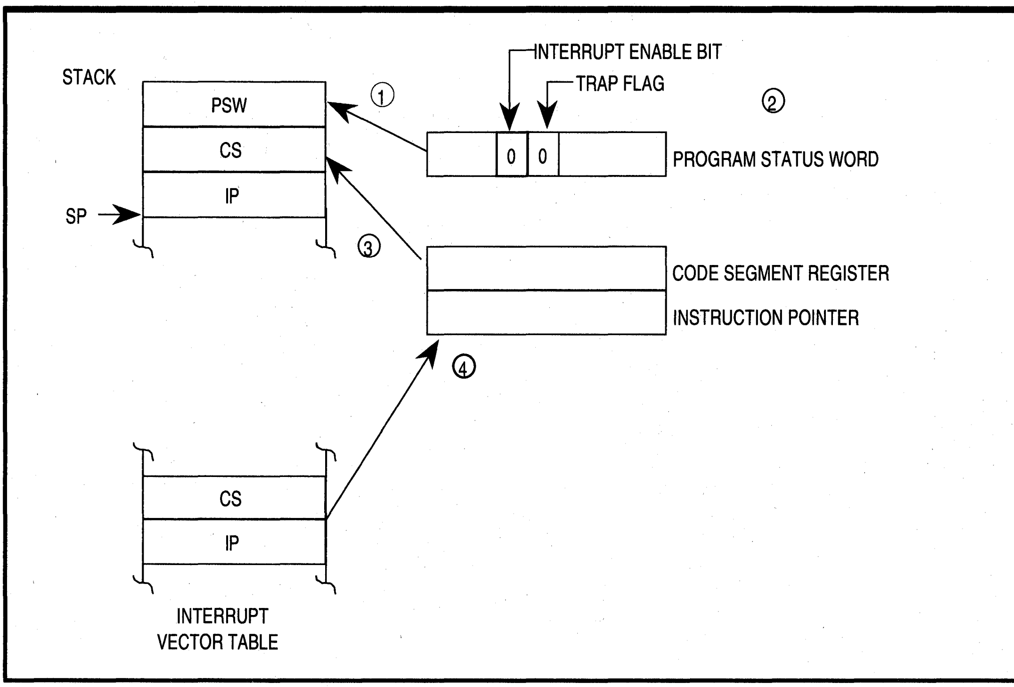


Figure 2.26. Interrupt Sequence

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 Section 2.3.4 on interrupt latency). 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 Program 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 and is described in a different section.

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. This depends on when they are detected and if the instruction which 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 way, the instruction can be restarted. A trap is detected and serviced immediately after the instruction which 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 Program Status Word. This allows programs to execute one instruction at a time. Interrupts will not be generated after prefix instructions (e.g. REP), instructions which modify segment registers (e.g. POP DS) or 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:

This is a single byte version of the INT instruction. The breakpoint interrupt 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 Program Status Word and the INT0 instruction is executed. Interrupt on Overflow is a common way to conditionally handle arithmetic overflows.

Array Bounds Check - Type 5:

If the array index is outside the array bounds during execution of the BOUND instruction (see ***80C186 Instruction Set Additions and Extensions***), an array bounds trap occurs.

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, the escape opcode fault is enabled by setting the Escape Trap (ET) bit in the Relocation Register (see *Peripheral Control Block*). When a floating point instruction is executed with the Escape Trap bit set, the Escape Opcode Fault exception occurs. The Escape Opcode service routine then 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. The 80C186XL will generate the Escape Opcode Fault regardless of the state of the Escape Trap bit unless it is in Numerics Mode.

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 only checks the $\overline{\text{ERROR}}$ pin 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 which 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 parameter n corresponds to the specific interrupt type to be executed. The interrupt type can be any number between 0 and 255. If the parameter n corresponds to an interrupt type associated with a hardware interrupt (NMI, Timers), the vectors will be fetched and the routine executed, but the corresponding bits in the Interrupt Status register **will not be 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 only recognizes interrupts 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 Interrupt Enable bit must be set for the CPU to recognize the Maskable Interrupt.
2. The CPU will not recognize interrupts during HOLD.
3. Once communication is completely established with an 80C187, the CPU will not recognize interrupts until the numerics instruction is finished.

The CPU can only recognize interrupts 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 will delay 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

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. The response time for the maskable interrupt is covered in the Interrupt Control Unit section.

Figure 2.27 shows the sequence of events which dictate interrupt response time for the interrupts which 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.

		Clocks
	IDLE	5
	READ IP	4
	IDLE	5
	READ CS	4
	IDLE	4
	PUSH FLAGS	4
	IDLE	3
	PUSH CS	4
	PUSH IP	4
	IDLE	5
FIRST INSTRUTION FETCH FROM INTERRUPT ROUTINE	------------------------------------>	
		Total 42

Figure 2.27. Interrupt Response Factors

2.3.5. INTERRUPT AND EXCEPTION PRIORITY

Interrupts can only be recognized 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 the exceptions must be the single step. Single step is a special case which will be discussed later. By 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 will be taken, 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.

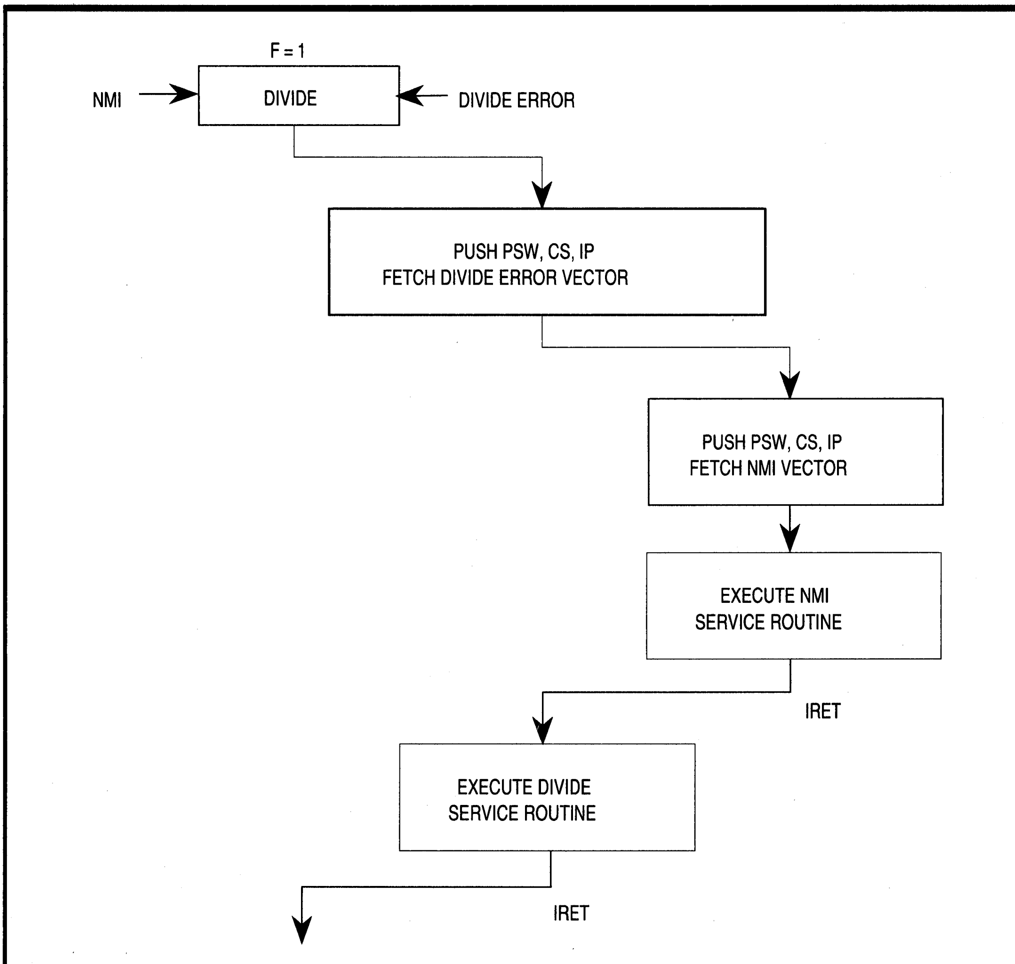


Figure 2.28. Simultaneous NMI and Exception

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, followed immediately by the single step vector. 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 setting the Trap Flag bit before 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.

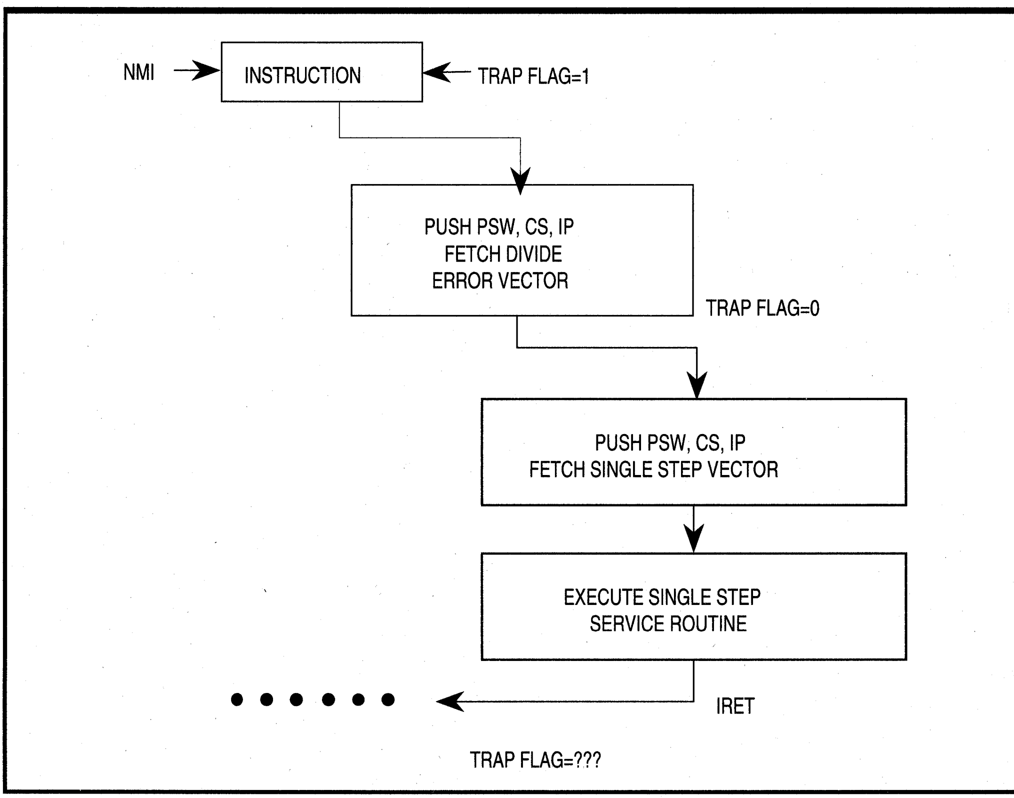


Figure 2.29. Simultaneous NMI and Single Step Interrupts

The most complicated case is when an NMI, maskable interrupt, 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: if the single step routine sets the Trap Flag bit before executing the IRET instruction, the NMI routine will also be single stepped.

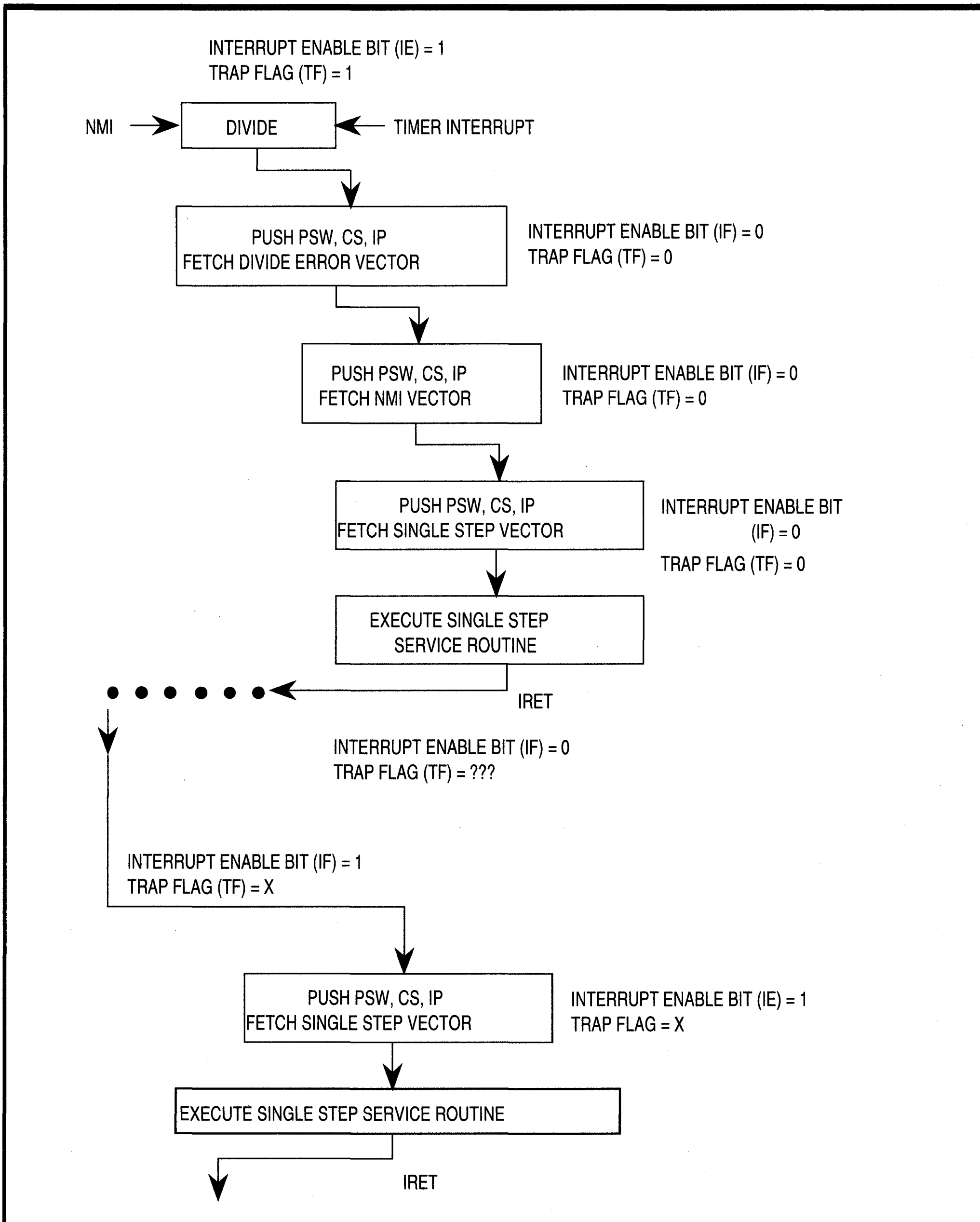


Figure 2.30. Simultaneous NMI, Single Step and Maskable Interrupt

Bus Interface Unit 3

CHAPTER 3
BUS INTERFACE UNIT

The Bus Interface Unit, abbreviated 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 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. Opon 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 Mbytes, while I/O space consists of 64 Kbytes. Any byte may contain an eight bit data element, and any two consecutive bytes may contain a sixteen 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 (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-A1 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.

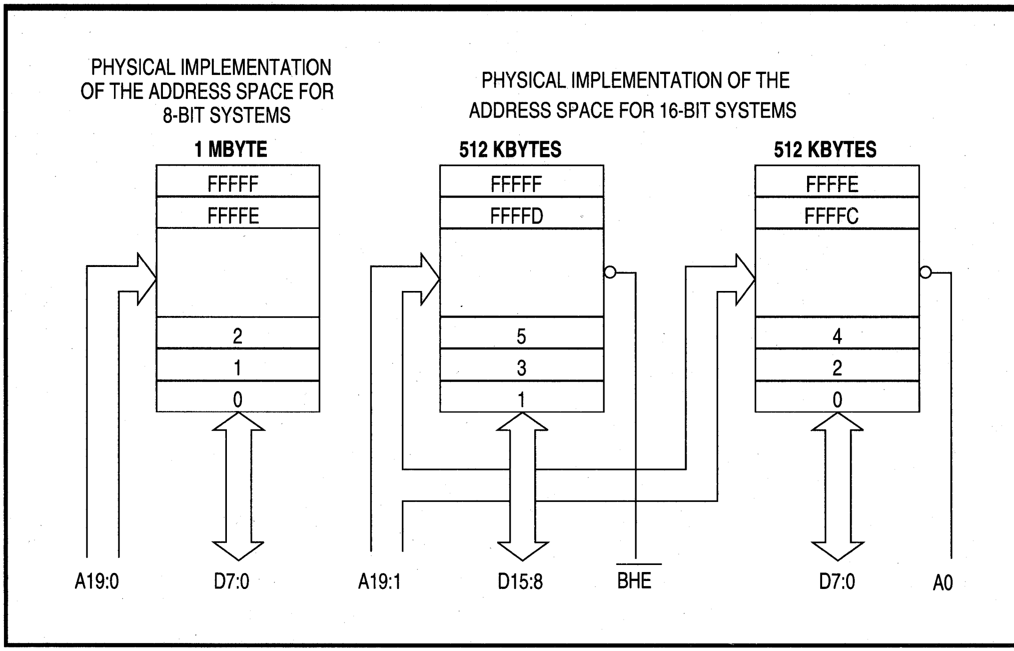


Figure 3.1. Physical Data Bus Models

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 $\overline{\text{BHE}}$ high disables the upper bank. The data value from the upper bank is ignored during a bus read cycle. $\overline{\text{BHE}}$ 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). $\overline{\text{BHE}}$ 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-A1 select the appropriate byte within each bank. A0 and $\overline{\text{BHE}}$ 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.

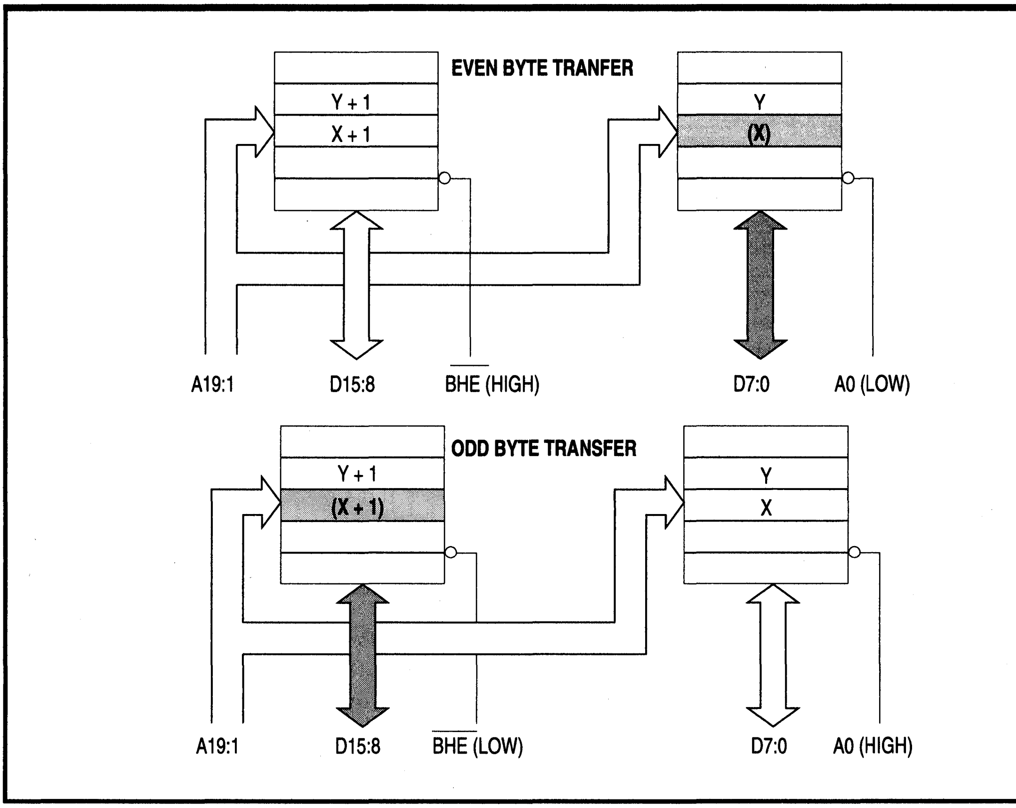


Figure 3.2. 16-Bit Data Bus Byte Transfers

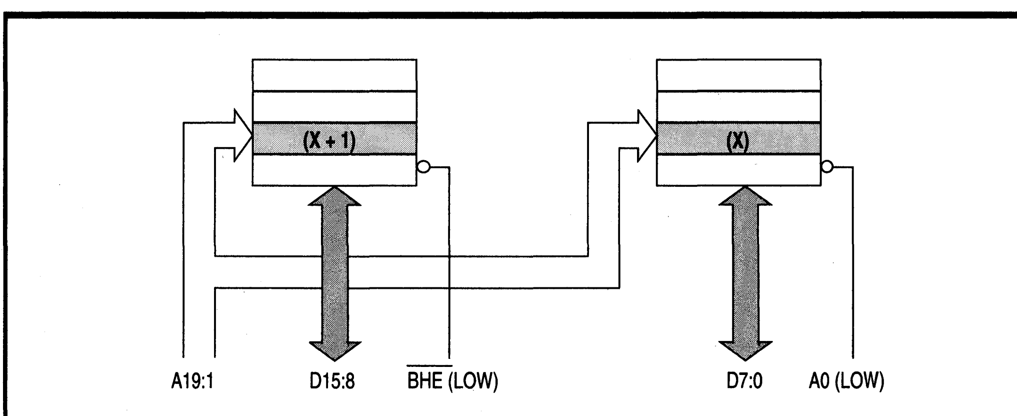


Figure 3.3. 16-Bit Data Bus Even Word Transfers

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, **both halves** of the bus can be driven and the BIU automatically accesses the correct half. The BIU drives both halves of the bus during a byte write operation. Information of 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.

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 Mbytes (see Figure 3.1). Address lines A19-A0 select a specific byte within the bank. Unlike a 16-bit bus, byte and word transfers (to even or odd addresses) all transfer data over the same 8-bit bus.

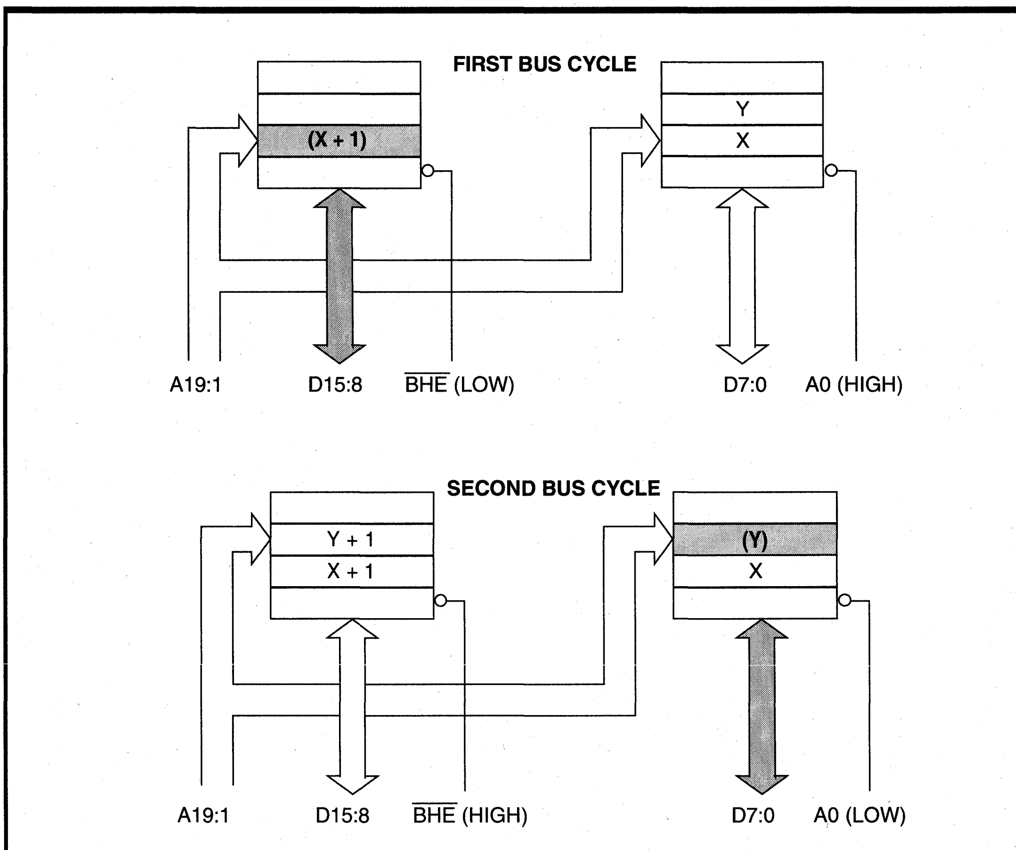


Figure 3.4. 16-Bit Data Bus Odd Word Transfers

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.

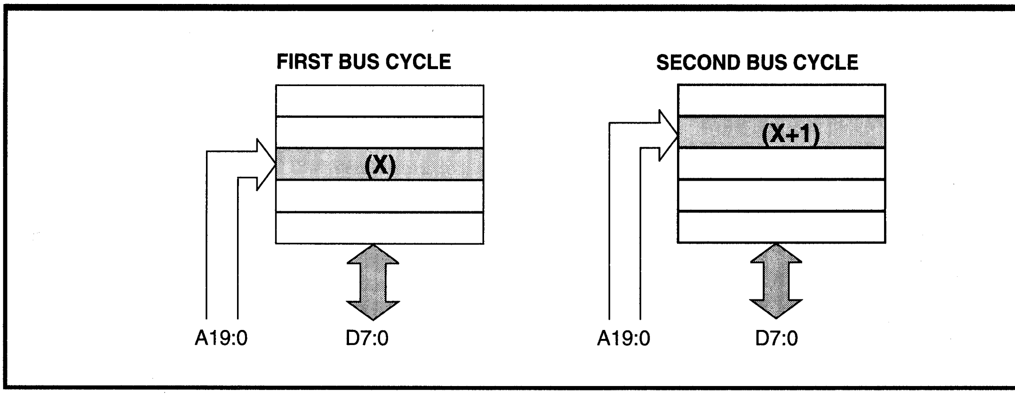


Figure 3.5. 8-Bit Data Bus Word Transfers

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 exchanged information during memory cycles. Memory mapped I/O allows the full power of the instruction set to be use 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 only be accessed indirectly through the DX register. I/O instructions always force address bits A19-A16 to zero.

Interrupt acknowledge, or INTA bus cycles access an I/O device intended to increase interrupt input capability. Valid address information is not generated as part of the INTA bus cycle, and data are 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 1FFH (1 Kbyte) hold 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 to or from any of the integrated units and 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 represent 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 3.8 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 symbol TW represents a wait state.

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

- Address/Status Phase
- Data Transfer Phase

The address/status phase starts just prior to T1 and continues through T1. The data transfer phase starts at T2 and continues through T4. Figure 3.9 illustrates the T-state relationship of the two phases.

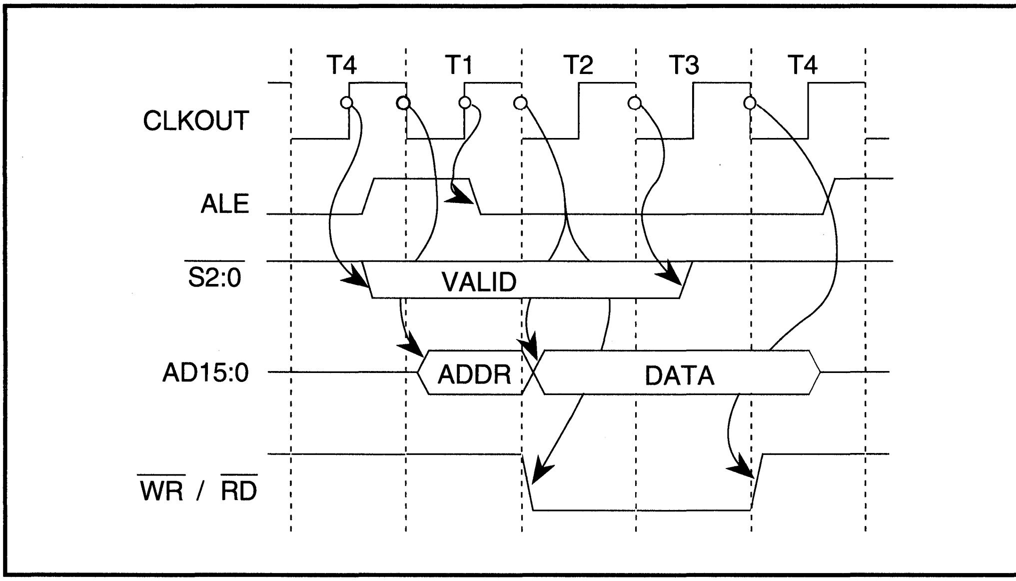


Figure 3.6. Typical Bus Cycle

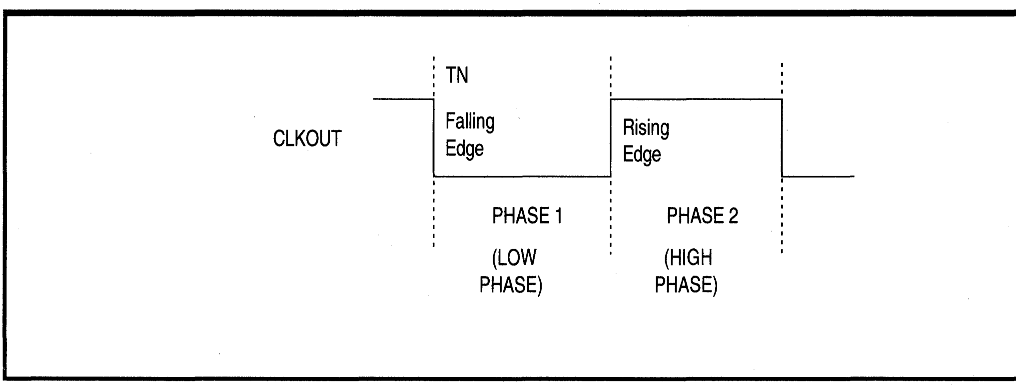


Figure 3.7. T-State Relation to CLKOUT

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 the ALE and $\overline{S2:0}$. These signals transition during phase 2 of the T-state just prior to T1. Referring back to Figure 3.8, T4 or TI precede T1 depending on the operation of the previous bus cycle.

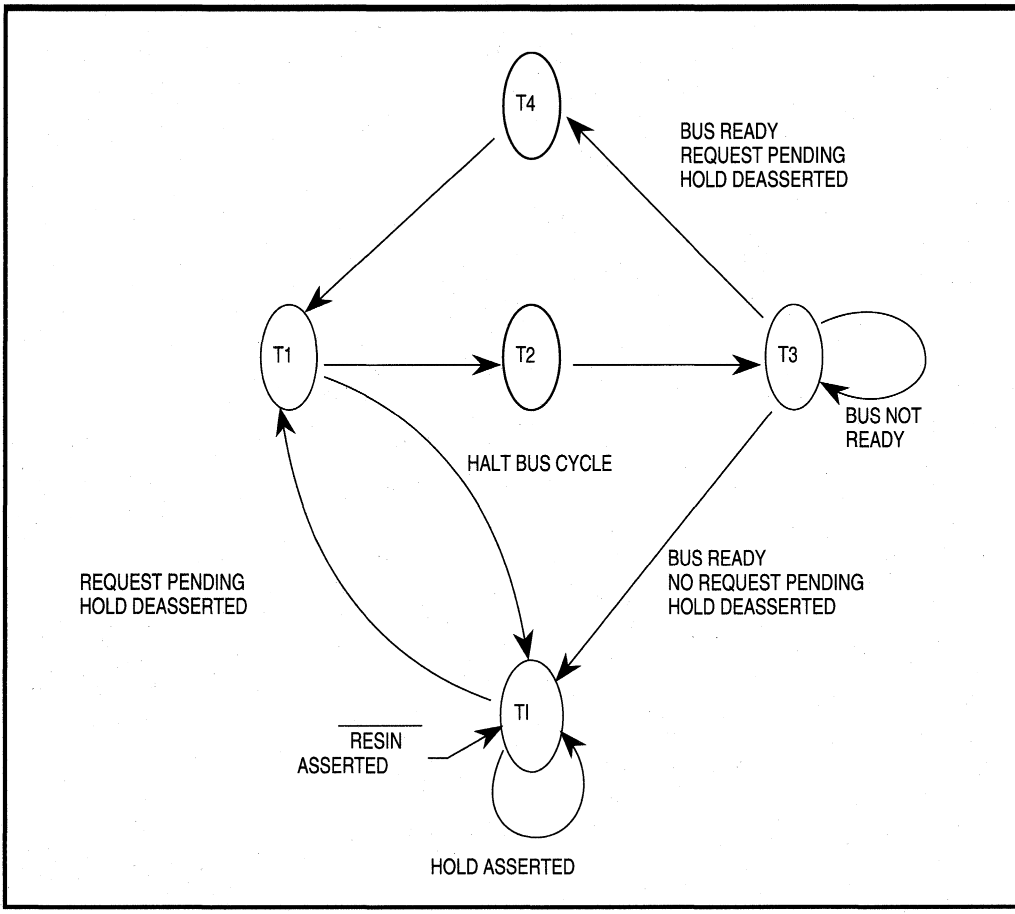


Figure 3.8. BIU State Diagram

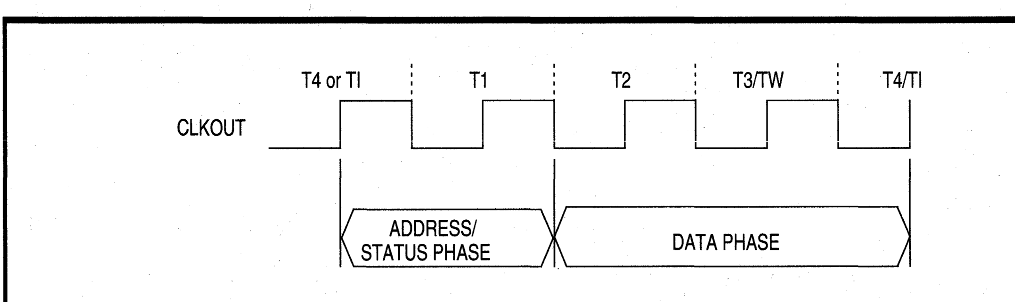
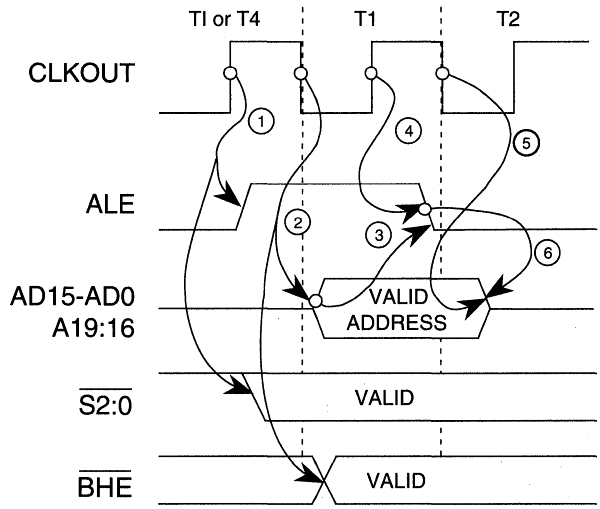


Figure 3.9. T-State and Bus Phases



NOTES:

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

Figure 3.10. Address/Status Signal Relationships

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 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 lists the possible bus cycle types. $\overline{S2:0}$ 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.

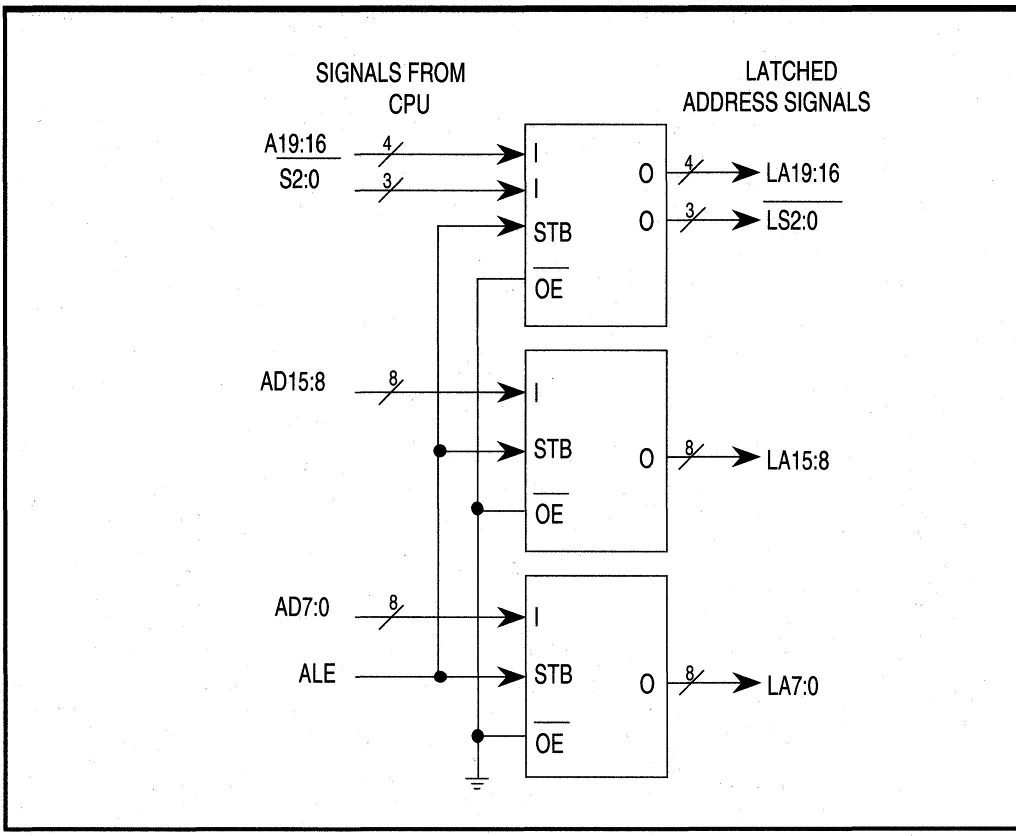
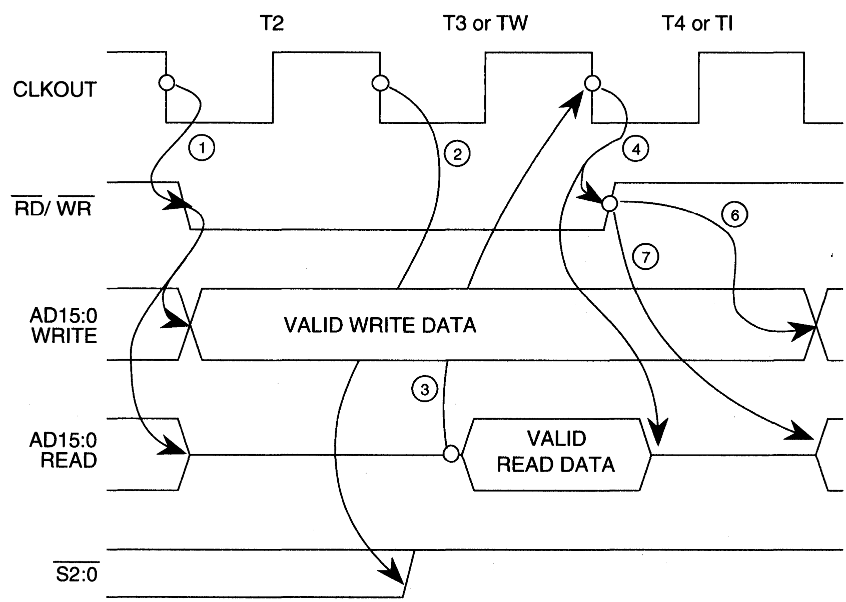


Figure 3.11. Demultiplexing Address Information

Table 3.1. Bus Cycle Types

STATUS BIT			OPERATION
$\overline{S2}$	$\overline{S1}$	$\overline{S0}$	
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)



NOTES:

1. $T_{CLRL/CLWL}$ T_{CLOV} : Clock low to valid $\overline{RD}$/ $\overline{WR}$ active; Write data valid
2. T_{CLSH} : Clock low to status inactive
3. T_{DVCL} : Data input valid to clock low
4. $T_{CLRH/CLWH}$: Clock valid to $\overline{RD}$/ $\overline{WR}$ inactive
5. T_{CLDX} : 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

Figure 3.12. Data Transfer Signal Relationships

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 can not 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.

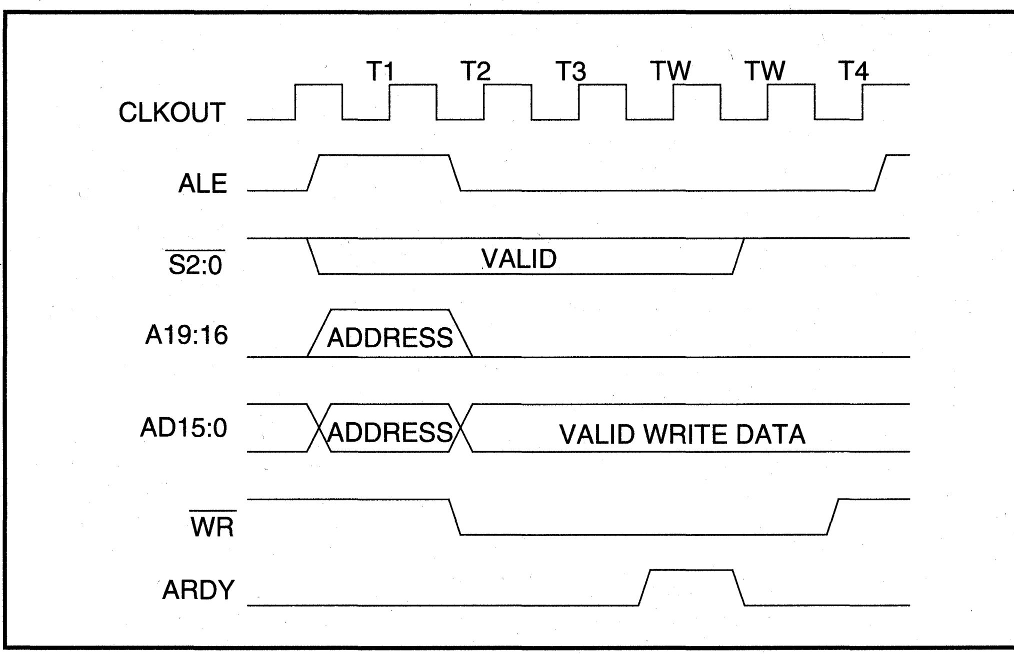


Figure 3.13. Typical Bus Cycle With Wait States

The bus ready pins and the Chip-Select Unit control bus cycle wait states. Only the bus ready pins are described in this section. Refer to Chapter 7 for a discussion of the Chip-Select Unit.

The SRDY and ARDY inputs control the wait state operation of the BIU. Figure 3.14 shows a simplified block diagram of the SRDY and ARDY inputs. Either ARDY or SRDY must be active to signal a bus ready condition. However, both ARDY and SRDY must be inactive to signal a bus not-ready condition. Depending on the size and characteristics of the system, ready implementation may take one of two approaches: normally not-ready or normally ready.

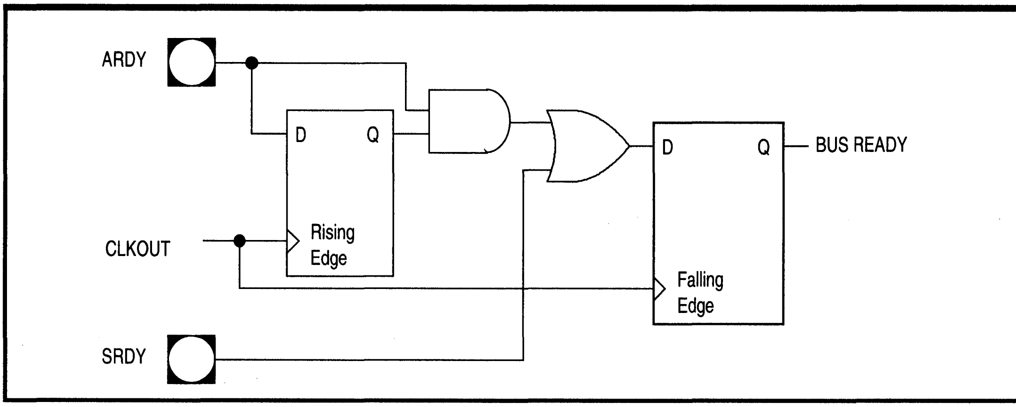


Figure 3.14. ARDY and SRDY Pin Block Diagram

The condition where ARDY and SRDY remain low at all times except to signal a ready condition defines a normally not-ready system. For any bus cycle, only the selected device drives either ready input high to allow the BIU to complete the bus cycle. The circuit shown in Figure 3.15 illustrates how 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 can not 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 ARDY low. After every rising edge of CLKOUT, INPUT1 and INPUT2 are shifted through the module and eventually drive ARDY 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. The same circuit works for SRDY, except no delay through the module results in no wait states.

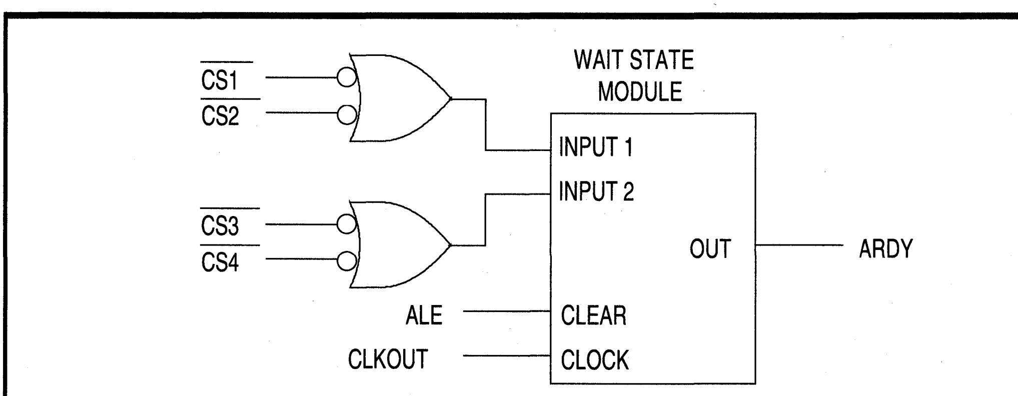


Figure 3.15. Generating a Normally Not-Ready Signal

A normally ready system drives ARDY or SRDY (or both) 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 (or inputs) 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 SRDY active (high). SRDY remains active if neither $\overline{CS1}$ or $\overline{CS2}$ go low. Should $\overline{CS1}$ or $\overline{CS2}$ go low, a series of zeros are shifted out every rising edge of CLKOUT causing SRDY to go inactive. At the end of the shift pattern SRDY 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 one wait state. Each additional zero shifted through the module generates one wait state. The same circuit works for ARDY, except shifting one "zero" through the module results in two wait states.

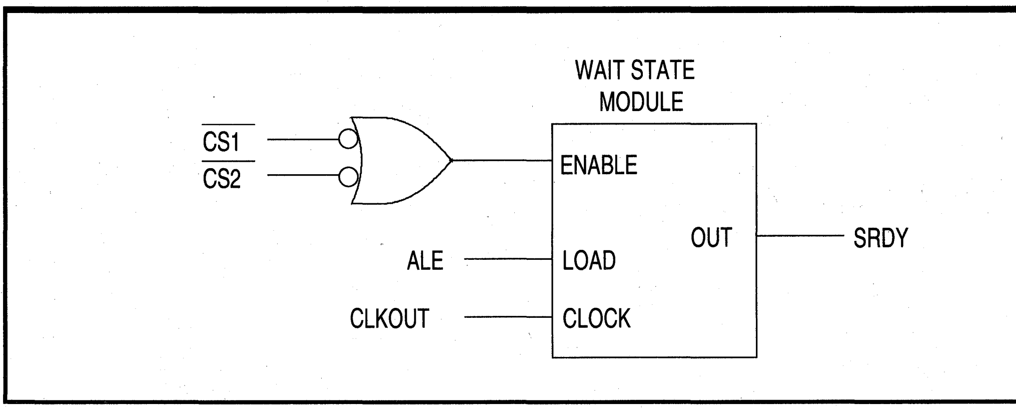


Figure 3.16. Generating a Normally Ready Signal

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 can not be kept full. Integrated peripheral performance suffers because the maximum bus bandwidth decreases.

3.4.3.1. ARDY INPUT

The ARDY input has two major timing concerns that can effect whether a normally ready or normally not-ready signal may be required. Referring to Figure 3.14, two latches capture the state of the ARDY input. The first latch captures ARDY on the phase 2 clock edge. The second latch captures ARDY **and** the result of the first latch on the phase 1 clock edge. The following equations define the requirements of the ARDY input (SRDY is inactive) to meet ready or not-ready bus conditions.

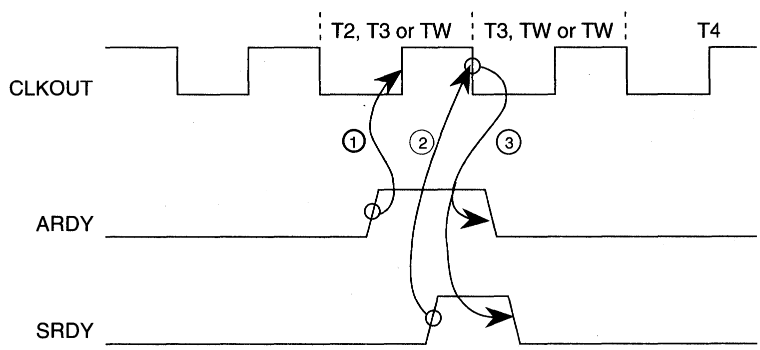
The bus is **ready** if:

1. ARDY is active prior to the phase 2 clock edge.
AND
2. ARDY is active prior to the phase 1 clock edge.

The bus is **not-ready** if:

1. ARDY is inactive prior to the phase 2 clock edge.
OR
2. ARDY 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 can not, then a normally ready system is required to run no wait states. Figure 3.17 illustrates the timing necessary to prevent wait states in a normally not-ready system. Figure 3.17 also illustrates how to terminate a bus cycle with wait states in a normally not-ready system.



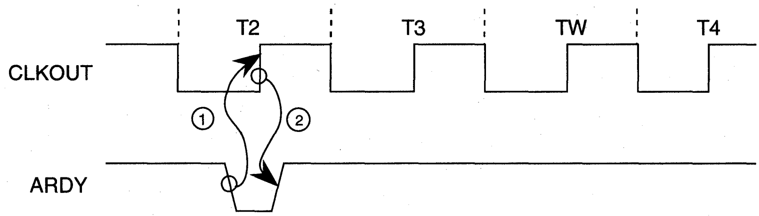
In a Normally-Not-Ready system, wait states are inserted until (1 or 2) and 3 are met.

1. T_{ARYCH} : ARDY active to clock high (assumes ARDY remains active until 3)
2. T_{SRYCL} : SRDY active to clock low
3. T_{CLARX} T_{CLSRY} : ARDY + SRDY hold from clock low

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

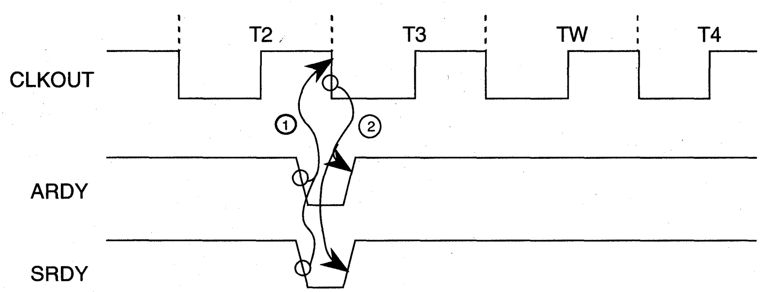
Figure 3.17. Normally Not-Ready System Timing

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 is required to run wait states if the not-ready condition can not 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 illustrates how to terminate a bus cycle with wait states in a normally ready system.



In a Normally-Not-Ready system, a wait state will be inserted when 1 & 2 are met. (Assumes SRDY is low.)

1. T_{ARYCH} : ARDY low to clock high
2. T_{ARYCHL} : Clock high to ARDY high (ARDY inactive hold time)



Alternatively, in a Normally-Ready system, a wait state will be inserted when 1 & 2 are met for both SRDY & ARDY.

1. T_{ARYLCL} T_{SRYCL} : ARDY/SRDY low to clock low
2. T_{CHARX} T_{CLSRY} : ARDY/SRDY low from clock low

⚠ Failure to meet ARDY & SRDY setup & hold time can cause a device failure (i.e., the bus hangs or operates inappropriately).

Figure 3.18. Normally Ready System Timing

3.4.3.2. SRDY INPUT

Referring to Figure 3.14, only one latch captures the state of the SRDY input. SRDY must be valid by phase 1 clock edge. The following equations define the requirements of the SRDY input (ARDY is inactive) to meet ready or not-ready bus conditions.

The bus is **ready** if:

1. SRDY is active prior to the phase 1 clock edge.

The bus is **not-ready** if:

1. SRDY is inactive prior to the phase 1 clock edge.

A normally not-ready system must generate a valid ready input at phase 1 of T3 to prevent wait states. If it can not, then a normally ready system is required to run no wait states. Figure 3.17 illustrates the timing necessary to prevent wait states in a normally not-ready system. Figure 3.17 also illustrates how to terminate a bus cycle with wait states in a normally not-ready system.

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 is required to run wait states if the not-ready condition can not 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 illustrates how to terminate a bus cycle with wait states in a normally ready system.

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), and may last an indefinite amount of time (depending on the instruction sequence). Conditions causing the BIU to become idle include:

- 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 does not drive any of the control strobes active in an idle state unless 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.

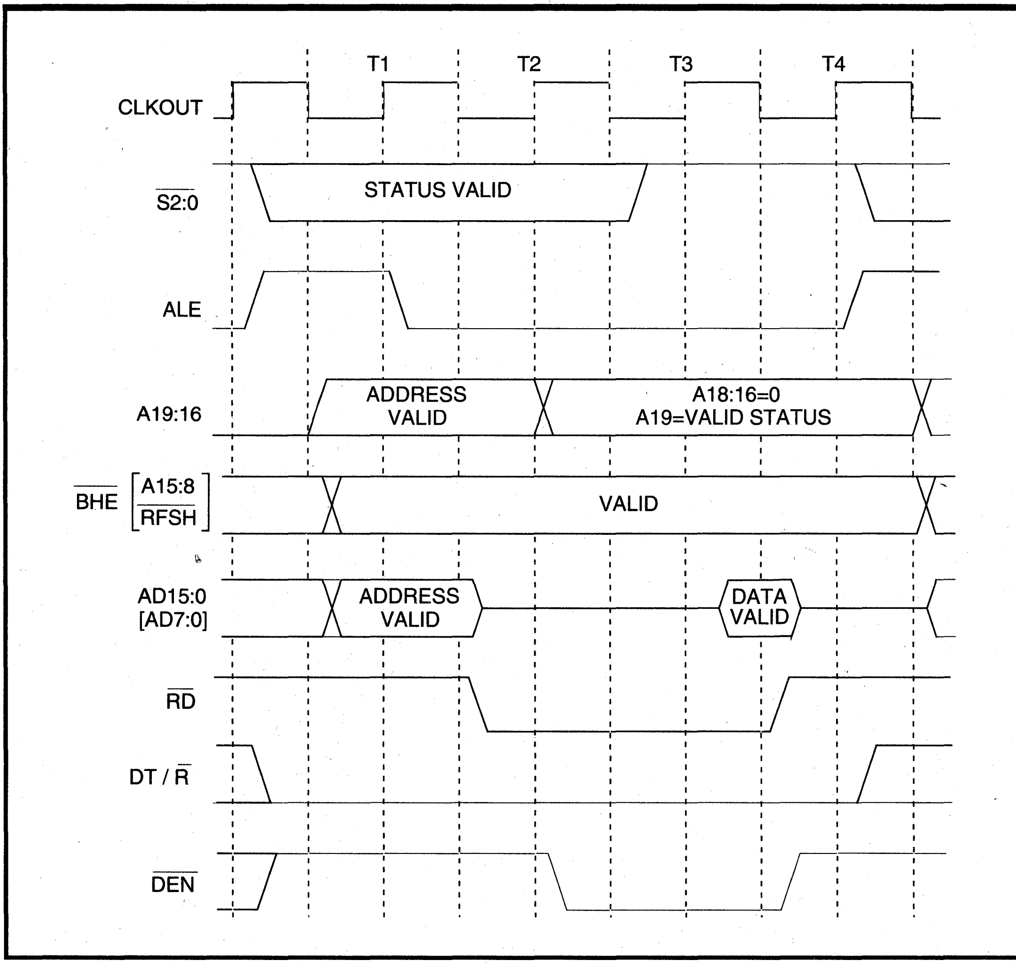


Figure 3.19. Typical Read Bus Cycle

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 no devices connect to an upper bus. Four parameters must be evaluated when determining the compatibility of a memory (or I/O) device. TADLTCH defines the delay through the address latch. Table 3.3 lists the four parameters.

TOE, TACC and TCE define the maximum data access requirements for the memory device. These device parameters must be **less** than the value calculated in the equation column. A equal to or greater than result indicates that wait states must be inserted into the bus cycle.

Table 3.2. Read Bus Cycle Types

STATUS BIT			BUS CYCLE TYPE
$\overline{S2}$	$\overline{S1}$	$\overline{S0}$	
0	0	1	Read I/O - Initiated by the Execution Unit for IN, OUT, INS, OUTS instructions or by the DMA Unit. A15:0 selects the desired I/O port. A19:16 drive to zero (see also DMA Unit).
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, the DMA Unit, or the Refresh Control Unit. A19:0 select the desired byte or word memory location

TDF determines the maximum time the memory device can float its outputs before the next bus cycle begins. A TDF 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.

TRHAX cannot be lengthened (other than slowing the clock rate). To resolve a buffer fight condition, chose a faster device or buffer the AD bus (see Section 3.6.1).

Table 3.3. Read Cycle Critical Timing Parameters

MEMORY DEVICE PARAMETER	DESCRIPTION	EQUATION
TOE	Output enable ($\overline{RD}$ low) to data valid	2TCLCL - TCLRL - TDVCL
TACC	Address valid to data valid	3TCLCL - TCLAV - TADLTCH - TDVCL
TCE	Chip enable ($\overline{UCS}$) to data valid	3TCLCL - TCLCSV - TDVCL
TDF	Output disable ($\overline{RD}$ high) to output float	TRHAV

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. The data value on the bus is ignored. $\overline{RFSH}$ has the same bus timing as $\overline{BHE}$.

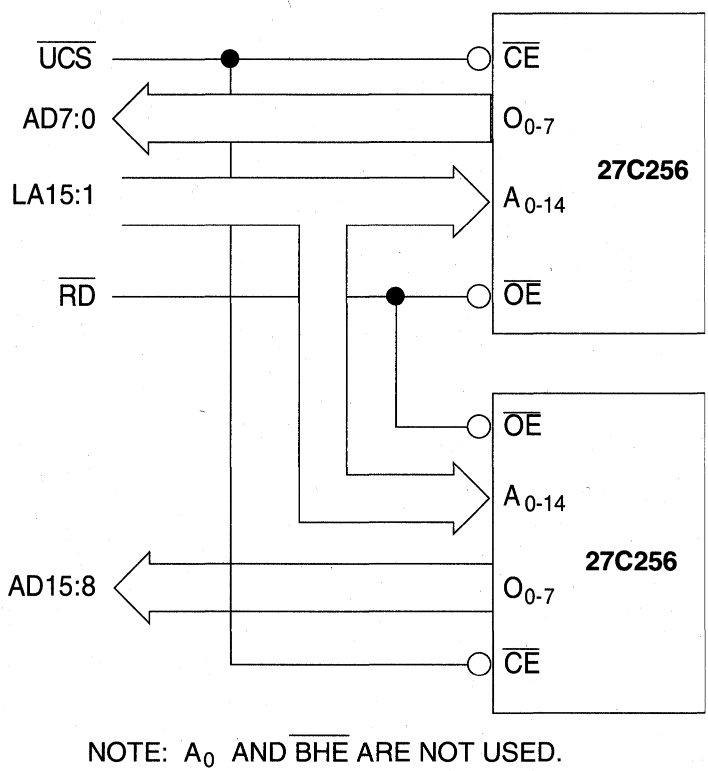


Figure 3.20. Read-Only Device Interface

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 3.3 lists the two types of write bus cycles.

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 3.4 lists some critical write bus cycle parameters.

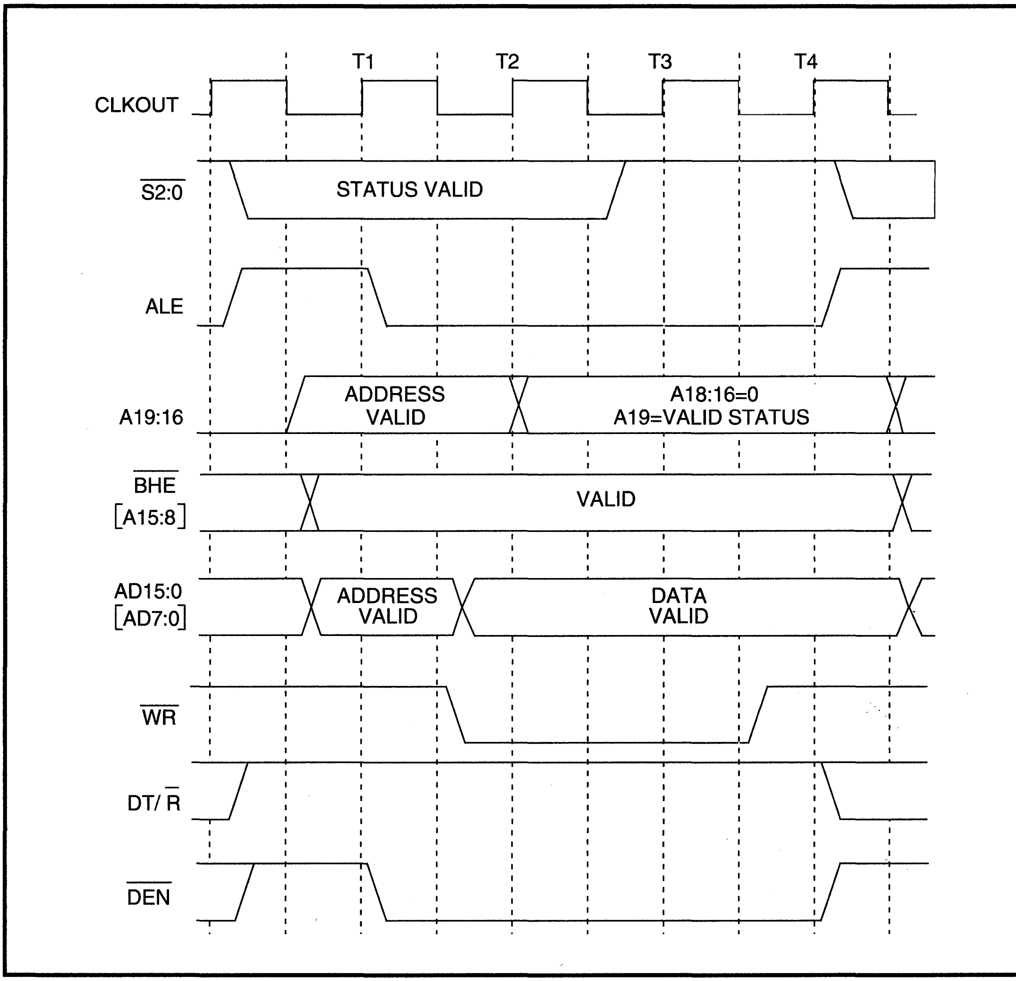


Figure 3.21. Typical Write Bus Cycle

Most memory and peripheral devices latch data on the rising edge of the write strobe. Address, chip-select and data must be valid (setup) prior to rising edge of $\overline{WR}$. TAW, TCW and TDW 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.

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

Table 3.4. Write Bus Cycle Types

STATUS BITS			BUS CYCLE TYPE
$\overline{S2}$	$\overline{S1}$	$\overline{S0}$	
0	1	0	Write I/O - Initiated by executing IN, OUT, INS, OUTS instructions or by the DMA Unit. A15:0 selects the desired I/O port. A19:16 are driven to zero (see also DMA Unit).
1	1	0	Write Memory - Initiated by any of the Byte/ Word memory instructions or the DMA Unit. A19:0 selects the desired byte or word memory location.

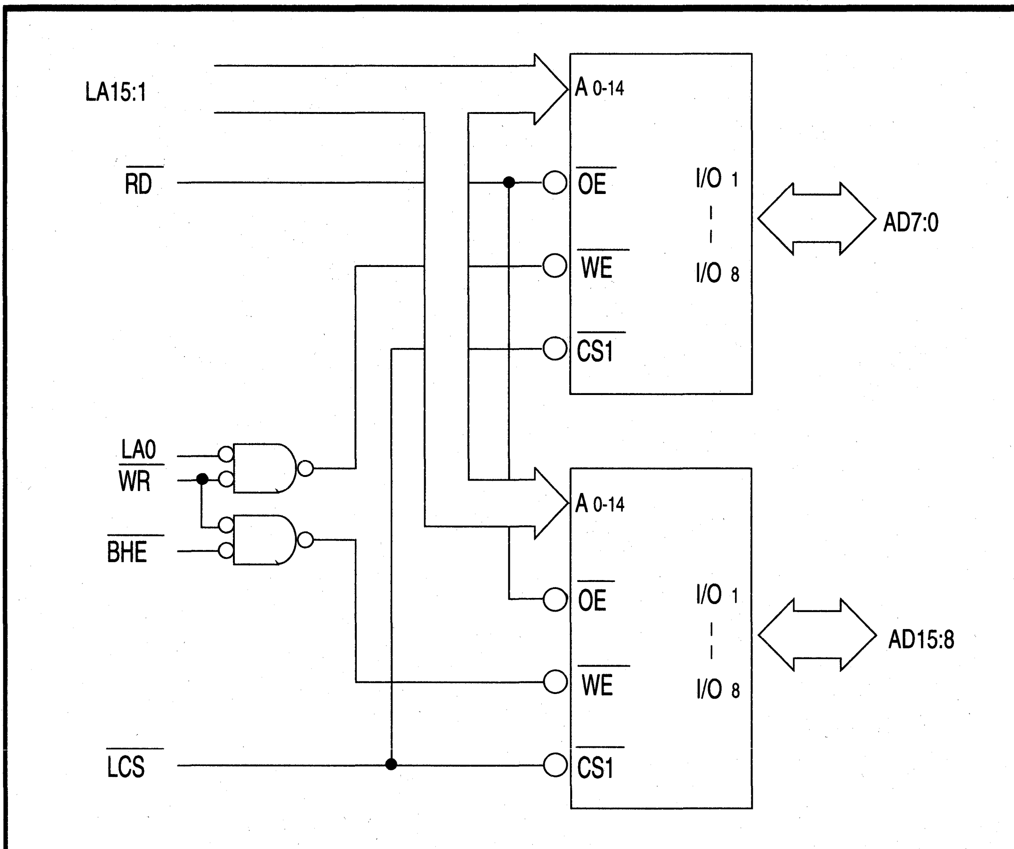


Figure 3.22. 16-Bit Bus Read/Write Device Interface

Table 3.5. Write Cycle Critical Timing Parameters

MEMORY DEVICE PARAMETER	DESCRIPTION	EQUATION
TWC	Write cycle time	4TCLCL
TAW	Address valid to end of write strobe ($\overline{\text{WR}}$ high)	3TCLCL - TADLTCH
TCW	Chip enable ($\overline{\text{LCS}}$) to end of write strobe ($\overline{\text{WR}}$ high)	3TCLCL
TWR	Write recover time	TWHLH
TDW	Data valid to write strobe ($\overline{\text{WR}}$ high)	2TCLCL
TDH	Data hold from write strobe ($\overline{\text{WR}}$ high)	TWHDX
TWP	Write pulse width	TWLWH

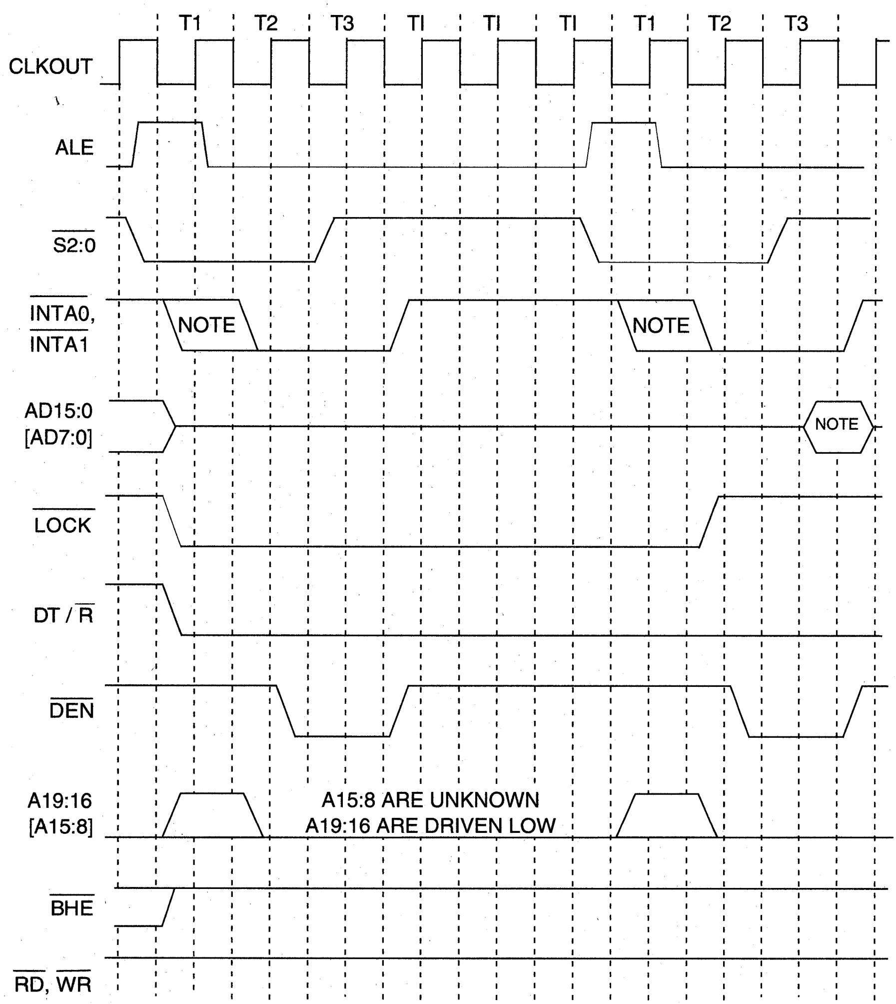
TWC and TWP define the minimum time (maximum frequency) a device can process write bus cycles. TWR determines the minimum time from the end of the current write cycle to the start of the next write cycle. All three parameters require calculated values be greater than device requirements. The calculated TWC and TWP values increase by inserting wait states. The calculated TWR value, however, can not be changed except 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. The BIU controls the bus cycles required to fetch vector information from the peripheral device, and then passes the information to the CPU. These bus cycles, collectively know as an INTA bus cycle, operate similarly to read bus cycles. However, instead of generating $\overline{\text{RD}}$ to enable the peripheral, the signal $\overline{\text{INTA}}$ is used. Figure 3.23 illustrates a typical Interrupt Acknowledge bus cycle.

An Interrupt Acknowledge bus cycle consists of two consecutive bus cycles. $\overline{\text{LOCK}}$ 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, the upper half of the bus floats.

Figure 3.25 shows a typical 82C59A interface example. Bus ready must be provided to terminate both bus cycles in the interrupt acknowledge sequence.



NOTE: Vector Type is read from AD7:0 only.
INTA occurs during T2 in Slave Mode

Figure 3.23. Interrupt Acknowledge Bus Cycle

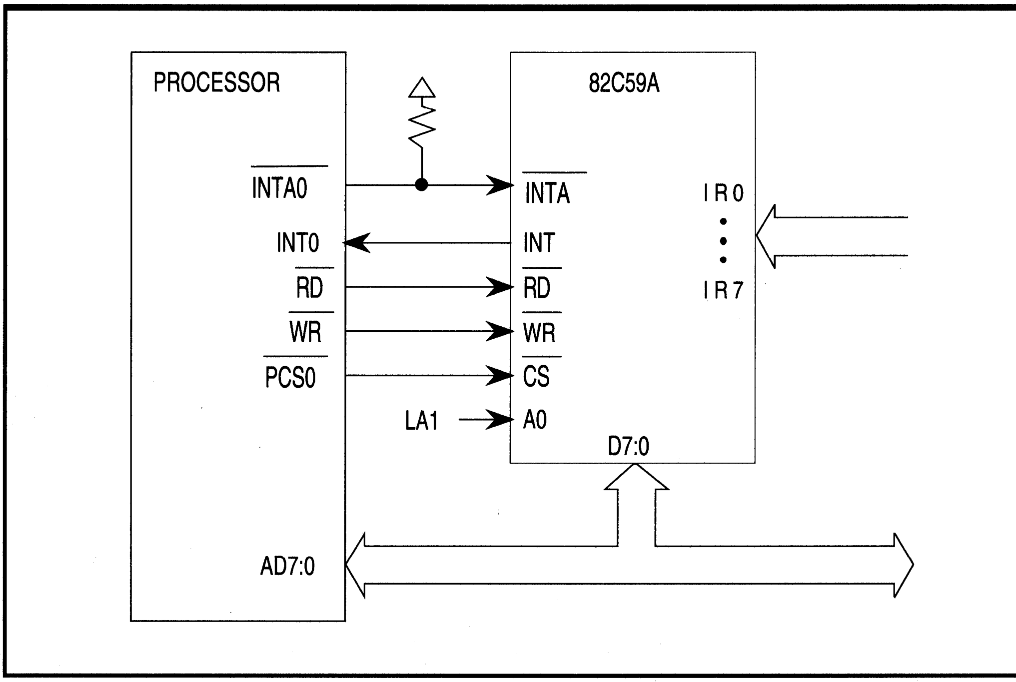


Figure 3.24 Typical 82C59A Interface

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 decode circuitry must be disabled for Interrupt Acknowledge bus cycles to prevent erroneous operation.

3.5.4. HALT BUS CYCLE

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

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

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

- An interrupt is generated
- A bus HOLD is generated
- A DMA request is generated
- A refresh request is generated

Figure 3.25 shows the operation of a HALT bus cycle. During T1, the AD bus either floats or drives depending on the next bus cycle to be executed by the BIU. Under most instruction sequences, the BIU floats the AD bus because the next operation would most likely be an instruction prefetch. However, the AD bus drives either data or address information during T1 if the HALT occurs just after a bus write operation. A19:16 continues to drive the previous bus cycle information under most instruction sequences (it drives the next prefetch address otherwise). The BIU always operates 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.

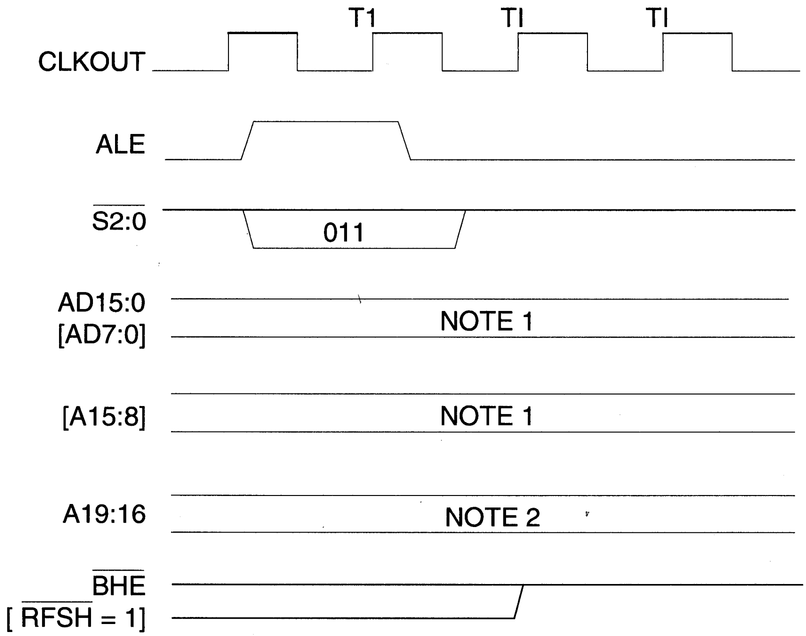
Table 3.6 lists the state of each pin after entering the HALT bus state.

Table 3.6. HALT Bus Cycle Pin States

PIN(S)	PIN STATE
AD15:0 (AD7:0 for 8-bit)	Float
A15:8 (8-bit)	Drive Address
A19:16	Drive 8H or Zero
$\overline{\text{BHE}}$ (16-bit)	Drive Last Value
$\overline{\text{RD}}$, $\overline{\text{WR}}$, $\overline{\text{DEN}}$, DT/$\overline{\text{R}}$, $\overline{\text{RFSH}}$ (8-bit), $\overline{\text{S2:0}}$	Drive One

3.5.5. TEMPORARILY EXITING THE HALT BUS STATE

A DMA request, refresh request or bus hold request cause the BIU to temporarily exit the HALT bus state. 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, 3.27, and 3.28 illustrate how the BIU temporarily exits and then returns to the HALT bus state.



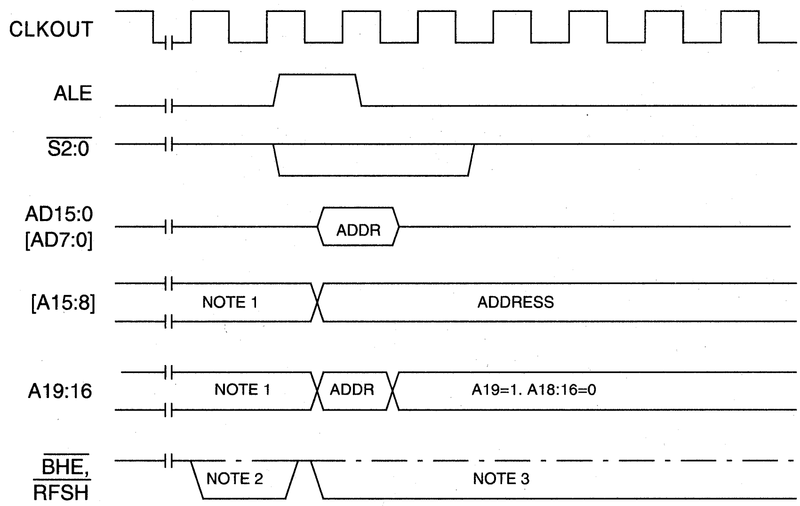
NOTES:

1. The AD15:0 [AD7:0] bus can be floating, driving a previous write data value, or driving the next instruction prefetch address value. For an 8-bit device, A15:8 either drives the previous bus address value or the next instruction prefetch address value.
2. The A19:16 bus either drives zero (all low) or 8H (all low except A19/S6, which can be high if the previous bus cycle was a DMA or refresh operation).

Figure 3.25. HALT Bus Cycle

3.5.6. EXITING HALT

Any NMI or non-masked INTx interrupt forces the BIU to exit the HALT bus state. The first bus operations to occur after exiting HALT are read cycles to reload the CS:IP registers. Figure 3.29 shows how the HALT bus state is exited when and NMI or INTx occurs.



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

Figure 3.26. Returning to HALT After a Refresh Bus Cycle

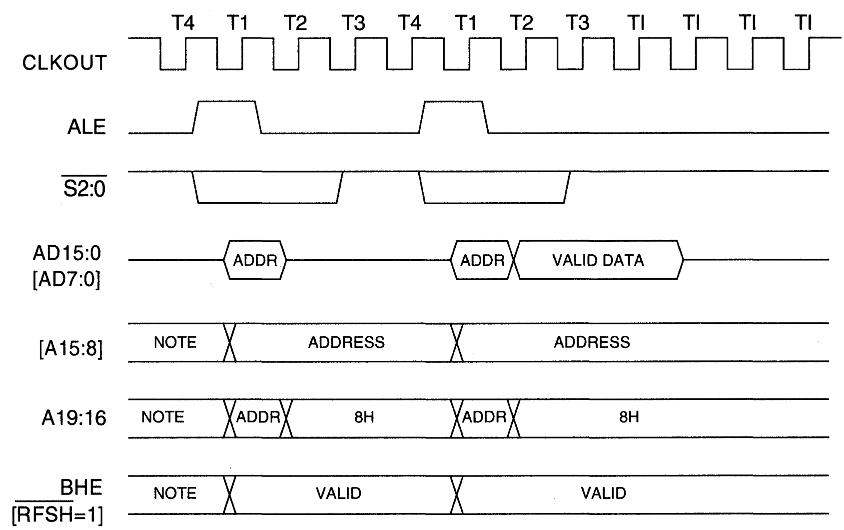
3.6. SYSTEM DESIGN ALTERNATIVES

Most system designs do not require any additional signaling requirements 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, $\overline{\text{DEN}}$ and DT/$\overline{\text{R}}$, to control bidirectional buffers or transceivers. The timing relationship of $\overline{\text{DEN}}$ and DT/$\overline{\text{R}}$ is shown in Figure 3.30. Conditions requiring transceivers include:

- The capacitive load on the AD bus gets too large
- The current load on the AD bus exceeds device specifications
- Additional VOL and VOH drive is required
- A memory or I/O device can not float its outputs in time to prevent a buffer fight



NOTE: Drives previous bus cycle value

Figure 3.27. Returning to HALT After a DMA Bus Cycle

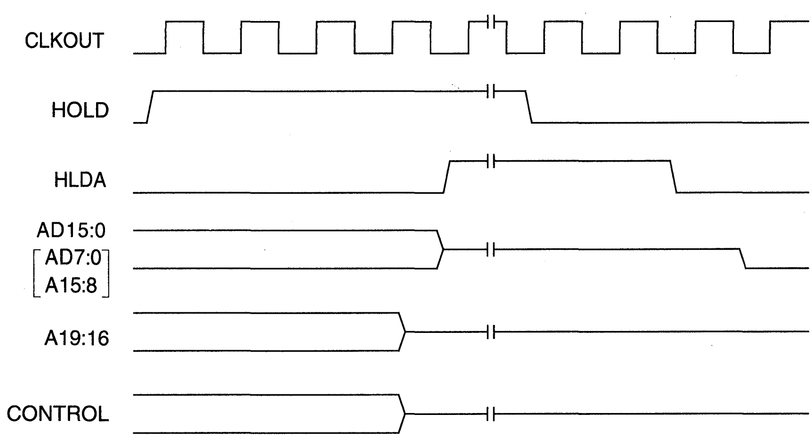
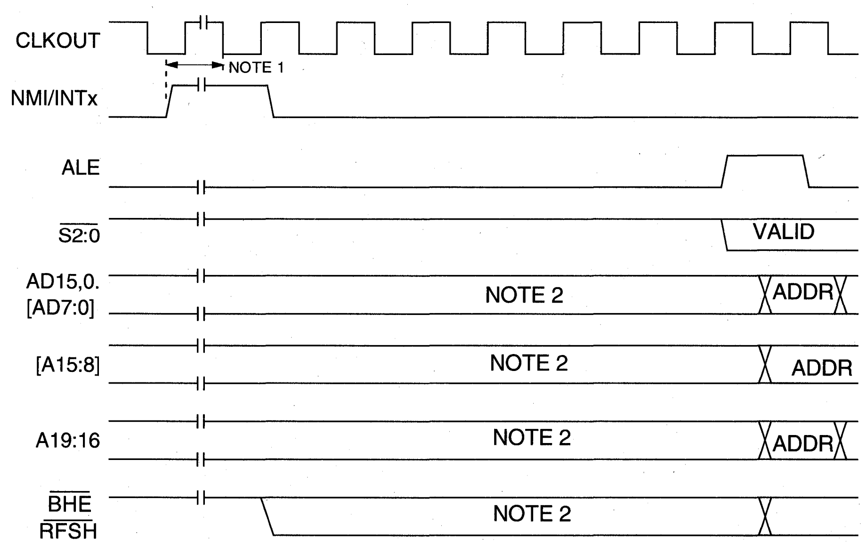


Figure 3.28. Returning to HALT After a HOLD/HLDA Bus Exchange



NOTE:

1. For NMI, delay = 4 1/2 clock
   For INTx, delay = 7 1/2 clocks
2. Previous bus cycle value

Figure 3.29. Exiting HALT

The circuit shown in Figure 3.31 illustrates how to use transceivers to buffer the AD bus. The connection between the processor and the transceiver is known as the "local bus." Connections between the transceiver and other memory or I/O devices is known as the "buffered bus." A fully buffered system does not have any devices attached to the local bus. A partially buffered system has devices on both the local and buffered buses.

$\overline{\text{DEN}}$ drives the transceiver output enable directly in a fully buffered system. A partially buffered system requires $\overline{\text{DEN}}$ to 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 $\overline{\text{DEN}}$.

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

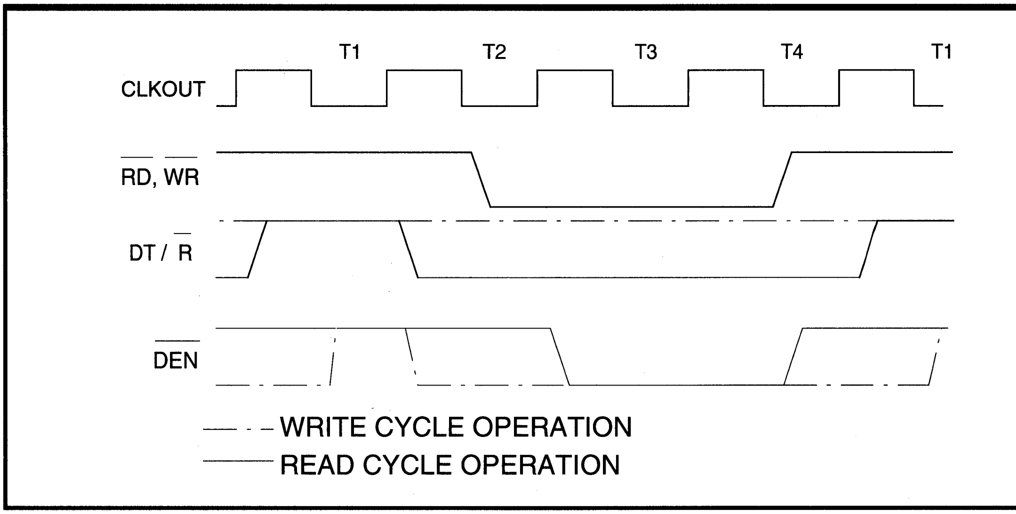


Figure 3.30. $\overline{\text{DEN}}$ and DT/$\overline{\text{R}}$ Timing Relationship

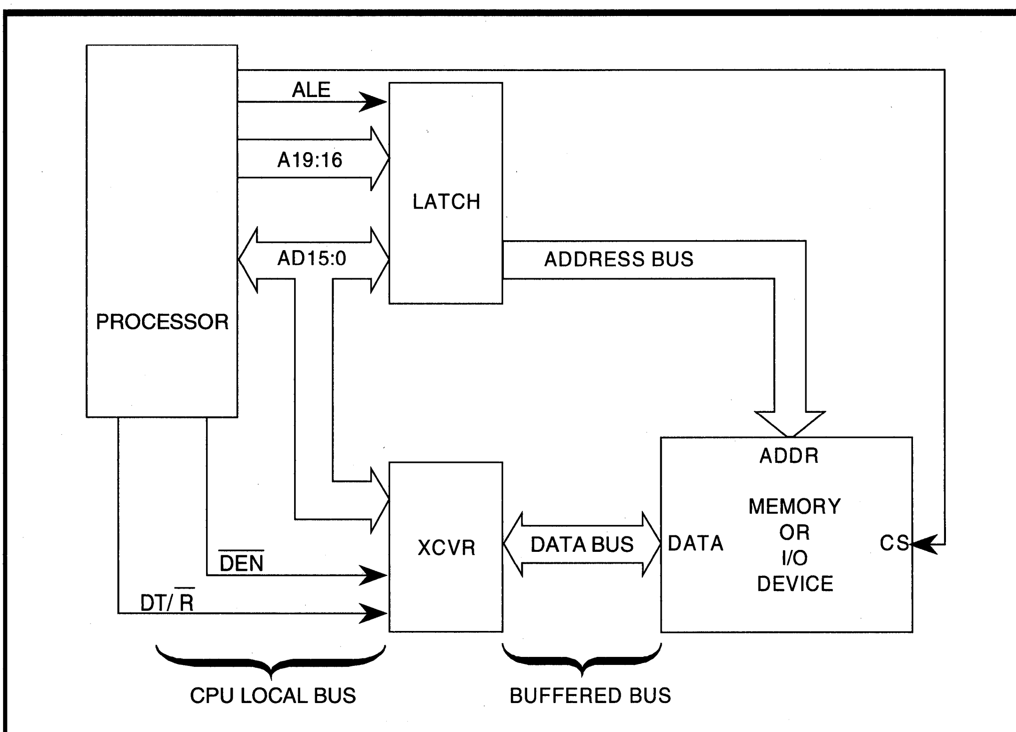


Figure 3.31. Buffered AD Bus System

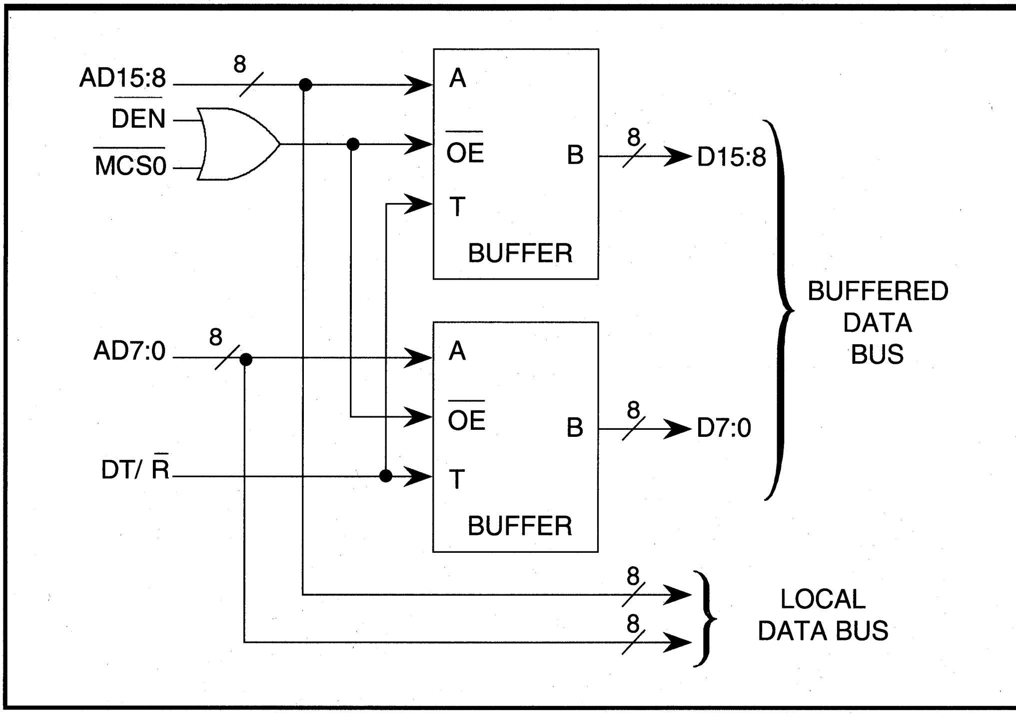


Figure 3.32. Qualifying $\overline{\text{DEN}}$ with Chip-Selects

3.6.2. SOFTWARE SYNCHRONIZATION

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, and the program execution continues.

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 once again 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 $\overline{\text{TEST}}$ input pin is sampled low. Unlike interrupts, the $\overline{\text{TEST}}$ input pin does not require program execution to be transferred to a new location (i.e., an interrupt routine is not executed). In processing the WAIT instruction, as long as $\overline{\text{TEST}}$ remains high program execution remains suspended (at least until an interrupt occurs). When $\overline{\text{TEST}}$ is sampled low, program execution resumes.

The 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. LOCKED BUS OPERATION

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

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 LOCK signal should be incorporated into the system bus arbitration logic resident to 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 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 LOCK is active), a bus hold, DMA or refresh request are recorded but not acknowledged until completion of the locked instruction. However, LOCK has no affect on interrupts. As an example, a locked HALT instruction causes bus hold, DMA or refresh bus requests to be ignored, but still allows the CPU to exit the HALT state on an interrupt.

In general, prefix bytes (like LOCK) are considered extensions of the instructions they preceded. Interrupts, DMA requests and refresh requests that occur during execution of prefix are not acknowledged until completion of the instruction following the prefix (except for instructions which are servicing interrupts during their execution, (i.e., HALT, WAIT and repeated string primitive). Note that multiple prefix bytes may precede an instruction.

Another example is a "string primitive" preceded by the repetition prefix (REP) which is interruptible 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, DMA and refresh requests remain locked out until $\overline{\text{LOCK}}$ is removed (either by completing the block operation or after an interrupt occurs).

3.6.4. QUEUE STATUS OPERATION

The queue status indicates what information is being removed from the internal queue and when the queue is being reset due to a transfer of control (e.g., jump, interrupt, etc.). Since the Execution Unit can remove information from the queue on any clock boundary, the queue status pins **can** change state on every phase 1 clock edge (see Figure 3.33). The queue status signals can not be related to any specific T-state, although for a given sequence of instructions the relationship between the operation of the BIU and the sequence of queue status information always remains the same.

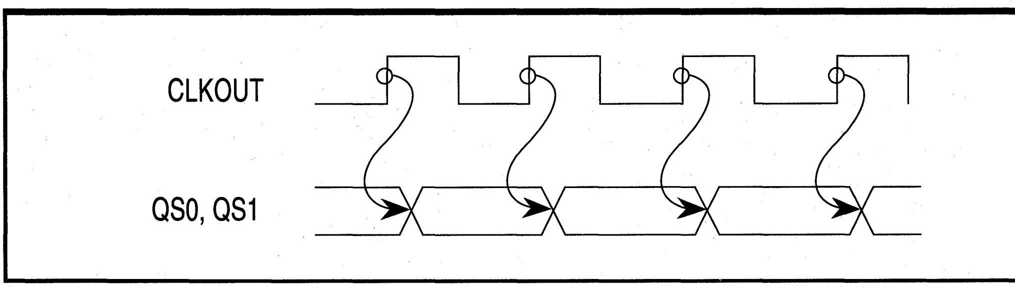


Figure 3.33. Queue Status Timing

The queue status signals QS0 and QS1 become alternate functions of the ALE and $\overline{\text{WR}}$ signals, respectively. To enable QS0 and QS1, the $\overline{\text{RD}}$ signal pin must be directly shorted to ground. $\overline{\text{RD}}$, $\overline{\text{WR}}$ and ALE are no longer available for use by the system and must be generated by external hardware. A device like the 82C88 or a programmable logic device can recreate the function of $\overline{\text{RD}}$, $\overline{\text{WR}}$ and ALE. Table 3.7 shows the encoding of the QS0 and QS1 signals.

Table 3.7. Queue Status Bit Encoding

QS1	QS2	DEFINITION
0	0	No queue operation occurred
0	1	First byte of a new instruction has been taken from the queue.
1	0	The queue was reinitialized. Signals the flush of all prefetch information. BIU must begin prefetching new queue information.
1	1	Subsequent byte of instruction taken from queue. The current instruction contains multiple opcode bytes or immediate data.

Queue status mode is required in older generation devices for the purposes of interfacing with an 8087 Math Coprocessor. However, the 8087 Math Coprocessor has been replaced by the

80187 Math Coprocessor, which has an I/O port interface similar to a peripheral device. This new interface no longer requires queue status mode.

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, and then waits until the HLDA output goes active before driving the bus. After HLDA has gone 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. Table 3.8 lists the state of the BIU pins when HLDA is asserted. Figure 3.35 illustrates the timing sequence when acknowledging the hold request. Of those device pins not mentioned in Table 3.8 or shown in Figure 3.35, all other pins either remain active (e.g., CLKOUT and TMR OUT1) or remain in their inactive state (e.g., $\overline{\text{UCS}}$ and $\overline{\text{INTA}}$). Refer to the data sheet for specific details of pin functioning during a bus hold.

Table 3.8. Signal Condition Entering HOLD

SIGNAL	HOLD CONDITION
A19:16, $\overline{\text{S2:0}}$, $\overline{\text{RD}}$, $\overline{\text{WR}}$, DT/$\overline{\text{R}}$, $\overline{\text{BHE}}$, $\overline{\text{RFSH}}$, DT/$\overline{\text{R}}$, $\overline{\text{LOCK}}$	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), $\overline{\text{DEN}}$	These signals float the same clock HLDA is generated (i.e., phase 1).

3.7.1.1. HOLD BUS LATENCY

The duration of time between the assertion of HOLD by the external device and the assertion of HLDA by the BIU is known as bus latency. In Figure 3.34, the two clock delay between HOLD and HLDA represents the shortest bus latency. Normally this only occurs if the bus is idle, halted or the bus hold request occurs just prior to the BIU beginning another bus cycle.

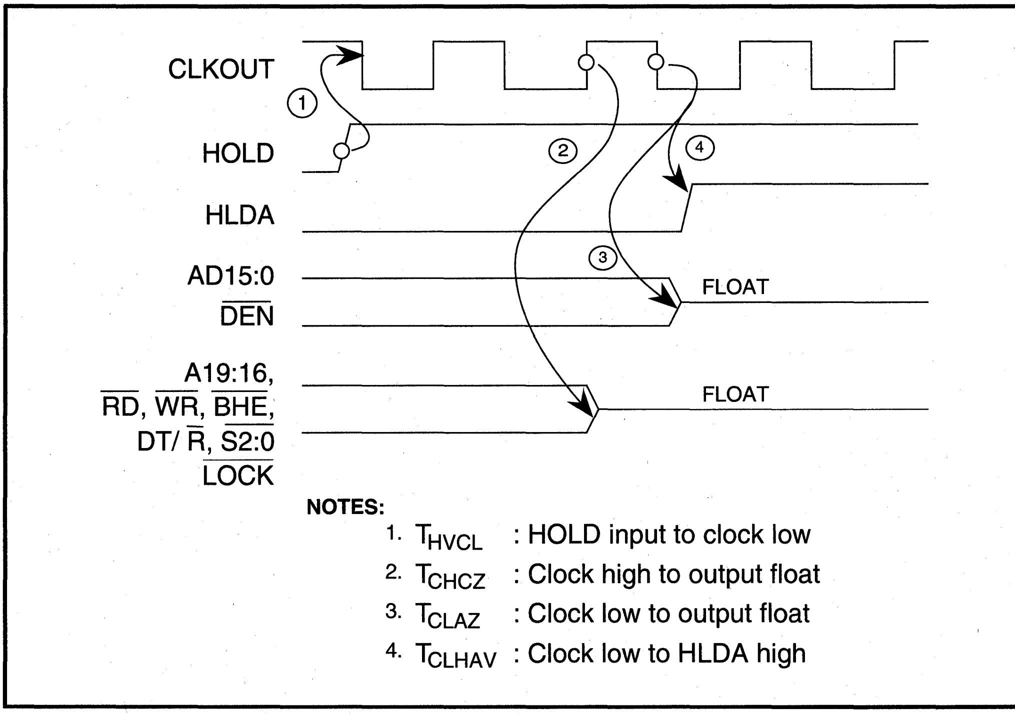


Figure 3.34. Timing Sequence Entering HOLD

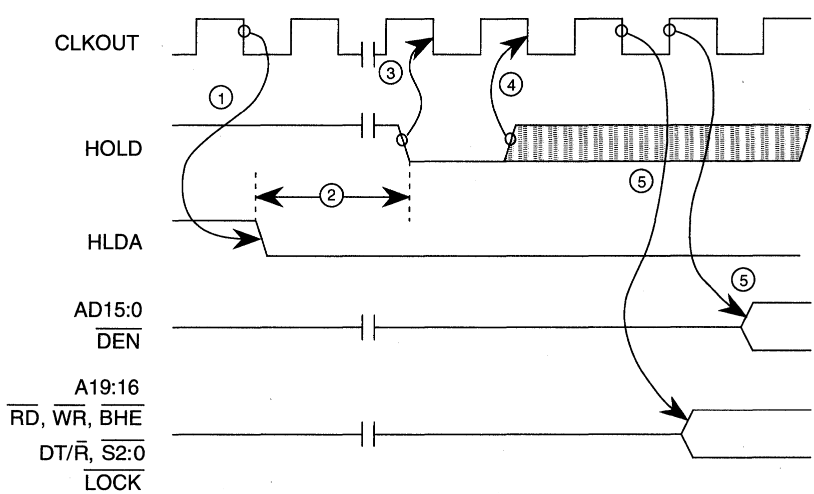
The major factors that influence bus latency are listed below (in order of longest delay to shortest delay).

1. Bus Not Ready — As long as the bus remains not ready a bus hold request can not be serviced.
2. Locked Bus Cycle — As long as $\overline{\text{LOCK}}$ remains asserted a bus hold request can not be serviced. Performing a locked move string operation can take several thousands of clocks.
3. Completion of Current Bus Cycle — A bus hold request is not 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 INTA bus cycle has completed. An INTA bus cycle drives $\overline{\text{LOCK}}$ active.
5. DMA and Refresh Bus Cycles — A bus hold request is not serviced until after the DMA request or refresh bus cycle has completed. Refresh bus cycles have a higher priority than hold bus requests. A bus hold request can not separate the bus cycles associated with a DMA transfer (worst case is an odd aligned transfer, which takes four bus cycles to complete).

3.7.1.2. REFRESH OPERATION DURING A BUS HOLD

Under normal operating conditions, once HDLA 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 3.35 shows the timing associated with the occurrence of 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, so when both occur simultaneously the refresh request occurs before HLDA becomes active.



NOTES:
1. HLDA deasserted, signaling need to run refresh bus cycle
2. External bus master terminates use of the bus.
3. HOLD deasserted.
4. HOLD may be reasserted after one clock.
5. BIU runs refresh bus cycle

Figure 3.35. Refresh Request During Bus Hold

The device requesting a bus hold must be able to detect a one clock wide HLDA pulse. A bus lockup (hang) condition may result because the requesting device did not 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 3.36 can be used to latch HLDA.

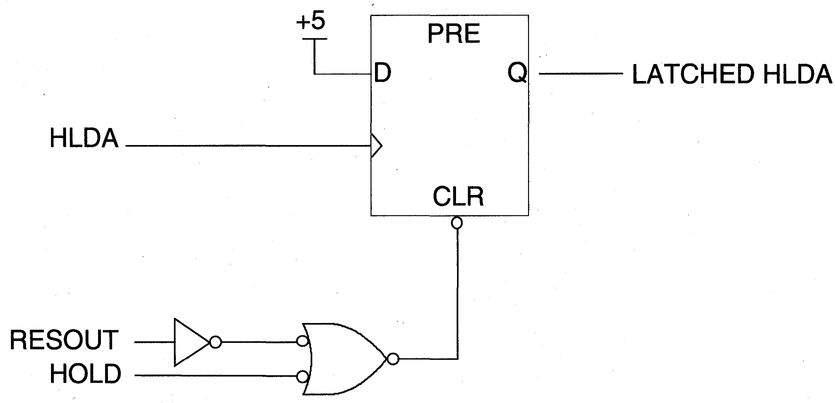


Figure 3.36. Latching HLDA

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. The BIU will release the bus and generate HLDA should HOLD go active prior to completing the refresh bus cycle.

3.7.2. EXITING HOLD

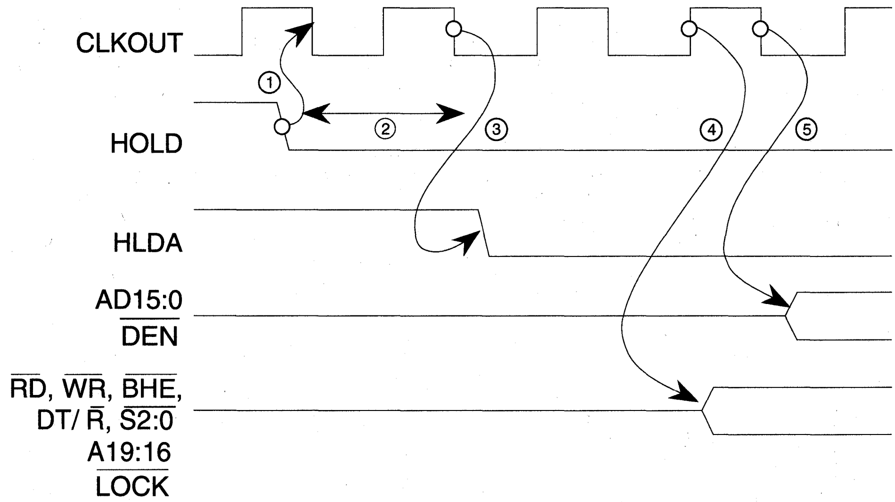
Figure 3.38 shows the timing associated with exiting the bus hold state. Normally a bus operation (e.g., instruction prefetch) occurs just after HOLD is released. However, if no bus cycle is pending when leaving a bus hold state, the bus and associated control signals remain floating.

3.8. BUS CYCLE PRIORITIES

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

1. Instruction execution reads or writes 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, DMA) interrupt vectoring sequence.
8. 80C187 Math Coprocessor error interrupt vectoring sequence.
9. DMA bus cycles.
10. General instruction execution. This category includes read and 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., EXCHG) may be separated from the first cycle by other bus cycles.
    - Successive bus cycles of string instructions (e.g., MOVS) may be separated by other bus cycles.
    - When a locked instruction begins, its associated bus cycles become the highest priority and can not be separated (or preempted) until completed.
11. Bus cycles necessary to fill the prefetch queue.



NOTES:
1. T_{HVCL} : HOLD recognition setup to clock low
2. : HOLD internally synchronized
3. T_{CLHAV} : Clock low to HLDA low
4. T_{CHCV} : Clock high to signal active (high or low)
5. T_{CLAV} : Clock low to signal active (high or low)

Figure 3.37. Exiting HOLD

Peripheral Control Block 4

CHAPTER 4
PERIPHERAL CONTROL BLOCK

All integrated peripherals are controlled by sets of registers within an integrated Peripheral Control Block (PCB). These 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. The control block can be located on any 256 byte boundary of memory or I/O space. Table 4.1 shows a map of these registers. Unused locations are reserved.

4.1. SETTING THE BASE LOCATION

The Peripheral Control Block contains the Peripheral Control Block Relocation Register, in addition to control registers for each integrated peripheral device. The Relocation Register allows the Peripheral Control Block to be relocated to any 256 byte boundary within memory or I/O space, depending on the state of the Memory I/O (MEM) bit and R19:8. Figure 4.1 shows the layout of the Relocation Register.

The Relocation Register is located at a fixed offset within the Peripheral Control Block. If the Peripheral Control Block is moved, the Relocation Register will also move.

The Peripheral Control Block Relocation Register contains the Escape Trap (ET) bit. When set, this bit forces the processor to trap whenever an ESC (coprocessor) instruction is encountered.

The Relocation Register also contains the Slave Master (SL) bit. This bit controls the function of the Interrupt Control Unit. See Chapter 8 for further explanation of this bit.

The Relocation Register contains the value 00FFH upon RESET. This means the Peripheral Control Block will be located at the top of I/O space (0FF00H to 0FFFFH).

As an example, to relocate the Peripheral Control Block to the memory range 10000-100FFH, the user would program the 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.

All communication between integrated peripherals and the Modular CPU Core occurs over a special bus called the *F-Bus*. The F-Bus always carries 16 bit data.

Table 4.1. Peripheral Control Block Register

PCB Offset	Function	PCB Offset	Function	PCB Offset	Function	PCB Offset	Function
00H	Reserved	40H	Reserved	80H	Reserved	C0H	D0SRCL
02H	Reserved	42H	Reserved	82H	Reserved	C2H	D0SRCH
04H	Reserved	44H	Reserved	84H	Reserved	C4H	D0DSTL
06H	Reserved	46H	Reserved	86H	Reserved	C6H	D0DSTH
08H	Reserved	48H	Reserved	88H	Reserved	C8H	D0TC
0AH	Reserved	4AH	Reserved	8AH	Reserved	CAH	D0CON
0CH	Reserved	4CH	Reserved	8CH	Reserved	CCH	Reserved
0EH	Reserved	4EH	Reserved	8EH	Reserved	CEH	Reserved
10H	Reserved	50H	T0CNT	90H	Reserved	D0H	D1SRCL
12H	Reserved	52H	T0CMPA	92H	Reserved	D2H	D1SRCH
14H	Reserved	54H	T0CMPB	94H	Reserved	D4H	D1DSTL
16H	Reserved	56H	T0CON	96H	Reserved	D6H	D1DSTH
18H	Reserved	58H	T1CNT	98H	Reserved	D8H	D1TC
1AH	Reserved	5AH	T1CMPA	9AH	Reserved	DAH	D1CON
1CH	Reserved	5CH	T1CMPB	9CH	Reserved	DCH	Reserved
1EH	Reserved	5EH	T1CON	9EH	Reserved	DEH	Reserved
20H	Reserved	60H	T2CNT	A0H	UMCS	E0H	RFBASE
22H	EOI	62H	T2CMPA	A2H	LMCS	E2H	RFTIME
24H	POLL	64H	Reserved	A4H	PACS	E4H	RFCON
26H	POLLSTS	66H	T2CON	A6H	MMCS	E6H	RFADDR
28H	IMASK	68H	Reserved	A8H	MPCS	E8H	Reserved
2AH	PRIMSK	6AH	Reserved	AAH	Reserved	EAH	Reserved
2CH	INSERV	6CH	Reserved	ACH	Reserved	ECH	Reserved
2EH	REQST	6EH	Reserved	AEH	Reserved	EEH	Reserved
30H	INTSTS	70H	Reserved	B0H	Reserved	F0H	PWRSAV
32H	TCUCON	72H	Reserved	B2H	Reserved	F2H	PWRCON
34H	DMA0CON	74H	Reserved	B4H	Reserved	F4H	Reserved
36H	DMA1CON	76H	Reserved	B6H	Reserved	F6H	STEPID
38H	I0CON	78H	Reserved	B8H	Reserved	F8H	Reserved
3AH	I1CON	7AH	Reserved	BAH	Reserved	FAH	Reserved
3CH	I2CON	7CH	Reserved	BCH	Reserved	FCH	Reserved
3EH	I3CON	7EH	Reserved	BEH	Reserved	FEH	RELREG

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). Accesses to the Peripheral Control Block, like all integrated peripherals, are always done 16 bits at a time. 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 is used with an 80C188 Modular Core family member (i.e., OUT DX, AX), 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 will no longer be within the control block (the Peripheral Control Block was moved on the first cycle). Generation of external READY is now needed to complete the cycle. For this reason, we recommend byte operations for the Relocation Register. Byte instructions should also be used for the other registers in the Peripheral Control Block of an 80C188 Modular Core family member. This requires half of the bus cycles of word operations. Byte operations are only valid for even addressed writes to the Peripheral Control Block. A word read (i.e., IN AX, DX) must be performed to read a 16-bit Peripheral Control Block register.

Register Name: PCB Relocation Register
Register Mnemonic: RELREG
Register Function: Relocates the PCB within memory or I/O space.

15															0
ET	SL		MEM	R19	R18	R17	R16	R15	R14	R13	R12	R11	R10	R9	R8

BIT MNEMONIC	BIT NAME	RESET STATE	FUNCTION
ET	*Escape Trap*	0	If set, the CPU will trap when an ESC instruction is executed.
SL	*Slave Master*	0	If set, the Interrupt Control Unit operates in slave mode. If clear, the Interrupt Control Unit operates in master mode.
MEM	*Memory I/O*	0	If set, the PCB is located in memory space. If clear, the PCB is located in I/O space.
R19:8	*PCB Base Address Upper Bits*	1	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 grey shading. Reserved register bits must be written with a logic zero value to maintain compatibility with future Intel products.

Figure 4.1. PCB Relocation Register

4.2. PERIPHERAL CONTROL BLOCK 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. Many locations within the Peripheral Control lock are not assigned to any peripheral. If a write is made to these locations, a bus cycle will occur, but data will not be stored. If a subsequent read is made to the same location, the value written will not be read back. Unused Peripheral Control Block locations are reserved.

The processor will run an external bus cycle for any memory or I/O cycle accessing a location within the Peripheral Control Block. Address, data and control information will be driven on the external pins as with an ordinary bus cycle. Information returned by an external device will be 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 word accesses made to integrated registers will be performed in two bus cycles.

The processor generates an internal READY signal whenever an integrated peripheral is accessed. External READY is ignored. READY will also be generated if an access is made to the Peripheral Control Block not corresponding to an integrated peripheral control register. The processor will not insert wait states for any access to the integrated Peripheral Control Block. The exceptions to this are accesses to timer registers. Accesses to timer control and counting registers insert one wait state. This is required to properly multiplex processor and counter element accesses to the timer control registers.

The F-Bus does not function identically to 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 cause the modification of [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 do not work normally. 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. For the 80C186 Modular Core:

Word reads: Access only even aligned words with IN AX, DX or MOV <word register>, <even PCB address>.

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

Word writes: Always write even aligned words. Writing an odd aligned word will give unexpected results. Use either OUT DX, AX or OUT DX, AL (or MOV <even PCB address>, <word register>).

Byte writes: Do not perform unaligned byte writes. Even aligned byte writes will modify the entire word PCB location.

For the 80C188 Modular Core:

Word reads: Access only even aligned words with IN AX, DX or MOV <word register>, <even PCB address>.

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

Word writes: Always write even aligned words. Writing an odd aligned word will give unexpected results. Use OUT DX, AL or MOV <even aligned byte PCB address>, <byte register low byte>. Using OUT DX, AX will perform an unnecessary bus cycle.

Byte writes: Do not perform unaligned byte writes. Even aligned byte writes will modify the entire word PCB location.

4.3. RESERVED LOCATIONS AND THE NUMERICS INTERFACE

Locations within the Peripheral Control Block not explicitly used 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. **Failure to follow this guideline could result in incompatibilities with future 80C186 Modular Core family products.**

Systems using the 80C187 Numeric Processor Extension must not relocate the Peripheral Control Block to location 0H in I/O space. The 80C187 interface uses I/O locations 0F8H through 0FFH. If the Peripheral Control Block were relocated to these locations, the processor would be communicating with the Peripheral Control Block, not the 80C187 interface circuitry. This will cause indeterminate system operation if a numerics instruction is encountered when the Escape Trap bit is clear.

Clock Generation and Power Management

5

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. 80C186 Modular Core Family processors have additional logic which controls the clock signals to provide power management functions.

5.1. CLOCK GENERATION

The clock generation circuit includes a crystal oscillator, a divide-by-two counter and power-save and reset circuitry (see Figure 5.1).

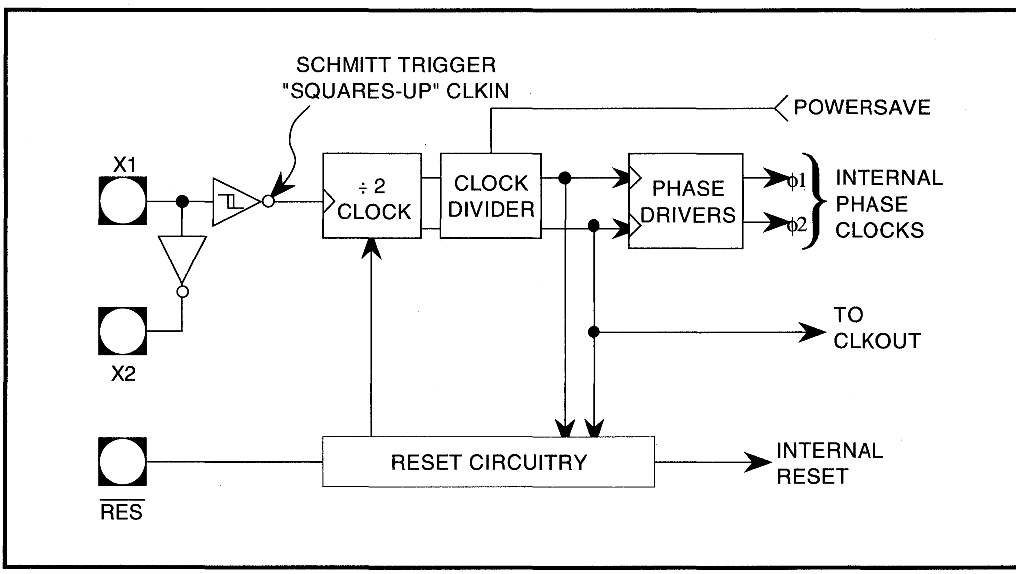


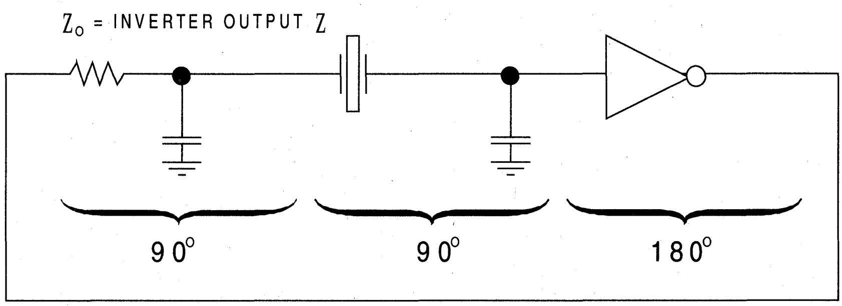
Figure 5.1. Clock Generator

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 degree phase shift around the loop will sustain the feedback in the oscillator. The on-chip inverter provides a 180 degree phase shift. The combination of the inverter's output impedance and the first load capacitor (see Figure 5.2) provides another 90 degree phase shift. At resonance, the crystal becomes primarily resistive. The combination of the crystal and the second load capacitor provides the final 90 degree phase shift. Above and below resonance the crystal is reactive and forces the oscillator back toward the crystal's nominal frequency.



NOTE: At resonance, the crystal is essential resistive.
Above resonance, the crystal is inductive.
Below resonance, the crystal is capacitive.

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 X1 and C_{X2} at X2, 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. Section 5.1.1.2 discusses crystal vibration modes in more detail.

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 criteria makes the circuit inductive at the fundamental frequency. The inductive circuit cannot make the 90 degree 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 criteria makes the circuit capacitive at the third overtone frequency, necessary for oscillation.
- The LC components form an equivalent parallel resonant circuit at a frequency about halfway between the fundamental frequency and the third overtone frequency. This

criteria 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.

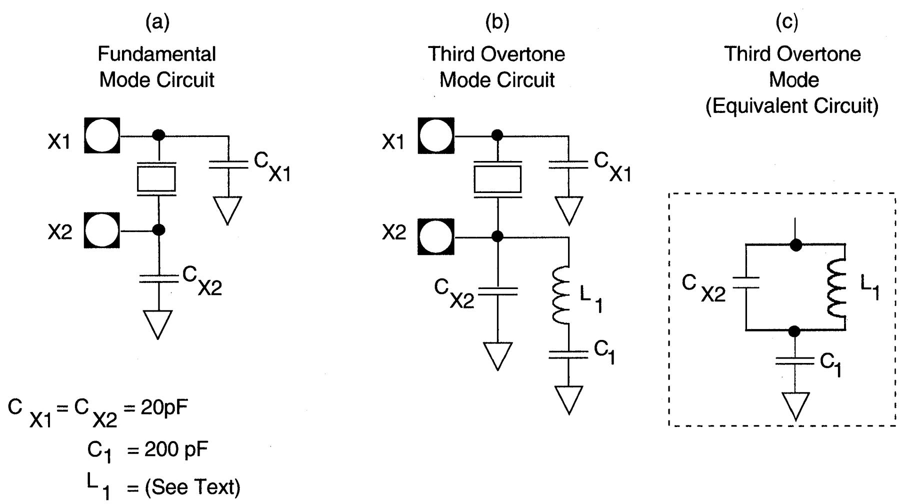


Figure 5.3. Crystal Connections to Microprocessor

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

(a) Series or Parallel Resonant Frequency

$$Ceq = \frac{\omega^2 C_1 C_{X2} L_1 - C_1 - C_{X2}}{\omega^2 C_1 L_1 - 1}$$

(b) Equivalent Capacitance

Figure 5.4. Equations for Crystal Calculations

Choosing C_1 as 200 pF (at least 10X 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 percent) and the equation in Figure 5.4(a) now yields the parallel resonant frequency.

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

f_{CLKOUT} (MHz)	$f_{3\ O.T.}$ (MHz)	L_1 (μH)
10	20	10.0, 12.0, 15.0
12.5	25	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. Above 32 MHz, vendors usually offer a third overtone component. You must know the vibrational mode to know whether to add the LC circuit at X2.

- 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 Vcc. 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 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 degrees 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 percent 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 percent) 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 X1 input. Leave X2 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 as 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 percent 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 X1 signal, even during reset and bus hold. CLKOUT is also available during Idle Mode but not during Powerdown Mode (see Sections 5.2.2 and 5.2.3).

In a CMOS circuit, significant current only flows 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 (RESET). The $\overline{\text{RES}}$ input generates RESOUT and the clock generator synchronizes it to the CLKOUT signal.

A Schmitt trigger in the $\overline{\text{RES}}$ 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 four CLKOUT cycles after V_{CC} and CLKOUT stabilize. The hysteresis allows a simple RC circuit to drive the $\overline{\text{RES}}$ input (see Figure 5.5). Typical applications can use about 100 ms. as an RC time constant.

Reset may be either cold (power-up) or warm. Figure 5.6 illustrates a cold reset. Assert the $\overline{\text{RES}}$ input during power supply and oscillator startup. The processor's pins assume their reset pin states a maximum of 28 X1 periods after X1 and V_{CC} stabilize. Assert $\overline{\text{RES}}$ four additional X1 periods after the device pins assume their reset states.

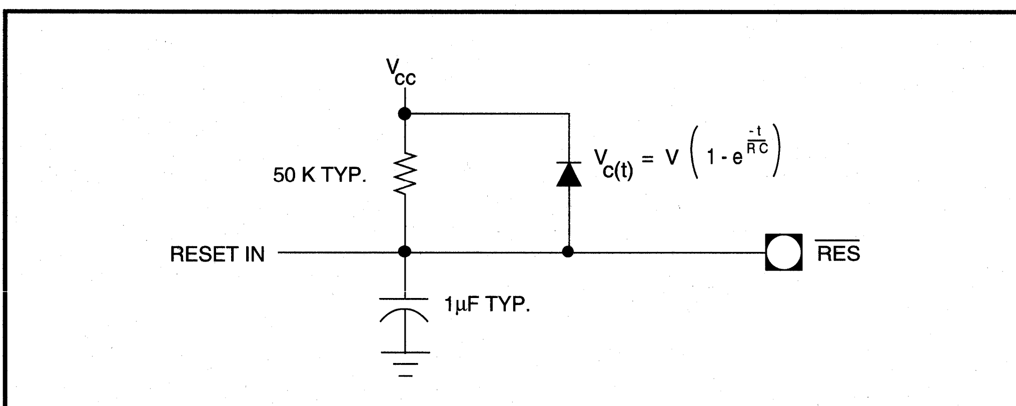
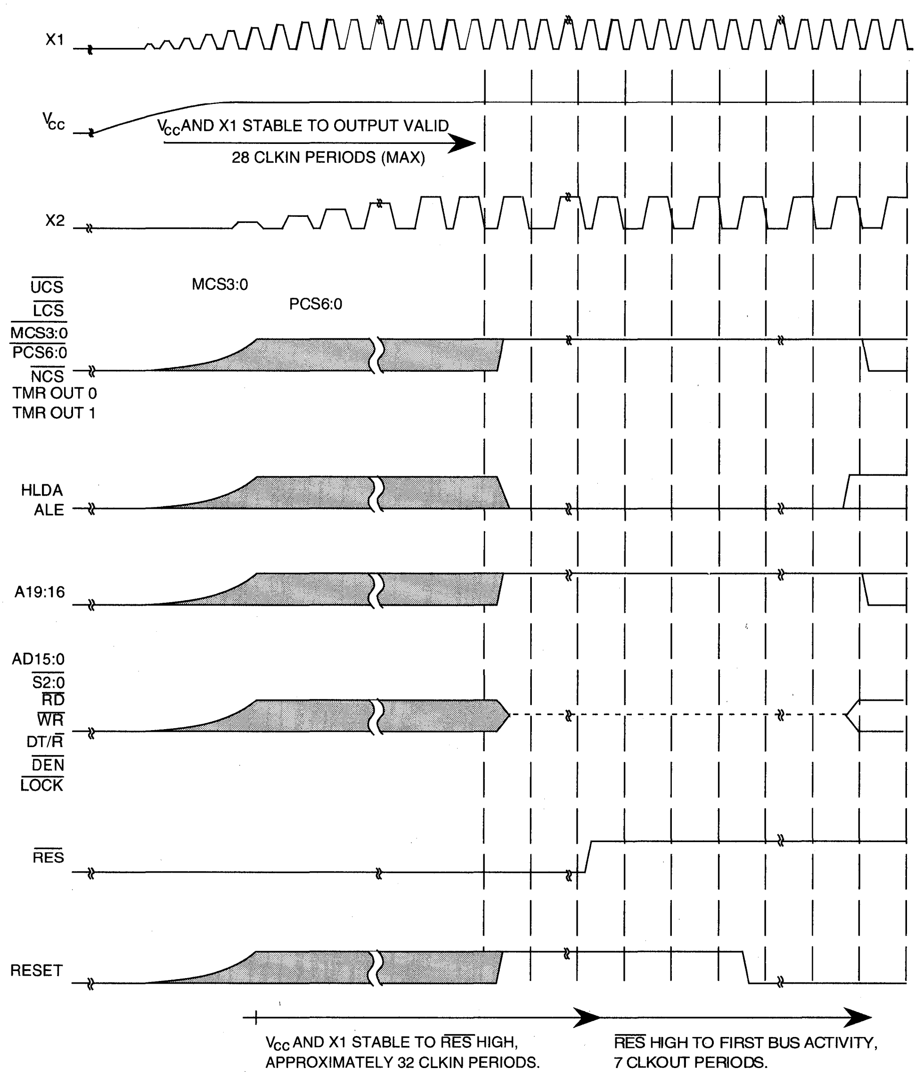


Figure 5.5. Simple RC Circuit for Powerup Reset

Applying $\overline{RES}$ when the device is running constitutes a warm reset (see Figure 5.7). In this case, assert $\overline{RES}$ at least 4 CLKOUT periods. The device pins will assume their reset states on the second falling X1 edge following the assertion of $\overline{RES}$.



NOTE:

CLKOUT synchronization occurs approximately 1 - 1/2 CLKIN periods after $\overline{RES}$ is sampled low.

Figure 5.6. Cold Reset Waveform

The falling $\overline{RES}$ edge generates an internal RESYNC pulse (Figure 5.8) resynchronizing the divide-by-two internal phase clock. The clock generator samples $\overline{RES}$ on the falling X1 edge. If $\overline{RES}$ is sampled high while CLKOUT is high, the processor forces CLKOUT high for the next two X1 cycles. The clock essentially "skips a beat" to synchronize the internal phases. If $\overline{RES}$ is sampled high while CLKOUT is low, CLKOUT is already in phase.

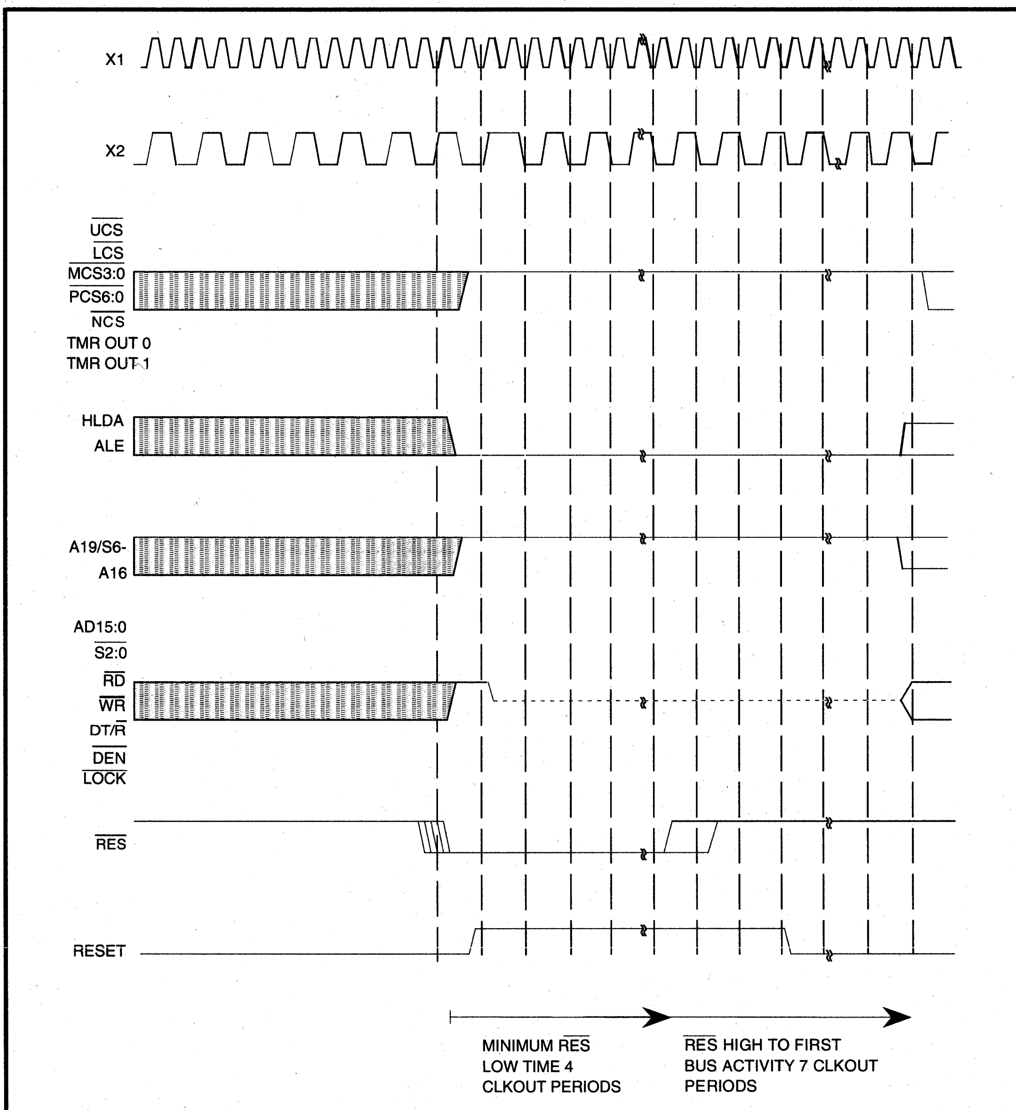
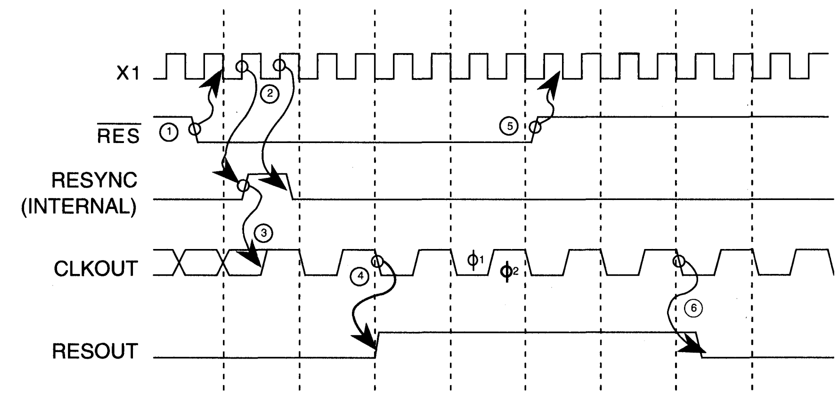


Figure 5.7. Warm Reset Waveform

At the second falling CLKOUT edge after sampling $\overline{RES}$ inactive, the processor deasserts RESOUT. Bus activity starts 6-1/2 CLKOUT periods after recognition of $\overline{RES}$ in the logic high state. If an alternate bus master asserts HOLD during reset, the processor will immediately assert HLDA and will not prefetch instructions.



NOTES:
1. Setup of $\overline{RES}$ to falling CLKIN.
2. RESYNC pulse generated.
3. RESYNC pulse drives CLKOUT high, resynchronizing the clock generator.
4. RESOUT goes active.
5. $\overline{RES}$ allowed to go inactive after minimum 4 CLKOUT cycles.
6. RESOUT goes inactive 1-1/2 CLKOUT cycles after $\overline{RES}$ sampled inactive.

Figure 5.8. Clock Synchronization at Reset

5.2. POWER MANAGEMENT

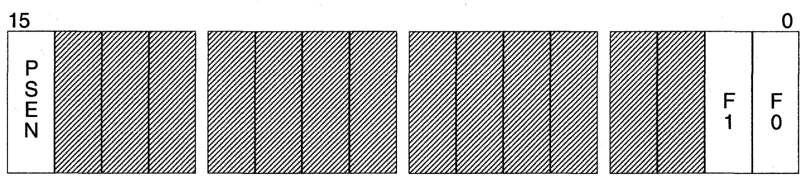
Many VLSI devices available today use dynamic circuitry. A dynamic circuit uses a capacitor (usually parasitic gate or diffusion capacitance) to store information. The stored charge decays over time due to leakage currents in the silicon. If the device does not use the stored information before it decays, the state of the entire device may be lost. Circuits must periodically refresh dynamic RAMs, for example, to ensure data retention. Any microprocessor which 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 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.

5.2.1. POWER-SAVE MODE

Power-Save Mode is a means to reduce operating current. Power-Save Mode enables a programmable clock divider in the clock generation circuit. This divider operates in addition to the divide-by-two counter mentioned in Section 5.1.

Register Name: Power Save Register
Register Mnemonic: PWRSAV
Register Function: Enables and sets clock division factor.



BIT MNEMONIC	BIT NAME	RESET STATE	FUNCTION
PSEN	*Power Save Enable*	0	Setting this bit enables Power Save Mode and divides the internal operating clock by the value defined by F1:0. This bit is cleared to disable Power-Save mode and force the CPU to operate at full speed. PSEN is automatically cleared whenever an interrupt occurs.
F1:0	*Clock Division Factor*	0H	These bits control the division factor used when Power Save mode is enabled. The allowable values are listed below: F1 F0 Divisor 0 0 By 1 0 1 By 4 1 0 By 8 1 1 By 16

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

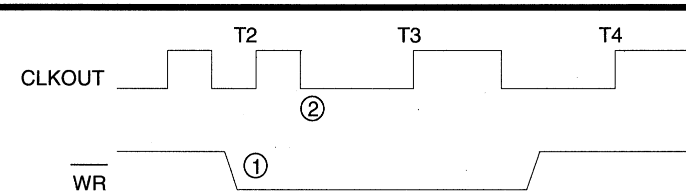
Figure 5.9. Power-Save Register

Possible clock divisor settings are 1, 4, 8 and 16 (1 has no effect). The divided frequency feeds the core, the integrated peripherals and CLKOUT. The processor operates at the divided clock rate exactly as if the crystal or external oscillator frequency were lower by the same amount.

It may be necessary to reprogram units such as the Timer Counter Unit and the Refresh Control Unit to compensate for the overall reduced clock rate.

5.2.1.1. ENTERING POWER-SAVE MODE

The Power-Save Register (see Figure 5.9) controls Power-Save Mode operation. The lower two bits select the divisor. When program execution sets the PSEN bit, the processor enters Power-Save Mode. The internal clock frequency changes at the falling edge of T_3 of the write to the Power-Save Register. CLKOUT changes simultaneously and does not glitch. Figure 5.10 illustrates the change at CLKOUT.



NOTES: 1. Write to Power-Save Register (as viewed on the bus).
2. Low-going edge of T3 starts new clock rate.

Figure 5.10. Power-Save Clock Transition

5.2.1.2. LEAVING POWER-SAVE MODE

Power-Save Mode continues until one of three events: execution clears the PSEN bit in the Power-Save Register, an unmasked interrupt occurs or an NMI occurs.

When the PSEN bit clears, the clock returns to its undivided frequency (standard divide-by-two) at the falling T_3 edge of the write to the Power-Save Register. The same result happens from reprogramming the clock divisor to a new value. The Power-Save Register can be read or written at any time.

Unmasked interrupts include those from the Interrupt Control Unit but not software interrupts. If an NMI occurs, or an unmasked interrupt request has sufficient priority to pass to the core, Power-Save Mode will end. The PSEN bit clears and the clock resumes full speed operation at the falling edge of a bus cycle T_3 state. However, the exact bus cycle of the transition is undefined. The Return from Interrupt instruction (IRET) does not automatically set the PSEN bit again. If you still want Power-Save Mode operation, you can set the PSEN bit as part of the interrupt service routine.

5.2.1.3. EXAMPLE POWER-SAVE INITIALIZATION CODE

Example 5.1 illustrates programming the Power-Save Unit for a typical system. The program also includes code to change the DRAM refresh rate to compensate for the reduced clock rate.

```
$mod186
name            example_PSU_code

;FUNCTION:      This function reduces CPU power consumption
;               by dividing the CPU operating frequency by a
;               divisor.
;   SYNTAX:     extern void far power_save(int divisor);
;   INPUTS:     divisor - This variable represents F0 and F1 of
;               PWRSAV.
; OUTPUTS:      None
;     NOTE:     Parameters are passed on the stack as required
;               by high-level languages
;
                                        ;substitute register offset
PWRSAV          equ     xxxxH           ;Power-Save Register
RFTIME          equ     xxxxH           ;Refresh Interval Count
Register
RFCON           equ     xxxxH           ;Refresh Control Register
PSEN            equ     8000H           ;Power-Save enable bit

data            segment public 'data'
FreqTable       dw      1, 4, 8, 16
data            ends

lib_80C186      segment public 'code'
                assume cs:lib_80C186, ds:data

                public _power_save
_power_save     proc far

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

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

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

Example 5.1. Power-Save Initialization Code

```
            mov    dx, RFCON            ;get current DRAM refresh
            in     ax, dx               ;rate
            and    ax, 01ffh            ;mask off unwanted bits

            div    FreqTable[_divisor]        ;divide refresh rate
                                              ;by _divisor
            mov    dx, RFTIME           ;set new refresh rate
            out    dx, ax
            mov    dx, PWRSAV           ;select Power-Save Register
            mov    ax, _divisor         ;get divisor
            and    ax, 3                ;mask off unwanted bits
            or     ax, PSEN             ;set enable bit
            out    dx, ax               ;divide frequency
            pop    dx                   ;restore saved registers
            pop    ax
            pop    bp                   ;restore caller's bp
            ret
_power_save endp

lib_80C186  ends
            end
```

Example 5.1. Power-Save Initialization Code (Continued)

Chip Select Unit 6

CHAPTER 6
CHIP SELECT UNIT

Every system requires some form of component select mechanism so the CPU can access a specific memory or peripheral device. The signal selecting 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 too slow to keep up with the maximum bus bandwidth can use wait states to slow the bus down.

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 a 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 3FFFFH, 5FFFFH, 7FFFFH and so on also selects the 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 upper most address bits. Each of the eight decoded outputs are 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 may require circuit changes to accommodate the additional memory.

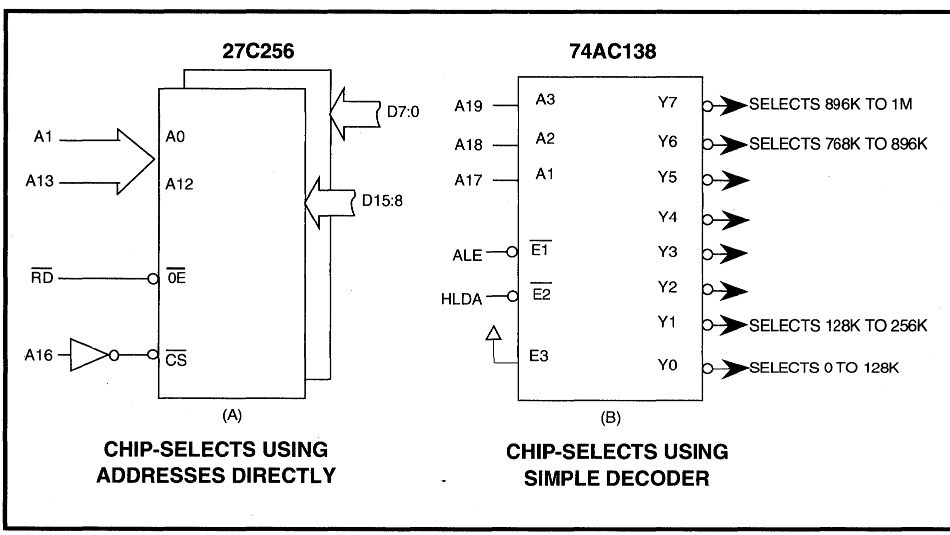


Figure 6.1. Common Chip-Select Generation Methods

The Chip-Select Unit overcomes limitations found in the above designs and has the following features:

- Thirteen chip-select outputs
- Programmable chip-select active range
- Memory or I/O bus cycle decoder
- Programmable wait state generator
- Provision to override bus ready

Figure 6.2 illustrates the logic blocks that generate a chip-select.

6.1. FUNCTIONAL OVERVIEW

The Chip-Select Unit, abbreviated 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 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 CPU, DMA Control Unit or Refresh Control Unit initiated bus cycles.

Six of the thirteen chip-selects only map into memory address space. The remaining seven chip-selects can map into memory or I/O address space. The chip-selects typically associate with memory and peripheral devices as follows:

$\overline{UCS}$ Mapped only to upper memory address space and selects the BOOT memory device (EPROM or FLASH memory types).

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

$\overline{MCS0:3}$ Mapped only to memory address space and selects additional SRAM memory, DRAM memory or system bus.

$\overline{PCS7:0}$ Mapped to memory or I/O address space and selects peripheral devices or generates a DMA acknowledge strobe.

The $\overline{LCS}$ chip-select always starts at address location 0H and has a programmable block size up to 256 Kbytes. The $\overline{UCS}$ chip-select always ends at address location 0FFFFH and has a programmable block size up to 256 Kbytes.

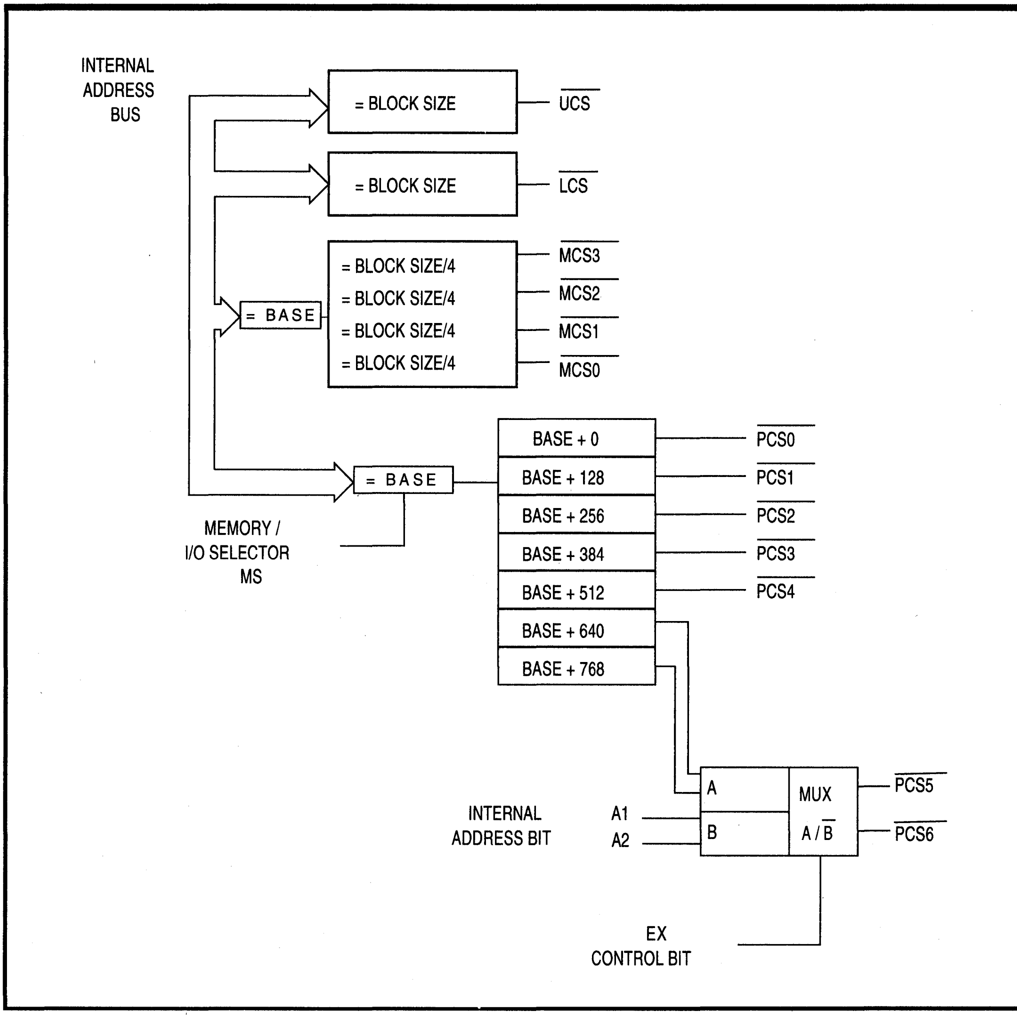


Figure 6.2. Chip-Select Block Diagram

The four $\overline{\text{MCS}}$ chip-selects access one contiguous block of memory address space. The block size can range from 8 Kbytes to 512 Kbytes and each chip-select goes active for one fourth of the block size. The block start address is programmable but must be an integer multiple of the block size. This start address limitation prevents the $\overline{\text{MCS}}$ chip-selects from covering the entire address space between the $\overline{\text{LCS}}$ and $\overline{\text{UCS}}$ chip-selects.

The $\overline{\text{PCS}}$ chip-selects access a contiguous block of memory or I/O address space. Each chip-select goes active for 128 bytes of the 896 byte block. The $\overline{\text{PCS}}$ block start address can begin on any 1 Kbyte boundary.

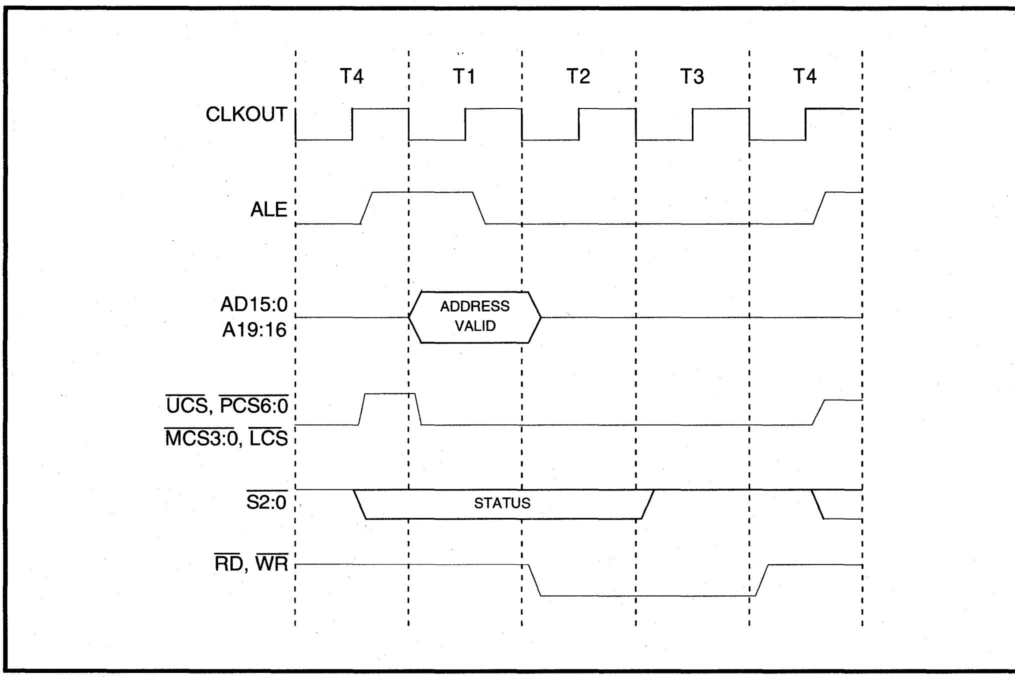


Figure 6.3. Chip-Select Relative Timings

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 default or programmed type (memory or I/O).
3) The bus cycle address is within the default or programmed block size.
4) 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 $\overline{UCS}$ chip-select is initialized and active (see Figure 6.4).

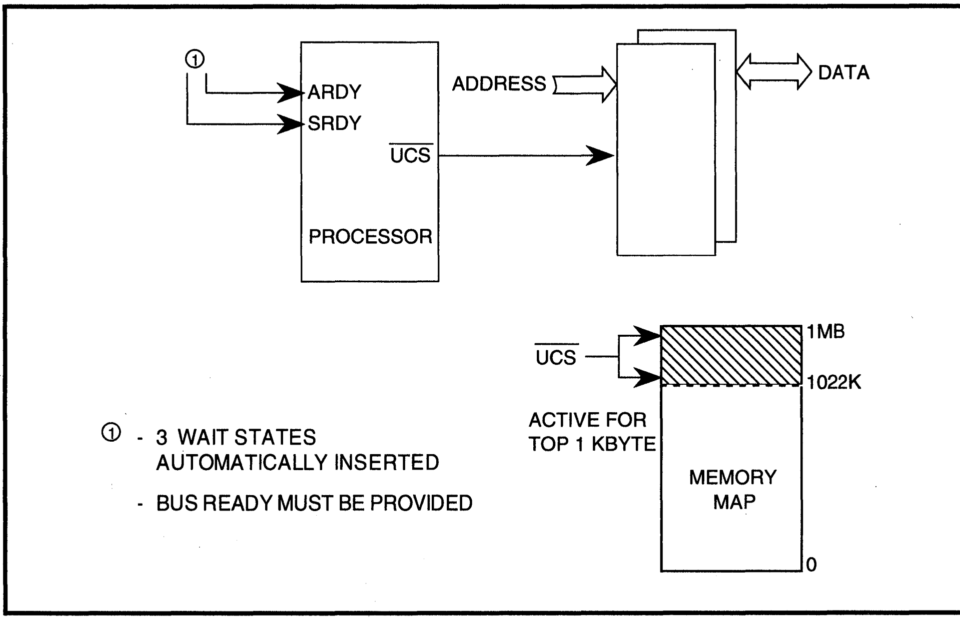


Figure 6.4. $\overline{\text{UCS}}$ Reset Configuration

6.2. PROGRAMMING

A set of registers determine the operating characteristics of the chip-selects. The Peripheral Control Block defines the location of the Chip-Select Unit registers. Table 6.1 lists all of the Chip-Select Unit registers and their associated programming names.

The $\overline{\text{UCS}}$ and $\overline{\text{LCS}}$ chip-selects each have one register that defines their operation (see Figure 6.5 and Figure 6.6).

Table 6.1. Chip-Select Unit Registers

REGISTER MNEMONIC	REGISTER MNEMONIC	CHIP-SELECT AFFECTED
UMCS		$\overline{\text{UCS}}$
LMCS		$\overline{\text{LCS}}$
MMCS	MPCS	$\overline{\text{MCS3:0}}$
PACS	MPCS	$\overline{\text{PCS7:0}}$

Register Name: $\overline{\text{UCS}}$ Control Register
Register Mnemonic: UMCS
Register Function: Controls the operation of the $\overline{\text{UCS}}$ chip-select.

15															0
		U17	U16	U15	U14	U13	U12	U11	U10				R2	R1	R0

BIT MNEMONIC	BIT NAME	RESET STATE	FUNCTION
U17:10	*Start Address*	0FFH	Defines the starting address for the $\overline{\text{UCS}}$ chip-select. During memory bus cycles, address bits A17:10 are compared against U17:10 and an equal to or greater than result enables the chip-select (A19 and A18 must be 1 also). Allowable bit programming combinations are as follows: U17:0 — Starting Address — Block Size 00H — 0C0000H — 256 Kbytes 80H — 0E0000H — 128 Kbytes C0H — 0F0000H — 64 Kbytes E0H — 0F8000H — 32 Kbytes F0H — 0FC000H — 16 Kbytes F8H — 0FE000H — 8 Kbytes FCH — 0FF000H — 4 Kbytes FEH — 0FF800H — 2 Kbytes FFH — 0FFC00H — 1 Kbytes
R2	*Bus Ready Disable*	0	Clearing R2 requires bus ready be active to complete a bus cycle. When R2 is cleared, R1:0 control the number of bus wait states (bus ready is ignored).
R1:0	*Wait State Value*	3H	R1:0 define the minimum number of wait states inserted into the bus cycle.

NOTE: Reserved register bits are shown with grey shading and must contain a value of zero when writing this register (to ensure compatibility with future products). Do not program U17:10 with values other than what is shown. Failure to do so results in unreliable chip-select operation. Reading this register (prior to writing it) enables the chip-select, however, none of the programmable fields will have been properly initialized.

Figure 6.5. UMCS Register Definition

Register Name: $\overline{\text{LCS}}$ Control Register
Register Mnemonic: LMCS
Register Function: Controls the operation of the $\overline{\text{LCS}}$ chip-select.

15															0
		U17	U16	U15	U14	U13	U12	U11	U10				R2	R1	R0

BIT MNEMONIC	BIT NAME	RESET STATE	FUNCTION
U17:10	*End Address*	XXH	Defines the ending address for the $\overline{\text{LCS}}$ chip-select. During memory bus cycles, address bits A17:10 are compared against U17:10 and a less than result enables the chip-select (A19 and A18 must be 0 also). Allowable bit programming combinations are as follows: U17:0 — Ending Address — Block Size 00H — 003FFH — 1 Kbytes 01H — 007FFH — 2 Kbytes 03H — 00FFFH — 4 Kbytes 07H — 01FFFH — 8 Kbytes 0FH — 03FFFH — 16 Kbytes 1FH — 07FFFH — 32 Kbytes 3FH — 0FFFFH — 64 Kbytes 7FH — 1FFFFH — 128 Kbytes FFH — 3FFFFH — 256 Kbytes
R2	*Bus Ready Disable*	X	Clearing R2 requires bus ready be active to complete a bus cycle. When R2 is cleared, R1:0 control the number of bus wait states (bus ready is ignored).
R1:0	*Wait State Value*	XH	R1:0 define the minimum number of wait states inserted into the bus cycle. A zero value means no wait states (unless R2 is zero, which means bus ready controls wait states)

NOTE: Reserved register bits are shown with grey shading and must contain a value of zero when writing this register (to ensure compatibility with future products). Do not program U17:10 with values other than what is shown. Failure to do so results in unreliable chip-select operation. Reading this register (prior to writing it) enables the chip-select, however, none of the programmable fields will have been properly initialized.

Figure 6.6. LMCS Register Definition

The $\overline{\text{MCS}}$ and $\overline{\text{PCS}}$ chip-selects require two registers to define their operation. One register is shared between them. The MMCS and MPCS registers control the $\overline{\text{MCS}}$ chip-selects. The PACS and MPCS registers control the $\overline{\text{PCS}}$ chip-selects. Figure 6.7, Figure 6.8 and Figure 6.9 define the programming attributes for each of the registers.

Register Name: $\overline{\text{MCS}}$ Control Register
Register Mnemonic: MMCS
Register Function: Controls the operation of the $\overline{\text{MCS}}$ chip-selects

15															0
U19	U18	U17	U16	U15	U14	U13	U12	U11	U10				R2	R1	R0

BIT MNEMONIC	BIT NAME	RESET STATE	FUNCTION
U19:13	*Start Address*	XXH	Defines the starting (base) address for the block of $\overline{\text{MCS}}$ chip-selects. During memory bus cycles, address bits A19:13 are compared against U19:13 and an equal to or greater than result enables the chip-select. The start address must be an integer multiple of the $\overline{\text{MCS}}$ block size (defined in the MPCS register).
R2	*Bus Ready Disable*	XH	Clearing R2 requires bus ready be active to complete a bus cycle. When R2 is cleared, R1:0 control the number of bus wait states (bus ready is ignored).
R1:0	*Wait State Value*	XH	R1:0 define the minimum number of wait states inserted into the bus cycle. A zero value means no wait states (unless R2 is zero, which means bus ready controls wait states)

NOTE: Reserved register bits are shown with grey shading and must contain a value of zero when writing this register (to ensure compatibility with future products). Reading this register and the MPCS register (prior to writing them) enables the $\overline{\text{MCS}}$ chip-selects, however, none of the programmable fields will have been properly initialized.

Figure 6.7. MMCS Register Definition

Register Name: $\overline{\text{MCS}}$ and $\overline{\text{PCS}}$ Alternate Control Register
Register Mnemonic: MPCS
Register Function: Controls the operation for both the $\overline{\text{MCS}}$ and $\overline{\text{PCS}}$ chip-selects.

15															0
	M6	M5	M4	M3	M2	M1	M0	EX	MS				R2	R1	R0

BIT MNEMONIC	BIT NAME	RESET STATE	FUNCTION
M6:0	*Block Size*	XXH	Defines the block size for the $\overline{\text{MCS}}$ chip-selects. Allowable bit programming combinations are as follows: M6 M5 M4 M3 M2 M1 M0 — Block Size 0 0 0 0 0 0 1 — 8 Kbytes 0 0 0 0 0 1 X — 16 Kbytes 0 0 0 0 1 X X — 32 Kbytes 0 0 0 1 X X X — 64 Kbytes 0 0 1 X X X X — 128 Kbytes 0 1 X X X X X — 256 Kbytes 1 X X X X X X — 512 Kbytes X = Don't Care, but should be 0 for future compatibility.
EX	*Pin Selector*	XH	Setting EX configures $\overline{\text{PCS5}}$ and $\overline{\text{PCS6}}$ pins as chip-selects. When EX is cleared, $\overline{\text{PCS5}}$ becomes latched address bit 1 (A1) and $\overline{\text{PCS6}}$ becomes latched address bit 2 (A2).
MS	*Bus Cycle Selector*	XH	When MS is cleared the $\overline{\text{PCS}}$ chip-selects go active for I/O bus cycles. Setting MS activates the $\overline{\text{PCS}}$ chip-selects for memory bus cycles.
R2	*Bus Ready Disable*	XH	This bit applies to the $\overline{\text{PCS4}}$-$\overline{\text{PCS6}}$ chip-selects only. Clearing R2 requires bus ready be active to complete a bus cycle. When R2 is set, R1:0 control the number of bus wait states (bus ready is ignored).
R1:0	*Wait State Value*	XH	These bits apply to the $\overline{\text{PCS4}}$-$\overline{\text{PCS6}}$ chip-selects only. R1:0 define the minimum number of wait states inserted into the bus cycle. A zero value means no wait states.

NOTE: Reserved register bits are shown with grey shading and must contain a value of zero when writing this register (to ensure compatibility with future products). Reading this register and the MMCS register or PACS register (prior to writing them) enables the associated chip-selects, however, none of the programmable fields will have been properly initialized.

Figure 6.8. MPCS Register Definition

Register Name: $\overline{\text{PCS}}$ Control Register
Register Mnemonic: PACS
Register Function: Controls the operation of the $\overline{\text{PCS}}$ chip-selects.

15															0
U19	U18	U17	U16	U15	U14	U13	U12	U11	U10				R2	R1	R0

BIT MNEMONIC	BIT NAME	RESET STATE	FUNCTION
U19:10	*Start Address*	XXH	Defines the starting (base) address for the block of $\overline{\text{PCS}}$ chip-selects. During memory or I/O bus cycles, address bits A19:13 are compared against U19:13 and an equal to or greater than result enables the chip-select. U19:16 must be programmed to zero for proper I/O bus cycle operation.
R2	*Bus Ready Disable*	XH	Clearing R2 requires bus ready be active to complete a bus cycle. When R2 is set, R1:0 control the number of bus wait states (bus ready is ignored).
R1:0	*Wait State Value*	XH	R1:0 define the minimum number of wait states inserted into the bus cycle. A zero value means no wait states (unless R2 is zero, which means bus ready controls wait states)

NOTE: Reserved register bits are shown with grey shading and must contain a value of zero when writing this register (to ensure compatibility with future products). Reading this register and the MPCS register (prior to writing them) enables the $\overline{\text{PCS}}$ chip-selects, however, none of the programmable fields will have been properly initialized.

Figure 6.9. PACS Register Definition

6.2.1. 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.

Programming the UMCS and LMCS registers can be done in any sequence. To program the $\overline{\text{MCS}}$ and $\overline{\text{PCS}}$ chip-selects, follow the sequence shown below:

1) Program the MPCS register
2) Program the MMCS register to enable the $\overline{\text{MCS}}$ chip-selects
3) Program the PACS register to enable the $\overline{\text{PCS}}$ chip-selects

6.2.2. START ADDRESS

The $\overline{\text{LCS}}$ chip-select has a fixed starting address of zero in memory address space. The $\overline{\text{UCS}}$ chip-select defines its starting address as 100000H (1 Mbyte) minus the programmed block size (see Section 6.2.4). The $\overline{\text{MCS}}$ chip-selects have a programmable base address that determines their individual start addresses (see Figure 6.10). However, there are limitations on the location of the base address depending on the $\overline{\text{MCS}}$ block size.

Table 6.2 lists the limitations of the base address for the $\overline{\text{MCS}}$ chip-selects. Figure 6.10 illustrates how to calculate the starting address for each $\overline{\text{MCS}}$ chip-select.

Each $\overline{\text{PCS}}$ chip-select is active for 128 bytes and start at an offset above the programmed base address. The base address can start on any 1 Kbyte memory or I/O address location. Table 6.3 lists the range for each chip-select.

Table 6.2. MMCS Programming Restrictions

ALLOWABLE BLOCK SIZE	BASE ADDRESS RESTRICTIONS	NOTES
8 Kbytes	None	
16 Kbytes	U13 must be zero	
32 Kbytes	U13-14 must be zero	
64 Kbytes	U13-15 must be zero	
128 Kbytes	U13-16 must be zero	
256 Kbytes	U13-17 must be zero	
512 Kbytes	U13-18 must be zero	Will overlap $\overline{\text{UCS}}$ if U19 is 1

Table 6.3. $\overline{\text{PCS}}$ Chip-Selects Active Range

CHIP SELECT	ACTIVE RANGE	
$\overline{\text{PCS0}}$	Base	to Base + 127 (7FH)
$\overline{\text{PCS1}}$	Base + 128 (080H)	to Base + 255 (0FFH)
$\overline{\text{PCS2}}$	Base + 256 (100H)	to Base + 383 (17FH)
$\overline{\text{PCS3}}$	Base + 384 (180H)	to Base + 511 (1FFH)
$\overline{\text{PCS4}}$	Base + 512 (200H)	to Base + 639 (27FH)
$\overline{\text{PCS5}}$	Base + 640 (280H)	to Base + 767 (2FFH)
$\overline{\text{PCS6}}$	Base + 768 (300H)	to Base + 895 (37FH)

6.2.3. STOP ADDRESS

The $\overline{\text{UCS}}$ chip-select has a fixed ending address of 0FFFFFH in memory address space. The $\overline{\text{LCS}}$ chip-select defines its ending address as one byte less than the programmed block size (see Section 6.2.4).

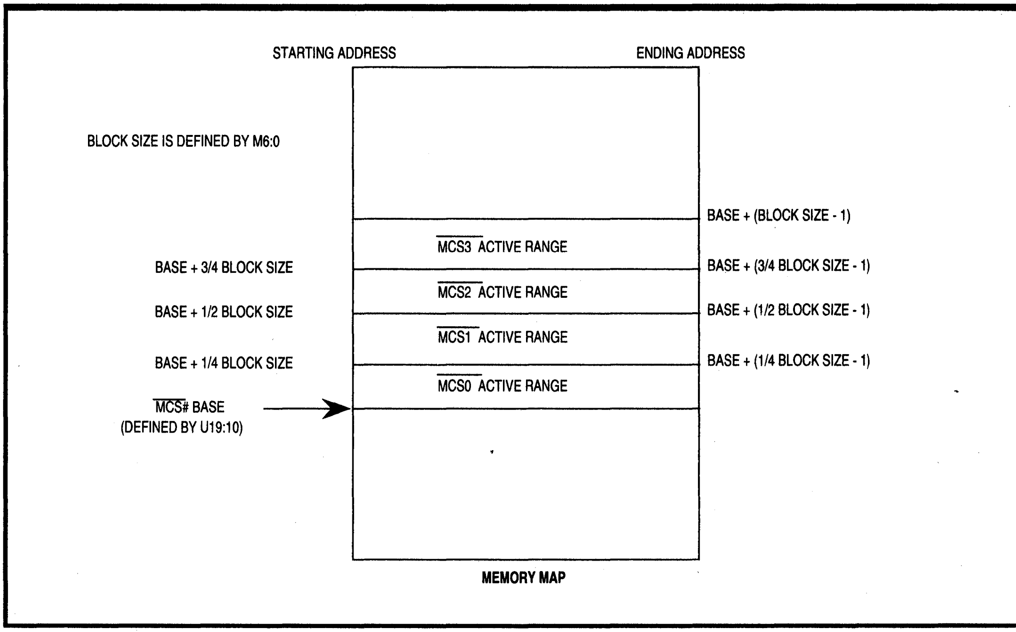


Figure 6.10. $\overline{\text{MCS}}$ Active Range

The ending address for the $\overline{\text{MCS}}$ chip-selects is defined by the programmed base address and the block size. Figure 6.10 illustrates how to calculate the ending address for each $\overline{\text{MCS}}$ chip-select.

The $\overline{\text{PCS}}$ chip-selects have fixed ending addresses defined by the programmed base address. Table 6.3 defines the ending address for each chip-select.

6.2.4. BLOCK SIZE

The $\overline{\text{LCS}}$, $\overline{\text{UCS}}$ and $\overline{\text{MCS}}$ chip-selects have programmable block sizes to define their active ranges. The $\overline{\text{PCS}}$ chip-selects have fixed block sizes of 128 bytes.

The LMCS and UMCS registers define the block size for the $\overline{\text{LCS}}$ and $\overline{\text{UCS}}$ chip selects, respectively. The allowable block sizes, in Kbytes, for the $\overline{\text{LCS}}$ and $\overline{\text{UCS}}$ chip-selects are 1, 2, 4, 8, 16, 32, 64, 128 and 256.

The combined $\overline{\text{MCS}}$ block size is controlled by the MPCS register. Each $\overline{\text{MCS}}$ chip-select is active for one quarter of the block size. Table 6.2 defines the allowable block sizes for the $\overline{\text{MCS}}$ chip-selects.

6.2.5. BUS WAIT STATE AND READY CONTROL

Normally the bus ready inputs 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 inputs to extend and complete the bus cycle automatically. Most memory and peripheral devices operate properly using three or less wait states. However, accessing devices such as a dual-port memory, an expansion bus interface, a system bus interface or remote peripheral devices can require more than three wait states to complete a bus cycle.

The Chip-Select Unit can insert up to three wait states and control the state of the bus ready inputs. The UMCS, LMCS, MMCS, MPCS and PACS registers define a three-bit field (R0, R1, R2) that control bus wait state and ready requirements. Figure 6.11 shows a simplified logic diagram of the wait state and ready control functions.

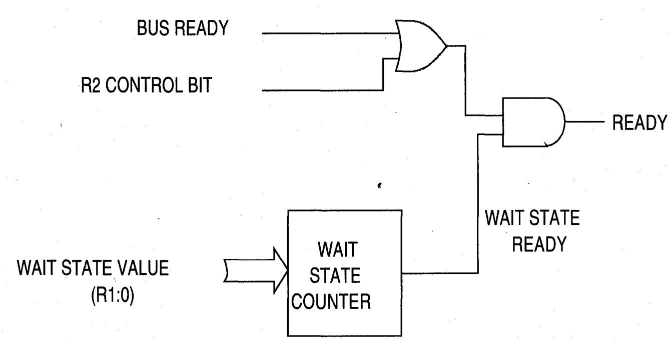


Figure 6.11. Wait State and Ready Control Functions

The R0 and R1 control bits define the number of wait states to insert into the bus cycle. The R2 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 three wait states or less can program R2 active to complete the bus cycle automatically. Devices that may require more than three wait states must program R2 inactive.

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

6.2.6. 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 range overlap (i.e., chip-selects ranges do not need to completely overlap to all go active). There are various reasons for overlapping chip-selects. For example, 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 value and bus ready programming, the following priority scheme exists:

1. If any $\overline{\text{MCS}}$ chip-select is active, the MPCS R2:0 bits are used.
2. If the $\overline{\text{PCS}}$ chip-selects overlap the $\overline{\text{LCS}}$ or $\overline{\text{UCS}}$ chip selects, the LMCS or UMCS R2:0 bits (respectively) are used.

As an example, consider the case where $\overline{\text{MCS3}}$ overlaps $\overline{\text{UCS}}$. $\overline{\text{MCS3}}$ is programmed for two wait states and requires bus ready. $\overline{\text{UCS}}$ is programmed for no wait states and ignores bus ready. An access to the overlapped region results in two wait states and bus ready is required.

Be cautious when overlapping chip selects with different wait state and bus ready programming. Here are two conditions that require special attention to ensure proper system operation.

1. When all overlapping chip-selects ignore bus ready but have different wait states, make sure each chip-select still works properly using the highest wait state value. A system failure may result when the **required** number of wait states does not occur in the bus cycle.
2. If one or more of the overlapping chip-selects requires bus ready, verify the following:
   A. All chip-selects that ignore bus ready work properly using the smallest wait state value.
   B. All chip-selects that ignore bus ready work properly for the longest bus cycle possible.

A system failure may result when not enough or too many wait states occur in the bus cycle.

6.2.7. MEMORY OR I/O BUS CYCLE DECODING

The $\overline{\text{PCS}}$ chip-selects go active for memory or I/O address space. The MS control bit in the MPCS register selects the appropriate address space. 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.

The $\overline{\text{UCS}}$, $\overline{\text{PCS}}$ and $\overline{\text{MCS}}$ chip-selects only go active for memory bus cycles. Chip-selects go active for CPU, DMA Control Unit and Refresh Control Unit initiated bus cycles.

6.3. PROGRAMMING CONSIDERATIONS

When programing the $\overline{\text{PCS}}$ chip-selects active for I/O bus cycles, remember that eight bytes of I/O are reserved by Intel. These eight bytes, located between 00F8H and 00FFH, control the interface to an 80C187 Numerics Coprocessor. A chip-select can overlap this reserved space provided there is no intention of using the 80C187. However, Intel recommends that the base address of the $\overline{\text{PCS}}$ chip-selects not start at 0H in I/O address space to avoid possible future compatibility issues.

An access to the appropriate chip-select register or registers, enables the chip-select. **An access is any read or write operation.** For instance, reading the LMCS register enables the $\overline{\text{LCS}}$ chip-select. However, reading the LMCS register does not ensure it has been programmed correctly.

Do not read any chip-select register unless it has been previously written. Reading a register before programming it enables the chip-select and results in indeterminate operation.

A chip-select can not be disabled once it has been enabled. However, the operating characteristics of the chip-select can be changed by writing the appropriate register.

Three of the MCS chip-selects are alternately used to support the 80C187 Numerics Processor interface when the device is configured in Enhanced Mode. However, the programming characteristics and operation of the $\overline{\text{MCS2}}$ chip-select remain active.

6.4. CHIP-SELECTS AND BUS HOLD

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

The circuit shown in Figure 6.12 allows an external bus master to access a device during bus HOLD.

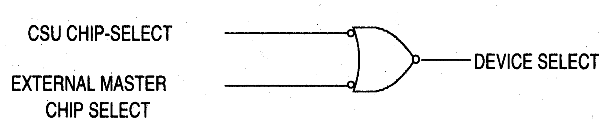


Figure 6.12. Using Chip-Selects During HOLD

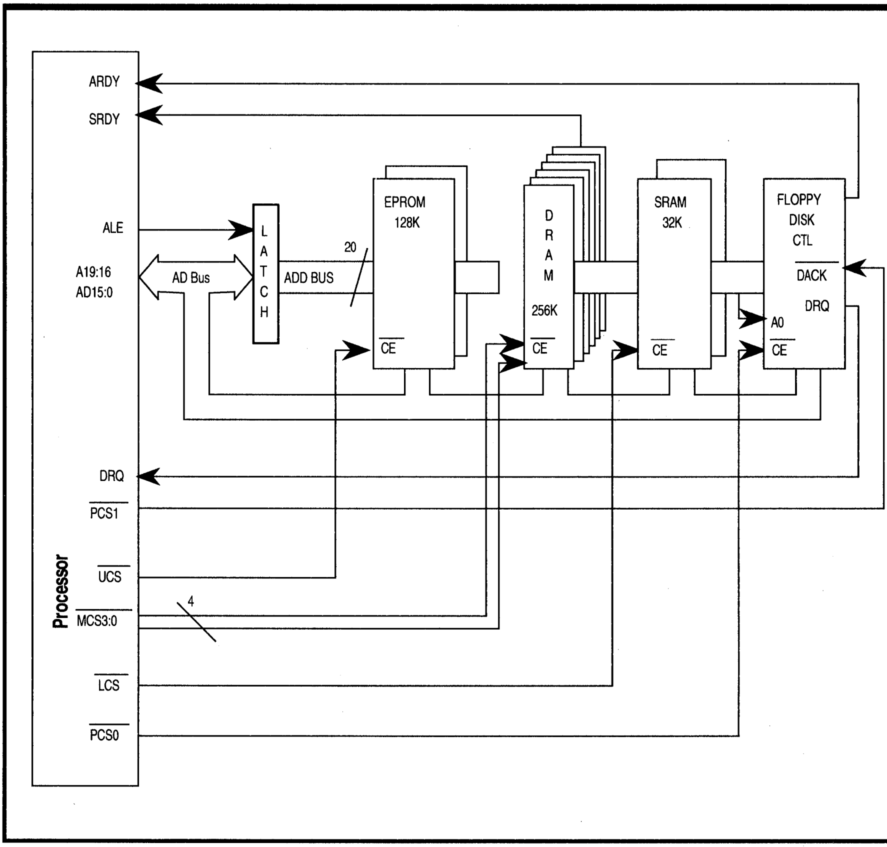


Figure 6.13. Typical System

6.5. 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.

6.5.1. EXAMPLE 1: TYPICAL SYSTEM CONFIGURATION

Figure 6.13 illustrates a block diagram of a typical system design. The EPROM memory has a total size of 128 Kbytes and the SRAM memory has a total size of 32 Kbytes also. The peripherals are mapped to I/O address space.

```
$       TITLE   (Chip-Select Unit Initialization)
$       MOD186  XREF
        NAME    CSU_EXAMPLE_1

;***************************************************************
;                                                              *
;       EXTERNAL REFERENCE FROM THIS MODULE                    *
;                                                              *
;***************************************************************

$       include(PCBMAP.INC)     ; File declares register
                                ; locations and names

;***************************************************************
;                                                              *
;       MODULE EQUATES                                         *
;                                                              *
;***************************************************************
;               CONFIGURATION EQUATES

INTRDY       EQU      0004H    ; Ixternal bus ready modifier
EXTRDY       EQU      0000H    ; External bus ready modifier
IO           EQU      0080H    ; PCS Memory/IO Modifier
ALLPCS       EQU      0040H    ; PCS PCS/Latched Address 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_SIZE   EQU    128                 ; Size in Kbytes
EPROM_BASE   EQU    1024 - EPROM_SIZE   ; Start address in Kbytes
EPROM_WAIT   EQU    2                   ; Wait states
EPROM_RDY    EQU    INTRDY              ; Ignore bus ready

; The UMCS regiser value is calculated using the above
; system constraints and the equations below.

UMCS_VAL     EQU    (EPROM_BASE SHL 6) OR (0C038H) OR
&                   (EPROM_RDY)        OR (EPROM_WAIT)
```

Example 6.1.

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

SRAM_SIZE     EQU     32        ; Size in Kbytes
SRAM_BASE     EQU     0         ; Start address in Kbytes
SRAM_WAIT     EQU     0         ; Wait states
SRAM_RDY      EQU     INTRDY    ; Ignore bus ready

; The LMCS register value is calculated using the above
; system constraints and the equation below

LMCS_VAL      EQU   ((SRAM_SIZE - 1) SHL 6) OR (00038H)     OR
&                    (SRAM_RDY)              OR (SRAM_WAIT)

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

DRAM_BASE     EQU     256       ; Window start address in Kbytes
DRAM_SIZE     EQU     256       ; Window size in Kbytes
DRAM_WAIT     EQU     0         ; Wait states
DRAM_RDY      EQU     INTRDY    ; Ignore bus ready

; The MPCS register is used to program both the MCS and PCS
; chip-selects. Below are the equates for the I/O peripherals
; (also used to program the PACS register).

IO_WAIT       EQU     4         ; IO Wait states
IO_RDY        EQU     INTRDY    ; Ignore bus ready
PCS_SPACE     EQU     IO        ; Put PCSx# in I/O Space
PCS_FUNC      EQU     ALLPCS    ; Generate PCS5# and PCS6#

; The MMCS and MPCS register values are calculated using the
; above system constraints and the equations below

MMCS_VAL      EQU     (DRAM_BASE SHL 6) OR (001F8H)    OR
&                     (DRAM_RDY)        OR (DRAM_WAIT)

MPCS_VAL      EQU     (DRAM_SIZE SHL 5) OR (08038H)    OR
&                     (PCS_SPACE)       OR (PCS_FUNC)  OR
&                     (IO_RDY)          OR (IO_WAIT)
```

Example 6.1. (Continued)

```
; I/O is selected using the PCS0# chip-select.  Wait states
; assume operation at 16MHz.  For this example, the Floppy Disk
; Controller is connect to PCS2# and PCS1# provides the DACK#
; signal.

IO_BASE     EQU     1         ; IO start address in KBytes

; The PACS register value is calculated using the above
; system contraints and the equation below

PACS_VAL    EQU     (IO_BASE SHL 6)  OR (00038H)    OR
&                   (IO_RDY)         OR (IO_WAIT)

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

        ASSUME  CS:CODE
        ASSUME  DS:DATA
        ASSUME  SS:DATA
        ASSUME  ES:DATA

CODE    SEGMENT PUBLIC  'CODE'

;****************************************************************
;                                                               *
;       ENTRY POINT ON POWER UP                                 *
;                                                               *
;****************************************************************

FW_START    LABEL   FAR       ; FORCES FAR JUMP

        CLI                   ; Disable Interrupts

; Place register initialization code here
```

Example 6.1. (Continued)

```
;   SET UP CHIP SELECTS
;
;   UCS - EPROM Select  (Initialized during POWER_ON code)
;   LCS - SRAM  Select  (Set to SRAM Size)
;   PCS - I/O   Select  (PCS0-1 Support Floppy)
;   MCS - DRAM  Select  (Set to DRAM Size)

        MOV  DX, LMCS_REG       ; Set up LCS Register
        MOV  AX, LMCS_VAL
        OUT  DX, AL             ; Remember, BYTE Writes OK

        MOV  DX, MPCS_REG       ; READY FOR PCS LINES 4-6
        MOV  AX, MPCS_VAL       ; AS WELL AS MCS PROGRAMMING
        OUT   X, AL

        MOV  DX, MMCS_REG       ; SET UP DRAM Chip-Select
        MOV  AX, MMCS_VAL
        OUT  DX, AL

        MOV  DX, PACS_REG       ; SET UP IO Chip-Select
        MOV  AX, PACS_VAL
        OUT  DX, AL

CODE    ENDS

;       POWER ON RESET CODE TO GET STARTED

        ASSUME  CS:POWER_ON

        POWER_ON SEGMENT AT 0FFFFH

        MOV  DX, UMCS_REG       ; Point to UMCS Register
        MOV  AX, UMCS_VAL       ; Reprogram UMCS to match
        OUT  DX, AL             ; system requirements
        JMP  FW_START           ; Jump to init code

POWER_ON  ENDS
```

Example 6.1. (Continued)

```
;*************************************************************
;                                                           *
;       DATA SEGMENT                                        *
;                                                           *
;*************************************************************

DATA    SEGMENT PUBLIC  'DATA'

        DD   256 DUP (?)      ; Reserved for Interrupt Vectors

;Place memory variables Here

        DW   500 DUP (?)      ; Stack Allocation

STACK_TOP     LABEL   WORD

DATA    ENDS

; Program ends

        END
```

Example 6.1. (Continued)

Refresh Control Unit 7

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.

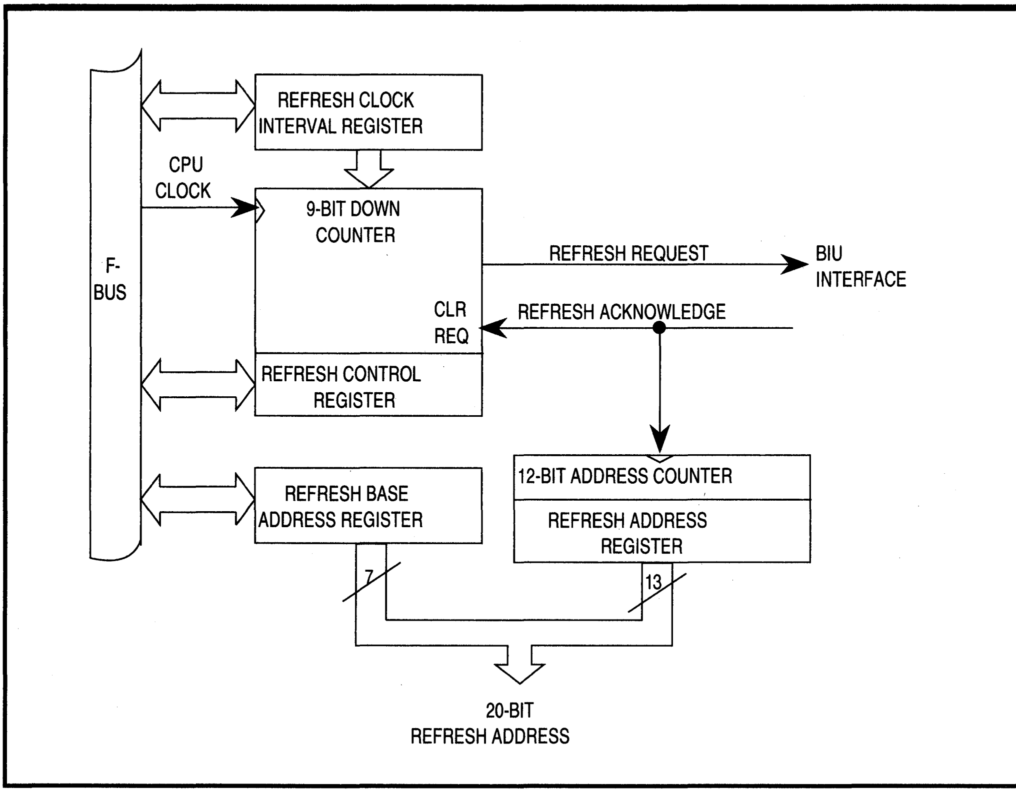


Figure 7.1. Refresh Control Unit Block Diagram

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 nine-bit address counter forms the refresh addresses, supporting any dynamic memory devices with up to nine rows of memory cells (nine refresh address bits). This includes all practical DRAM sizes for the processor's one 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 flow chart form.

The 9-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. Power-Save Mode divides the Refresh Control Unit clocks, so reprogramming the Refresh Interval Register becomes necessary.

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/HLDA bus arbitration protocol (see Section 7.8).

The 9-bit refresh clock counter does not wait until the BIU services the refresh request to continue counting. This operation ensures 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.

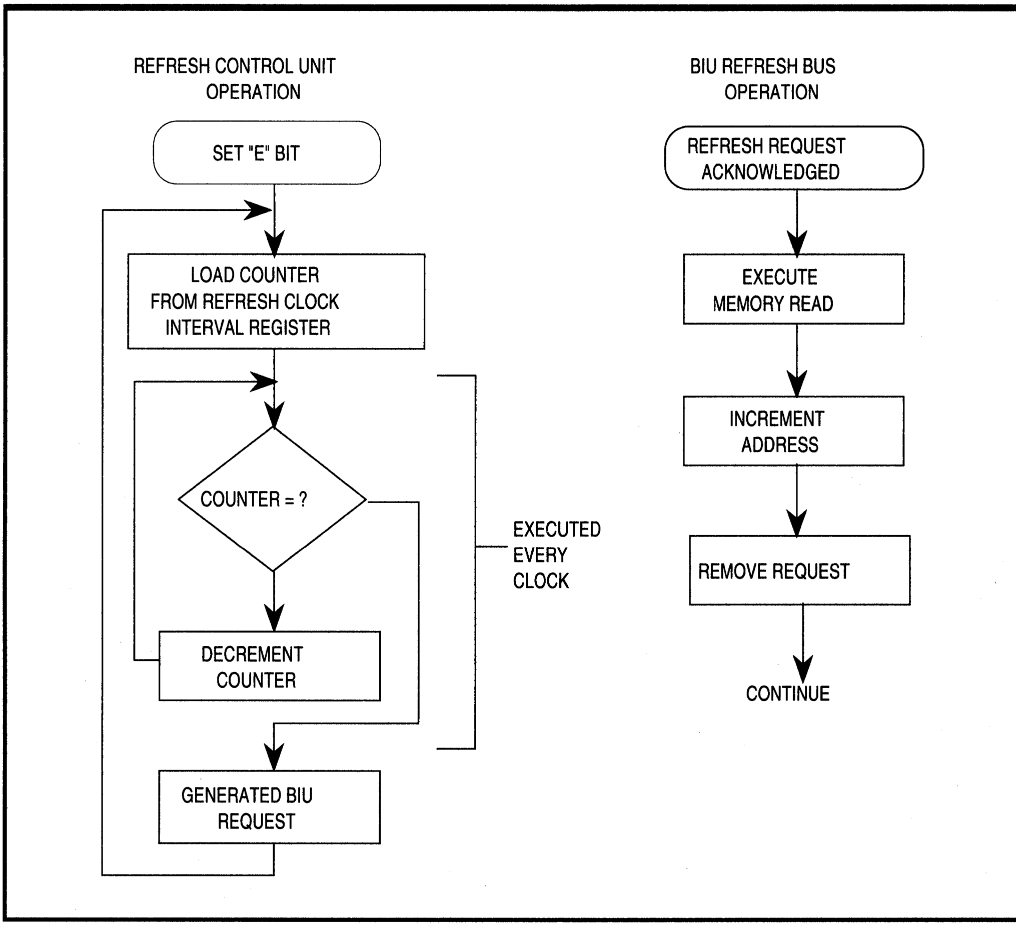


Figure 7.2. Refresh Control Unit Operation Flow Chart

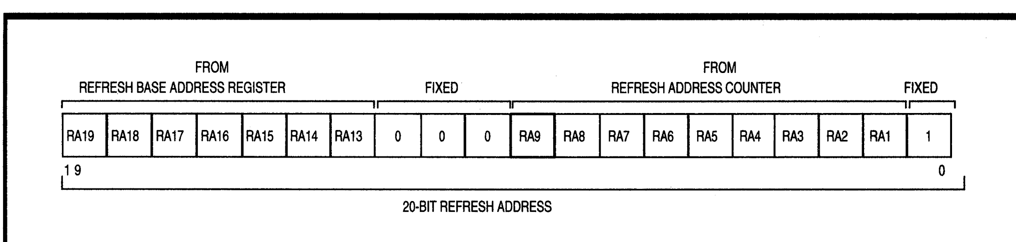


Figure 7.3. Refresh Address Formation

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 described in Section 7.7.2.1.

Refresh address bits RA12:10 are always zero. A linear-feedback shift counter generates address bits RA9:1. The counter does not increment linearly from 0 through 1FFH. However, the counting algorithm cycles uniquely through all possible 9-bit values. It only matters that each row of DRAM memory cells gets refreshed at a specific interval. The order of the rows is unimportant.

Address bit A0 is fixed at zero 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 indicated in Table 7.1. The 16-bit bus processor drives both the $\overline{\text{BHE}}$ and A0 pins high during refresh cycles. These signals may be AND'ed in a DRAM controller to detect a refresh bus cycle. The 8-bit bus version replaces the $\overline{\text{BHE}}$ pin with $\overline{\text{RFSH}}$, which is low during refresh cycles. $\overline{\text{RFSH}}$ and $\overline{\text{BHE}}$ timings are the same. A0 is also high during refresh cycles on the 8-bit bus processor.

Table 7.1. Identification of Refresh Bus Cycles

DATA BUS WIDTH	$\overline{\text{BHE}}/\overline{\text{RFSH}}$	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 controller asserts a Row Address Strobe ($\overline{RAS}$), which latches the row address inside the DRAMs.
3. The controller supplies a column address to the DRAMs.
4. The 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 will not be 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.

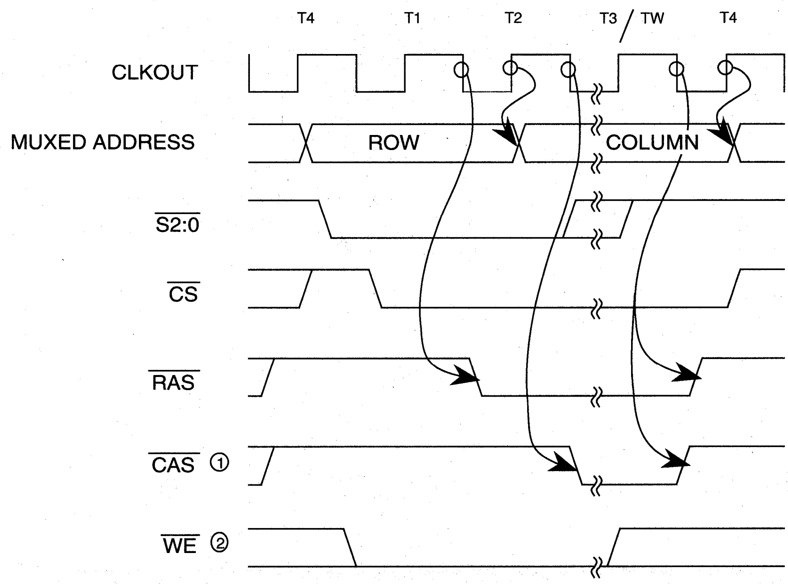
DRAM controllers may be either clocked or unclocked. An unclocked DRAM controller requires a tapped digital delay line to derive the proper timings.

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). The cycle begins with presentation of the row address. $\overline{RAS}$ should go active on the falling edge of T_2. At the rising edge of T_2, the address lines should switch to a column address. $\overline{CAS}$ goes active on the falling edge of T_3. 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 T_4. At the rising edge of T_4, 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.



NOTES:

1. $\overline{CAS}$ is unnecessary for refresh cycles only.
2. $\overline{WE}$ is necessary for write cycles only.

Figure 7.4. Suggested DRAM Control Signal Timing Relationships

$$\frac{R_{Period}\,(\mu s) \times f(MHz)}{\#\ \text{Refresh Rows} + \#\ (\text{Refresh Rows} \times \%\ \text{Overhead})} = \text{RFTIME Register Value}$$

R_{Period} = Maximum refresh period specified by DRAM manufacturer (microseconds).

f = Operating frequency in MHz.

Refresh 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

7.7.1. 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 indicated number of cycles is the same as the number of rows. Multiply the specified refresh period (convert to microseconds) by the microprocessor's CLKOUT frequency (MHz). Then divide the result by the number of rows in the DRAM. Figure 7.5 shows the formula.

Bus latency is the time the Refresh Control Unit needs to gain control of the bus. Reduce the calculated refresh interval by one to five percent to compensate. If an external bus master will be extremely slow to release the bus, reduce the interval even more. At standard operating frequencies, DRAM refresh bus overhead totals two or three percent of the total bus bandwidth.

If the processor enters Power-Save Mode, the refresh rate must increase to offset the reduced CPU clock rate to preserve memory. At lower frequencies, the refresh bus overhead increases. At frequencies less than about 1.5 MHz, the Bus Interface Unit will spend almost all its time running refresh cycles. There may not be enough bandwidth left for the processor to perform other activities, especially if the processor must share the bus with an external master.

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.

7.7.2.1. REFRESH BASE ADDRESS REGISTER

The Refresh Base Address Register (see Figure 7.6) programs the base (upper 7 bits) of the refresh address. Seven-bit mapping places the refresh address at any 4 Kbyte boundary within the one Mbyte address space. When the partial refresh address from the 9-bit address counter (see Section 7.3) passes 1FFH, the Refresh Control Unit does not increment the refresh base address.

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 1, the Refresh Control Unit generates a refresh request and the counter again loads the value from the register.

Register Name: Refresh Base Address Register
Register Mnemonic: RFBASE (MDRAM)
Register Function: Determines upper 7 bits of refresh address.

15															0
RA19	RA18	RA17	RA16	RA15	RA14	RA13									

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. Always program reserved register bits with a "0" to insure proper device functionality and compatibility with future Intel products.

Figure 7.6. Refresh Base Address Register

Register Name: Refresh Clock Interval Register
Register Mnemonic: RFTIME (CDRAM)
Register Function: Sets refresh rate.

15															0
							RC8	RC7	RC6	RC5	RC4	RC3	RC2	RC1	RC0

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. Always program reserved register bits with a "0" to insure proper device functionality and compatibility with future Intel products.

Figure 7.7. Refresh Clock Interval Register

Register Name: Refresh Control Register
Register Mnemonic: RFCON (EDRAM)
Register Function: Controls Refresh Unit operation.

15															0
REN							RC8	RC7	RC6	RC5	RC4	RC3	RC2	RC1	RC0

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 which triggers refresh requests.

NOTE: Reserved register bits are shown with gray shading. Always program reserved register bits with a "0" to insure proper device functionality and compatibility with future Intel products.

Figure 7.8. Refresh Control 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 9-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.

7.7.3. PROGRAMMING EXAMPLE

Example 7.1 contains sample code to initialize the Refresh Control Unit. Example 5.2 shows the additional code to reprogram the Refresh Control Unit upon entering Power-Save Mode.

```
$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      equu   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
                                      ;will be modified
            push cx
            push dx
            push di
```

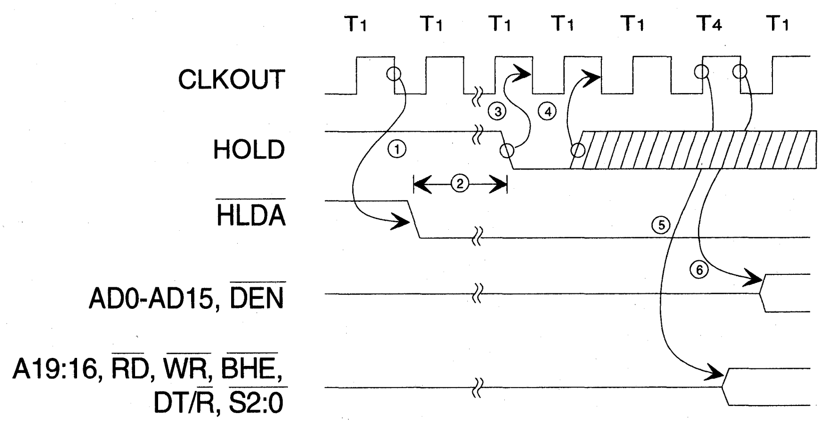
Example 7.1. Refresh Control Unit Intialization Code

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

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

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

            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. Refresh Control Unit Initialization Code (Continued)

7.8. REFRESH OPERATION AND BUS HOLD

When another bus master controls the bus, the processor keeps 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 HLDA signal inactive, indicating to the current bus master that it wishes to regain bus control (see Figure 7.9). The BIU begins a refresh bus cycle only after the alternate master removes HOLD. The user must design the system so the processor can regain bus control. If the alternate master asserts HOLD after the processor starts the refresh cycle, the CPU will give up the bus afterwards.



NOTES: 1. HLDA deasserted; signaling needs 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

Figure 7.9. Regaining Bus Control to Run a DRAM Refresh Bus Cycle

Interrupt Control Unit 8

CHAPTER 8
INTERRUPT CONTROL UNIT

The 80C186 Modular Core has a single maskable interrupt input (See Section 2.3.1.2). An Interrupt Control Unit is needed to expand the interrupt capabilities beyond a single input. To fulfill this function, the Interrupt Control Unit has two different modes of operation; Master Mode and Slave Mode.

In Master Mode, the Interrupt Control Unit processes all maskable interrupt sources and presents them to the CPU through the single maskable interrupt input. The Interrupt Control Unit synchronizes and prioritizes interrupt sources and provides the interrupt type vector to the CPU. The interrupts can originate from on-chip peripherals and from four external interrupt pins. Most systems use Master Mode.

In Slave Mode, an external 8259A interrupt controller acts as the master interrupt controller. The 8259A now actually controls the maskable interrupt input to the CPU. The Interrupt Control Unit is only responsible for processing the on-chip interrupt sources and must request service from the external 8259A.

Features of the Interrupt Control Unit are:

- Programmable priority of each interrupt source
- Support for polled operation
- Individual masking of each interrupt source
- Nesting of interrupt sources
- External 8259As can be used for expanding external interrupt sources (Cascade Mode)

8.1. FUNCTIONAL OVERVIEW

All microprocessor systems must communicate in some way with the external world. A typical system may have a set of peripherals, for example, a keyboard, communications port and a display. Each peripheral requires the attention of the CPU at different times. There are two distinct ways to process peripheral I/O requests; polling and interrupts.

Polling requires the CPU to check each peripheral in the system periodically to see if an I/O request is pending. However, polling is not a very efficient use of CPU time and in most cases is detrimental to system throughput.

Interrupts eliminate polling by allowing the peripheral to signal the CPU that it has an I/O request pending. The CPU then stops execution of the current task, saves its state and begins executing the peripheral servicing routine (interrupt handler). At the end of the interrupt handler, the CPU restores its original state and returns to executing the original task.

The Interrupt Control Unit is responsible for processing interrupts from multiple peripherals and presenting them to the CPU in an orderly and defined fashion.

8.2. MASTER MODE

A block diagram of the Interrupt Control Unit in Master Mode is shown in Figure 8.1.

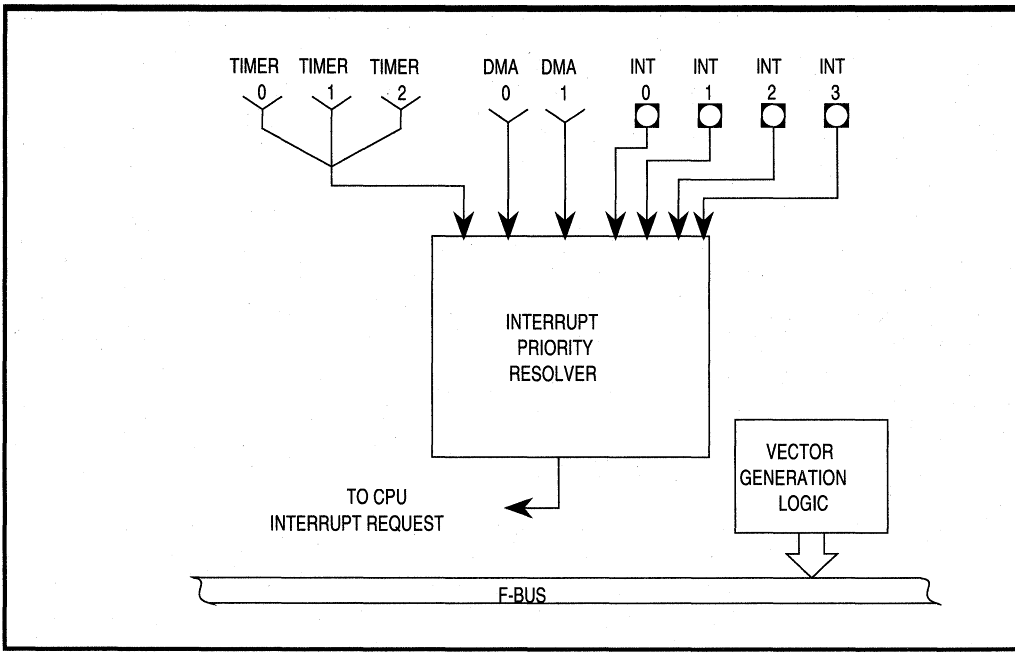


Figure 8.1. Interrupt Control Unit Block Diagram

8.2.1. GENERIC FUNCTIONS IN MASTER MODE

There are several functions of the Interrupt Control Unit which are common among most interrupt controllers. This section covers how these generic functions are implemented on the Interrupt Control Unit.

8.2.1.1. INTERRUPT MASKING

There are several instances where a programmer may want to disable an interrupt source temporarily. Executing time-critical sections of code or servicing a high priority task are common examples of when interrupt sources may need to be disabled. This is called interrupt masking. All interrupts from the Interrupt Control Unit may be globally masked or selectively masked on an individual basis.

8.2.1.1.1. GLOBAL MASKING OF INTERRUPT SOURCES

The Interrupt Enable Bit in the Program Status Word globally enables or disables the maskable interrupt request from the Interrupt Control Unit. The programmer controls the Interrupt Enable Bit by using the STI (Set Interrupt) and the CLI (Clear Interrupt) instructions.

8.2.1.1.2. INDIVIDUAL MASKING OF INTERRUPT SOURCES

In addition to 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. By setting or clearing a bit in the Interrupt Mask Register, the programmer can selectively mask or unmask the corresponding interrupt source.

8.2.1.2. INTERRUPT PRIORITY

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

An interrupt source is assigned a priority between zero and seven. Zero is the highest possible priority and seven is the lowest. After reset, the interrupts default to the priority shown in Table 8.1. Because the timers share an interrupt source, they also share a priority. Within the assigned priority, they are prioritized relative to each other. Timer 0 has the highest relative priority, Timer 2 the lowest.

Different priorities can be assigned for each source. This is done by programming the Interrupt Control Register with a new priority. The priority must be between zero and seven. Interrupt sources can be programmed to share the same priority. The Interrupt Control Unit handles this by using the default priorities within the shared priority level. For example, assume INT0 and INT1 are programmed to priority seven. INT0 is serviced first because it has the higher default priority.

Interrupt sources can also be masked on the basis of their priority. The Priority Mask Register masks all interrupts with a lower priority than its programmed value. After reset, the Priority Mask Register contains priority seven, effectively enabling interrupts of any priority. The register can then be programmed with any valid priority.

Table 8.1. Default Interrupt Priorities

Interrupt Name	Relative Priority
Timer 0	0 (a)
Timer 1	0 (b)
Timer 2	0 (c)
DMA0	1
DMA1	2
INT0	3
INT1	4
INT2	5
INT3	6

8.2.1.2.1. OPERATION WHEN INTERRUPT NESTING IS NOT ENABLED

When entering an interrupt handler, the Program Status Word is pushed onto the stack. The Interrupt Enable Bit is cleared. The processor enters all interrupt handlers with maskable interrupts disabled. Maskable interrupts will not be enabled again 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; an interrupt handler must finish before another executes.

The simplest way to use the Interrupt Control Unit is when nesting is not needed. 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 simplest terms, the Interrupt Control Unit asserts the maskable interrupt request to the CPU and waits for the interrupt acknowledge. When the Interrupt Control Unit receives the acknowledge, it presents the highest priority unmasked interrupt type at that time to the CPU. The CPU then executes the interrupt handler for that interrupt. Because the Interrupt Enable Bit is never set within the interrupt handler, the interrupt handler can never be interrupted.

8.2.1.2.2. OPERATION WHEN NESTING INTERRUPTS

The function of the Interrupt Control Unit is more complicated when nesting interrupts. An interrupt now can occur within an interrupt handler. The term used here is an interrupt preempting another interrupt. The following rules apply for nesting interrupts:

- An interrupt source can only preempt other interrupts of equal or higher priority.
- An interrupt source cannot preempt itself. The interrupt handler must finish executing before the interrupt is serviced again. (An exception to this is Special Fully Nested Mode, which is covered in Section 8.3.3.1)

8.3. MASTER MODE OPERATION

This section covers the process in which the Interrupt Control Unit receives interrupts and asserts the Maskable Interrupt Request to the CPU.

8.3.1. TYPICAL INTERRUPT SEQUENCE

When the Interrupt Control Unit first detects an interrupt, it sets the corresponding bit in the Interrupt Request Register. That interrupt is pending or waiting to be serviced. The Interrupt Control Unit checks all pending interrupt sources. If the interrupt is not masked and it meets the priority criteria (see Section 8.3.2 on Priority Resolution), the Interrupt Control Unit asserts the maskable interrupt request to the CPU.

The Interrupt Control Unit then waits for the interrupt acknowledge from the CPU. At that time, it passes the interrupt type to the CPU and the interrupt processing sequence takes place. See Section 2.3.1 for a detailed explanation of the interrupt processing sequence. The Interrupt Control Unit always passes the highest priority interrupt vector **at the time** the acknowledge is received. If a higher priority interrupt occurs before the interrupt acknowledge, the higher priority interrupt has precedence.

When the interrupt acknowledge occurs, the corresponding bit in the Interrupt Request Register is cleared. The corresponding bit in the In-Service Register is set. The In-Service Register keeps track of which interrupt handlers are being processed. At the end of Interrupt Handler, the programmer must explicitly clear the bit in the In-Service Register by issuing an End-Of-Interrupt (EOI) command. If the bit remains set, the Interrupt Control Unit **cannot** process any more interrupts from that source.

8.3.2. PRIORITY RESOLUTION

The criteria for asserting 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 bit is set in the Interrupt Request Register. The Interrupt Control Unit then asserts the maskable interrupt request to the CPU based on the following criteria:

1. The interrupt is not masked.
2. The interrupt has higher priority than the Priority Mask Register.
3. The interrupt must not have its own In-Service bit set.
4. An interrupt has equal or higher priority than any interrupt whose In-Service bit is set.

The In-Service Register keeps track of any currently executing interrupt handler. The Interrupt Control Unit uses this information to decide if another interrupt source has enough priority to preempt an interrupt handler that is currently executing.

The following example illustrates the priority resolution:

The initial conditions are:

- The Interrupt Control Unit has been initialized.
- There are no pending interrupts.
- No bits are set in the In-Service Register.
- All interrupts are unmasked and the Interrupt Enable bit is set.
- The default priority scheme is used.
- The Priority Mask Register is set to the lowest priority (seven).

1. A low to high transition on INT0 sets its bit in the Interrupt Request Register. The interrupt is now pending.
2. Because INT0 is the only interrupt pending, it must meet 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. The Interrupt Control Unit passes the interrupt type (in this case type 12) to the CPU.
4. The Interrupt Control Unit clears the INT0 in the Interrupt Request Register and sets the INT0 bit in the In-Service Register.
5. The CPU executes the interrupt processing sequence and begins executing the interrupt handler for INT0.
6. During execution of the interrupt handler, a low to high transition on INT3 sets its bit in the Interrupt Request Register.
7. INT3 has lower priority than INT0, whose interrupt handler is currently executing (INT0's In-Service bit is set). INT3 does not meet the priority criteria and thus no interrupt request is sent to the CPU. If INT3 had been programmed with an equal or higher priority than INT0, the interrupt request would have been sent to the CPU. INT3 remains pending in the Interrupt Request Register.
8. The INT0 interrupt handler completes and an EOI command clears the INT0 bit in the In-Service Register.
9. INT3 is still pending and now meets all the priority criteria. An interrupt request is sent to the CPU and the process begins again.

8.3.2.1. INTERRUPTS WHICH SHARE A SINGLE SOURCE

Multiple interrupt requests can share a single source input to the Interrupt Control Unit (the three timer interrupts, for example). Although these interrupts share a source input, each has its own interrupt vector. The actual vectoring sequence is transparent to the user (i.e., when a Timer0 interrupt occurs, the Timer0 interrupt handler gets executed). The application consequences of how these interrupts get prioritized and serviced is covered in this section. We will use the three timer interrupts as an example.

The Interrupt Status Register acts as a second level request register to process the three timer interrupts. The Interrupt Status Register contains a bit for each timer interrupt. Lets assume a timer interrupt occurs. The specific bit for that timer in the Interrupt Status Register and the shared timer interrupt bit in the Interrupt Request Register are both set. Now the shared timer interrupt is processed like any other interrupt source. Multiple timer interrupt bits can be set at one time in the Interrupt Status Register.

When the shared interrupt is acknowledged, the highest priority timer interrupt *at that time* gets serviced first (see Table 8.1). The highest priority timer bit is cleared in the Interrupt Status Register. Any other timer interrupts remain pending and their bits set. If only one timer interrupt is pending, the timer bit in the Interrupt Request Register is also cleared. Otherwise, it remains set, signalling other timer interrupts are pending.

The shared In-Service Bit is set when the timer interrupt is acknowledged. *No other timer interrupts can occur when the In-Service Bit is set.* For example, assume a lower priority timer interrupt is being serviced and a higher priority timer interrupt occurs. The In-Service Bit is already set for the shared timer interrupt. The higher priority timer interrupt remains pending until the lower priority timer interrupt handler is finished and the In-Service Bit cleared.

8.3.3. CASCADING WITH EXTERNAL 8259As

For some applications, the number of external interrupt pins on the Interrupt Control Unit is not enough. The Interrupt Control Unit has Cascade Mode which expands the number of external interrupt pins using 8259A interrupt controllers. The INT2/$\overline{\text{INTA0}}$ and INT3/$\overline{\text{INTA1}}$ have two functions. They can function as external interrupt pins or as interrupt acknowledge outputs in Cascade Mode. $\overline{\text{INTA0}}$ is the acknowledge for INT0 and $\overline{\text{INTA1}}$ is the acknowledge for INT1 as shown in Figure 8.2.

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

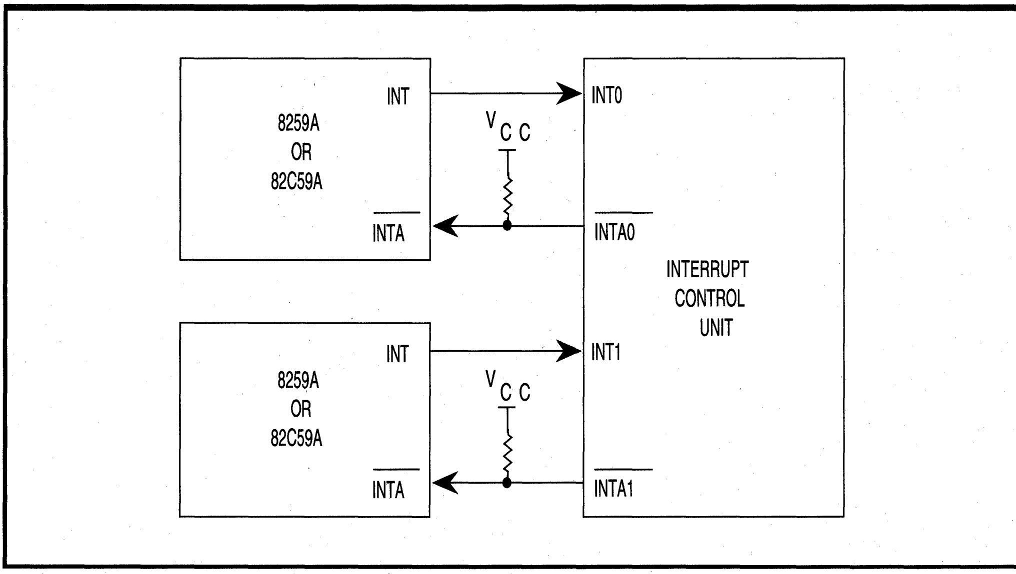


Figure 8.2. Using 8259As in Cascade Mode

8.3.3.1. SPECIAL FULLY NESTED MODE

Special Fully Nested Mode is an optional feature normally used with Cascade Mode and is only applicable to INT0 and INT1. In Special Fully Nested Mode, a request from an interrupt source **is serviced** even if its In-Service Bit is set.

In Cascade Mode, up to eight external interrupts share a single interrupt pin under the control of an 8259A. Special Fully Nested Mode allows the priority structure of the 8259A to be maintained. For example, let's assume the CPU is currently 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 pin to the Interrupt Control Unit. Special fully Nested Mode would allow that 8259A interrupt to be serviced even though the In-Service Bit is already set for that interrupt source. A higher priority interrupt has preempted a lower priority interrupt therefore fully maintaining interrupt nesting.

Special Fully Nested Mode can still be used without Cascade Mode. This allows a single external interrupt pin, (either INT0 or INT1) to preempt itself.

8.3.4. 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 get the interrupt vector address in the interrupt vector table. See Section 2.3.1.

The interrupt types for all the 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 will be the CPU fetching from the interrupt vector table.

Table 8.2. Fixed Interrupt Types

Interrupt Name	Interrupt Type
Timer 0	8
Timer 1	18
Timer 2	19
DMA0	10
DMA1	11
INT0	12
INT1	13
INT2	14
INT3	15

In Cascade Mode, the external 8259A supplies the interrupt type to the CPU. Therefore, the CPU runs an external interrupt acknowledge cycle (see Section 3.5.3) to fetch the interrupt type from the 8259A.

8.3.5. POLLING

In some applications, it is desirable to poll the Interrupt Control Unit. The CPU asks or polls, the Interrupt Control Unit for any pending interrupts. The user can then service interrupts whenever it is convenient. The Interrupt Control Unit has the Poll and Poll Status Registers to support polling.

By reading the Poll Register, the user gets the type of the highest priority pending interrupt. Now the user must call that interrupt handler. Reading the poll register also acknowledges the interrupt. The specific bit in the Request Register is cleared and the bit in the In-Service Register is set. The Poll Status Register has the same format as the Poll Register. Reading the Poll Status Register *does not* acknowledge the interrupt.

8.3.6. EDGE AND LEVEL TRIGGERING

The external interrupt pins (INT3-0) are programmable for either edge or level triggering. Both types of triggering are active high.

Edge triggering is defined as a zero to one transition on an external interrupt pin. The pin must remain high until after the CPU acknowledges the interrupt. The external interrupt pin must go low again to reset the edge detect circuitry (see the data sheet for timing information). No further interrupts will occur unless the external interrupt pin goes low after being acknowledged.

Level triggering is defined as a valid logic one on the external interrupt pin. The logic one must remain until after the CPU acknowledges the interrupt. Unlike edge triggering, level triggering will continue to generate interrupts if the pin remains high. A level triggered external interrupt pin must be deasserted before the EOI command or another interrupt occurs.

8.3.7. ADDITIONAL LATENCY AND RESPONSE TIME OF MASTER MODE

The Interrupt Control Unit adds five clocks to the interrupt latency of the CPU. The Interrupt Control Unit also adds an extra 13 clocks to the interrupt response time when the Cascade Mode is used because the interrupt acknowledge bus cycles must be run. (See Figure 8.3).

Section 2.3.3 defines the interrupt latency and interrupt response time of the 80C186 Modular CPU.

		Clocks	
INTERRUPT PRESENTED TO INTERRUPT CONTROL UNIT →			
		5	
INTERRUPT PRESENTED TO CPU →			
	INTA	4	CASCADE MODE ONLY
	IDLE	2	CASCADE MODE ONLY
	INTA	4	CASCADE MODE ONLY
	IDLE	5	CASCADE MODE ONLY
	READ IP	4	
	IDLE	3	(5 IF NOT CASCADE MODE)
	READ CS	4	
	IDLE	4	
	PUSH FLAGS	4	
	IDLE	3	
	PUSH CS	4	
	PUSH IP	4	
	IDLE	5	
FIRST INSTRUCTION FETCH FROM INTERRUPT ROUTINE →			
		Total 55	

Figure 8.3. Interrupt Control Unit Latency and Response Time

8.4. MASTER MODE INTERRUPT UNIT PROGRAMMING

The Peripheral Control Block map of the Interrupt Control Unit registers in Master Mode is shown in Table 8.3.

Table 8.3. Interrupt Control Unit Registers in Master Mode

Register Name	Offset Address
INT3 Control Register	3EH
INT2 Control Register	3CH
INT1 Control Register	3AH
INT0 Control Register	38H
DMA1 Control Register	36H
DMA0 Control Register	34H
Timer Control Register	32H
Interrupt Status Register	30H
Interrupt Request Register	2EH
In-Service Register	2CH
Priority Mask Register	2AH
Interrupt Mask Register	28H
Poll Status Register	26H
Poll Register	24H
EOI Register	22H

8.4.1. INTERRUPT CONTROL UNIT REGISTER DEFINITIONS

The following sections define the bit-level functionality of the individual Interrupt Control Unit Registers.

8.4.1.1. INTERRUPT CONTROL REGISTERS

Each interrupt source has its own Interrupt Control Register (See Figures 8.4-8.6). Each Interrupt Control Register has three bits which can be programmed with the priority level for the interrupt source (see Figure 8.4). Also, each register has a mask bit which enables the interrupt source. The mask bit is the same bit in the Interrupt Mask Register. Modifying one bit in either register also modifies the other bit.

Register Name: Interrupt Control Register (Internal Sources)
Register Mnemonic: TCUCON, DMA0CON, DMA1CON
Register Function: Control Register for the internal interrupt sources.

15															0
												MSK	PM2	PM1	PM0

BIT MNEMONIC	BIT NAME	RESET STATE	FUNCTION
MSK	*Interrupt Mask*	1	Cleared to enable interrupts from this source.
PM2:0	*Priority Level Field*	111	Sets 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 insure compatibility with future Intel products.

Figure 8.4. Interrupt Control Register Template for Internal Sources

Each Interrupt Control Register for the external interrupt pins also has a LVL bit (see Figure 8.5). The LVL bit selects between Level-triggered and Edge-triggered mode for the corresponding external interrupt pin. In Edge-triggered Mode, a low to high transition causes the interrupt. The pin must remain low at least one clock before the low to high transition. The interrupt pin must still must remain asserted until the CPU acknowledges the interrupt. Otherwise, the interrupt is lost.

In Level-triggered Mode, an interrupt pin left asserted after the EOI causes another interrupt. Level-triggered Mode is useful when interrupt requests are wire-ORed to a single interrupt pin.

Register Name: Interrupt Control Register (Non-cascadable external pins)
Register Mnemonic: I2CON, I3CON
Register Function: Control Register for non-cascadable external interrupt pins.

15															0
											LVL	MSK	PM2	PM1	PM0

BIT MNEMONIC	BIT NAME	RESET STATE	FUNCTION
LVL	*Level-trigger*	0	0 = Edge-triggered mode 1 = Level-triggered mode
MSK	*Interrupt Mask*	1	Cleared to enable interrupts from this source.
PM2:0	*Priority Level Field*	111	Sets 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 insure compatibility with future Intel products.

Figure 8.5. Interrupt Control Register Template for Non-Cascadeable Interrupt Pins

Level-triggered mode must be used when external 8259As are cascaded into the Interrupt Control Unit.

To support external 8259As, the INT0 and INT1 Interrupt Control Registers have the CAS and SFNM bits (see Figure 8.6). The CAS bit enables Cascade Mode operation and the SFNM bit enables the Special Fully Nested Mode.

Register Name: Interrupt Control Register (Cascadable external pins)
Register Mnemonic: I0CON, I1CON
Register Function: Control register for the cascadable external interrupt pins.

15															0
									SFNM	CAS	LVL	MSK	PM2	PM1	PM0

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	0 = Edge-trigger mode 1 = Level-trigger mode
MSK	*Interrupt Mask*	1	Cleared to enable interrupts from this source.
PM2:0	*Priority Level Field*	111	Sets the priority level for this interrupt source.

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

Figure 8.6. Interrupt Control Register Template for Cascadeable Interrupt Pins

8.4.1.2. THE INTERRUPT REQUEST REGISTER

The Interrupt Request Register has seven bits, one for each interrupt source (see Figure 8.7). When an interrupt occurs, the corresponding bit is set in the Interrupt Request Register. The bit is set whether the interrupt is masked or unmasked. The bit is cleared when the interrupt is acknowledged.

Register Name: Interrupt Request Register
Register Mnemonic: REQST
Register Function: Stores pending interrupt requests.

15															0
								INT3	INT2	INT1	INT0	DMA1	DMA0		TMR

BIT MNEMONIC	BIT NAME	RESET STATE	FUNCTION
INT3:0	*External Interrupts*	0	When set, the corresponding INT pin has an interrupt pending.
DMA1:0	*DMA Interrupts*	0	DMA channel interrupt requests. When set, the corresponding DMA channel has an interrupt pending.
TMR	*Timer Interrupt*	0	Timer/Counter Unit interrupt request. When set, the TCU has an interrupt pending.

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

Figure 8.7. Interrupt Request Register

For the external interrupt pins, the request must remain asserted until the interrupt is acknowledged. Otherwise, that bit in the Interrupt Request Register will be cleared and the interrupt will not be serviced.

8.4.1.3. INTERRUPT MASK REGISTER

The Interrupt Mask Register contains a mask bit for each interrupt source (see Figure 8.8). The bit for an interrupt source is the same as the mask bit in the Interrupt Control Register. The Interrupt Mask Register may be read or written.

Register Name: Interrupt Mask Register
Register Mnemonic: IMASK
Register Function: Masks individual interrupt sources.

15															0
								INT3	INT2	INT1	INT0	DMA1	DMA0		TMR

BIT MNEMONIC	BIT NAME	RESET STATE	FUNCTION
INT3:0	*External ilterrupts*	1111	Set to mask interrupt requests from the corresponding INT pin.
DMA1:0	*DMA Interrupts*	11	Set to mask interrupt requests from the corresponding DMA channel.
TMR	*Timer Interrupt*	1	Set to mask interrupt requests from the Timer/Counter Unit.

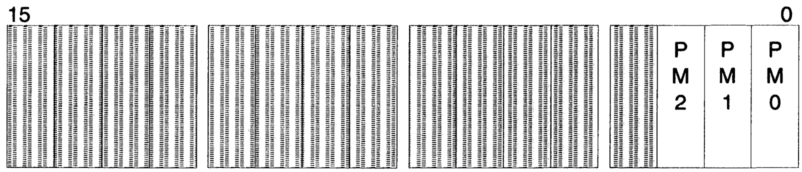
NOTE: Reserved register bits are shown with gray shading. Reserved bits must be written to a logic zero to insure compatibility with future Intel products.

Figure 8.8. Interrupt Mask Register

8.4.1.4. PRIORITY MASK REGISTER

The Priority Mask Register (see Figure 8.9) indicates the lowest interrupt priority that will be serviced. Any interrupts with a lower priority will be masked. After reset, the Priority Mask Register is set to the lowest priority (seven) to enable interrupts of any priority.

Register Name: Priority Mask Register
Register Mnemonic: PRIMSK
Register Function: Masks all interrupts with a lower priority.



BIT MNEMONIC	BIT NAME	RESET STATE	FUNCTION
PM2:0	*Priority Mask Field*	111	Interrupts with a lower priority than PM2:0 will not be serviced.

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

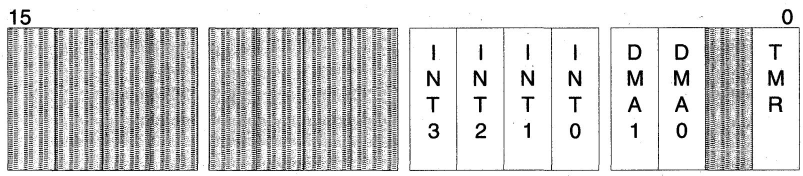
Figure 8.9. Priority Mask Register

8.4.1.5. IN-SERVICE REGISTER

The In-Service Register (see Figure 8.10) has a bit for each interrupt source. The bits indicate which source's interrupt handlers are executing. The bit in the In-Service Register is set when the interrupt is acknowledged. The bit is then cleared at the end of the interrupt handler by the 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 currently in process.



BIT MNEMONIC	BIT NAME	RESET STATE	FUNCTION
INT3:0	*External Interrupts*	0	When set, the corresponding INT pin's interrupt request is in-service.
DMA1:0	*DMA Interrupts*	0	When set, the corresponding DMA interrupt request is in-service.
TMR	*Timer Interrupt*	0	When set, the corresponding Timer interrupt request is in-service.

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

Figure 8.10. In-Service Register

8.4.1.6. POLL AND POLL STATUS REGISTERS

The Poll and Poll Status Registers (see Figures 8.11 and 8.12) support polling the Interrupt Control Unit. They indicate an interrupt is pending and also the type of the highest priority pending interrupt. The programmer reads these registers to service interrupts through software.

The Poll Register and Poll Status Register both contain the same information. If an interrupt of sufficient priority is pending, the IREQ bit is set and the highest priority vector type is contained in bits VT4:0.

Register Name: Poll Register
Register Mnemonic: POLL
Register Function: Read to check for pending interrupts when polling.

15															0
IREQ											VT4	VT3	VT2	VT1	VT0

BIT MNEMONIC	BIT NAME	RESET STATE	FUNCTION
IREQ	*Interrupt Request*	0	Set if an interrupt is pending.
VT4:0	*Poll Status*	0	Indicate the type of the highest pending interrupt. Reading the Poll Register acknowledges highest pending interrupt.

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

Figure 8.11. Poll Register

Reading the Poll Register acknowledges the pending interrupt the same as if the CPU had started the interrupt vectoring sequence. The processor will not actually run any interrupt acknowledge sequence or fetch the vector from the vector table. The user has the responsibility to use this information and execute the proper routine to service the interrupt. The Interrupt Control Unit updates the Interrupt Request, In-Service, Poll and Poll Status Registers the same as in the normal interrupt acknowledge sequence.

The Poll Status Register may be read to get the same information as the Poll Register. However, the interrupt is not actually acknowledged and none of the other registers in the Interrupt Control Unit will be modified.

Register Name: Poll Status Register
Register Mnemonic: POLLSTS
Register Function: Read to check for pending interrupts when polling.

15							0
IREQ	[reserved]	[reserved]	[reserved]	VT4	VT3	VT2	VT1	VT0

BIT MNEMONIC	BIT NAME	RESET STATE	FUNCTION
IREQ	*Interrupt Request*	0	Set if an interrupt is pending.
VT4:0	*Poll Status*	0	Indicate the type of the highest pending interrupt. Reading the poll status register will NOT acknowledge the interrupt.

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

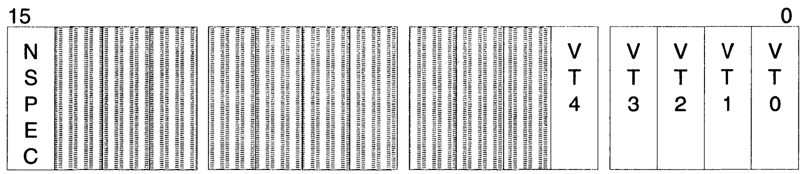
Figure 8.12. Poll Status Register

8.4.1.7. END-OF-INTERRUPT REGISTER

The End-Of-Interrupt Register (see Figure 8.13) is used to issue the EOI (End-Of-Interrupt) command to the Interrupt Control Unit. The EOI command is usually issued at the end of an interrupt handler and clears the bit in the In-Service Register.

There are two types of EOIs, specific and non-specific. A non-specific EOI simply clears the In-Service bit of the highest priority interrupt. A non-specific EOI is performed by writing a word to the End-Of-Interrupt Register with the NSPEC bit set (8000H).

Register Name: End of Interrupt Register
Register Mnemonic: EOI
Register Function: Used to issue the EOI command.



BIT MNEMONIC	BIT NAME	RESET STATE	FUNCTION
NSPEC	*Non-specific EOI*	0	Set to issue a non-specific EOI.
VT4:0	*Interrupt Type Number*	0	Specifies the interrupt type when issuing a specific EOI.

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

Figure 8.13. End-Of-Interrupt Register

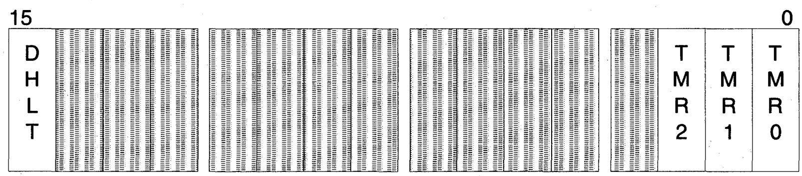
A specific EOI clears a particular bit in the In-Service Register. To perform a specific EOI, write a word to the End-Of-Interrupt Register with the interrupt type in bits VT4:0 of the In-Service bit to be cleared. The NSPEC bit must be cleared when issuing specific EOI command.

The timer interrupts share a bit in the In-Service Register. Write the interrupt type 8 to the End-Of-Interrupt Register to clear any timer interrupt with a specific EOI.

8.4.1.8. INTERRUPT STATUS REGISTER

All three timer interrupts share a single interrupt source. The Interrupt Status Register distinguishes between the interrupts which share an interrupt source (see Figure 8.14). The bits in the Interrupt Status Register are cleared when the interrupt request is acknowledged. More than one of these bits may be set at a time.

Register Name: Interrupt Status Register
Register Mnemonic: INTSTS
Register Function: Indicates which interrupt(s) is(are) pending for those interrupts which share a source.



BIT MNEMONIC	BIT NAME	RESET STATE	FUNCTION
DHLT	*DMA Halt*	0	Set to prevent any DMA activity.
TMR2:0	*Timer Interrupts*	0	Set when a timer has an interrupt request pending.

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

Figure 8.14. Interrupt Status Register

8.4.2. INTERRUPT CONTROL UNIT INITIALIZATION SEQUENCE

To initialize the Interrupt Control Unit, follow these steps:

1. Determine which interrupt sources will be utilized.
2. Determine if the default priority scheme will be used or figure out your own priority.
3. Initialize the Interrupt Control Registers for all used interrupt sources.
   A. For the external interrupt pins, determine whether edge or level triggered will be used.
   B. For either INT0 or INT1 determine whether The Cascade Mode and/or the Special Fully Nested Mode will be used.
   C. If using your own priority scheme, program the priority levels.
4. Initialize the Priority Mask Register if seven is too low a priority for your application.
5. Unmask all desired interrupt sources with the Interrupt Mask Register.
6. Set the Interrupt Enable bit by executing the STI instruction.

8.4.3. MASTER MODE INITIALIZATION EXAMPLE

The following example shows how to initialize the Interrupt Control Unit.

```
$mod186
name            example_80186_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,0100111B     :Cascade Mode
                mov             dx,IOCON        ;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

8.5. SLAVE MODE

Although Master Mode is the most common mode used in the Interrupt Control Unit, Slave Mode has some unique features that make it useful in larger system designs. In Slave Mode, an external 8259A acts as the master interrupt controller. The 8259A now controls the maskable interrupt input to the CPU. The Interrupt Control Unit acts as an interrupt input to the 8259A. In simplest terms, the Interrupt Control Unit behaves like a cascaded 8259A to the master 8259A (See Figure 8.15).

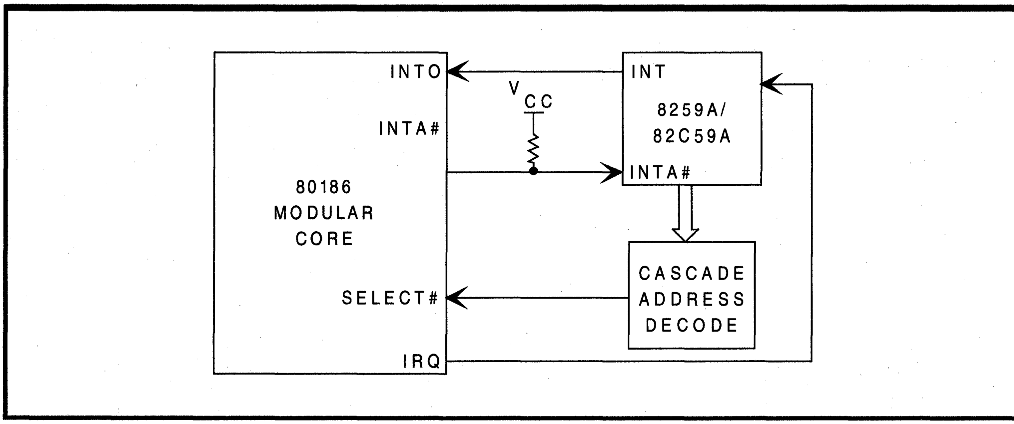


Figure 8.15. Interrupt Control Unit In Slave Mode

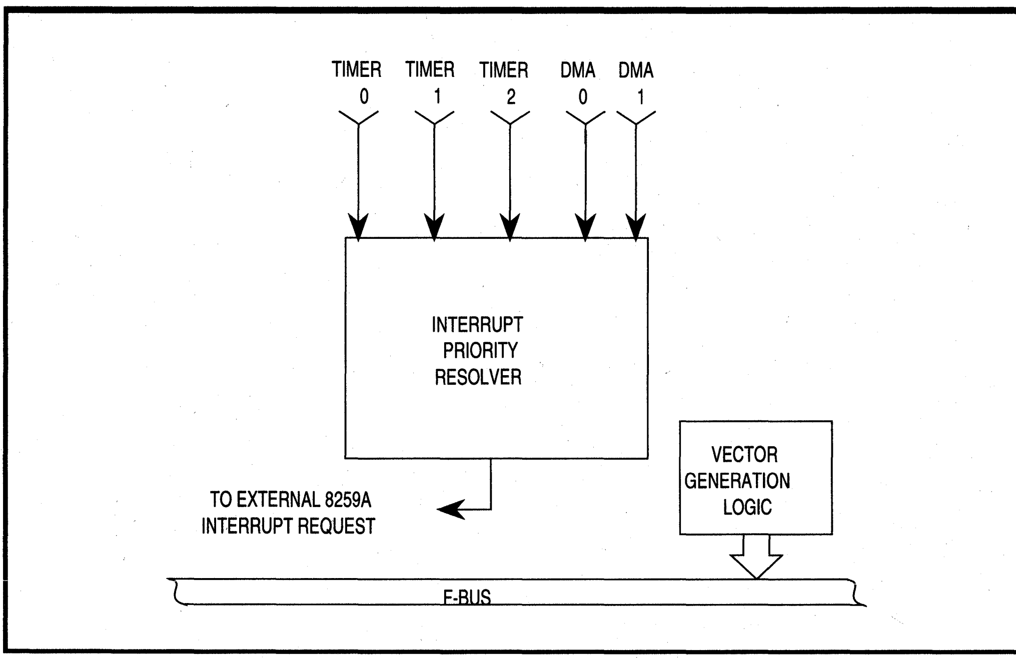


Figure 8.16. Interrupt Sources In Slave Mode

8.5.2. SLAVE MODE PROGRAMMING

Slave Mode adds one new register. Most of the registers retain the same functionality as in Master Mode. Many of the bit positions have changed, to account for each timer interrupt now being its own source to the Interrupt Control Unit. The register positions in the Peripheral Control Block have also changed (See Table 8.4).

8.5.2.1. INTERRUPT VECTOR REGISTER

The Interrupt Vector Register (see Figure 8.17) is the additional register in Slave Mode. In Slave Mode, the interrupt vector types are programmable. While in Master Mode, the interrupt vector types are fixed and unalterable. The Interrupt Vector Register specifies the five most significant bits of the interrupt vector type. The three least significant bits are fixed according to Table 8.5.

Table 8.4. Interrupt Control Unit Registers In Slave Mode

Register Name	Offset Address
Timer 2 Control Register	3AH
Timer 1 Control Register	38H
DMA1 Control Register	36H
DMA0 Control Register	34H
Timer 0 Control Register	32H
Interrupt Status Register	30H
Interrupt Request Register	2EH
In-Service Register	2CH
Priority Mask Register	2AH
Interrupt Mask Register	28H
EOI Register	22H
Interrupt Vector Register	20H

Table 8.5. Slave Mode Interrupt Type Bits

Interrupt Source	Type bits 2-0
Timer 0	000
(reserved)	001
DMA0	010
DMA1	011
Timer 1	100
Timer 2	101

Register Name: Interrupt Vector Register (Slave Mode)
Register Mnemonic: INTVEC
Register Function: Sets the five most significant bits of the interrupt types for the interrupt sources in Slave Mode.

15															0
											T 4	T 3	T 2	T 1	T 0

BIT MNEMONIC	BIT NAME	RESET STATE	FUNCTION
T4:0	*Interrupt Type Field*	0	Sets the five most significant bits of the interrupt types for the internal sources.

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

Figure 8.17. Interrupt Vector Register

8.5.2.2. END-OF-INTERRUPT REGISTER

The End-Of-Interrupt Register (see Figure 8.18) retains the same function in Slave Mode. However, only specific EOIs can be issued to the Interrupt Control Register in Slave Mode. Non-specific EOIs are not supported. To clear an In-Service Bit in Slave Mode, write the three least significant bits of the interrupt type to VT2:0 in the End-Of-Interrupt Register.

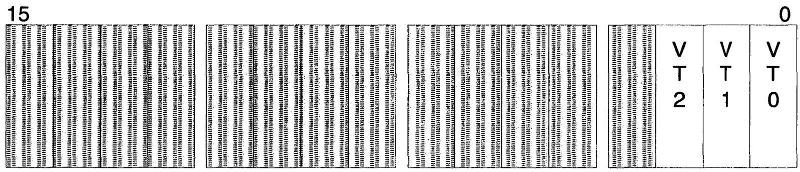
8.5.2.3. OTHER REGISTERS IN SLAVE MODE

The Interrupt Control, Interrupt Request, Interrupt Mask, In-Service and Interrupt Status Registers all retain the same functionality in Slave Mode as in Master Mode. The individual bits are different to account for the addition of the separate timer sources and the deletion of the external interrupt pins (see Figure 8.19).

The Priority Mask Register maintains the exact function and bit definitions in Slave Mode as in Master Mode.

The Poll and Poll Status Registers are not supported in Slave Mode.

Register Name: End of Interrupt Register (Slave Mode)
Register Mnemonic: EOI
Register Function: Used to issue the EOI command in Slave Mode.



BIT MNEMONIC	BIT NAME	RESET STATE	FUNCTION
VT2:0	*Interrupt Type Number*	0	Write three LSBs of the interrupt type to VT2:0 to issue an EOI in Slave Mode.

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

Figure 8.18. End-Of-Interrupt Register In Slave Mode

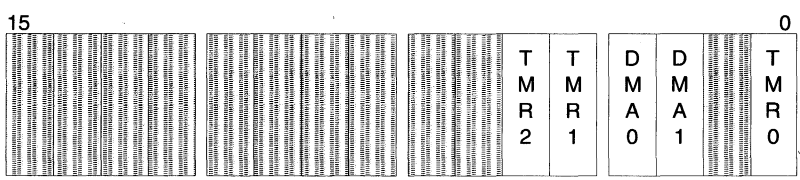
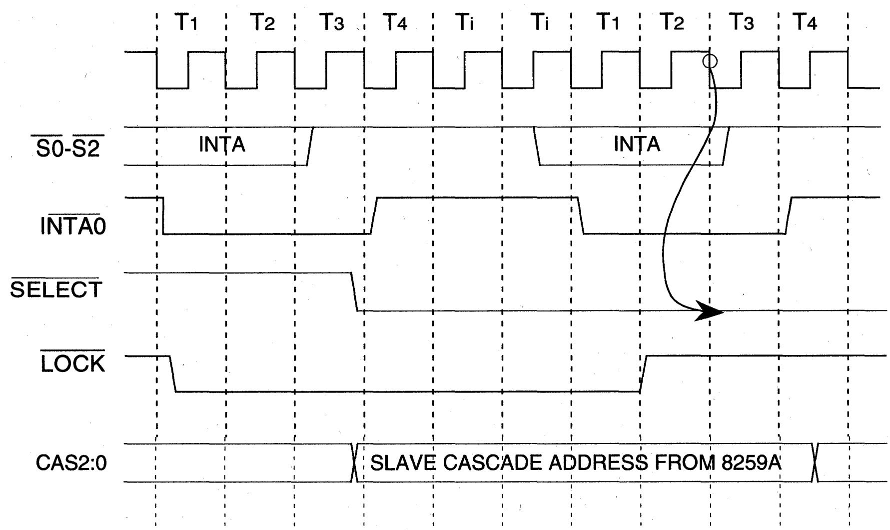


Figure 8.19. Other Registers In Slave Mode

8.5.2.4. INTERRUPT VECTORING IN SLAVE MODE

The external 8259A acts as the master interrupt controller in Slave Mode. Therefore, interrupt acknowledge cycles must be run for every interrupt. This includes any interrupts from the integrated peripherals. During the first interrupt acknowledge cycle, the external 8259A determines which slave interrupt controller has the highest priority interrupt request. The external 8259A then drives the address of that interrupt controller onto its CAS2:0 pins (see Figure 8.20). External logic must decode the correct slave address of the Interrupt Control Unit from the CAS2:0 signals to drive the $\overline{\text{SELECT}}$ pin.



NOTES:

1. INT1/$\overline{\mathrm{SELECT}}$ HAS THE $\overline{\mathrm{SELECT}}$ FUNCTION IN SLAVE MODE
2. INT2/$\overline{\mathrm{INTA0}}$ HAS THE $\overline{\mathrm{INTA0}}$ FUNCTION IN SLAVE MODE
3. CASCADE ADDRESS IS DRIVEN BY THE EXTERNAL 8259A
4. $\overline{\mathrm{SELECT}}$ MUST BE DRIVEN BEFORE PHASE 2 OF T2 OF THE SECOND INTA
5. $\overline{\mathrm{SELECT}}$ READ BY PROCESSOR
6. ALE IS GENERATED FOR EACH INTA
7. $\overline{\mathrm{RD}}$ IS INACTIVE

Figure 8.20. Interrupt Vectoring In Slave Mode

The $\overline{\mathrm{SELECT}}$ pin is used as the slave-select input to the Interrupt Control Unit. During the second interrupt acknowledge cycle, the slave interrupt controller with the highest priority transfers the interrupt type to the CPU of its highest priority interrupt. If the Interrupt Control Unit is selected, it passes the interrupt type internally to the CPU. However, the interrupt acknowledge cycle still must be run for the benefit of the external 8259A.

External interrupt acknowledge cycles must be run for every maskable interrupt. Therefore, the interrupt response time for every interrupt will be 55 clocks. This is shown in Figure 8.21.

		Clocks
INTERRUPT PRESENTED TO INTERRUPT CONTROL UNIT		
		5
INTERRUPT PRESENTED TO EXTERNAL 82C59A		
	INTA	4
	IDLE	2
	INTA	4
	IDLE	5
	READ IP	4
	IDLE	3
	READ CS	4
	IDLE	4
	PUSH FLAGS	4
	IDLE	3
	PUSH CS	4
	PUSH IP	4
	IDLE	5
FIRST INSTRUCTION FETCH FROM INTERRUPT ROUTINE		
		Total 55

Figure 8.21. Slave Mode Interrupt Response Time

Timer/Counter Unit 9

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. Code examples configuring the Timer/Counter Unit to function as a real-time clock, a square-wave generator, and a digital one-shot are provided in Section 9.4.

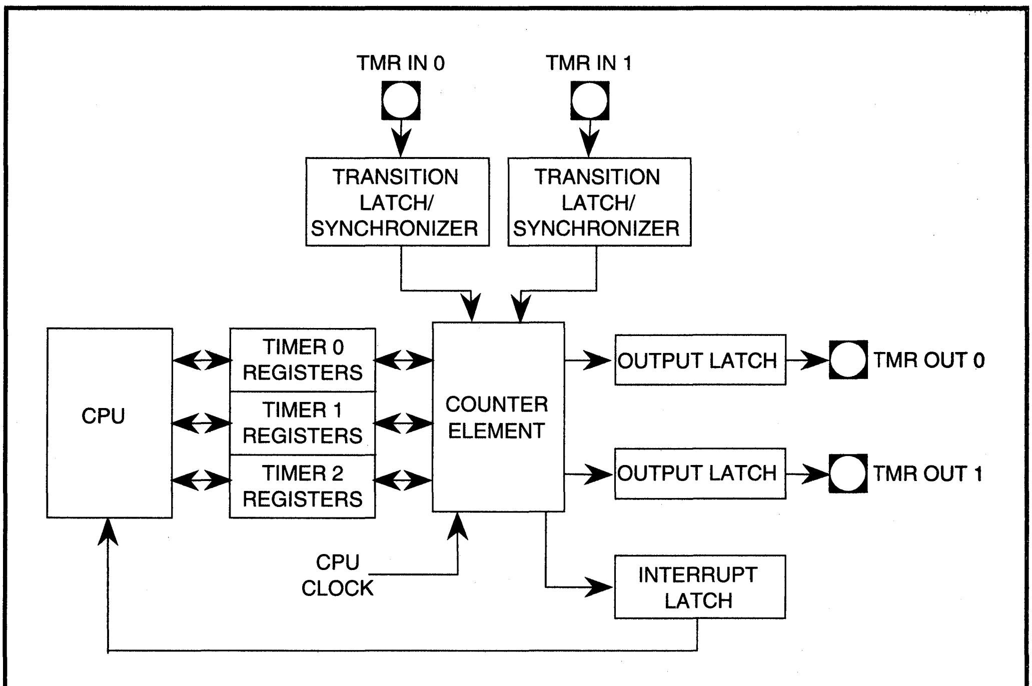
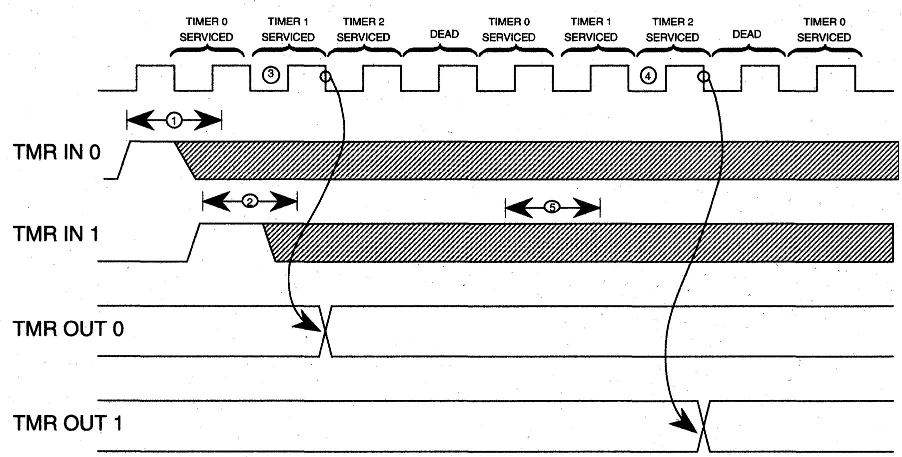


Figure 9.1. Timer/Counter Unit Block Diagram

9.1. FUNCTIONAL OVERVIEW

The Timer/Counter Unit is composed of three independent 16-bit timers (see Figure 9.1). These timers operate independently of the CPU. The internal Timer/Counter Unit can be modeled as a single counter element, time multiplexed to three register banks. The unit is serviced over 4 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 2 1/2 to 6 1/2 CLKOUT period delay from timer input to timer output.

The register banks are dual-ported between the counter element and the CPU. During a given bus cycle, the counter element and CPU may both access the register banks. Counter element and CPU accesses to the register banks are synchronized.



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

Figure 9.2. Counter Element Multiplexing and Timer Input Synchronization

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 only use 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 to either stop after a single cycle or run continuously.

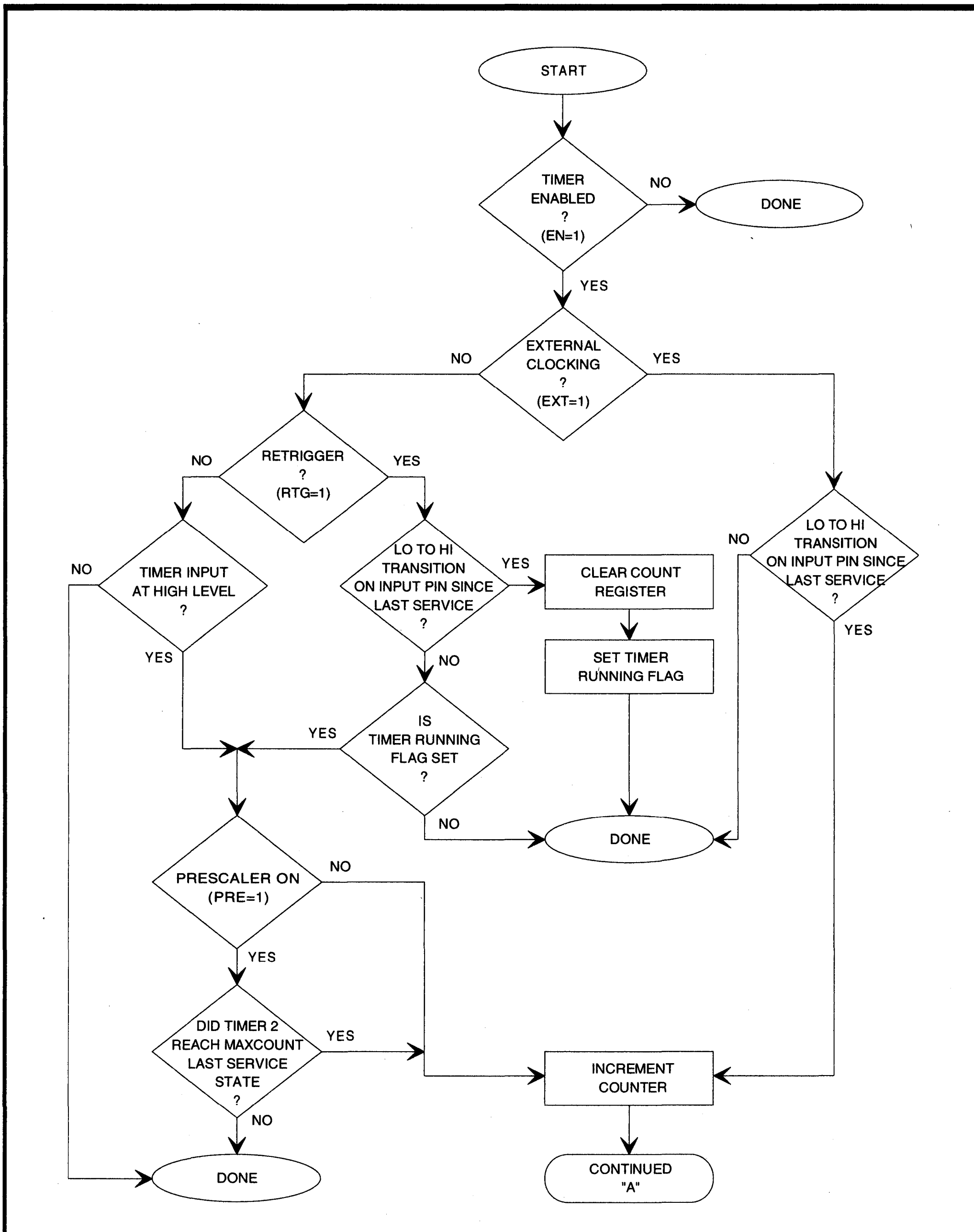


Figure 9.3(a). Timers 0 and 1 Flow Chart

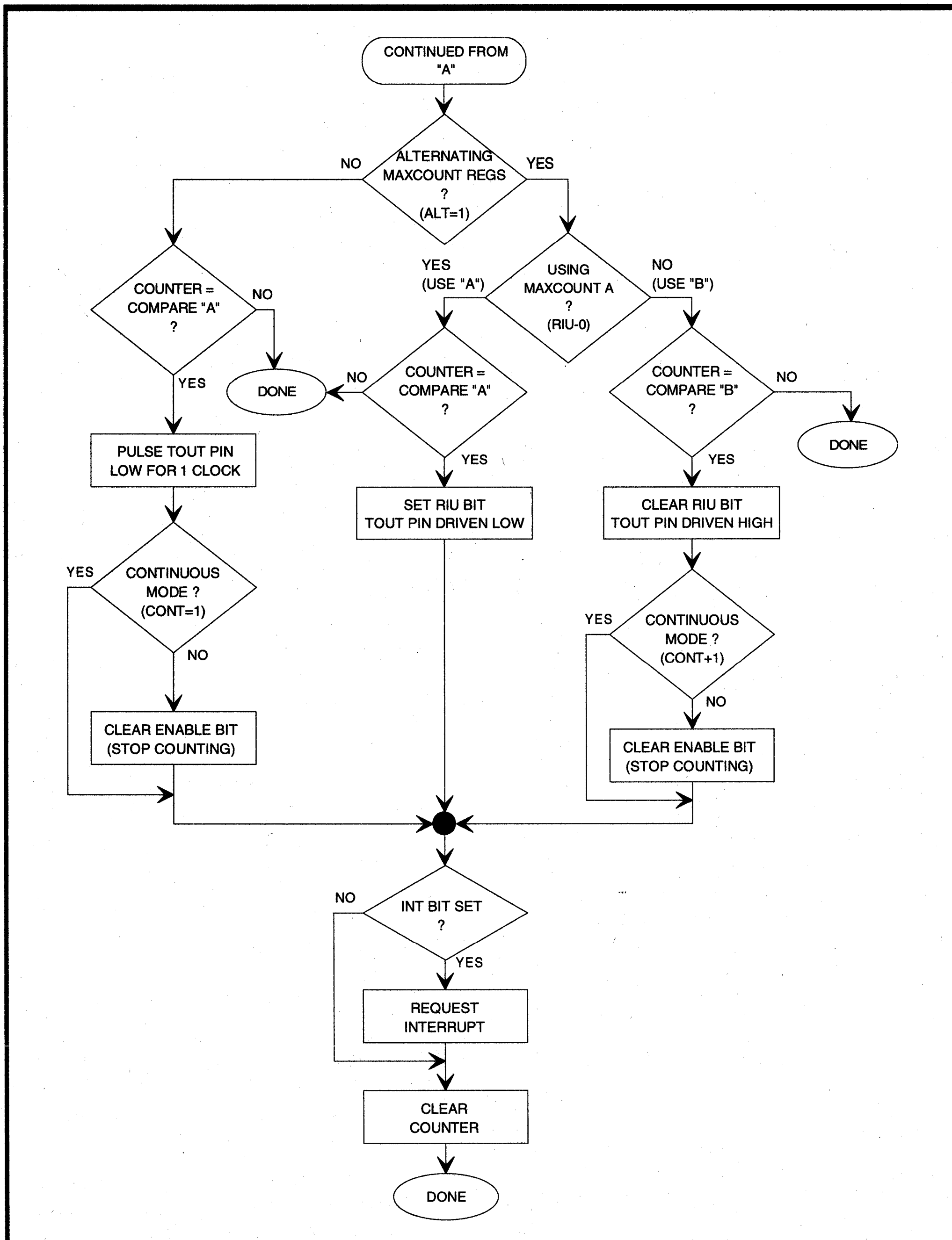


Figure 9.3(b). Timers 0 and 1 Flow Chart (Continued)

Timers 0 and 1 are functionally identical. Each has a latched, synchronized input pin and a single output pin. Each timer may be clocked internally or externally. Internally, the timer may increment at either 1/4 CLKOUT frequency or be prescaled by Timer 2. If a timer is prescaled by Timer 2, when Timer 2 reaches its maximum count value, the timer increments. When configured for internal clocking, the Timer/Counter Unit uses the input pins to either enable timer counting or retrigger the associated timer. Externally, a timer will increment on LOW-TO-HIGH transitions on its input pin (up to 1/4 CLKOUT frequency). A flow chart for Timer 0 and 1 operation is given in Figures 9.3(a) and 9.3(b).

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 (see Figure 9.4). Interrupts can be generated at the end of every timing cycle.

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

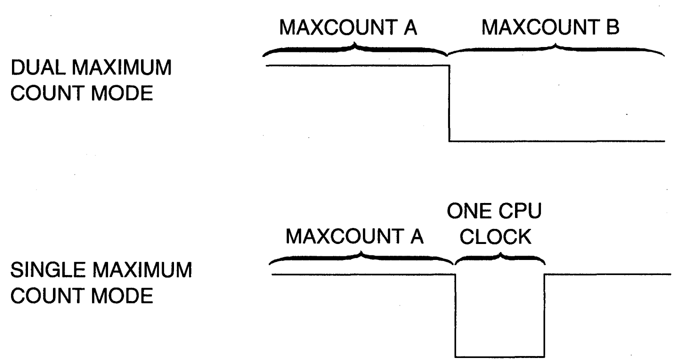


Figure 9.4. Timer/Counter Unit Output Modes

9.2. PROGRAMMING THE TIMER/COUNTER UNIT

Each timer has three registers: a Timer Control register (see Figures 9.5 and 9.6), a Timer Count register (see Figure 9.7) and a Timer Maxcount Compare register (see 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. 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.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EN	$\overline{\text{INH}}$	INT	RIU							MC	RTG	P	EXT	ALT	CONT

BIT MNEMONIC	BIT NAME	RESET STATE	FUNCTION
EN	*Enable*	0	If set, the timer is enabled. This bit cannot be written to unless the $\overline{\text{INH}}$ bit is set.
$\overline{\text{INH}}$	*Inhibit*	X	If set, writes to the Enable bit are allowed. If clear, writes to the Enable bit are ignored. This bit is not stored and is always read as zero.
INT	*Interrupt*	X	If set, an interrupt request is generated when the Count register equals a maximum count. If clear, the timer will not issue interrupt requests.
RIU	*Register In Use*	X	If set, Maxcount Compare register B is being used. If clear, Maxcount Compare register A is being used.
MC	*Maximum Count*	X	If set, counter has reached a maximum count. If clear, counter has not reached a maximum count.
RTG	*Retrigger*	X	If set, 0 to 1 edge on TMR INx resets count. If clear, high input enables counting. This bit is ignored with external clocking (EXT=1).
P	*Prescaler*	X	If set, timer is prescaled by Timer 2. If clear, timer counts 1/4 CLKOUT. This bit is ignored with external clocking (EXT=1).
EXT	*External Clock*	X	If set, use external clock. If clear, use internal clock.
ALT	*Alternate Compare Register*	X	If set, count to Maxcount Compare A, reset Count register to zero, count to Maxcount Compare B, reset Count register to zero again. If clear, count to Maxcount Compare A and reset Count register to zero.
CONT	*Continuous Mode*	X	If set, timer runs continuously. If clear, EN is cleared after each timer counting sequence.

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

Figure 9.5. Timer 0 and Timer 1 Control Registers

Register Name: Timer 2 Control Register
Register Mnemonic: T2CON
Register Function: Defines Timer 2 operation.

15															0
EN	$\overline{\text{INH}}$	INT								MC					CONT

BIT MNEMONIC	BIT NAME	RESET STATE	FUNCTION
EN	*Enable*	0	If set, the timer is enabled. If clear, the timer is disabled. This bit cannot be written to unless the $\overline{\text{INH}}$ bit is set.
$\overline{\text{INH}}$	*Inhibit*	X	If set, writes to the Enable bit are allowed. If clear, writes to the Enable bit are ignored. This bit is not stored and is always read as zero.
INT	*Interrupt*	X	If set, an interrupt request is generated when the Count register equals a maximum count. If clear, the timer will not issue interrupt requests.
MC	*Maximum Count*	X	If set, counter has reached a maximum count. If clear, counter has not reached a maximum count. This bit must be cleared by the user after maximum count is reached.
CONT	*Continuous Mode*	X	If set, timer runs continuously. If clear, EN is cleared after each timer counting sequence.

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

Figure 9.6. Timer 2 Control Register

9.2.1. INITIALIZATION

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.
3. Set Timer Maxcount Compare register to maximum count value. Make sure to program Maxcount Compare A and B if dual maximum count mode is used.
4. Program Timer Control register to enable timer.

Register Name: Timer Count Register
Register Mnemonic: T0CNT, T1CNT, T2CNT
Register Function: Contains the current timer count.

15															0
TC15	TC14	TC13	TC12	TC11	TC10	TC9	TC8	TC7	TC6	TC5	TC4	TC3	TC2	TC1	TC0

BIT MNEMONIC	BIT NAME	RESET STATE	FUNCTION
TC15:0	*Timer Count Value*	XXXXH	Register contains the current count of the associated timer.

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

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.

15															0
TC15	TC14	TC13	TC12	TC11	TC10	TC9	TC8	TC7	TC6	TC5	TC4	TC3	TC2	TC1	TC0

BIT MNEMONIC	BIT NAME	RESET STATE	FUNCTION
TC15:0	*Timer Compare Value*	XXXXH	Register contains the maximum value a timer will count to before resetting its Count register to zero.

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

Figure 9.8. Timer Maxcount Compare Registers

The programmer must clear the Timer Count register before enabling the timer because the count register is undefined at reset. This ensures counting begins at zero.

When using Timer 2 to prescale another timer, Timer 2 should be enabled 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 time-out of Timer 2 (Timers 0 and 1). Up to 65536 (2^{16}) events may 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 1/4 the CPU clock rate (measured at CLKOUT) because the timers are only serviced 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 may only use 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 1/4 CLKOUT frequency.
0	1	Timer clocked internally, prescaled by Timer 2.
1	X	Timer clocked externally at up to 1/4 CLKOUT frequency.

9.2.3. COUNTING SEQUENCE

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 may be written to 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 may be configured to execute different counting sequences. The timers may operate in single maximum count mode (all timers) or dual maximum count mode (Timers 0 and 1 only). They may 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 may 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 only operate 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 if 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 if 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 will reset 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 LOW, 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 LOW and the RTG bit is HIGH, 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 which can perform two functions. First, the output may be a single pulse, indicating the end of a timing cycle (single maximum count mode). Second, the output may 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 2 1/2 to 6 1/2 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 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 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). This occurs when the count value equals the Maxcount Compare A value. If programmed to run continuously, the timer generates periodic pulses.

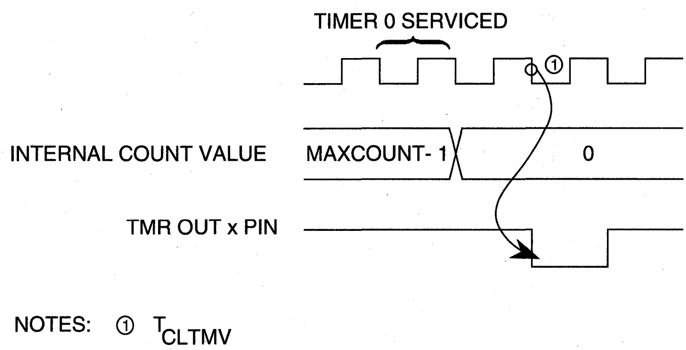


Figure 9.9. TxOUT Signal Timing

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). 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 ($\overline{\text{INH}}$) 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 will 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 to either enable the timer or 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 interrupt priority.

Timer Interrupts are enabled or disabled via 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 will be 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 will still be 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 will be 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 the 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. The timer input signal 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 may only operate up to 1/4 of the CLKOUT frequency. Clocking at greater frequencies will result in missed clocks.

9.4. 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.

9.4.1. 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.

```
$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
;
```

Example 9.1.

```
;---------------------------------------------------------------
                        ;                   ;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

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
```

Example 9.1. (Continued)

```
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, ax

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

                mov DX, TCUCON              ;set up interrupt controller
                xor ax, ax                  ;unmask highest
                out DX, ax                  ;priority interrupt

                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
```

Example 9.1. (Continued)

```
                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, ax
                pop DX
                pop ax
                iret

timer_2_interrupt_routine endp

lib_80186       ends
                end
```

Example 9.1. (Continued)

9.4.2. SQUARE WAVE GENERATOR

A square-wave generator can be useful to act as a system clock tick. Example 9.2 illustrates how to configure the Timer 1 to operate this way.

```
$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  equu    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
                                        ;modified
                push bx
                push DX
```

Example 9.2.

```
                mov DX, T1CMPA             ;set mark time
                mov ax, _mark
                out DX, ax

                mov DX, T1CMPB             ;set space time
                mov ax, _space
                out DX, ax

                mov DX, T1CNT              ;Clear Timer 1 Counter
                xor ax, ax
                out DX, ax

                mov DX, T1CON              ;start Timer 1
                mov ax, C003H
                out DX, ax

                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. (Continued)

9.4.3. DIGITAL ONE-SHOT

Example 9.3 configures Timer 1 to act as a digital one-shot.

```
$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);
;
```

Example 9.3.

```
;  INPUTS:       CMPB - This is the T1CMPB value required to
;                generate a pulse of 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          equu  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

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

               push ax                   ;save registers that will be
                                         ;modified
               push DX

               mov DX, T1CNT             ;Clear Timer 1 Counter
               xor ax, ax
               out DX, ax

               mov DX, T1CMPA            ;set time before  t_shot to 0
               mov ax, 1
               out DX, ax
```

Example 9.3. (Continued)

```
                mov DX, T1CMPB              ;set pulse time
                mov ax, _CMPB
                out DX, ax

                mov DX, T1CON
                mov ax, C002H               ;start Timer 1
                out DX, ax

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, ax                  ;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. (Continued)

Direct Memory Access Unit

10

CHAPTER 10
DIRECT MEMORY ACCESS UNIT

In many applications, large blocks of data must be transferred between memory and I/O space. A disk drive, for example, usually reads and writes data in blocks that may be thousands of bytes long. If the CPU were required to handle each byte of the transfer, the main tasks would suffer a severe performance penalty. Even if the data transfers were interrupt driven, the overhead for transferring control to the interrupt handler would still have a detrimental effect on system throughput.

Direct Memory Access, or DMA, allows data to be transferred between memory and peripherals **without the intervention of the CPU.** Systems that use DMA have a special device, known as the DMA controller, that takes control of the system bus and performs the transfer between memory and the peripheral device. When the DMA controller receives a request for a transfer from a peripheral, it signals the CPU that it needs control of the system bus. The CPU then releases control of the bus and the DMA controller performs the transfer. In many cases, the CPU will release the bus and continue to execute instructions from the prefetch queue. If the DMA transfers are relatively infrequent there will be no degradation of software performance; the DMA transfer is transparent to the CPU.

The DMA Unit has two channels. Each channel can accept DMA requests from one of 3 sources: an external request pin, the Timer/Counter Unit or by direct programming. Data can be transferred between any combination of memory and I/O space. The DMA Unit can access the entire memory and I/O space in either byte or word increments.

10.1. FUNCTIONAL OVERVIEW

The DMA Unit is comprised of two identical channels. Both channels are functionally identical. The following discussion is hierarchical beginning with an overview of a single channel and ending with a description of the two channel unit.

10.1.1. THE DMA TRANSFER

A DMA transfer begins with a request. The requesting device may either have data to transmit (a source request) or it may require data (a destination request). Alternatively, transfers may be initiated by the system software without an external request.

When the DMA request is granted, the Bus Interface Unit provides the bus signals for the DMA transfer while the DMA channel provides the address information for the source and destination devices. The DMA Unit does not provide a discrete DMA acknowledge signal, unlike other DMA controller chips (an acknowledge can be synthesized, however). The DMA channel will continue transferring data as long as the request is active and it has not exceeded its programmed transfer limit.

Every DMA transfer consists of two distinct bus cycles: a fetch and a deposit (see Figure 10.1). During the fetch cycle, the byte or word is read from the data source and placed in an internal temporary storage register. The data in the temporary storage register is written to the destination during the deposit cycle. The two bus cycles are indivisible; they cannot be separated by a bus hold request, a refresh request or another DMA request.

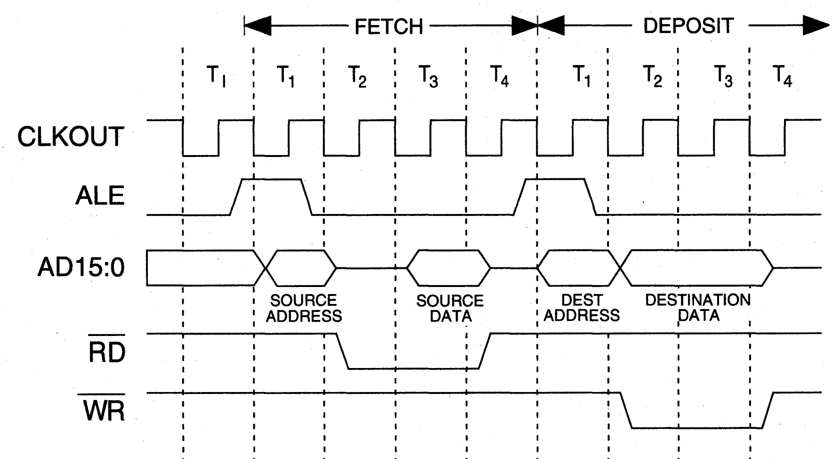


Figure 10.1. Typical DMA Transfer

10.1.1.1. DMA TRANSFER DIRECTIONS

The source and destination addresses for a DMA transfer are programmable and can be in either memory or I/O space. DMA transfers can be programmed for any of the following four directions:

- From memory space to I/O space
- From I/O space to memory space
- From memory space to memory space
- From I/O space to I/O space

DMA transfers can access the Peripheral Control Block.

10.1.1.2. BYTE AND WORD TRANSFERS

DMA transfers can be programmed to handle either byte or word sized transfers. The handling of byte and word data is the same as that for normal bus cycles and is processor bus width dependent. For example, odd aligned word DMA transfers on a 16-bit bus processor requires two fetches and two deposits (all back-to-back). BIU bus cycles are covered in greater detail in Chapter 3. Word transfers are illegal on the 8-bit bus device.

10.1.2. SOURCE AND DESTINATION POINTERS

Each DMA channel maintains a twenty bit pointer for the source of data and a twenty bit pointer for the destination of data. The twenty bit pointers allow access to the full 1 Mbyte of memory space. The DMA Unit views memory as a linear (unsegmented) array.

With a twenty bit pointer it is possible to create an I/O address that is above the CPU limit of 64 Kbytes. The DMA Unit will run I/O DMA cycles above 64K even though these addresses are not accessible through CPU instructions (e.g., IN and OUT). Some applications may wish to make use of this by swapping pages of data from I/O space above 64K to standard CPU memory.

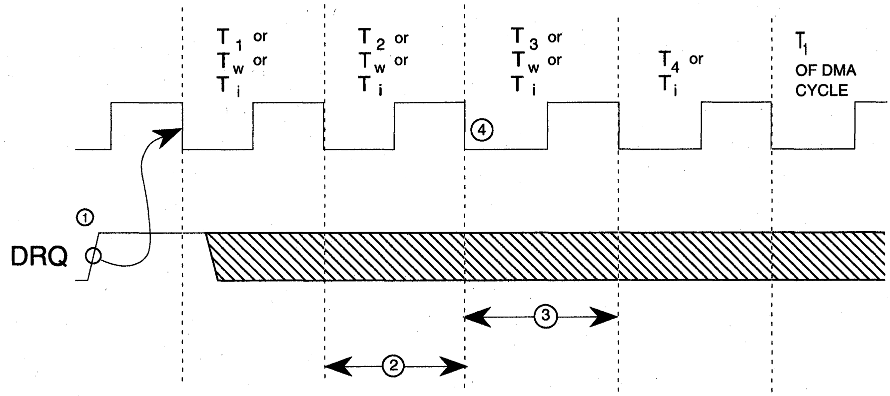
The source and destination pointers can be individually programmed to increment, decrement or remain constant after each transfer. The amount that a pointer is incremented or decremented is dependent on the programmed data width, byte or word, for the channel. Word transfers will change the pointer by two, byte transfers change the pointer by one.

10.1.3. DMA REQUESTS

There are three distinct sources of DMA requests: the external DRQ pin, the internal DMA request line and the system software. In all three cases, the system software must *arm* a DMA channel before it recognizes DMA requests. Arming a DMA channel is discussed in the programming section of this chapter.

10.1.4. EXTERNAL REQUESTS

External DMA requests are asserted on the DRQ pins. The DRQ pins are sampled on the falling edge of CLKOUT. It takes a minimum of four clocks before the DMA cycle is initiated by the BIU (see Figure 10.2). The DMA request is cleared four clocks before the end of the DMA cycle (effectively re-arming the DRQ input).



NOTES:

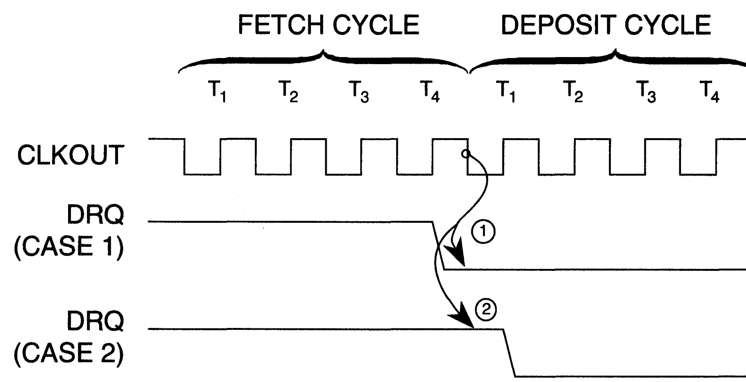
1. T_{INVCL} : DMA request to clock low.
2. Synchronizer resolution time.
3. DMA unit priority arbitration and overhead.
4. Bus interface Unit latches DMA request and decides to run DMA cycle.

Figure 10.2. DMA Request Minimum Response Time

External requests (and the resulting DMA transfer) are classified as either source synchronized or destination synchronized. A source synchronized request originates from the peripheral **from** which data is transferred. For example, a disk controller in the process of reading data from a disk would use a source synchronized request. A destination synchronized request originates from the peripheral **to** which data is transferred. If the previously mentioned disk controller were writing data to the disk, it would use destination synchronization since the data would be moving from memory to the disk. The type of synchronization a channel uses is programmable.

10.1.4.1. SOURCE SYNCHRONIZATION

A typical source synchronized transfer is shown in Figure 10.3. Most DMA driven peripherals do not deassert their DRQ line until after the DMA transfer has begun. The DRQ signal must be deasserted at least 4 clocks before the end of the DMA transfer (at the T1 state of the deposit phase) in order to prevent another DMA cycle from occurring. A source synchronized transfer provides the source device at least three clock cycles from when it is accessed (acknowledged) to deassert its request line if further transfers are not required.



NOTES:

① Current source synchronized transfer will **not** be immediately followed by another DMA transfer.

② Current source synchronized transfer will be immediately followed by another DMA transfer.

Figure 10.3. Source Synchronized Transfers

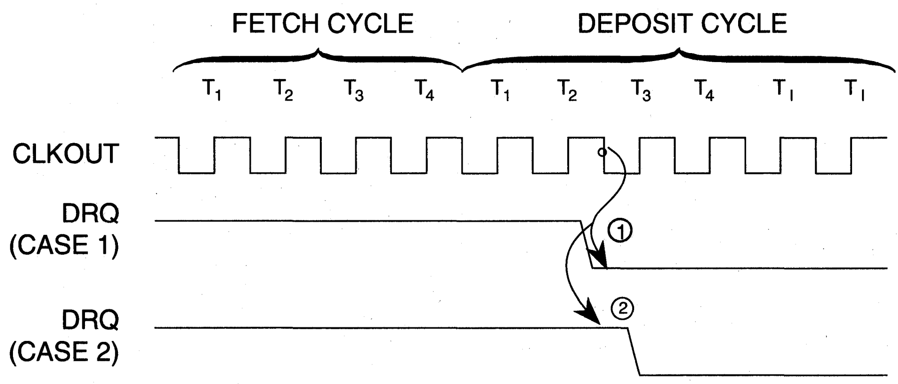
10.1.4.2. DESTINATION SYNCHRONIZATION

A destination synchronized transfer differs from a source synchronized transfer by the addition of two idle states at the end of the deposit cycle (Figure 10.4). The two idle states extend the DMA cycle to allow the destination device to deassert its DRQ pin four clocks before the end of the cycle. If the two idle states **were not** inserted, the destination device would not be able to deassert its request in time to prevent another DMA cycle from occurring.

The insertion of two idle states at the end of a destination synchronization transfer has an important side effect. **A destination synchronized DMA channel gives up the bus during the idle states allowing any other bus master to gain ownership.** This includes the CPU, the Refresh Control Unit, an external bus master or another DMA channel.

10.1.5. INTERNAL REQUESTS

Internal DMA requests can come from either Timer 2 or from the system software.



NOTES:

1. Current destination synchronized transfer will **not** be immediately followed by another DMA transfer.
2. Current destination synchronized transfer will be immediately followed by another DMA transfer.

Figure 10.4. Destination Synchronized Transfers

10.1.5.1. TIMER 2 INITIATED TRANSFERS

When programmed for Timer 2 initiated transfers, the DMA channel performs one DMA transfer every time that Timer 2 reaches its maximum count. Timer 2 initiated transfers are useful for servicing time based peripherals. For example, an A/D converter would require data every 22 microseconds in order to produce an audio range waveform. In this case the DMA source would point at the waveform data, the destination would point to the A/D converter and Timer 2 would request a transfer every 22 microseconds.

10.1.5.2. UNSYNCHRONIZED TRANSFERS

DMA transfers can be initiated directly by the system software by selecting unsynchronized transfers. Unsynchronized transfers continue, back-to-back, at the full bus bandwidth, until the channel's transfer count reaches zero or DMA transfers are suspended by an NMI.

10.1.6. DMA TRANSFER COUNTS

Each DMA Unit maintains a programmable 16-bit transfer count value that controls the total number of transfers the channel runs. The transfer count is decremented by one after each transfer (regardless of data size). The DMA channel can be programmed to terminate transfers when the transfer count reaches zero (also referred to as *terminal count*).

10.1.7. TERMINATION AND SUSPENSION OF DMA TRANSFERS

When DMA transfers for a channel are *terminated*, no further DMA requests for that channel will be granted until the channel is re-started by direct programming. A *suspended* DMA transfer temporarily disables transfers in order to perform a specific task. A suspended DMA channel does not need to be re-started by direct programming.

10.1.7.1. TERMINATION AT TERMINAL COUNT

When programmed to terminate on terminal count, the DMA channel disarms itself when the transfer count value reaches zero. No further DMA transfers take place on the channel until it is re-armed by direct programming.

Unsynchronized transfers always terminate when the transfer count reaches zero regardless of programming.

10.1.7.2. SOFTWARE TERMINATION

A DMA channel can be disarmed by direct programming. Any DMA transfer that is in progress will complete but no further transfers are run until the channel is re-armed.

10.1.7.3. SUSPENSION OF DMA DURING NMI

DMA transfers are inhibited during the service of Non-Maskable Interrupts (NMI). DMA activity is halted in order to give the CPU full command of the system bus during the NMI service. Exit from the NMI via an IRET instruction re-enables the DMA Unit. DMA transfers can be enabled during an NMI service routine by the system software.

10.1.7.4. SOFTWARE SUSPENSION

DMA transfers can be temporarily suspended by direct programming. In time critical sections of code, interrupt handlers for example, it may be necessary to temporarily shut off DMA activity in order to give the CPU total control of the bus.

10.1.8. DMA UNIT INTERRUPTS

Each DMA channel can be programmed to generate an interrupt request when its transfer count reaches zero.

10.1.9. DMA CYCLES AND THE BIU

The DMA Unit uses the Bus Interface Unit to perform its transfers. When the DMA Unit has a pending request, it signals the BIU. If the BIU has no other higher priority request pending it runs the DMA cycle (BIU priority is described in Chapter 3). The BIU signals that it is running a bus cycle initiated by a master other than the CPU by driving the S6 status bit high.

The Chip-Select Unit monitors the BIU addresses to determine which chip-select, if any, to activate. Because the DMA Unit uses the BIU, chip-selects are active for DMA cycles. If a DMA channel accesses a region of memory or I/O space within a chip-select's programmed range, then that chip-select is asserted during the cycle. The Chip-Select Unit will not recognize DMA cycles that access I/O space above 64K.

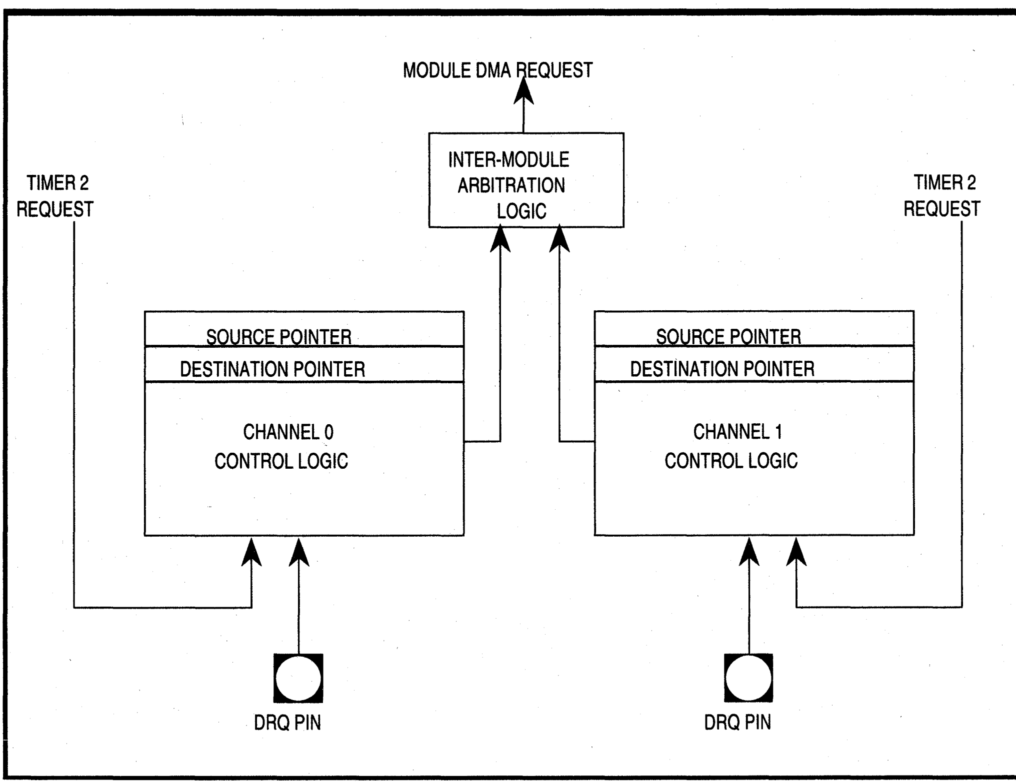


Figure 10.5. Two Channel DMA Unit

10.1.10. THE 2 CHANNEL DMA UNIT

Two DMA channels are combined with arbitration logic to form the two channel DMA Unit (see Figure 10.5).

10.1.10.1. DMA CHANNEL ARBITRATION

Within a two channel DMA module, the arbitration logic decides which channel takes precedence when both channels simultaneously request transfers. Each channel can be set to either low priority or high priority. If the two channels are set to the same priority (either both high or both low) then the channels rotate priority.

10.1.10.1.1. FIXED PRIORITY

Fixed priority results when one channel in a module is programmed to high priority and the other is set to low priority. If both DMA requests occur simultaneously, the high priority channel will perform its transfer (or transfers) first. The high priority channel continues to perform transfers as long as the following conditions are met:

- the channel's DMA request is still active
- the channel has not terminated or suspended transfers (through programming or interrupts)
- the channel has not released the bus (through the insertion of idle states for destination synchronized transfers)

The last point is extremely important when the two channels use different synchronization. For example, consider the case where channel 1 is programmed for high priority and destination synchronization and channel 0 is programmed for low priority and source synchronization. If a DMA request occurred for both channels simultaneously channel 1 would perform the first transfer. At the end of channel 1's deposit cycle two idle states are inserted (thus releasing the bus). With the bus released, channel 0 is free to perform its transfer **even though the higher priority channel 0 has not completed all of its transfers**. Channel 1 would regain the bus at the end of channel 0's transfer. The transfers would alternate as long as both requests remained active.

A higher priority DMA channel will interrupt the transfers of a lower priority channel. Figure 10.6 shows several transfers with different combinations of channel priority and synchronization.

10.1.10.1.2. ROTATING PRIORITY

Channel priority rotates when both channels are programmed as both high or both low priority. The highest priority is initially assigned to channel 1 of the module. After a channel performs a transfer it is assigned the lower priority. When requests are active for both channels, the transfers alternate between the two as long as the bus is not released by the DMA Unit. Channel 1 is reassigned high priority whenever the bus is released (i.e., at the end of a destination synchronized transfer, or when DMA requests are no longer active).

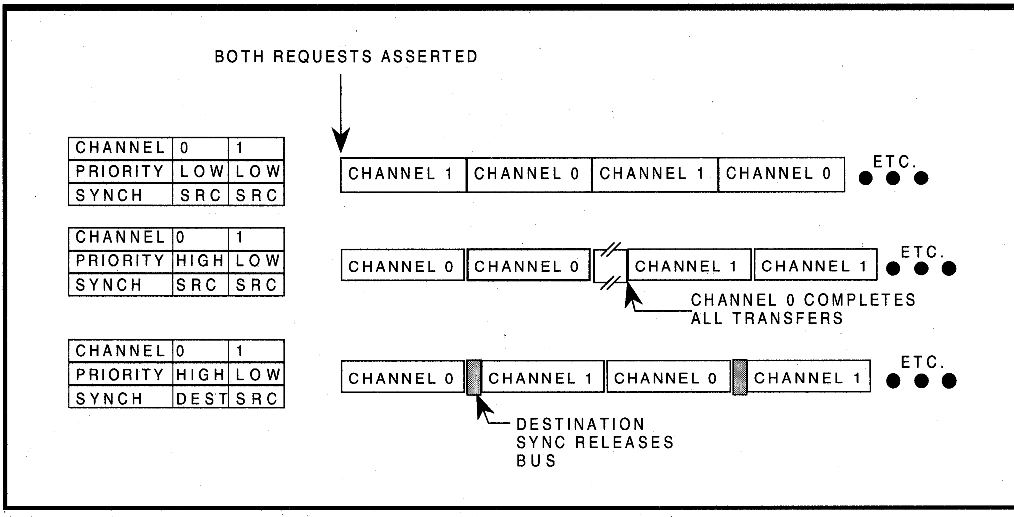


Figure 10.6. Examples of DMA Priority

10.2. PROGRAMMING THE DMA UNIT

A total of six Peripheral Control Block registers configure each DMA channel.

10.2.1. DMA CHANNEL PARAMETERS

The first step in programming the DMA Unit is to set up the parameters for each of the channels.

10.2.1.1. PROGRAMMING THE SOURCE AND DESTINATION POINTERS

The following parameters are programmable for the source and destination pointers:

- pointer address
- address space (memory or I/O)
- automatic pointer indexing (increment/decrement) after transfer

Two 16-bit Peripheral Control Block registers define each of the 20-bit pointers. Figures 10.7 through 10.10 show the layout of the DMA Source and DMA Destination pointer address registers. The DS19:16 and DD19:16 (high order address bits) are driven on the bus even if I/O transfers have been programmed. When performing I/O transfers within the normal 64K I/O space **only**, the high order bits in the pointer registers must be cleared.

Register Name: DMA Source Address Pointer (High)
Register Mnemonic: DxSRCH
Register Function: Contains the upper 4 bits of the DMA Source pointer.

15															0
												DSA19	DSA18	DSA17	DSA16

BIT MNEMONIC	BIT NAME	RESET STATE	FUNCTION
DSA19:16	*DMA Source Address*	XXXXH	DSA19:16 are driven on A19:16 during the fetch phase of a DMA transfer.

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

Figure 10.7. DMA Source Pointer (High Order Bits)

Register Name: DMA Source Address Pointer (Low)
Register Mnemonic: DxSRCL
Register Function: Contains the lower 16 bits of the DMA Source pointer.

15															0
DSA15	DSA14	DSA13	DSA12	DSA11	DSA10	DSA9	DSA8	DSA7	DSA6	DSA5	DSA4	DSA3	DSA2	DSA1	DSA0

BIT MNEMONIC	BIT NAME	RESET STATE	FUNCTION
DSA15:0	*DMA Source Address*	XXXXH	DSA15:0 are driven on the lower 16 bits of the address bus during the fetch phase of a DMA transfer.

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

Figure 10.8. DMA Source Pointer (Low Order Bits)

The address space referenced by the source and destination pointers is programmed in the DMA Control Register for the channel (see Figure 10.13). The SMEM and DMEM bits control the address space (memory or I/O) for source pointer and destination pointer, respectively.

Automatic pointer indexing is also controlled by the DMA Control Register. Each pointer has a two bit field, increment and decrement, that controls the indexing. If the increment and decrement bits for a pointer are programmed to the same value then the pointer will remain constant. The amount that a pointer is incremented or decremented is automatically controlled by the programmed data width, byte or word, for the channel.

Register Name: DMA Destination Address Pointer (High)
Register Mnemonic: DxDSTH
Register Function: Contains the upper 4 bits of the DMA Source pointer.

15							0
				DDA19	DDA18	DDA17	DDA16

BIT MNEMONIC	BIT NAME	RESET STATE	FUNCTION
DDA19:16	*DMA Destination Address*	XXXXH	DDA19:16 are driven on A19:16 during the deposit phase of a DMA transfer.

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

Figure 10.9. DMA Destination Pointer (High Order Bits)

Register Name: DMA Destination Address Pointer (Low)
Register Mnemonic: DxDSTL
Register Function: Contains the lower 16 bits of the DMA Source pointer.

15															0
DDA15	DDA14	DDA13	DDA12	DDA11	DDA10	DDA9	DDA8	DDA7	DDA6	DDA5	DDA4	DDA3	DDA2	DDA1	DDA0

BIT MNEMONIC	BIT NAME	RESET STATE	FUNCTION
DDA15:0	*DMA Destination Address*	XXXXH	DDA15:0 are driven on the lower 16 bits of the address bus during the deposit phase of a DMA transfer.

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

Figure 10.10. DMA Destination Pointer (Low Order Bits)

Register Name: DMA Control Register
Register Mnemonic: DxCON
Register Function: Controls DMA channel parameters.

15															0
DMEM	DDEC	DINC	SMEM	SDEC	SINC	TC	INT	SYN1	SYN0	P	IDRQ		CHG	STRT	WORD

Figure 10.11(a).DMA Control Register Bit Positions

BIT MNEMONIC	BIT NAME	RESET STATE	FUNCTION
SMEM/DMEM	*Source/ Destination Address Space Select*	X	Selects memory or I/O space for the corresponding pointer. Set SMEM/DMEM to select memory space; Clear SMEM/DMEM to select I/O space. SMEM corresponds to the source pointer. DMEM corresponds to the destination pointer.
SINC/DINC	*Source/ Destination Increment*	X	Set to automatically increment the source/destination pointer after each transfer. A pointer will remain constant if its increment and decrement bits are equal.
SDEC/DDEC	*Source/ Destination Decrement*	X	Set to automatically decrement the source/destination pointer after each transfer. A pointer will remain constant if its increment and decrement bits are equal.
TC	*Terminal Count*	X	Set to terminate transfers on Terminal Count.
INT	*Interrupt*	X	Set to generate an interrupt request on Terminal Count. The TC bit must be set to generate an interrupt.
SYN1:0	*Synchron- ization Type*	XX	Selects channel synchronization: **SYN1:0 Synchronization Type** 00 Unsynchronized 01 Source Synchronized 10 Destination Synchronized 11 Reserved (Do Not Use)
P	*Relative Priority*	X	Setting P selects high priority for the channel.
IDRQ	*Internal DMA Request Select*	X	Setting IDRQ selects internal (Timer 2) DMA requests. When IDRQ is set the external DRQ pin is ignored. Clearing IDRQ selects the DRQ pin as the source of DMA requests.
CHG	*Change Start Bit*	X	CHG must be set to modify the STRT bit.
STRT	*Start DMA Channel*		The DMA channel is armed by setting the STRT bit. The STRT bit can only be modified when the CHG bit is set.
WORD	*Word Transfer Select*	X	The WORD bit selects between byte and word transfers. Setting WORD selects word transfers; clearing WORD selects byte transfers.

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

Figure 10.11(b). DMA Channel Control Register Bit Descriptions

10.2.1.2. SELECTING BYTE OR WORD SIZE TRANSFERS

The WORD bit in the DMA Control Register is used to control the data size for a channel. When WORD is set, the channel transfers data in 16-bit words. Byte transfers are selected by clearing the WORD bit. The data size for a channel also affects pointer indexing. Word sized transfers modify (increment or decrement) the pointer registers by two for each transfer whereas byte transfers modify the pointer registers by one.

10.2.1.3. SELECTING THE SOURCE OF DMA REQUESTS

DMA requests can come from either an internal source (Timer 2) or an external source.

Timer 2 DMA requests are selected by setting the IDRQ bit in the DMA Control Register for the channel. The DMA channel ignores its DRQ pin when internal requests are programmed. Similarly, the DMA channel only responds to the DRQ pin (and ignores internal requests) when external requests are selected.

10.2.1.4. ARMING THE DMA CHANNEL

Each DMA channel must be armed before it will recognize DMA requests. A channel is armed by setting its STRT (Start) bit in the DMA Control Register. The STRT bit can only be modified if the CHG (Change Start) bit is set at the same time. The CHG bit is a safeguard to prevent unwanted arming of a DMA channel while modifying other channel parameters.

A DMA channel is disarmed by clearing its STRT bit. The STRT bit is cleared either directly by software or by the channel itself when programmed to terminate on terminal count.

10.2.1.5. SELECTING CHANNEL SYNCHRONIZATION

The synchronization method for a channel is controlled by the SYN1:0 bits in the DMA Control Register. The combination SYN1:0=11 is reserved and will result in unpredictable operation, if used.

When programmed for unsynchronized transfers (SYN1:0=00) the DMA channel will begin to transfer data as soon as the STRT bit is set.

Transfers requested by Timer 2 must always be programmed for source synchronization.

10.2.1.6. PROGRAMMING THE TRANSFER COUNT OPTIONS

The Transfer Count Register and the TC bit in the DMA Control Register are used to stop DMA transfers for a channel after a specified number of transfers have occurred.

Register Name: DMA Transfer Count
Register Mnemonic: DxTC
Register Function: Contains the DMA channel's transfer count.

15															0
TC15	TC14	TC13	TC12	TC11	TC10	TC9	TC8	TC7	TC6	TC5	TC4	TC3	TC2	TC1	TC0

BIT MNEMONIC	BIT NAME	RESET STATE	FUNCTION
TC15:0	*Transfer Count*	XXXXH	Contains the transfer count for a DMA channel. This value is decremented by one after each transfer.

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

Figure 10.12. Transfer Count Register

The number of transfers desired are written to the DMA Transfer Count Register (see Figure 10.12) The Transfer Count Register is 16-bits wide limiting the total number of transfers for a channel to 65,536 (without reprogramming). The Transfer Count Register is decremented by one after each transfer (for both byte and word transfers).

The TC bit, when set, instructs the DMA channel to disarm itself (by clearing the STRT bit) when the transfer count reaches zero. If the TC bit is cleared, the channel continues to perform transfers regardless of the state of the Transfer Count Register. Unsynchronized (software initiated) transfers always terminate when the transfer count reaches zero; the TC bit is ignored.

10.2.1.7. GENERATING INTERRUPTS ON TERMINAL COUNT

A channel can be programmed to generate an interrupt request whenever the transfer count reaches zero. Both the TC bit and the INT bit (in the DMA Control Register) must be set to generate an interrupt request.

10.2.1.8. SETTING THE RELATIVE PRIORITY OF A CHANNEL

The priority of a channel within a module is controlled by the Priority bit in the DMA Control Register. A channel may be assigned either high or low priority. If the priority for both channels is programmed to the same priority (i.e., both high or both low) then the channels will rotate priority.

10.2.2. SUSPENSION OF DMA TRANSFERS

Whenever an NMI is received by the CPU, all DMA activity is suspended at the end of the current transfer. The CPU suspends transfers by setting the DHLT (DMA Halt) bit in the Interrupt Status Register (see Chapter 8). The DHLT bit is automatically cleared upon execution of an IRET instruction. DMA transfers resume when the DHLT bit is cleared.

The DHLT bit may be read and written by the user. **Do not write to the DHLT bit while Timer/Counter Unit interrupts are enabled; a conflict with the internal use of the register may lead to incorrect timer interrupt processing.**

The DHLT bit does not function when the interrupt controller is in Slave Mode.

10.2.3. INITIALIZING THE DMA UNIT

Use the following sequence when programming the DMA Unit:

1. Program the source and destination pointers for all used channels.
2. Program the DMA Control Registers in order of highest priority channel to lowest priority channel.

10.3. HARDWARE CONSIDERATIONS AND THE DMA UNIT

The following sections cover hardware interfacing and performance factors for the DMA Unit.

10.3.1. DRQ PIN TIMING REQUIREMENTS

The DRQ pins are sampled on the falling edge of CLKOUT. The DRQ pins must be setup a minimum of T_{INVCL} before CLKOUT falling to guarantee recognition at a specific clock edge. Refer to the datasheet for specific values.

The DRQ pins have an internal synchronizer. Violating the setup and hold times may only result in a missed DMA request, not a processor malfunction.

10.3.2. DMA LATENCY

DMA Latency is the delay between a DMA request being asserted and the DMA cycle being run. The DMA latency for a channel is controlled by many factors, including:

- **Bus HOLD:** Bus HOLD takes precedence over internal DMA requests. Using bus HOLD will degrade DMA latency.
- **LOCKed Instructions:** Long LOCKed instructions (e.g., LOCK REP MOVS) will monopolize the bus preventing access by the DMA Unit.
- **Inter-channel Priority Scheme:** Setting a channel at low priority will affect its latency.

The minimum latency in all cases is four CLKOUT cycles. This is the amount of time it takes to synchronize and prioritize a request.

10.3.3. DMA TRANSFER RATES

The maximum DMA transfer rate is a function of processor operating frequency and synchronization mode. For unsynchronized and source synchronized transfers, 2 bytes can be transferred every eight CLKOUT cycles for the 80C186XL and one byte can be transferred for the 80C188XL. Maximum transfer rate for the 80C186XL is calculated by:

Maximum DMA Transfer Rate in Mbytes/sec = $.25*F_{CPU}$
(Source and Unsynchronized)

Where F_{CPU} is the CPU operating frequency in megahertz.

For destination synchronized transfers, the addition of two idle T-states reduces the bandwidth by two clocks per word:

Maximum DMA Transfer Rate in Mbytes/sec = $.20*F_{CPU}$
(Source and Unsynchronized)

Where F_{CPU} is the CPU operating frequency in megahertz.

Maximum transfer rates for the 80C188XL are half those calculated by the above equations as the 80C188XL can only transfer one byte per cycle.

10.3.4. GENERATING A DMA ACKNOWLEDGE

The DMA channels do not provide a distinct DMA acknowledge signal. A chip select line can be programmed to active for the memory or I/O range that requires the acknowledge. The chip select must be programmed to active only when a DMA is in progress. Latched status line S6 can be used as a qualifier to the chip select in situations where the chip select line will be active for both DMA and normal data accesses.

10.4. DMA UNIT EXAMPLES

In Example 10.1, channel 0 is set up to perform an unsynchronized burst transfer from memory to memory while channel 1 is used to service an external DMA request from a hard disk controller.

Timed DMA transfers are shown in Example 10.2. A sawtooth waveform is created using DMA transfers to an A/D converter.

```
$MOD186
NAME    DMA_EXAMPLE_1

; This example shows code necessary
; to setup of two DMA channels. One
; channel performs an unsynchronized
; transfer from memory to memory.
; The second channel is used by a
; hard disk controller located in
; I/O space.


; It is assumed that the constants for PCB register
; addresses are defined elsewhere with EQUates.





CODE_SEG        SEGMENT
                ASSUME CS:CODE_SEG


START:          MOV     AX, DATA_SEG     ; DATA SEGMENT POINTER
                MOV     DS, AX
                ASSUME DS:DATA_SEG


; First we must initialize DMA channel 0. DMA0 will
; an unsynchronized transfer from SOURCE_DATA_1 to
; DEST_DATA_1. The first step is to calculate the
; proper values for the source and destination
; pointers.

            MOV   AX, SEG SOURCE_DATA_1

            ROL   AX, 4                ; GET HIGH 4 BITS
               MOV     BX, AX          ; SAVE ROTATED VALUE
               AND     AX, 0FFF0H      ; GET SHIFTED LOW 4
                                       ; NIBBLES


               ADD     AX, OFFSET SOURCE_DATA_1
```

Example 10.1. DMA Unit Initialization

```
        ; NOW LOW BYTES OF
        ; POINTER ARE IN AX

        ADC     BX, 0           ; ADD IN THE CARRY
                                ; TO THE HIGH NIBBLE
        AND     BX, 000FH       ; GET JUST THE HIGH
                                ; NIBBLE
        MOV     DX, D0SRCL
        OUT     DX, AX          ; AX=LOW 4 BYTES

        MOV     DX, D0SRCH
        MOV     AX, BX          ; GET HIGH NIBBLE
        OUT     DX, AX

        ; SOURCE POINTER DONE. REPEAT FOR DEST.

    MOV   AX, SEG DEST_DATA_1

    ROL   AX, 4                 ; GET HIGH 4 BITS
        MOV     BX, AX          ; SAVE ROTATED VALUE
        AND     AX, 0FFF0H      ; GET SHIFTED LOW 4
                                ; NIBBLES


        ADD     AX, OFFSET DEST_DATA_1

        ; NOW LOW BYTES OF
        ; POINTER ARE IN AX

        ADC     BX, 0           ; ADD IN THE CARRY
                                ; TO THE HIGH NIBBLE
        AND     BX, 000FH       ; GET JUST THE HIGH
                                ; NIBBLE
        MOV     DX, D0DSTL
        OUT     DX, AX          ; AX=LOW 4 BYTES

        MOV     DX, D0DSTH
        MOV     AX, BX          ; GET HIGH NIBBLE
        OUT     DX, AX
```

Example 10.1. DMA Unit Initialization (Continued)

```
; THE POINTER ADDRESSES HAVE BEEN SET UP. NOW
; WE SET UP THE TRANSFER COUNT.

MOV     AX, 29          ; THE MESSAGE IS
                        ; 29 BYTES LONG.
MOV     DX, D0TC        ; XFER COUNT REG
OUT     DX, AX

; NOW WE NEED TO SET THE PARAMETERS FOR
; THE CHANNEL AS FOLLOWS:
;
;       DESTINATION     SOURCE
;       -----------     ------
;       MEMORY SPACE    MEMORY SPACE
;       INCREMENT PTR   INCREMENT PTR
;
; TERMINATE ON TC, NO INTERRUPT, UNSYNCHRONIZED,
; LOW PRIORITY RELATIVE TO CHANNEL 1, BYTE XFERS.
; WE START THE CHANNEL

MOV     AX, 1011011000000110B
MOV     DX, D0CON
OUT     DX, AX

; THE UNSYNCHRONIZED BURST IS NOW RUNNING ON
; THE BUS...

; NOW SET UP CHANNEL 1 TO SERVICE THE DISK
; CONTROLLER. FOR THIS EXAMPLE WE WILL ONLY
; BE READING FROM THE DISK.

; THE SOURCE IS THE I/O PORT FOR THE
; DISK CONTROLLER.

MOV     AX, DISK_IO_ADDR
MOV     DX, D1SRCL
OUT     DX, AL          ; PROGRAM LOW ADDR

XOR     AX, AX
MOV     DX, D1SRCH      ; HI ADDR FOR IO=0
OUT     DX, AL
```

Example 10.1. DMA Unit Initialization (Continued)

```
        ; THE DESTINATION IS THE DISK BUFFER IN MEMORY

    MOV   AX, SEG DISK_BUFF

    ROL   AX, 4                   ; GET HIGH 4 BITS
        MOV     BX, AX            ; SAVE ROTATED VALUE
        AND     AX, 0FFF0H        ; GET SHIFTED LOW 4
                                  ; NIBBLES


        ADD     AX, OFFSET DISK_BUFF

        ; NOW LOW BYTES OF
        ; POINTER ARE IN AX

        ADC     BX, 0             ; ADD IN THE CARRY
                                  ; TO THE HIGH NIBBLE
        AND     BX, 000FH         ; GET JUST THE HIGH
                                  ; NIBBLE
        MOV     DX, D1DSTL
        OUT     DX, AX            ; AX=LOW 4 BYTES

        MOV     DX, D1DSTH
        MOV     AX, BX            ; GET HIGH NIBBLE
        OUT     DX, AX


        ; THE POINTER ADDRESSES HAVE BEEN SET UP. NOW
        ; WE SET UP THE TRANSFER COUNT.

        MOV     AX, 512           ; THE DISK READS IN
                                  ; 512 BYTE SECTORS.
        MOV     DX, D1TC          ; XFER COUNT REG
        OUT     DX, AX
```

Example 10.1. DMA Unit Initialization (Continued)

```
                ; NOW WE NEED TO SET THE PARAMETERS FOR
                ; THE CHANNEL AS FOLLOWS:
                ;
                ;       DESTINATION     SOURCE
                ;       -----------     -------
                ;       MEMORY SPACE    I/O SPACE
                ;       INCREMENT PTR   CONSTANT PTR
                ;
                ; TERMINATE ON TC, INTERRUPT, SOURCE SYNC,
                ; HIGH PRIORITY RELATIVE TO CHANNEL 0, BYTE XFERS,
                ; USE DRQ PIN FOR REQUEST SOURCE.

                ; THE CHANNEL IS ARMED.

                MOV     AX, 1010001101100110B
                MOV     DX, D0CON
                OUT     DX, AX

                ; REQUESTS ON DRQ1 WILL NOW RESULT IN TRANSFERS



CODE_SEG         ENDS

DATA_SEG         SEGMENT

SOURCE_DATA_1    DB       '80C186XL INTEGRATED PROCESSOR'
DEST_DATA_1      DB       30 DUP('DUMMY')            ; JUNK DATA FOR TEST

DISK_BUFF       DB       512 DUP(?)

DATA_SEG         ENDS


                 END START
```

Example 10.1. DMA Unit Initialization (Continued)

```
$MOD186
NAME    DMA_EXAMPLE_1

; This example sets up the DMA Unit
; to perform a memory to I/O space
; transfer every 22uS. The data is
; sent to an A/D converter.

; It is assumed that the constants for PCB register
; addresses are defined elsewhere with EQUates.




CODE_SEG        SEGMENT
                ASSUME CS:CODE_SEG


START:          MOV     AX, DATA_SEG     ; DATA SEGMENT POINTER
                MOV     DS, AX
                ASSUME DS:DATA_SEG


; First, setup the pointers. The source is in memory.

            MOV    AX, SEG WAVEFORM_DATA

            ROL    AX, 4                ; GET HIGH 4 BITS
               MOV     BX, AX           ; SAVE ROTATED VALUE
               AND     AX, 0FFF0H       ; GET SHIFTED LOW 4
                                        ; NIBBLES


               ADD     AX, OFFSET WAVEFORM_DATA
```

Example 10.2. Timed DMA Transfers

```
        ; NOW LOW BYTES OF
        ; POINTER ARE IN AX

        ADC     BX, 0           ; ADD IN THE CARRY
                                ; TO THE HIGH NIBBLE
        AND     BX, 000FH       ; GET JUST THE HIGH
                                ; NIBBLE
        MOV     DX, D0SRCL
        OUT     DX, AX          ; AX=LOW 4 BYTES

        MOV     DX, D0SRCH
        MOV     AX, BX          ; GET HIGH NIBBLE
        OUT     DX, AX

    MOV   AX, DA_CNVTR; I/O ADDRESS OF D/A
        MOV     DX, D0DSTL
        OUT     DX, AX          ;

        MOV     DX, D0DSTH
        XOR   AX, AX            ; CLEAR HIGH NIBBLE
        OUT     DX, AX

        ; THE POINTER ADDRESSES HAVE BEEN SET UP. NOW
        ; WE SET UP THE TRANSFER COUNT.

        MOV     AX, 255         ; 8-BIT D/A SO
                                ; WE SEND 256 BYTES
        MOV     DX, D0TC        ; TO GET A FULL SCALE
        OUT     DX, AX
```

Example 10.2. Timed DMA Transfers (Continued)

```
                ; NOW WE NEED TO SET THE PARAMETERS FOR
                ; THE CHANNEL AS FOLLOWS:
                ;
                ;       DESTINATION     SOURCE
                ;       -----------     ------
                ;       I/O SPACE       MEMORY SPACE
                ;       CONSTANT PTR    INCREMENT PTR
                ;
                ; TERMINATE ON TC, INTERRUPT, SOURCE SYNCHRONIZE,
                ; INTERNAL REQUESTS,
                ; LOW PRIORITY RELATIVE TO CHANNEL 1, BYTE XFERS.

                MOV     AX, 0001011101010110B
                MOV     DX, D0CON
                OUT     DX, AX

                ; NOW WE ASSUME THAT TIMER 2 HAS BEEN PROPERLY
                ; PROGRAMMED FOR A 22US DELAY.

                ; WHEN THE TIMER IS STARTED, A DMA
                ; TRANSFER WILL OCCUR EVERY 22US.

CODE_SEG        ENDS

DATA_SEG        SEGMENT

WAVEFORM_DATA   DB      0,1,2,3,4,5,6,7,8,9,10,11,12,13
                DB      14,15,16,17,18,19,20,21,22,23,24

                ; ETC. UP TO 255


DATA_SEG        ENDS


                END START
```

Example 10.2. Timed DMA Transfers (Continued)

Math Coprocessing 11

CHAPTER 11
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 provided by the 80C186 Modular Core.

11.1. 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 satisfies the need for powerful mathematics. The 80C187 can increase the math performance of the microprocessor system by 50 to 100 times.

11.2. AVAILABILITY OF MATH COPROCESSING

The processor supports the 80C187 with a hardware interface under microcode control. To execute numerics instructions, the 80C186XL must exit reset in Enhanced Mode. The processor checks its $\overline{\text{TEST}}$ pin at reset and enters Enhanced Mode automatically if the Math Coprocessor is present.

The core has a TRAP bit in the Relocation Register to control the availability of math coprocessing. If the bit is a one, 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 8-bit bus version will automatically trap ESC (numerics) opcodes to the Type 7 interrupt regardless of Relocation Register programming.

11.3. 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, usable 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 387™ DX and 387™ SX math coprocessors. The 80C187 conforms to ANSI/IEEE Standard 754-1985.

11.3.1. 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; FSQRT always takes the square root of the number in the top stack element, for example. Other instructions allow or require the programmer to specify explicitly the operand(s) 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 may alter the register when it receives the result of the operation. For most destination operands, the coprocessor usually replaces the destinations with results.

11.3.1.1. 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 of the data types 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 11.1 summarizes the data transfer instructions.

Table 11.1. 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

11.3.1.2. 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 11.2 summarizes the available operation and operand forms for basic arithmetic. In addition to the four normal operations, two "reversed" instructions make subtraction and division "symmetrical" like addition and multiplication. In summary, the arithmetic instructions are highly flexible because:

- The 80C187 uses register or memory operands
- The 80C187 may 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 11.2. 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)

11.3.1.3. COMPARISON INSTRUCTIONS

Each comparison instruction (see Table 11.3) 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 11.3. 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

11.3.1.4. TRANSCENDENTAL INSTRUCTIONS

Transcendental instructions 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 11.4 lists the transcendental instructions.

Table 11.4. 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

11.3.1.5. CONSTANT INSTRUCTIONS

Each constant instruction (see Table 11.5) 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 two bytes long, save memory space.

Table 11.5. 80C187 Constant Instructions

FLDZ	Load +0.1
FLD1	Load +1.0
FLDPI	Load π
FLDL2T	Load $\log_2 10$
FLDL2E	Load $\log_2 e$
FLDLG2	Load $\log_{10} 2$
FLDLG2	Load $\log_e 2$

11.3.1.6. PROCESSOR CONTROL INSTRUCTIONS

Computations do not use the processor control instructions; they are available for activities at the operating system level. This group (see Table 11.6) includes initialization, exception handling and task switching instructions.

Table 11.6. 80C187 Processor Control Instructions

FINIT/FNINIT	Initialize processor
FDISI/FNDISI	Disable interrupts
FENI/FNENI	Enable interrupts
FLDCW	Load control word
FSTCW/FNSTCW	Store control word
FSTSW/FNSTSW	Store status word
FCLEX/FNCLEX	Clear exceptions
FSTENV/FNSTENV	Store environment
FLDENV	Load environment
FSAVE/FNSAVE	Save state
FRSTOR	Restore state
FINCSTP	Increment stack pointer
FDECSTP	Decrement stack pointer
FFREE	Free register
FNOP	No operation
FWAIT	CPU wait

11.3.2. 80C187 DATA TYPES

The microprocessor/math coprocessor combination supports the following 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 11.1 graphically represents these data types.

11.4. MICROPROCESSOR AND COPROCESSOR OPERATION

The 80C187 interfaces directly to the microprocessor (see Figure 11.2) 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 multiplexed with $\overline{\text{MCS3}}$, $\overline{\text{TEST}}$, $\overline{\text{MCS0}}$ and $\overline{\text{MCS1}}$, respectively. When the processor leaves reset, the presence of the 80C187 automatically enables Enhanced Mode and configures the pins correctly. Note that $\overline{\text{MCS2}}$ always retains its function as a chip select. The processor also retains the wait state and ready programming for the entire mid-range memory block, even though $\overline{\text{MCS0}}$, $\overline{\text{MCS1}}$ and $\overline{\text{MCS3}}$ are no longer available.

11.4.1. CLOCKING THE 80C187

The microprocessor and math coprocessor operate asynchronously and their clock rates may differ. The 80C187 has a CKM pin which 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 2X frequency 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.

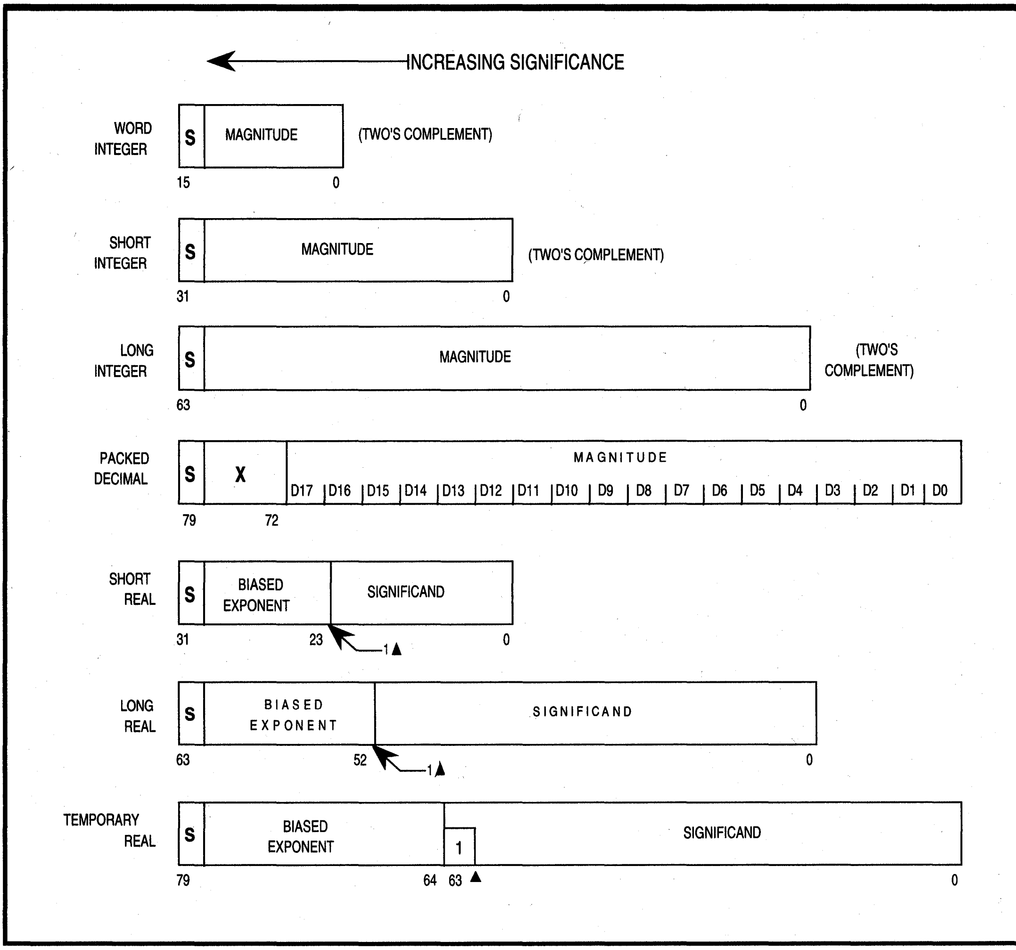


Figure 11.1. 80C187-Supported Data Types

11.4.2. 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 11.7. 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.

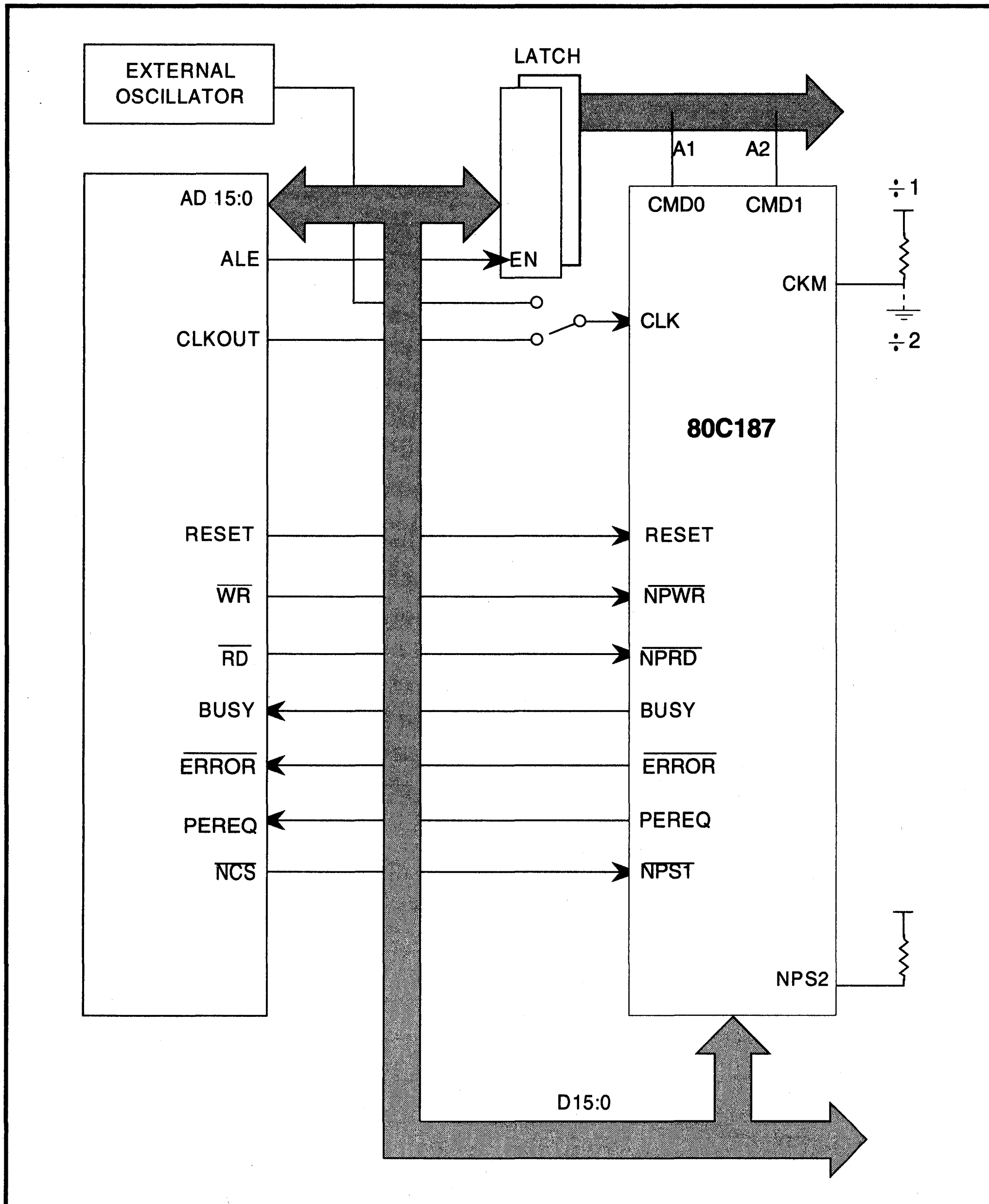


Figure 11.2. 80C186 Modular Core Family/80C187 System Configuration

Table 11.7. 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 with the TRAP bit in the Relocation Register 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) will fail without the 80C187. If an application offers the 80C187 as an option, problems can be prevented in three ways:

- Remove all numerics (ESC) instructions, including code which 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. The next section covers integration of the 80C187 into the overall system.

11.4.3. SYSTEM DESIGN TIPS

All 80C187 operations require that bus ready be asserted. The simplest way to return the ready indication is via hardware connected to the processor's ARDY or SRDY pin. If you program a chip select to cover the math coprocessor port addresses, its ready programming will be in force and can provide bus ready for coprocessor accesses. The user must verify there are no conflicts from other hardware connected to that chip select pin.

A chip select pin will go 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 chip select) regardless of programming.

In a buffered system, it is customary to place the 80C187 on the local bus. Since DT/$\overline{R}$ and $\overline{DEN}$ function normally during 80C187 transfers, you must qualify $\overline{DEN}$ with $\overline{NCS}$ (see Figure 11.3). 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 and DMA 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/HLDA protocol) can suspend coprocessor bus cycles for an indefinite period.

The programmer may lock 80C187 instructions. The CPU asserts the $\overline{LOCK}$ pin for the entire duration of a numerics instruction, monopolizing the bus for a very long time.

11.4.4. 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 $\overline{ERROR}$ pin. The processor tests $\overline{ERROR}$ at the beginning of numerics instructions, so it traps an exception on the **next** attempted numerics instruction after it occurs. When $\overline{ERROR}$ 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 $\overline{ERROR}$ 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 11.4). With this arrangement, use a flip-flop to latch BUSY upon assertion of $\overline{ERROR}$. 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.

11.5. EXAMPLE MATH COPROCESSOR ROUTINES

Example 11.1 shows the initialization sequence for the 80C187. Example 11.2 is an example of a floating point routine using the 80C187. The FSINCOS instruction yields both sine and cosine in one operation.

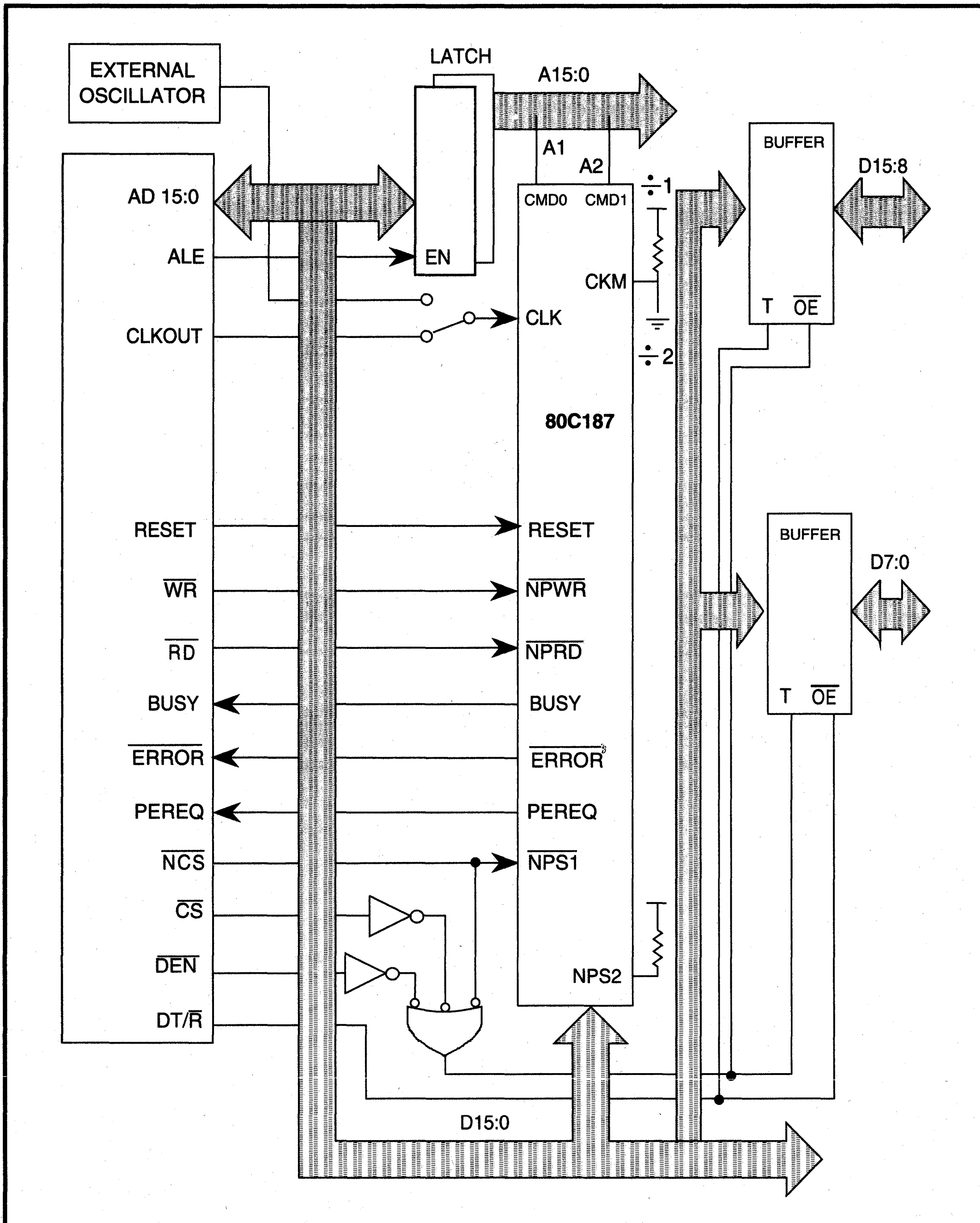


Figure 11.3. 80C187 Configuration with Partially Buffered Bus

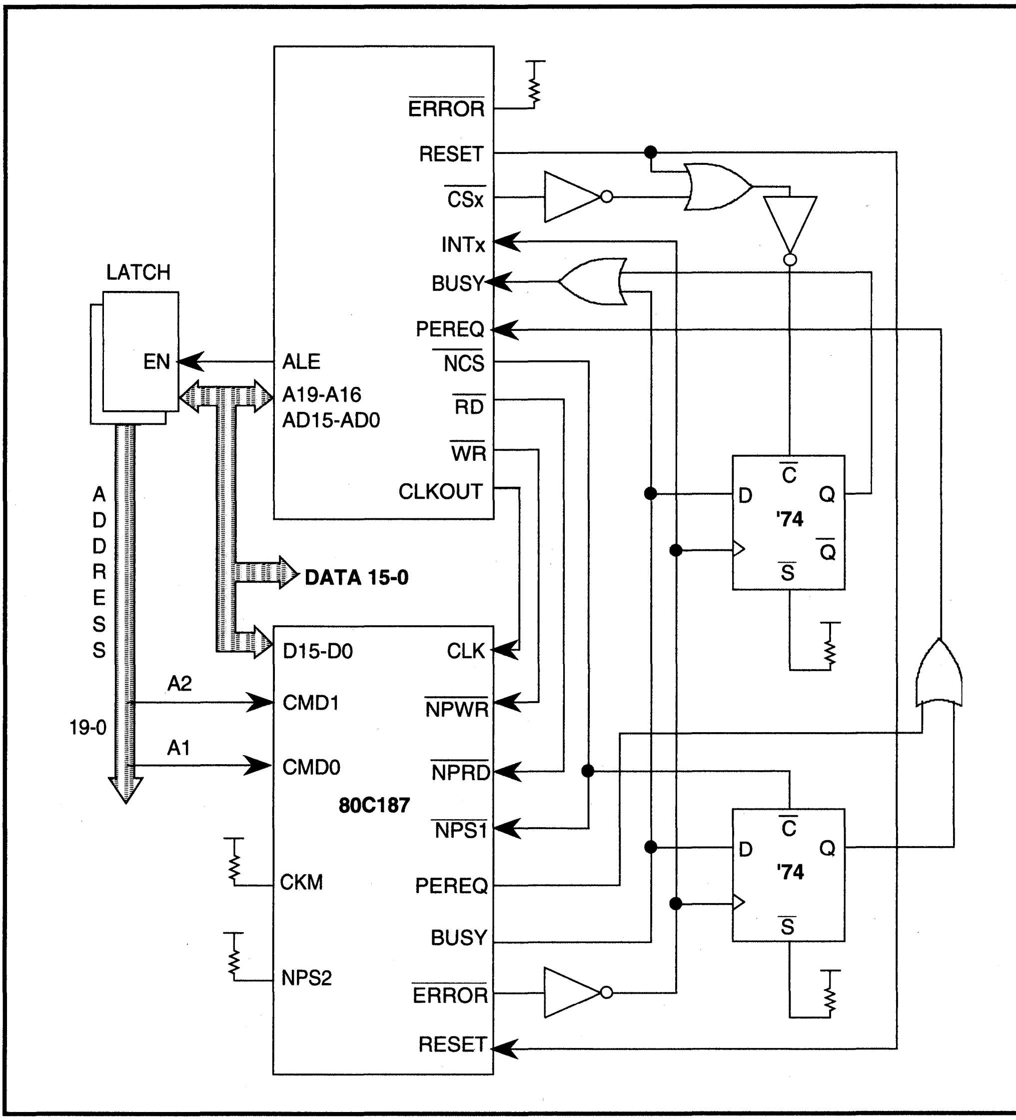


Figure 11.4. 80C187 Exception Trapping via Processor Interrupt Pin

```
$mod186
name          example_80C187_init

;
; FUNCTION: This function initializes the 80C187 numerics
;           co-processor.
;
;   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_init   proc 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)
```

Example 11.1. Initialization Sequence for 80C187 Math Coprocessor

```
            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_init   endp

lib_80186   ends
            end
```

Example 11.1. Initialization Sequence for 80C187 Math Coprocessor (Continued)

```
$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 a HLL-callable routine.
;
;                 This code assumes that the 80C187 has already
;                 been initialized.
;
                  assume cs:code, ds:data
```

Example 11.2. Floating Point Math Routine Using FSINCOS

```
            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 11.2. Floating Point Math Routine Using FSINCOS (Continued)

ONCE™ Mode 12

CHAPTER 12
ONCE™ MODE

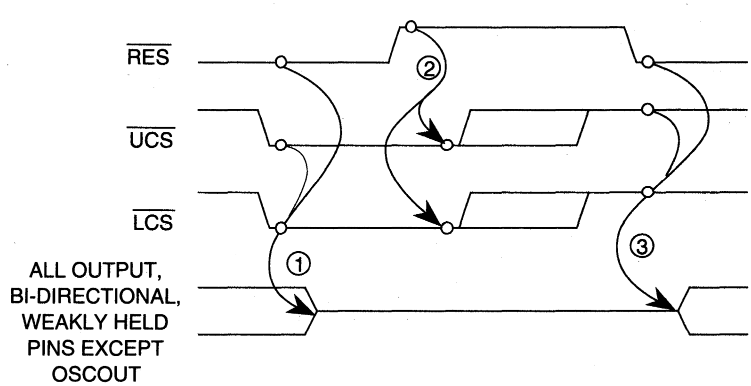
ONCE (pronounced: ahnce) Mode provides the ability to three-state all output, bidirectional, or weakly held high/low pins except X2. X2 does not three-state to allow device operation with a crystal network.

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).

12.1. ENTERING/LEAVING ONCE MODE

Forcing $\overline{UCS}$ and $\overline{LCS}$ low while $\overline{RES}$ is asserted (low) enables ONCE Mode (see Figure 12.1). Maintaining $\overline{UCS}$, $\overline{LCS}$ and $\overline{RES}$ low continues to keep ONCE Mode active. Returning $\overline{UCS}$ and/or $\overline{LCS}$ back high exits the ONCE Mode.

However, it is possible to always keep ONCE Mode active by deasserting $\overline{RES}$ while keeping $\overline{UCS}$ and $\overline{LCS}$ low. Removing $\overline{RES}$ "latches" ONCE Mode and allows $\overline{UCS}$ and $\overline{LCS}$ to be driven to any level. $\overline{UCS}$ and $\overline{LCS}$ must remain low for at least one clock beyond the time $\overline{RES}$ is driven high. Asserting $\overline{RES}$ exits ONCE Mode, assuming $\overline{UCS}$ and $\overline{LCS}$ do not remain low also (see Figure 12.1).



NOTES: 1. Entering ONCE Mode.
2. Latching ONCE Mode.
3. Leaving ONCE Mode (assuming 2. occurred)

Figure 12.1. Entering/Leaving ONCE Mode

Appendix A
80C186 Instruction Set Additions and Extensions

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, there are several additional instructions that were not available in the 8086/8088 instruction set. Second, there are several 8086/8088 instructions that have been enhanced for the 80C186 Modular Core family instruction set.

A.1. 80C186 INSTRUCTION SET ADDITIONS

The following sections describe instructions 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.

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: This instruction does not save segment registers (CS, DS, SS, ES), the Instruction Pointer (IP), the Program 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 1 for byte transfers or 2 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 1 for byte transfers or 2 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: the higher the lexical nesting level, the lower the procedure is in the nesting hierarchy.

The lexical nesting level determines the number 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, 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.

ENTER treats a reentrant procedure as a procedure calling another procedure at the same lexical level. A reentrant procedure can only address 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, as shown in Figure A.2, 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 PROCEDURE B because they operate at the same lexical nesting level. The following is a summary of the variable access for Figure A.2.

The formal definition of the ENTER instruction for all cases is given by the following listing: (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

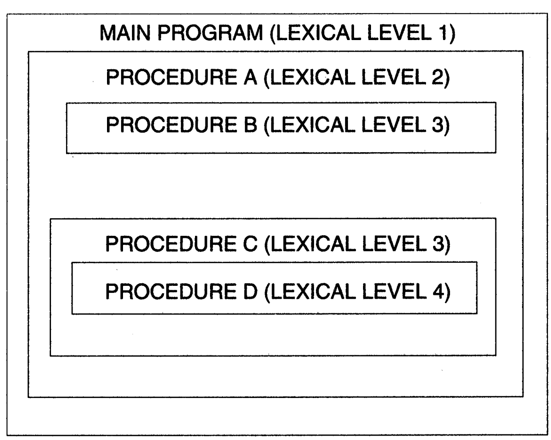


Figure A.2. Variable Access in Nested Procedures

1. MAIN PROGRAM 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.

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).

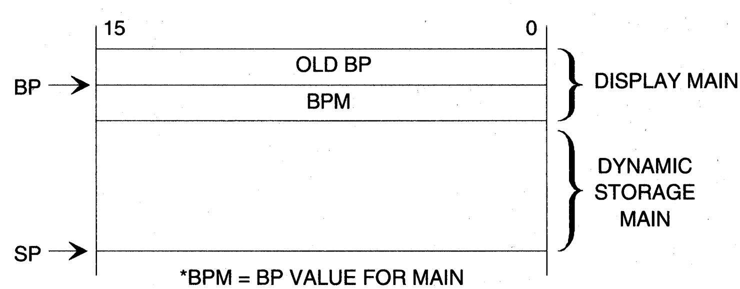


Figure A.3. Stack Frame for MAIN at Level 1

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).

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).

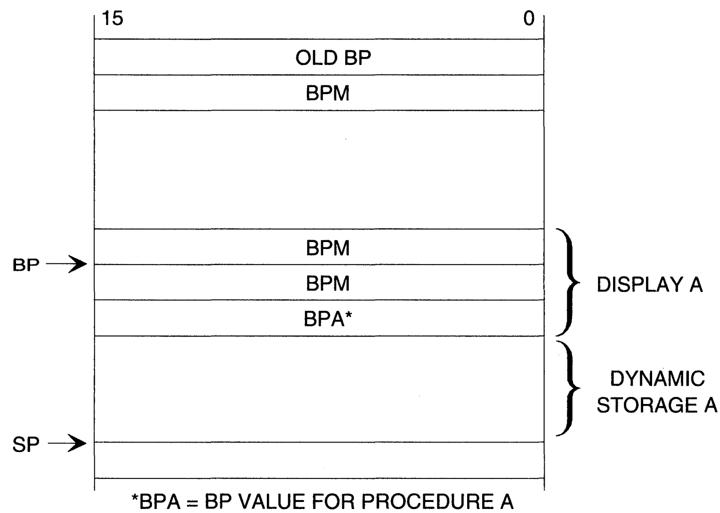


Figure A.4. Stack Frame for Procedure A at Level 2

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).

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. An 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.

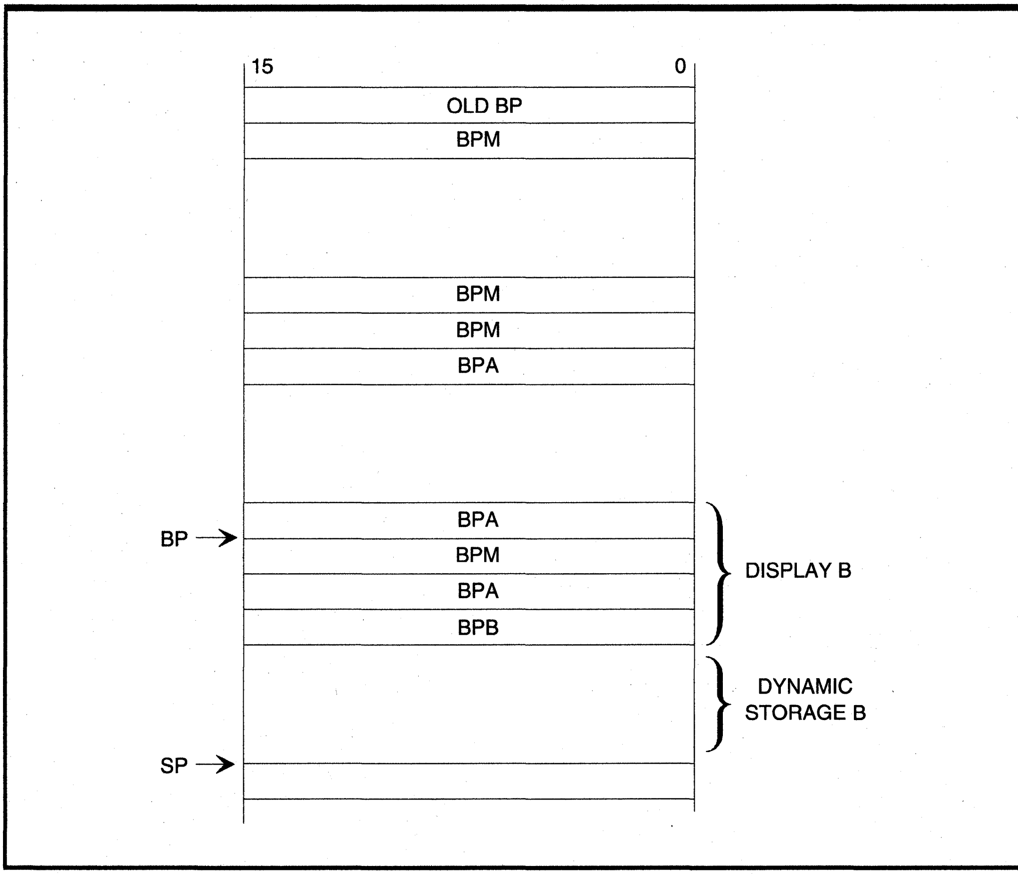


Figure A.5. Stack Frame for Procedure B at Level 3 Called from A

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).

BOUND is useful for checking array bounds before attempting to access an array element. This avoids the program overwriting information outside the limits of the array.

A.2. 80C186 INSTRUCTION SET ENHANCEMENTS

The following sections describe enhancements to the 8086/8088 instruction set available with the 80C186 Modular Core family. These instructions were available with the 8086/8088 instruction set, but have been expanded to be more useful.

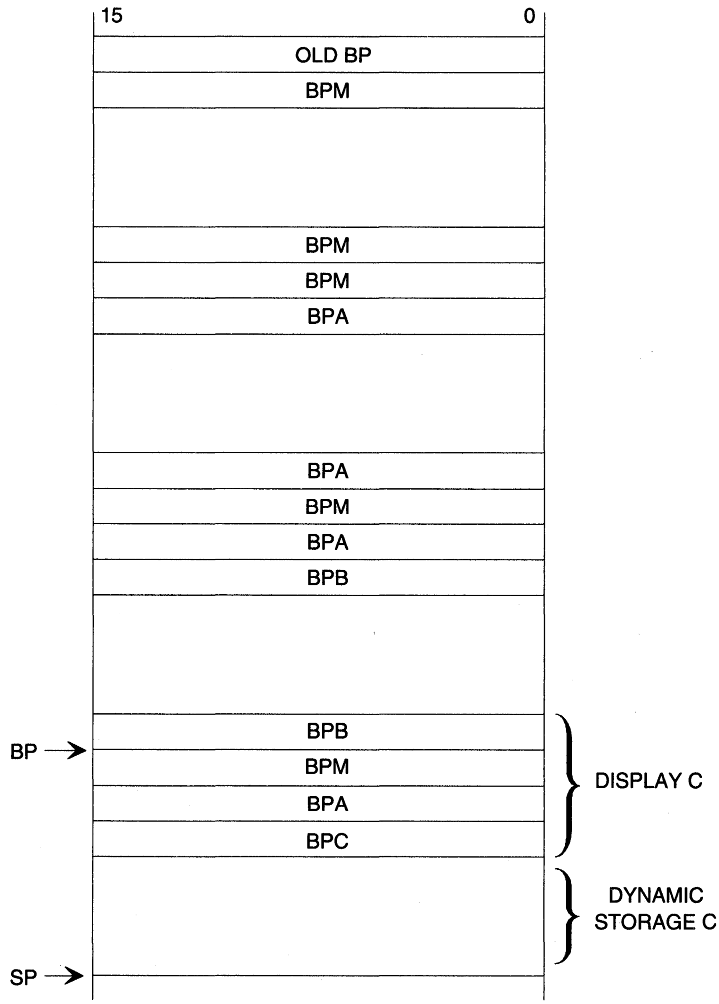


Figure A.6. Stack Frame for Procedure C at Level 3 Called from B

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 will be 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 be 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

The 80C186 Modular Core instruction set includes enhancements to the bit manipulation instructions. The following sections describe these enhancements.

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.

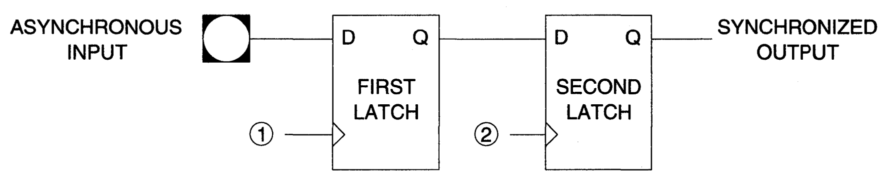
RCR ***destination, count***

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.

Appendix B
Input Synchronization

APPENDIX B
INPUT SYNCHRONIZATION

Many input signals to an embedded processor are asynchronous. Asynchronous signals do not **require** a specified set up 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) which 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.



NOTES:

1. First latch sample clock, can be phase 1 or phase 2 depending on pin function
2. Second latch sample clock, opposite phase of first latch sample clock (e.g. if first latch is sampled with phase 1, the second latch is sampled with phase 2).

Figure B.1. Input Synchronization Circuit

B.1. WHY SYNCHRONIZERS ARE REQUIRED

Every data latch requires a specific set up and hold time to operate properly. The duration of the setup and hold time defines a **window** where 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 a larger window be specified to cover all conditions.

Should the input to the data latch transition during the sample and hold window, the output of the latch eventually attains a stable state. Reaching this stable state must occur before the second stage of sychroniztion requires a valid input. To synchronize an asynchronous signal, the circuit in Figure B-1 samples the input into the first latch, allows to output to stabilize, then samples the stabilized value into a second latch. With the asynchronous signal resolved in this way, the input signal can not cause a 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

Those inputs that use the two stage synchronization circuit are: TMR IN 0, TMR IN 1, NMI, $\overline{\text{TEST}}$/BUSY, INT0-3, HOLD, DRQ0, and DRQ1.

Appendix C

APPENDIX C

Table C.1. Instruction Set Summary

Function	Format	Clock Cycles	Comments
DATA TRANSFER			
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 0 0 0 r/m | data | data if w=1	12-13	8/16-bit
Immediate to register	1 0 1 1 w reg | data | data if w=1	3-4	8/16-bit
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 1 1 0 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	
PUSHA = Push All	0 1 1 0 0 0 0 0	36	
POP = Pop:			
Memory	1 0 0 0 1 1 1 1 | mod 0 0 0 r/m	20	
Register	0 1 0 1 1 reg	10	
Segment register	0 0 0 reg 1 1 1 (reg?01)	8	
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	
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 to:			
Fixed port	1 1 1 0 0 1 1 w | port	9	
Variable port	1 1 1 0 1 1 1 w	7	
XLAT = Translate byte to AL	1 1 0 1 0 1 1 1	11	
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	
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	

Shaded areas indicate instructions not available in iAPX 86, 88 microsystems.

Table C.1. Instruction Set Summary (Continued)

Function	Format				Clock Cycles	Comments
DATA TRANSFER (Continued)						
SEGMENT = Segment Override:						
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	
ARITHMETIC						
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 0 0 0 r/m	data	data if s w=01	4/16	
Immediate to accumulator	0 0 0 0 0 1 0 w	data	data if w=1		3/4	8/16-bit
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 0 1 0 r/m	data	data if s w=01	4/16	
Immediate to accumulator	0 0 0 1 0 1 0 w	data	data if w=1		3/4	8/16-bit
INC = Increment						
Register/memory	1 1 1 1 1 1 1 w	mod 0 0 0 r/m			3/15	
Register	0 1 0 0 0 reg				3	
SUB = Subtract						
Reg/memory and 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 1 0 1 r/m	data	data if s w=01	4/16	
Immediate from accumulator	0 0 1 0 1 1 0 w	data	data if w=1		3/4	8/16-bit
SBB = Subtract with borrow						
Reg/memory and 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 0 1 1 r/m	data	data if s w=01	4/16	
Immediate from accumulator	0 0 0 1 1 1 0 w	data	data if w=1		3/4	8/16-bit
DEC = Decrement:						
Register/memory	1 1 1 1 1 1 1 w	mod 0 0 1 r/m			3/15	
Register	0 1 0 0 1 reg				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 1 1 1 r/m	data	data if s w=01	3/10	
Immediate with accumulator	0 0 1 1 1 1 0 w	data	data if w=1		3/4	8/16-bit
NEG = Change sign	1 1 1 1 0 1 1 w	mod 0 1 1 r/m			3	
AAA = ASCII adjust for Add	0 0 1 1 0 1 1 1				8	
DAA = Decimal adjust for add	0 0 1 0 0 1 1 1				4	
AAS = ASCII adjust for subtract	0 0 1 1 1 1 1 1				7	
DAS = Decimal adjust for subtract	0 0 1 0 1 1 1 1				4	
MUL = Multiply (unsigned):	1 1 1 1 0 1 1 w	mod 1 0 0 r/m				
Register-Byte					26-28	
Register-Word					35-37	
Memory-Byte					32-34	
Memory-Word					41-43	

Shaded areas indicate instructions not available in iAPX 86, 88 microsystems.

Table C.1. Instruction Set Summary (Continued)

Function	Format	Clock Cycles	Comments
ARITHMETIC (Continued)			
IMUL = Integer multiply (signed):	1 1 1 1 0 1 1 w | mod 1 0 1 r/m		
Register-Byte		25-28	
Register-Word		34-37	
Memory-Byte		31-34	
Memory-Word		40-43	
IMUL = Integer immediate multiply (signed):	0 1 1 0 1 0 s 1 | mod reg r/m | data | data if s=0	22-25/29-32	
DIV = Divide (unsigned):	1 1 1 1 0 1 1 w | mod 1 1 0 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 1 1 1 r/m		
Register-Byte		44-52	
Register-Word		53-61	
Memory-Byte		50-58	
Memory-Word		59-67	
AAM = ASCII adjust for multiply	1 1 0 1 0 1 0 0 | 0 0 0 0 1 0 1 0	19	
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	
LOGIC			
Shift/Rotate Instructions:			
Register/Memory by 1	1 1 0 1 0 0 0 w | mod TTT r/m	2/15	
Register/Memory by CL	1 1 0 1 0 0 1 w | mod TTT r/m	5+n/17+n	
Register/Memory by Count	1 1 0 0 0 0 0 w | mod TTT r/m | count	5+n/17+n	

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 1	SAR

Function	Format	Clock Cycles	Comments
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 1 0 0 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	8/16-bit
TEST = And function to flags, no result:			
Register/memory and register	1 0 0 0 0 1 0 w | mod reg r/m	3/10	
Immediate data and register/memory	1 1 1 1 0 1 1 w | mod 0 0 0 r/m | data | data if w=1	4/10	
Immediate data and accumulator	1 0 1 0 1 0 0 w | data | data if w=1	3/4	8/16-bit
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 0 0 1 r/m | data | data if w=1	4/16	
Immediate to accumulator	0 0 0 0 1 1 0 w | data | data if w=1	3/4	8/16-bit

Shaded areas indicate instructions not available in iAPX 86, 88 microsystems.

Table C.1. Instruction Set Summary (Continued)

Function	Format				Clock Cycles	Comments
LOGIC (Continued)						
XOR = Exclusive or:						
Reg/memory and register to either	0 0 1 1 0 0 d w	mod reg r/m			3/10	
Immediate to register/memory	1 0 0 0 0 0 0 w	mod 1 1 0 r/m	data	data if w=1	4/16	
Immediate to accumulator	0 0 1 1 0 1 0 w	data	data if w=1		3/4	8/16-bit
Not = Invert register/memory	1 1 1 1 0 1 1 w	mod 0 1 0 r/m			3	
STRING MANIPULATION:						
MOVS = Move byte/word	1 0 1 0 0 1 0 w				14	
CMPS = Compare byte/word	1 0 1 0 0 1 1 w				22	
SCAS = Scan byte/word	1 0 1 0 1 1 1 w				15	
LODS = Load byte/wd to AL/AX	1 0 1 0 1 1 0 w				12	
STOS = Stor byte/wd from AL/A	1 0 1 0 1 0 1 w				10	
INS = Input byte/wd from DX port	0 1 1 0 1 1 0 w				14	
OUTS = Output byte/wd to DX port	0 1 1 0 1 1 1 w				14	
Repeated by count in CX						
MOVS - Move string	1 1 1 1 0 0 1 0	1 0 1 0 0 1 0 w			8+8n	
CMPS - Compare string	1 1 1 1 0 0 1 z	1 0 1 0 0 1 1 w			5+22n	
SCAS - Scan string	1 1 1 1 0 0 1 z	1 0 1 0 1 1 1 w			5+15n	
LODS - Load string	1 1 1 1 0 0 1 0	1 0 1 0 1 1 0 w			6+11n	
STOS - Store string	1 1 1 1 0 1 0 0	0 1 0 1 0 0 1 w			6+9n	
INS - Input string	1 1 1 1 0 0 1 0	0 1 1 0 1 1 0 w			8+8n	
OUTS - Output string	1 1 1 1 0 0 1 0	0 1 1 0 1 1 1 w			8+8n	
CONTROL TRANSFER						
CALL = Call:						
Direct within segment	1 1 1 0 1 0 0 0	disp-low	disp-hour		15	
Register memory indirect within segment	1 1 1 1 1 1 1 1	mod 0 1 0 r/m			13/19	
Direct intersegment	1 0 0 1 1 0 1 0	segment offset			23	
		selector				
Indirect intersegment	1 1 1 1 1 1 1 1	mod 0 1 1 r/m	(mod ? 11)		38	
JMP = Unconditional jump:						
Short/long	1 1 1 0 1 0 1 1	disp-low			14	
Direct within segment	1 1 1 0 1 0 0 1	disp-low	disp-high		14	
Register/memory indirect with segment	1 1 1 1 1 1 1 1	mod 1 0 0 r/m			26	
Direct intersegment	1 1 1 0 1 0 1 0	segment offset			14	
		selector				
Indirect intersegment	1 1 1 1 1 1 1 1	mod 1 0 1 r/m	(mod ? 11)		11/17	
RET = Return from CHPS:						
Within segment	1 1 0 0 0 0 1 1				16	
With seg adding immed to SP	1 1 0 0 0 0 1 0	data-low	data-high		18	
Intersegment	1 1 0 0 1 0 1 1				22	
Intersegment adding immediate to SP	1 1 0 0 1 0 1 0	data-low	data-high		25	
JE/JZ = Jump on equal zero	0 1 1 1 0 1 0 0	disp			4/13	13 if JMP taken
JL/JNGE = Jump on less/not greater or equal	0 1 1 1 1 1 0 0	disp			4/13	4 if Jmp
JLE/JNG = Jump on less or equal/not greater	0 1 1 1 1 1 1 0	disp			4/13	not taken

Shaded areas indicate instructions not available in iAPX 86, 88 microsystems.

Table C.1. Instruction Set Summary (Continued)

Function	Format	Clock Cycles	Comments
Control Transfer (Continued)			
JB/JNAE = Jump on below/not above or equal	0 1 1 1 0 0 1 0 | disp	4/13	
JBE/JNA = Jump on below or equal/not above	0 1 1 1 0 1 1 0 | disp	4/13	
JP/JPE = Jump on parity/parity even	0 1 1 1 1 0 1 0 | disp	4/13	
JO = Jump on overflow	0 1 1 1 0 0 0 0 | disp	4/13	
JS = Jump on sign	0 1 1 1 1 0 0 0 | disp	4/13	
JNE/JNZ = Jump on not equal/not zero	0 1 1 1 0 1 0 1 | disp	4/13	
JNL/JGE = Jump on not less/greater or equal	0 1 1 1 1 1 0 1 | disp	4/13	
JNLE/JG = Jump on not less or equal/greater	0 1 1 1 1 1 1 1 | disp	4/13	
JNB/JAE = Jump on not below/above or equal	0 1 1 1 0 0 1 1 | disp	4/13	
JNBE/JA = Jump on not below or equal/above	0 1 1 1 0 1 1 1 | disp	4/13	
JNP/JPO = Jump on not par/par odd	0 1 1 1 1 0 1 1 | disp	4/13	
JNO = Jump on not overflow	0 1 1 1 0 0 0 1 | disp	4/13	
JNS = Jump on not sign	0 1 1 1 1 0 0 1 | disp	5/15	
JCXZ = Jump on CX zero	1 1 1 0 0 0 1 1 | disp	6/16	
LOOP = Loop CX times	1 1 1 0 0 0 1 0 | disp	6/16	
LOOPZ/LOOPE = Loop while zero/equal	1 1 1 0 0 0 0 1 | disp	16	JMP taken/
LOOPNZ/LOOPNE = Loop while not zero/equal	1 1 1 0 0 0 0 0 | disp	5	JMP not taken
ENTER = Enter Procedure	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 = Leave Procedure	1 1 0 0 1 0 0 1	8	
INT = Interrupt:			
Type specified	1 1 0 0 1 1 0 1 | type	47	if INT taken/
Type 3	1 1 0 0 1 1 0 0	45	if INT not
INTO = Interrupt on overflow	1 1 0 0 1 1 1 0	48/4	taken
IRET = Interrupt return	1 1 0 0 1 1 1 1	28	
BOUND = Detect value out of range	0 1 1 0 0 0 1 0 | mod reg r/m	33-35	
PROCESSOR CONTROL			
CLC = Clear carry	1 1 1 1 1 0 0 0	2	
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	if $\overline{\text{test}}$ = 0
LOCK = Bus lock prefix	1 1 1 1 0 0 0 0	2	
ESC = Processor extension escape	1 1 0 1 1 T T T | mod L L L r/m	6	
	(TTT LLL are opcode to processor extension)		

Shaded areas indicate instructions not available in iAPX 86, 88 microsystems.

FOOT NOTES

The Effective Address (EA) of the memory operand is computed according to the mod and r/m fields:

if mod = 11 then r/m is treated as a REG field
if mod = 00 then DISP = 0*, disp-low and disp-high are absent
if mod = 01 then DISP = disp-low sign-extended to 16-bits, disp-high is absent
if mod = 10 then DISP = disp-high:disp-low

if r/m = 000 then EA = (BX) + (SI) + DISP
if r/m = 001 then EA = (BX) + (DI) + DISP
if r/m = 010 then EA = (BP) + (SI) + DISP
if r/m = 011 then EA = (BP) + (DI) + DISP
if r/m = 100 then EA = (SI) + DISP
if r/m = 101 then EA = (DI) + DISP
if r/m = 110 then EA = (BP) + DISP*
if r/m = 111 then EA = (BX) + DISP

DISP follows 2nd byte of instruction (before data if required)

*except if mod = 00 and r/m = 110 then EA = disp-high:disp-low.

SEGMENT OVERRIDE PREFIX

0	0	1	reg	1	1	0

reg is assigned according to the following:

reg	**Segment Register**
00	ES
01	CS
10	SS
11	DS

REG is assigned according to the following table:

16-Bit (w=1)	**8-Bit (w=0)**
000 AX	000 AL
001 CX	001 CL
010 DX	010 DL
011 BX	011 BL
100 SP	100 AH
101 BP	101 CH
110 SI	110 DH
111 DI	111 BH

The physical address of all operands addressed by the BP register are computed using the SS segment register. The physical addresses of the destination operands of the string primitive operation (those addressed by the DI register) are computed using the ES segment, which may not be overridden.

Table C.2. Machine Instruction Decoding Guide

1ST BYTE		2ND BYTE	BYTES 3,4,5,6	ASM-86 INSTRUCTION FORMAT	
HEX	BINARY				
00	0000 0000	MOD REG R/M	(DISP-LO),(DISP-HI)	ADD	REG8/MEM8,REG8
01	0000 0001	MOD REG R/M	(DISP-LO),(DISP-HI)	ADD	REG16/EM16,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
09	0000 1001	MOD REG R/M	(DISP-LO),(DISP-HI)	OR	REG16/MEM16,REG16
0A	0000 1010	MOD REG R/M	(DISP-LO),(DISP-HI)	OR	REG8,REG8/MEM8
0B	0000 1011	MOD REG R/M	(DISP-LO),(DISP-HI)	OR	REG16,REG16/MEM16
0C	0000 1100	DATA-8		OR	AL,IMMED8
0D	0000 1101	DATA-LO	DATA-HI	OR	AX,IMMED16
0E	0000 1110			PUSH	CS
0F	0000 1111			(not used)	
10	0001 0000	MOD REG R/M	(DISP-LO),(DISP-HI)	ADC	REG8/MEM8,REG8
11	0001 0001	MOD REG R/M	(DISP-LO),(DISP-HI)	ADC	REG16/MEM16,REG16
12	0001 0010	MOD REG R/M	(DISP-LO),(DISP-HI)	ADC	REG8,REG8/MEM8
13	0001 0011	MOD REG R/M	(DISP-LO),(DISP-HI)	ADC	REG16,REG16/MEM16
14	0001 0100	DATA-8		ADC	AL,IMMED8
15	0001 0101	DATA-LO	DATA-HI	ADC	AX,IMMED16
16	0001 0110			PUSH	SS
17	0001 0111			POP	SS
18	0001 1000	MOD REG R/M	(DISP-LO),(DISP-HI)	SBB	REG8/MEM8,REG8
19	0001 1001	MOD REG R/M	(DISP-LO),(DISP-HI)	SBB	REG16/MEM16,REG16
1A	0001 1010	MOD REG R/M	(DISP-LO),(DISP-HI)	SBB	REG8,REG8/MEM8
1B	0001 1011	MOD REG R/M	(DISP-LO),(DISP-HI)	SBB	REG16,REG16/MEM16
1C	0001 1100	DATA-8		SBB	AL,IMMED8
1D	0001 1101	DATA-LO	DATA-HI	SBB	AX,IMMED16
1E	0001 1110			PUSH	DS
1F	0001 1111			POP	DS
20	0010 0000	MOD REG R/M	(DISP-LO),(DISP-HI)	AND	REG8/MEM8,REG8
21	0010 0001	MOD REG R/M	(DISP-LO),(DISP-HI)	AND	REG16/MEM16,REG16
22	0010 0010	MOD REG R/M	(DISP-LO),(DISP-HI)	AND	REG8,REG8/MEM8
23	0010 0011	MOD REG R/M	(DISP-LO),(DISP-HI)	AND	REG16,REG16/MEM16
24	0010 0100	DATA-8		AND	AL,IMMED8
25	0010 0101	DATA-LO	DATA-HI	AND	AX,IMMED16
26	0010 0110			ES:	(segment override prefix)
27	0010 0111			DAA	
28	0010 1000	MOD REG R/M	(DISP-LO),(DISP-HI)	SUB	REG8/MEM8,REG8
29	0010 1001	MOD REG R/M	(DISP-LO),(DISP-HI)	SUB	REG16/MEM16,REG16
2A	0010 1010	MOD REG R/M	(DISP-LO),(DISP-HI)	SUB	REG8,REG8/MEM8
2B	0010 1011	MOD REG R/M	(DISP-LO),(DISP-HI)	SUB	REG16,REG16/MEM16
2C	0010 1100	DATA-8		SUB	AL,IMMED8
2D	0010 1100	DATA-LO	DATA-HI	SUB	AX,IMMED16

Table C.2. Machine Instruction Decoding Guide (Continued)

1ST BYTE		2ND BYTE	BYTES 3,4,5,6	ASM-86 INSTRUCTION FORMAT	
HEX	BINARY				
2E	0010 1110			CS:	(segment override prefix)
2F	0010 1111			DAS	
30	0011 0000	MOD REG R/M	(DISP-LO),(DISP-HI)	XOR	REG8/MEM8,REG8
31	0011 0001	MOD REG R/M	(DISP-LO),(DISP-HI)	XOR	REG16/MEM16,REG16
32	0011 0010	MOD REG R/M	(DISP-LO),(DISP-HI)	XOR	REG8,REG8/MEM8
33	0011 0011	MOD REG R/M	(DISP-LO),(DISP-HI)	XOR	REG16,REG16/MEM16
34	0011 0100	DATA-8		XOR	AL,IMMED8
35	0011 0100	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)	CMP	REG8/MEM8,REG8
39	0011 1001	MOD REG R/M	(DISP-LO),(DISP-HI)	CMP	REG16/MEM16,REG16
3A	0011 1010	MOD REG R/M	(DISP-LO),(DISP-HI)	CMP	REG8,REG8/MEM8
3B	0011 1011	MOD REG R/M	(DISP-LO),(DISP-HI)	CMP	REG16,REG16/MEM16
3C	0011 1100	DATA-8		CMP	AL,IMMED8
3D	0011 1101	DATA-LO	DATA-HI	CMP	AX,IMMED16
3E	0011 1110			DS:	(segment override prefix)
3F	0011 1111			AAS	
40	0100 0000			INC	AX
41	0100 0001			INC	CX
42	0100 0010			INC	DX
43	0100 0011			INC	BX
44	0100 0100			INC	SP
45	0100 0101			INC	BP
46	0100 0110			INC	SI
47	0100 0111			INC	DI
48	0100 1000			DEC	AX
49	0100 1001			DEC	CX
4A	0100 1010			DEC	DX
4B	0100 1011			DEC	BX
4C	0100 1100			DEC	SP
4D	0100 1101			DEC	BP
4E	0100 1110			DEC	SI
4F	0100 1111			DEC	DI
50	0101 0000			PUSH	AX
51	0101 0001			PUSH	CX
52	0101 0010			PUSH	DX
53	0101 0011			PUSH	BX
54	0101 0100			PUSH	SP
55	0101 0101			PUSH	BP
56	0101 0110			PUSH	SI
57	0101 0111			PUSH	DI
58	0101 1000			POP	AX
59	0101 1001			POP	CX
5A	0101 1010			POP	DX
5B	0101 1011			POP	BX

Table C.2. Machine Instruction Decoding Guide (Continued)

1ST BYTE		2ND BYTE	BYTES 3,4,5,6	ASM-86 INSTRUCTION FORMAT	
HEX	BINARY				
5C	0101 1100			POP	SP
5D	0101 1101			POP	BP
5E	0101 1110			POP	SI
5F	0101 1111			POP	DI
60	0110 0000			PUSHA	(186/8 ONLY)
61	0110 0001			POPA	(186/8 ONLY)
62	0110 0010	MOD REG R/M		BOUND	REG16,MEM16(186/8 ONLY)
63	0110 0011			(not used)	
64	0110 0100			(not used)	
65	0110 0101			(not used)	
66	0110 0110			(not used)	
67	0110 0111			(not used)	
68	0110 1000	DATA-LO	DATA-HI	PUSH	IMMED16(186/8 ONLY)
69	0110 1001	MOD REG R/M	DATA-LO,DATA-HI	IMUL	IMMED16(186/8 ONLY)
6A	0110 1010	DATA-8		PUSH	IMMED8(186/8 ONLY)
6B	0110 1011	MOD REG R/M	DATA-8	IMUL	IMMED8(186/8 ONLY)
6C	0110 1100			INS	MEM8,DX(186/8 ONLY)
6D	0110 1101			INS	MEM16,DX(186/8 ONLY)
6E	0110 1110			OUTS	MEM8,CX(186/8 ONLY)
6F	0110 1111			OUTS	MEM16,DX(186/8 ONLY)
70	0111 0000	IP-INC8		JO	SHORT-LABEL
71	0111 0001	IP-INC8		JNO	SHORT-LABEL
72	0111 0010	IP-INC8		JB/ JNAE/ JC	SHORT-LABEL
73	0111 0011	IP-INC8		JNB/ JAE/ JNC	SHORT-LABEL
74	0111 0100	IP-INC8		JE/JZ	SHORT-LABEL
75	0111 0101	IP-INC8		JNE/JNZ	SHORT-LABEL
76	0111 0110	IP-INC8		JBE/JNA	SHORT-LABEL
77	0111 0111	IP-INC8		JNBE/ JA	SHORT-LABEL
78	0111 1000	IP-INC8		JS	SHORT-LABEL
79	0111 1001	IP-INC8		JNS	SHORT-LABEL
7A	0111 1010	IP-INC8		JP/JPE	SHORT-LABEL
7B	0111 1011	IP-INC8		JNP/JPO	SHORT-LABEL
7C	0111 1100	IP-INC8		JL/ JNGE	SHORT-LABEL
7D	0111 1101	IP-INC8		JNLJGE	SHORT-LABEL
7E	0111 1110	IP-INC8		JLE/ JNG	SHORT-LABEL
7F	0111 1111	IP-INC8		JNLE/ JG	SHORT-LABEL
80	1000 0000	MOD 000 R/M	(DISP LO),(DISP HI) DATA-8	ADD	REG8/MEM8,IMMED8

Table C.2. Machine Instruction Decoding Guide (Continued)

1ST BYTE		2ND BYTE	BYTES 3,4,5,6	ASM-86 INSTRUCTION FORMAT
HEX	BINARY			
80	1000 0000	MOD 001 R/M	(DISP-LO),(DISP-HI), DATA-8	OR REG8/MEM8,IMMED8
80	1000 000	MOD 010 R/M	(DISP-LO),(DISP-HI), DATA-8	ADC REG8/MEM8,IMMED8
80	1000 0000	MOD 011 R/M	(DISP-LO),(DISP-HI), DATA-8	SBB REG8/MEM8,IMMED8
80	1000 0000	MOD 100 R/M	(DISP-LO),(DISP-HI), DATA-8	AND REG8/MEM8,IMMED8
80	1000 0000	MOD 101 R/M	(DISP-LO),(DISP-HI), DATA-8	SUB REG8/MEM8,IMMED8
80	1000 0000	MOD 110 R/M	(DISP-LO),(DISP-HI), DATA-8	XOR REG8/MEM8,IMMED8
80	1000 0000	MOD 111 R/M	(DISP-LO),(DISP-HI), DATA-8	CMP REG8/MEM8,IMMED8
81	1000 0001	MOD 000 R/M	(DISP-LO),(DISP-HI), DATA-LO,DATA-HI	ADD REG16/MEM16,IMMED16
81	1000 0001	MOD 001 R/M	(DISP-LO),(DISP-HI), DATA-LO,DATA-HI	OR REG16/MEM16,IMMED16
81	1000 0001	MOD 010 R/M	(DISP-LO),(DISP-HI), DATA-LO,DATA-HI	ADC REG16/MEM16,IMMED16
81	1000 0001	MOD 011 R/M	(DISP-LO),(DISP-HI), DATA-LO,DATA-HI	SBB REG16/MEM16,IMMED16
81	1000 0001	MOD 100 R/M	(DISP-LO),(DISP-HI), DATA-LO,DATA-HI	AND REG16/MEM16,IMMED16
81	1000 0001	MOD 101 R/M	(DISP-LO),(DISP-HI), DATA-LO,DATA-HI	SUB REG16/MEM16,IMMED16
81	1000 0001	MOD 110 R/M	(DISP-LO),(DISP-HI), DATA-LO,DATA-HI	XOR REG16/MEM16,IMMED16
81	1000 0001	MOD 111 R/M	(DISP-LO),(DISP-HI), DATA-LO,DATA-HI	CMP REG16/MEM16,IMMED16
82	1000 0010	MOD 000 R/M	(DISP-LO),(DISP-HI), DATA-8	ADD REG8/MEM8,IMMED8
82	1000 0010	MOD 001 R/M		(not used)
82	1000 0010	MOD 010 R/M	(DISP-LO),(DISP-HI), DATA-8	ADC REG8/MEM8,IMMED8
82	1000 0010	MOD 011 R/M	(DISP-LO),(DISP-HI), DATA-8	SBB REG8/MEM8,IMMED8
82	1000 0010	MOD 100 R/M		(not used)
82	1000 0010	MOD 101 R/M	(DISP-LO),(DISP-HI), DATA-8	SUB REG8/MEM8,IMMED8
82	1000 0010	MOD 110 R/M		(not used)
82	1000 0010	MOD 111 R/M	(DISP-LO),(DISP-HI), DATA-8	CMP REG8/MEM8,IMMED8
83	1000 0011	MOD 000 R/M	(DISP-LO),(DISP-HI), DATA-SX	ADD REG16/MEM16,IMMED8
83	1000 0011	MOD 001 R/M		(not used)

Table C.2. Machine Instruction Decoding Guide (Continued)

1ST BYTE			2ND BYTE	BYTES 3,4,5,6	ASM-86 INSTRUCTION FORMAT	
HEX	BINARY					
83	1000	0011	MOD 010 R/M	(DISP-LO),(DISP-HI), DATA-SX	ADC	REG16/MEM16,IMMED8
83	1000	0011	MOD 011 R/M	(DISP-LO),(DISP-HI), DATA-SX	SBB	REG16/MEM16,IMMED8
83	1000	0011	MOD 100 R/M		(not used)	
83	1000	0011	MOD 101 R/M	(DISP-LO),(DISP-HI), DATA-SX	SUB	REG16/MEM16,IMMED8
83	1000	011	MOD 110 R/M		(not used)	
83	1000	0011	MOD 111 R/M	(DISP-LO),(DISP-HI), DATA-SX	CMP	REG16/MEM16,IMMED8
84	1000	0100	MOD REG R/M	(DISP-LO),(DISP-HI)	TEST	REG8,MEM8,REG8
85	1000	0101	MOD REG R/M	(DISP-LO),(DISP-HI)	TEST	REG16/MEM16,REG16
86	1000	0110	MOD REG R/M	(DISP-LO),(DISP-HI)	XCHG	REG8,REG8/MEM8
87	1000	0111	MOD REG R/M	(DISP-LO),(DISP-HI)	XCHG	REG16,REG16,MEM16
88	1000	1000	MOD REG R/M	(DISP-LO),(DISP-HI)	MOV	REG8/MEM8,REG8
89	1000	1001	MOD REG R/M	(DISP-LO),(DISP-HI)	MOV	REG16/MEM16/REG16
8A	1000	1010	MOD REG R/M	(DISP-LO),(DISP-HI)	MOV	REG8,REG8/MEM8
8B	1000	1011	MOD REG R/M	(DISP-LO),(DISP-HI)	MOV	REG16,REG16/MEM16
8C	1000	1100	MOD OSR R/M	(DISP-LO),(DISP-HI)		REG16/MEM16,SEGREG
8C	1000	1100	MOD 1 - RM		(not used)	
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
8E	1000	1110	MOD 1 - R/M		(not used)	
8F	1000	1111	MOD 000 R/M	(DISP-LO),(DISP-HI)		
8F	1000	1111	MOD 001 R/M		(not used)	
8F	1000	1111	MOD 010 R/M		(not used)	
8F	1000	1111	MOD 011 R/M		(not used)	
8F	1000	1111	MOD 100 R/M		(not used)	
8F	1000	1111	MOD 101 R/M		(not used)	
8F	1000	1111	MOD 110 R/M		(not used)	
90	1001	0000			NOP	(exchange AX,AX)
91	1001	0001			XCHG	AX,CX
92	1001	0010			XCHG	AX,DX
93	1001	0011			XCHG	AX,BX
94	1001	0100			XCHG	AX,SP
95	1001	0101			XCHG	AX,BP
96	1001	0110			XCHG	AX,SI
97	1001	0111			XCHG	AX,DI
98	1001	1000			CBW	
99	1001	1001			CWD	
9A	1001	1010	DISP-LO	DISP-HI,SEG-LO, SEG-HI	CALL	FAR_PROC
9B	1001	1011			WAIT	
9C	1001	1100			PUSHF	
9D	1001	1101			POPF	
9E	1001	1110			SAHF	

Table C.2. Machine Instruction Decoding Guide (Continued)

1ST BYTE			2ND BYTE	BYTES 3,4,5,6	ASM-86 INSTRUCTION FORMAT	
HEX	BINARY					
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,SR-STR8
A7	1010	0111			CMPS	DEST-STR16,SRC-STR16
A8	1010	1000	DATA-8		TEST	AL,IMMED8
A9	1010	1001	DATA-LO	DATA-HI	TEST	AX,IMMED16
AA	1010	1010			STOS	DEST-STR8
AB	1010	1011			STOS	DEST-STR16
AC	1010	1100			LODS	SRC-STR8
AD	1010	1101			LODS	SRC-STR16
AE	1010	1110			SCAS	DEST-STR8
AF	1010	1111			SCAS	DEST-STR16
B0	1011	0000	DATA-8		MOV	AL,IMMED8
B1	1011	0001	DATA-8		MOV	CL,IMMED8
B2	1011	0010	DATA-8		MOV	DL,IMMED8
B3	1011	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(186/8 ONLY)
C0	1100	0000	MOD 001 R/M	DATA-8	ROR	REG8/MEM8,IMMED8(186/8 ONLY)
C0	1100	0000	MOD 010 R/M	DATA-8	RCL	REG8/MEM8,IMMED8(186/8 ONLY)
C0	1100	0000	MOD 011 R/M	DATA-8	RCR	REG8/MEM8,IMMED8(186/8 ONLY)
C0	1100	0000	MOD 100 R/M	DATA-8	SHL/SAL	REG8/MEM8,IMMED8(186/8 ONLY)
C0	1100	0000	MOD 101 R/M	DATA-8	SHR	REG8/MEM8,IMMED8(186/8 ONLY)
C0	1100	0000	MOD 111 RIM	DATA-8	SAR	REG8/MEM8,IMMED8(186/8 ONLY)
C1	1100	0001	MOD 000 RIM	DATA-8	ROL	REG16/MDM16,IMMED8(186/8 ONLY)
C1	1100	0001	MOD 001 R/M	DATA-8	ROR	REG16/MDM16,IMMED8(186/8 ONLY)
C1	1100	0001	MOD 010 R/M	DATA-8	RCL	REG16/MDM16,IMMED8(186/8 ONLY)
C1	1100	0001	MOD 011 R/M	DATA-8	RCR	REG16/MDM16,IMMED8(186/8 ONLY)
C1	1100	0001	MOD 100 R/M	DATA-8	SHL/SAL	REG16/MDM16,IMMED8(186/8 ONLY)
C1	1100	0001	MOD 101 R/M	DATA-8	SHR	REG16/MDM16,IMMED8(186/8 ONLY)

Table C.2. Machine Instruction Decoding Guide (Continued)

1ST BYTE		2ND BYTE	BYTES 3,4,5,6	ASM-86 INSTRUCTION FORMAT	
HEX	BINARY				
C1	1100 0001	MOD 111 R/M	DATA-8	SAR	REG16/MDM16,IMMED8(186/8 ONLY)
C2	1100 0010	DATA-LO	DATA-HI	RET	IMMED16(intraseg)
C3	1100 0011			RET	(intrasegment)
C4	1100 0100	MOD REG R/M	(DISP-LO),(DISP-HI)	LES	REG16,MEM16
C5	1100 0101	MOD REG R/M	(DISP-LO),(DISP-HI)	LDS	REG16,MEM16
C6	1100 0110	MOD 000 R/M	(DISP-LO),(DISP-HI), DATA-8	MOV	MEM8,IMMED8
C6	1100 0110	MOD 001 R/M		(not used)	
C6	1100 0110	MOD 010 R/M		(not used)	
C6	1100 0110	MOD 011 R/M		(not used)	
C6	1100 0110	MOD 100 R/M		(not used)	
C6	1100 0110	MOD 101 R/M		(not used)	
C6	1100 0110	MOD 110 R/M		(not used)	
C6	1100 0110	MOD 111 R/M		(not used)	
C7	1100 0111	MOD 000 R/M	(DISP-LO),(DISP-HI), DATA-LO,DATA-HI	MOV	MEM16,IMMED16
C7	1100 0111	MOD 001 R/M		(not used)	
C7	1100 0111	MOD 010 R/M		(not used)	
C7	1100 0111	MOD 011 R/M		(not used)	
C7	1100 0111	MOD 100 R/M		(not used)	
C7	1100 0111	MOD 101 R/M		(not used)	
C7	1100 0111	MOD 110 R/M		(not used)	
C7	1100 0111	MOD 111 R/M		(not used)	
C8	1100 1000	DATA-LO	DATA-HI,LEVEL	ENTER	IMMED16,IMMED8(186/8 ONLY)
C9	1100 1001			LEAVE	(186/8 ONLY)
CA	1100 1010	DATA-LO	DATA-HI	RET	IMMED16 (intersegment)
CB	1100 1011			RET	(intersegment)
CC	1100 1100			INT	3
CD	1100 1101	DATA-8		INT	IMMED8
CE	1100 1110			INTO	
CF	1100 1111			IRET	
D0	1101 0000	MOD 000 R/M	(DISP-LO),(DISP-HI)	ROL	REG8/MEM8,1
D0	1101 0000	MOD 001 R/M	(DISP-LO),(DISP-HI)	ROR	REG8/MEM8,1
D0	1101 0000	MOD 010 R/M	(DISP-LO),(DISP-HI)	RCL	REG8/MEM8,1
D0	1101 0000	MOD 011 R/M	(DISP-LO),(DISP-HI)	RCR	REG8/MEM8,1
D0	1101 0000	MOD 100 R/M	(DISP-LO),(DISP-HI)	SAL/SHL	REG8/MEM8,1
D0	1101 `0000	MOD 101 R/M	(DISP-LO),(DISP-HI)	SHR	REG8/MEM8,1
D0	1101 0000	MOD 110 R/M		(not used)	
D0	1101 0000	MOD 111 R/M	(DISP-LO),(DISP-HI)	SAR	REG8/MEM8,1
D1	1101 0001	MOD 000 R/M	(DISP-LO),(DISP-HI)	SAR	REG16/MEM16,1
D1	1101 0001	MOD 001 R/M	(DISP-LO),(DISP-HI)	ROR	REG16/MEM16,1
D1	1101 0001	MOD 010 R/M	(DISP-LO),(DISP-HI)	RCL	REG16/MEM16,1
D1	1101 0001	MOD 011 R/M	(DISP-LO),(DISP-HI)	RCR	REG16/MEM16,1
D1	1101 0001	MOD 100 R/M	(DISP-LO),(DISP-HI)	SAL/SHL	REG16/MEM16,1
D1	1101 0001	MOD 101 R/M	(DISP-LO),(DISP-HI)	SHR	REG16/MEM16,1
D1	1101 0001	MOD 110 R/M		(not used)	

Table C.2. Machine Instruction Decoding Guide (Continued)

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

Table C.2. Machine Instruction Decoding Guide (Continued)

1ST BYTE		2ND BYTE	BYTES 3,4,5,6	ASM-86 INSTRUCTION FORMAT
HEX	BINARY			
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
F6	1111 0110	MOD 001 R/M		(not used)
F6	1111 0110	MOD 010 R/M	(DISP-LO),(DISP-HI)	NOT REG8/MEM8
F6	1111 0110	MOD 011 R/M	(DISP-LO),(DISP-HI)	NEG REG8/MEM8
F6	1111 0110	MOD 100 R/M	(DISP-LO),(DISP-HI)	MUL REG8/MEM8
F6	1111 0110	MOD 101 R/M	(DISP-LO),(DISP-HI)	IMUL REG8/MEM8
F6	1111 0110	MOD 110 R/M	(DISP-LO),(DISP-HI)	DIV REG8/MEM8
F6	1111 0110	MOD 111 R/M	(DISPLO),(DISPHI)	IDIV REG8/MEM8
F7	1111 0111	MOD 000 R/M	(DISP-LO),(DISP-HI), DATA-LO,DATA-HI	TEST REG16/MEM16,IMMED16
F7	1111 0111	MOD 001 R/M		(not used)
F7	1111 0111	MOD 010 R/M	(DISP-LO),(DISP-HI)	NOT REG16/MEM16
F7	1111 0111	MOD 011 R/M	(DISP-LO),(DISP-HI)	NEG REG16/MEM16
F7	1111 0111	MOD 100 R/M	(DISP-LO),(DISP-HI)	MUL REG16/MEM16
F7	1111 0111	MOD 101 R/M	(DISP-LO),(DISP-HI)	IMUL REG16/MEM16
F7	1111 0111	MOD 110 R/M	(DISP-LO),(DISP-HI)	DIV REG16/MEM16
F7	1111 0111	MOD 111 R/M	(DISP-LO),(DISP-HI)	IDIV REG16/MEM16
F8	1111 0100			CLC
F9	1111 1001			STC
FA	1111 1010			CLI
FB	1111 1011			STI
FC	1111 1100			CLD
FD	1111 1101			STD
FE	1111 1110	MOD 000 R/M	(DISP-LO),(DISP-HI)	INC REG8/MEM8
FE	1111 1110	MOD 001 R/M	(DISP-LO),(DISP-HI)	DEC REG8/MEM8
FE	1111 1110	MOD 010 R/M		(not used)
FE	1111 1110	MOD 011 R/M		(not used)
FE	1111 1110	MOD 100 R/M		(not used)
FE	1111 1110	MOD 101 R/M		(not used)
FE	1111 1110	MOD 110 R/M		(not used)
FE	1111 1110	MOD 111 R/M		(not used)
FF	1111 1111	MOD 000 R/M	(DISP-LO),(DISP-HI)	INC MEM16
FF	1111 1111	MOD 001 R/M	(DISP-LO),(DISP-HI)	DEC MEM16
FF	1111 1111	MOD 010 R/M	(DISP-LO),(DISP-HI)	CALL REG16/MEM16(intra)
FF	1111 1111	MOD 011 R/M	(DISP-LO),(DISP-HI)	CALL MEM16(intersegment)
FF	1111 1111	MOD 100 R/M	(DISP-LO),(DISP-HI)	JMP REG16/MEM16(intra)
FF	1111 1111	MOD 101 R/M	(DISP-LO),(DISP-HI)	JMP MEM16(intersegment)
FF	1111 1111	MOD 110 R/M	(DISP-LO),(DISP-HI)	PUSH MEM16
FF	1111 1111	MOD 111 R/M		(not used)

Table C.3. Mnemonic Encoding Matrix

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

where:

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

b = byte operation
d = direct
f = from CPU reg
i = immediate
ia = immed. to accum.
id = indirect
is = immed. byte, sign ext.
l = long ie. intersegment

m = memory
r/m = EA is second byte
si = short intrasegment
sr = segment register
t = to CPU reg
v = variable
w = word operation
z = zero

Appendix D
80C186XL/C188XL Compatibility With The 80C186/C188

APPENDIX D
80C186XL/C188XL COMPATIBILITY WITH THE 80C186/C188

This appendix details all known changes in AC and DC specifications and errata between the original Intel 80C186/C188 and the new 80C186XL/C188XL. The changes occur for two reasons: the XL parts have a new, fully static core and are produced on a faster 1 micron process. The new core and process provide 0 to 20 MHz operation and lower power consumption. The faster process also reduces minimum timings on some signals. In general, these changes will have no effect on system timings provided the system does not contain synchronous control logic. Additionally, standard 80C186/C188 errata have all been corrected on the 80C186XL/C188XL. Essentially, the 80C186XL/C188XL parts are higher performance, lower power, pin for pin replacements for the 80C186/C188 parts.

D.1. DC SPECIFICATION DIFFERENCES

Symbol	80C186		80C186XL		Units	Notes
	Min	Max	Min	Max		
V_{IH}	$0.2V_{CC} + 0.9$	$V_{CC} + 0.5$	$0.2V_{CC} + 0.9$	$V_{CC} + 0.5$	V	XL spec. now includes ARDY and SRDY. This corrects a previous 80C186 errata.
I_{CC}		N/A		100	mA	20 MHz, 0°C, V_{CC} = 5.5V(1)
		150			mA	16 MHz, 0°C, V_{CC} = 5.25V(1)
				80	mA	16 MHz, 0°C, V_{CC} = 5.5V(1)
		120		65	mA	12.5 MHz, 0°C, V_{CC} = 5.5V(1)
		100		50	mA	10 MHz, 0°C, V_{CC} = 5.5V(1)
		N/A		100	uA	0 MHz, 0°C, V_{CC} = 5.5V

NOTES:

1. Current is measured with the device in RESET with X1, X2, inputs, and bidirectional outputs driven.

D.1.1. V_{IH} Specifications

The extra V_{IH} specification on the 80C186 for ARDY and SRDY has been removed on the 80C186XL due to design improvements. The standard 80C186 has an errata associated with tolerances on input signals for guaranteed recognition of a high input voltage (see Errata section, "V_{IH} on SRDY and ARDY Pins"). All other V_{IH} specifications remain unchanged.

D.1.2. I_{CC} Specifications

The specifications for I_{CC} at all operating frequencies have been reduced significantly on the 80C186XL. This reduction is a direct result of the 1 micron process and the fully static core on the 80C186XL. The fully static core allows the processor clock to be stopped during operation without loss of the processor's current state. The 80C186XL part consumes only 100 uA at 0 MHz (due to leakage current).

D.1.3. V_{CC} Specifications

At 16 MHz, the 80C186XL has a 10% V_{CC} tolerance, while the standard 80C186 has a 5% V_{CC} tolerance at 16 MHz. This is an improved specification and will not affect existing designs converted to the XL parts.

D.2. AC SPECIFICATION DIFFERENCES

The 80C186XL is on a new 1 micron process. This process is inherently faster than the 1.5 micron process used to produce the 80C186. Due to the higher speed of this process, a number of timings have changed. The minimum delay timings have been reduced. This will have no effect on systems requiring 3 ns or less of hold time in synchronous control logic. Possible effects on system timings are discussed below.

Symbol	80C186		80C186XL		Units	Frequency (MHz)	Parameter
	Min	Max	Min	Max			
T_{CHSV}	5	45	3	45	ns	0	Status Active Delay
	5	35	3	35	ns	12.5	
	5	31	3	31	ns	16	
			3	25	ns	20	
T_{CLSH}	5	46	3	46	ns	10	Status Inactive Delay
	5	35	3	35	ns	12.5	
	5	30	3	30	ns	16	
			3	25	ns	20	
T_{CLAV}	5	44	3	44	ns	10	Address Valid Delay
	5	36	3	36	ns	12.5	
	5	33	3	33	ns	16	
			3	27	ns	20	
T_{CLDV}	5	40	3	40	ns	10	Data Valid Delay
	5	36	3	36	ns	12.5	
	5	33	3	33	ns	16	
			3	27	ns	20	
T_{CHCSX}	5	35	3	35	ns	10	Chip-Select Inactive Delay
	5	30	3	30	ns	12.5	
	5	25	3	25	ns	16	
			3	20	ns	20	

Symbol	80C186		80C186XL		Units	Frequency	Parameter
	Min	Max	Min	Max		(MHz)	
T_{CVDEX}	5	44	3	44	ns	10	$\overline{DEN}$ Inactive Delay
	5	37	3	37	ns	12.5	
	5	31	3	31	ns	16	
			3	22	ns	20	
T_{CHCTV}	5	44	3	44	ns	10	Control Active Delay 2
	5	37	3	37	ns	12.5	
	5	31	3	31	ns	16	
			3	22	ns	20	
T_{CLRL}	5	44	3	44	ns	10	$\overline{RD}$ Active Delay
	5	37	3	37	ns	12.5	
	5	31	3	31	ns	16	
			3	27	ns	20	
T_{CLRH}	5	44	3	44	ns	10	$\overline{RD}$ Inactive Delay
	5	37	3	37	ns	12.5	
	5	31	3	31	ns	16	
			3	27	ns	20	
T_{CKIN}	50	1000	50	∞	ns	10	CLKIN Period
	40	1000	40	∞	ns	12.5	
	31.25	1000	31.25	∞	ns	16	
			25	∞	ns	20	
T_{CLCK}	20		20	∞	ns	10	CLKIN Low Time
	16		16	∞	ns	12.5	
	13		13	∞	ns	16	
			10	∞	ns	20	
T_{CHCK}	20		20	∞	ns	10	CLKIN High Time
	16		16	∞	ns	12.5	
	13		13	∞	ns	16	
			10	∞	ns	20	
T_{CLCH} (Min)	$0.5\ T_{CLCL} - 8$		$0.5\ T_{CLCL} - 6$		ns	10	CLKOUT Low Time $C_L = 100$ pF
	$0.5\ T_{CLCL} - 7$		$0.5\ T_{CLCL} - 5$		ns	12.5	
	$0.5\ T_{CLCL} - 7$		$0.5\ T_{CLCL} - 5$		ns	16	
			$0.5\ T_{CLCL} - 5$		ns	20	
T_{CHCL}	$0.5\ T_{CLCL} - 8$		$0.5\ T_{CLCL} - 6$		ns	10	CLKOUT High Time $C_L = 100$ pF
	$0.5\ T_{CLCL} - 7$		$0.5\ T_{CLCL} - 5$		ns	12.5	
	$0.5\ T_{CLCL} - 7$		$0.5\ T_{CLCL} - 5$		ns	16	
			$0.5\ T_{CLCL} - 5$		ns	20	

D.2.1. Control Logic Considerations

The reduced minimum timings will affect hold time requirements relative to synchronous control logic. If the required hold time is 3 ns or less, no problems will occur. Additionally, no problems will occur in designs where the required hold times were exceeded by 2 or more nanoseconds (the amount that the minimum timings were reduced on the XL). Designs with tight timing margins for hold times should be evaluated to ensure the hold time requirements are still met. The timing specifications affected in this situation are: $T_{CHSV(min)}$, $T_{CLSH(min)}$, $T_{CLAV(min)}$ (BHE only), and $T_{CHCSX(min)}$. The specifications for $T_{CLRL(min)}$ and $T_{CLRH(min)}$ could also be an issue if $\overline{RD}$ is used in the synchronous control logic, but this is uncommon.

D.2.2. Address and Data Valid Considerations

$T_{CLAV(min)}$ (Address) should not cause a problem. Most designs are not affected by having the address valid to early. The same situation exists for $T_{CLDV(min)}$, data being valid earlier in the bus cycle. This should not cause difficulties in most designs.

D.2.3. Buffered Design Considerations

$T_{CVDEX(min)}$ and $T_{CHCTV(min)}$ changes will not affect system designs. This is due to the fact $T_{CLDX(min)}$ and $T_{CVDEX(min)}$ are both 3 ns. Therefore, you are guaranteed to meet the data hold time requirement. $T_{CHCTV(min)}$ should not affect system designs either. DT/$\overline{R}$ is used to control buffer data flow direction. If this signal goes valid/invalid earlier, it will not matter because the buffers are not enabled unless $\overline{DEN}$ is active. DT/$\overline{R}$ remains active long after $\overline{DEN}$ goes inactive.

D.2.4. X1 Considerations

Because the 80C186XL/C188XL are fully static devices, they can operate down to 0 MHz without losing their present state. The maximum timings for T_{CKIN}, T_{CLCK}, and T_{CHCK} have been set to infinity, reflecting the XL processor's ability to retain its current state, even with the clock stopped (infinite clock period).

D.2.5. CLKOUT High/Low Time Considerations

$T_{CLCH(min)}$ and $T_{CHCL(min)}$ are both improved for the 80C186XL/C188XL. The specifications for 50 pF loading were eliminated because the new 100 pF loading numbers are identical to the old 50 pF specifications. The same performance is achieved with heavier loading.

D.3. ERRATA COMPARISON

Below is a list of errata associated with the 80C186 and 80C188 devices. These errata have been fixed on the 80C186XL and 80C188XL devices. Always consult the latest data sheet for any 80C186XL/C188XL specific errata.

D.3.1. $\overline{\text{LOCK}}$/INTA Cycles

Description: If an interrupt arrives during a LOCK'ed bus cycle, a loss of synchronization can occur and LOCK may not be asserted between the first and second $\overline{\text{INTA}}$ pulses. Without

LOCK being active, the DMA controller or an auxiliary bus master can steal the bus, separating the INTA pulses. Some peripherals cannot tolerate separated INTA pulses, but the 82C59A will not be affected.

Disposition: Fixed within the Bus Interface Unit on the 80C186XL/C188XL.

D.3.2. FWAIT/ERROR

Description: During the execution of an FWAIT instruction, the 80C186 does not test the ERROR input pin. No other numerics instructions have this deficiency. This presents a problem when the FWAIT instruction is used to suspend program execution so that the result of the previous numerics instruction can be used immediately. Since FWAIT does not check for errors, the error may not be detected until the next numerics instruction is executed.

Disposition: This has been declassified as an errata. If 80C187 error synchronization is necessary, continue to follow the FWAIT instruction with a FNOP instruction or use an INT pin instead of the ERROR pin.

D.3.3. V_{IH} on SRDY and ARDY Input Pins

Description: The minimum V_{IH} specification for ARDY and SRDY is higher than those for other pins on the standard 80C186/C188. As a result, less noise margin exists when interfacing to TTL devices at low voltage.

Disposition: Switchpoints of ARDY and SRDY input buffers were fixed in the 80C186XL/C188XL. This change will have no effect on existing designs which required pullup resistors on the ARDY and SRDY inputs as a workaround.

D.3.4. Interrupt Status Register

Description: A timer interrupt request occurring during a write operation to the register may be ignored or redirected to the wrong interrupt vector. All instructions capable of affecting the register are implicated.

Disposition: Declassified as an errata. Continue to disable interrupts during accesses to the DHLT bit.

D.3.5. Bus Preemption

Description: An internal conflict between the HOLD/HOLDA protocol and the DRAM refresh unit can lock up the 80C186/C188 bus controller. There are three necessary conditions: an 80C186/C188 HOLD in progress, a pending non-pipelined effective address calculation, and a pending refresh cycle. The non-pipelined effective address calculation disturbs the normal bus controller priority scheme. The effective address calculation is given priority over the refresh. Hold must be deasserted additional clocks to allow the refresh cycle to begin, or the bus controller will lock-up.

Disposition: Errata corrected in the 80C186XL/C188XL Bus Interface Unit. Workarounds used in existing 80C186/C188 designs will be unaffected.

D.3.6. 80C188 $\overline{\text{RFSH}}$ pin

Description: The $\overline{\text{RFSH}}$ pin on the 80C188 goes active and inactive on T4 rather than T1 as indicated by the data sheet.

Disposition: Errata corrected on 80C188XL. Workaround to delay signal should be removed from designs converted to 80C188XL.

intel®

NORTH AMERICAN SALES OFFICES

ALABAMA

Intel Corp.
5015 Bradford Dr., #2
Huntsville 35805
Tel: (205) 830-4010
FAX: (205) 837-2640

ARIZONA

†Intel Corp.
410 North 44th Street
Suite 500
Phoenix 85008
Tel: (602) 231-0386
FAX: (602) 244-0446

CALIFORNIA

†Intel Corp.
21515 Vanowen Street
Suite 116
Canoga Park 91303
Tel: (818) 704-8500
FAX: (818) 340-1144

Intel Corp.
1 Sierra Gate Plaza
Suite 280C
Roseville 95678
Tel: (916) 782-8086
FAX: (916) 782-8153

†Intel Corp.
9665 Chesapeake Dr.
Suite 325
San Diego 92123
Tel: (619) 292-8086
FAX: (619) 292-0628

*†Intel Corp.
400 N. Tustin Avenue
Suite 450
Santa Ana 92705
Tel: (714) 835-9642
TWX: 910-595-1114
FAX: (714) 541-9157

*†Intel Corp.
San Tomas 4
2700 San Tomas Expressway
2nd Floor
Santa Clara 95051
Tel: (408) 986-8086
TWX: 910-338-0255
FAX: (408) 727-2620

COLORADO

Intel Corp.
4445 Northpark Drive
Suite 100
Colorado Springs 80907
Tel: (719) 594-6622
FAX: (303) 594-0720

*†Intel Corp.
600 S. Cherry St.
Suite 700
Denver 80222
Tel: (303) 321-8086
TWX: 910-931-2289
FAX: (303) 322-8670

CONNECTICUT

†Intel Corp.
301 Lee Farm Corporate Park
83 Wooster Heights Rd.
Danbury 06810
Tel: (203) 748-3130
FAX: (203) 794-0339

FLORIDA

†Intel Corp.
800 Fairway Drive
Suite 160
Deerfield Beach 33441
Tel: (305) 421-0506
FAX: (305) 421-2444

†Intel Corp.
5850 T.G. Lee Blvd.
Suite 340
Orlando 32822
Tel: (407) 240-8000
FAX: (407) 240-8097

GEORGIA

†Intel Corp.
20 Technology Parkway
Suite 150
Norcross 30092
Tel: (404) 449-0541
FAX: (404) 605-9762

ILLINOIS

*†Intel Corp.
Woodfield Corp. Center III
300 N. Martingale Road
Suite 400
Schaumburg 60173
Tel: (708) 605-8031
FAX: (708) 706-9762

INDIANA

†Intel Corp.
8910 Purdue Road
Suite 350
Indianapolis 46268
Tel: (317) 875-0623
FAX: (317) 875-8938

MARYLAND

*†Intel Corp.
10010 Junction Dr.
Suite 200
Annapolis Junction 20701
Tel: (410) 206-2860
FAX: (410) 206-3678

MASSACHUSETTS

*†Intel Corp.
Westford Corp. Center
3 Carlisle Road
2nd Floor
Westford 01886
Tel: (508) 692-0960
TWX: 710-343-6333
FAX: (508) 692-7867

MICHIGAN

†Intel Corp.
7071 Orchard Lake Road
Suite 100
West Bloomfield 48322
Tel: (313) 851-8096
FAX: (313) 851-8770

MINNESOTA

†Intel Corp.
3500 W. 80th St.
Suite 360
Bloomington 55431
Tel: (612) 835-6722
TWX: 910-576-2867
FAX: (612) 831-6497

NEW JERSEY

*†Intel Corp.
Lincroft Office Center
125 Half Mile Road
Red Bank 07701
Tel: (908) 747-2233
FAX: (908) 747-0983

NEW YORK

*Intel Corp.
850 Crosskeys Office Park
Fairport 14450
Tel: (716) 425-2750
TWX: 510-253-7391
FAX: (716) 223-2561

*†Intel Corp.
2950 Express Dr., South
Suite 130
Islandia 11722
Tel: (516) 231-3300
TWX: 510-227-6236
FAX: (516) 348-7939

†Intel Corp.
300 Westage Business Center
Suite 230
Fishkill 12524
Tel: (914) 897-3860
FAX: (914) 897-3125

OHIO

*†Intel Corp.
3401 Park Center Drive
Suite 220
Dayton 45414
Tel: (513) 890-5350
TWX: 810-450-2528
FAX: (513) 890-8658

*†Intel Corp.
25700 Science Park Dr.
Suite 100
Beachwood 44122
Tel: (216) 464-2736
TWX: 810-427-9298
FAX: (804) 282-0673

OKLAHOMA

Intel Corp.
6801 N. Broadway
Suite 115
Oklahoma City 73162
Tel: (405) 848-8086
FAX: (405) 840-9819

OREGON

†Intel Corp.
15254 N.W. Greenbrier Pkwy.
Building B
Beaverton 97006
Tel: (503) 645-8051
TWX: 910-467-8741
FAX: (503) 645-8181

PENNSYLVANIA

*†Intel Corp.
925 Harvest Drive
Suite 200
Blue Bell 19422
Tel: (215) 641-1000
FAX: (215) 641-0785

*†Intel Corp.
400 Penn Center Blvd.
Suite 610
Pittsburgh 15235
Tel: (412) 823-4970
FAX: (412) 829-7578

PUERTO RICO

†Intel Corp.
South Industrial Park
P.O. Box 910
Las Piedras 00671
Tel: (809) 733-8616

TEXAS

†Intel Corp.
8911 N. Capital of Texas Hwy.
Suite 4230
Austin 78759
Tel: (512) 794-8086
FAX: (512) 338-9335

*†Intel Corp.
12000 Ford Road
Suite 400
Dallas 75234
Tel: (214) 241-8087
FAX: (214) 484-1180

*†Intel Corp.
7322 S.W. Freeway
Suite 1490
Houston 77074
Tel: (713) 988-8086
TWX: 910-881-2490
FAX: (713) 988-3660

UTAH

†Intel Corp.
428 East 6400 South
Suite 104
Murray 84107
Tel: (801) 263-8051
FAX: (801) 268-1457

WASHINGTON

†Intel Corp.
155 108th Avenue N.E.
Suite 386
Bellevue 98004
Tel: (206) 453-8086
TWX: 910-443-3002
FAX: (206) 451-9556

Intel Corp.
408 N. Mullan Road
Suite 102
Spokane 99206
Tel: (509) 928-8086
FAX: (509) 928-9467

WISCONSIN

Intel Corp.
330 S. Executive Dr.
Suite 102
Brookfield 53005
Tel: (414) 784-8087
FAX: (414) 796-2115

CANADA

BRITISH COLUMBIA

Intel Semiconductor of
Canada, Ltd.
4585 Canada Way
Suite 202
Burnaby V5G 4L6
Tel: (604) 298-0387
FAX: (604) 298-8234

ONTARIO

†Intel Semiconductor of
Canada, Ltd.
2650 Queensview Drive
Suite 250
Ottawa K2B 8H6
Tel: (613) 829-9714
FAX: (613) 820-5936

†Intel Semiconductor of
Canada, Ltd.
190 Attwell Drive
Suite 500
Rexdale M9W 6H8
Tel: (416) 675-2105
FAX: (416) 675-2438

QUEBEC

†Intel Semiconductor of
Canada, Ltd.
1 Rue Holiday
Suite 115
Tour East
Pt. Claire H9R 5N3
Tel: (514) 694-9130
FAX: 514-694-0064

†Sales and Service Office
*Field Application Location

NORTH AMERICAN DISTRIBUTORS

ALABAMA

Arrow Electronics, Inc.
1015 Henderson Road
Huntsville 35806
Tel: (205) 837-6955
FAX: (205) 721-1581

Hamilton/Avnet Electronics
4960 Corporate Drive, #135
Huntsville 35805
Tel: (205) 837-7210
FAX: (205) 721-0356

MTI Systems Sales
4950 Corporate Drive
Suite 120
Huntsville 35805
Tel: (205) 830-9526
FAX: (205) 830-9557

Pioneer/Technologies Group, Inc.
4835 University Square, #5
Huntsville 35805
Tel: (205) 837-9300
FAX: (205) 837-9358

ARIZONA

†Arrow Electronics, Inc.
4134 E. Wood Street
Phoenix 85040
Tel: (602) 437-0750
FAX: (602) 252-9109

Avnet Computer
30 South McKemy Avenue
Chandler 85226
Tel: (602) 961-6460
FAX: (602) 961-4787

Hamilton/Avnet Electronics
30 South McKemy Avenue
Chandler 85226
Tel: (602) 961-6403
FAX: (602) 961-1331

Wyle Distribution Group
4141 E. Raymond
Phoenix 85040
Tel: (602) 437-2088
FAX: (602) 437-2124

CALIFORNIA

Arrow Commercial System Group
1502 Crocker Avenue
Hayward 94544
Tel: (415) 489-5371
FAX: (415) 489-9393

Arrow Commercial System Group
14242 Chambers Road
Tustin 92680
Tel: (714) 544-0200
FAX: (714) 731-8438

†Arrow Electronics, Inc.
19748 Dearborn Street
Chatsworth 91311
Tel: (818) 701-7500
FAX: (818) 772-8930

†Arrow Electronics, Inc.
9511 Ridgehaven Court
San Diego 92123
Tel: (619) 565-4800
FAX: (619) 279-8062

†Arrow Electronics, Inc.
1180 Murphy Avenue
San Jose 95131
Tel: (408) 441-9700
FAX: (408) 453-4810

†Arrow Electronics, Inc.
2961 Dow Avenue
Tustin 92680
Tel: (714) 838-5422
FAX: (714) 838-4151

Avnet Computer
3170 Pullman Street
Costa Mesa 92626
Tel: (714) 641-4121
FAX: (714) 641-4170

Avnet Computer
1361B West 190th Street
Gardena 90248
Tel: (800) 345-3870
FAX: (213) 327-5389

Avnet Computer
755 Sunrise Blvd., #150
Roseville 95661
Tel: (916) 781-2521
FAX: (916) 781-3819

Avnet Computer
1175 Bordeaux Drive, #A
Sunnyvale 94089
Tel: (408) 743-3304
FAX: (408) 743-3348

Avnet Computer
21150 Califa Street
Woodland Hills 91376
Tel: (808) 345-3870
FAX: (818) 594-8333

†Hamilton/Avnet Electronics
3170 Pullman Street
Costa Mesa 92626
Tel: (714) 641-4100
FAX: (714) 754-6033

†Hamilton/Avnet Electronics
1175 Bordeaux Drive, #A
Sunnyvale 94089
Tel: (408) 743-3300
FAX: (408) 745-6679

†Hamilton/Avnet Electronics
4545 Viewridge Avenue
San Diego 92123
Tel: (619) 571-1900
FAX: (619) 571-8761

†Hamilton/Avnet Electronics
21150 Califa St.
Woodland Hills 91367
Tel: (818) 594-0403
FAX: (818) 594-8234

†Hamilton/Avnet Electronics
1361B West 190th Street
Gardena 90248
Tel: (213) 516-8600
FAX: (213) 217-6822

†Hamilton/Avnet Electronics
755 Sunrise Avenue, #150
Roseville 95661
Tel: (916) 925-2216
FAX: (916) 925-3478

Pioneer/Technologies Group, Inc.
134 Rio Robles
San Jose 95134
Tel: (408) 954-9100
FAX: 408-954-9113

†Wyle Distribution Group
124 Maryland Street
El Segundo 90245
Tel: (213) 322-8100
FAX: (213) 416-1151

Wyle Distribution Group
7431 Chapman Ave.
Garden Grove 92641
Tel: (714) 891-1717
FAX: (714) 891-1621

†Wyle Distribution Group
2951 Sunrise Blvd., Suite 175
Rancho Cordova 95742
Tel: (916) 638-5282
FAX: (916) 638-1491

†Wyle Distribution Group
9525 Chesapeake Drive
San Diego 92123
Tel: (619) 565-9171
FAX: (619) 365-0512

†Wyle Distribution Group
3000 Bowers Avenue
Santa Clara 95051
Tel: (408) 727-2500
FAX: (408) 727-5896

†Wyle Distribution Group
17872 Cowan Avenue
Irvine 92714
Tel: (714) 863-9953
FAX: (714),263-0473

†Wyle Distribution Group
26010 Mureau Road, #150
Calabasas 91302
Tel: (818) 880-9000
FAX: (818) 880-5510

COLORADO

Arrow Electronics, Inc.
3254 C Frazer Street
Aurora 80011
Tel: (303) 373-5616
FAX: (303) 373-5760

†Hamilton/Avnet Electronics
9605 Maroon Circle, #200
Englewood 80112
Tel: (303) 799-7800
FAX: (303) 799-7801

†Wyle Distribution Group
451 E. 124th Avenue
Thornton 80241
Tel: (303) 457-9953
FAX: (303) 457-4831

CONNECTICUT

†Arrow Electronics, Inc.
12 Beaumont Road
Wallingford 06492
Tel: (203) 265-7741
FAX: (203) 265-7988

Avnet Computer
55 Federal Road, #103
Danbury 06810
Tel: (203) 797-2880
FAX: (203) 791-9050

†Hamilton/Avnet Electronics
55 Federal Road, #103
Danbury 06810
Tel: (203) 743-6077
FAX: (203) 791-9050

†Pioneer/Standard Electronics
112 Main Street
Norwalk 06851
Tel: (203) 853-1515
FAX: (203) 838-9901

FLORIDA

†Arrow Electronics, Inc.
400 Fairway Drive, #102
Deerfield Beach 33441
Tel: (305) 429-8200
FAX: (305) 428-3991

†Arrow Electronics, Inc.
37 Skyline Drive, #3101
Lake Mary 32746
Tel: (407) 333-9300
FAX: (407) 333-9320

Avnet Computer
3343 W. Commercial Blvd.
Bldg. C/D, Suite 107
Ft. Lauderdale 33309
Tel: (305) 979-9067
FAX: (305) 730-0368

Avnet Computer
3247 Tech Drive North
St. Petersburg 33716
Tel: (813) 573-5524
FAX: (813) 572-4324

†Hamilton/Avnet Electronics
5371 N.W. 33rd Avenue
Ft. Lauderdale 33309
Tel: (305) 484-5016
FAX: (305) 484-8369

†Hamilton/Avnet Electronics
3247 Tech Drive North
St. Petersburg 33716
Tel: (813) 573-3930
FAX: (813) 572-4329

†Hamilton/Avnet Electronics
7079 University Boulevard
Winter Park 32791
Tel: (407) 657-3300
FAX: (407) 678-1878

†Pioneer/Technologies Group, Inc.
337 Northlake Blvd., Suite 1000
Alta Monte Springs 32701
Tel: (407) 834-9090
FAX: (407) 834-0865

Pioneer/Technologies Group, Inc.
674 S. Military Trail
Deerfield Beach 33442
Tel: (305) 428-8877
FAX: (305) 481-2950

GEORGIA

Arrow Commercial System Group
3400 C. Corporate Way
Duluth 30136
Tel: (404) 623-8825
FAX: (404) 623-8802

†Arrow Electronics, Inc.
4250 E. Rivergreen Pkwy., #E
Duluth 30136
Tel: (404) 497-1300
FAX: (404) 476-1493

Avnet Computer
3425 Corporate Way, #G
Duluth 30136
Tel: (404) 623-5452
FAX: (404) 476-0125

Hamilton/Avnet Electronics
3425 Corporate Way, #G
Duluth 30136
Tel: (404) 446-0611
FAX: (404) 446-1011

Pioneer/Technologies Group, Inc.
4250 C. Rivergreen Parkway
Duluth 30136
Tel: (404) 623-1003
FAX: (404) 623-0665

ILLINOIS

†Arrow Electronics, Inc.
1140 W. Thorndale Rd.
Itasca 60143
Tel: (708) 250-0500

Avnet Computer
1124 Thorndale Avenue
Bensenville 60106
Tel: (708) 860-8573
FAX: (708) 773-7976

†Hamilton/Avnet Electronics
1130 Thorndale Avenue
Bensenville 60106
Tel: (708) 860-7700
FAX: (708) 860-8530

MTI Systems
1140 W. Thorndale Avenue
Itasca 60143
Tel: (708) 250-8222
FAX: (708) 250-8275

†Pioneer/Standard Electronics
2171 Executive Dr., Suite 200
Addison 60101
Tel: (708) 495-9680
FAX: (708) 495-9831

INDIANA

†Arrow Electronics, Inc.
7108 Lakeview Parkway West Dr.
Indianapolis 46268
Tel: (317) 299-2071
FAX: (317) 299-2379

Avnet Computer
485 Gradle Drive
Carmel 46032
Tel: (317) 575-8029
FAX: (317) 844-4964

Hamilton/Avnet Electronics
485 Gradle Drive
Carmel 46032
Tel: (317) 844-9333
FAX: (317) 844-5921

†Pioneer/Standard Electronics
9350 Priority Way West Dr.
Indianapolis 46250
Tel: (317) 573-0880
FAX: (317) 573-0979

†Certified VAD

NORTH AMERICAN DISTRIBUTORS (Contd.)

IOWA

Hamilton/Avnet Electronics
2335A Blairsferry Rd., N.E.
Cedar Rapids 52402
Tel: (319) 362-4757
FAX: (319) 393-7050

KANSAS

Arrow Electronics, Inc.
8208 Melrose Dr., Suite 210
Lenexa 66214
Tel: (913) 541-9542
FAX: (913) 541-0328

Avnet Computer
15313 W. 95th Street
Lenexa 61219
Tel: (913) 541-7989
FAX: (913) 541-7904

†Hamilton/Avnet Electronics
15313 W. 95th
Overland Park 66215
Tel: (913) 888-1055
FAX: (913) 541-7951

KENTUCKY

Hamilton/Avnet Electronics
805 A. Newtown Circle
Lexington 40511
Tel: (606) 259-1475
FAX: (606) 252-3238

MARYLAND

Arrow Commercial Systems Group
200 Perry Parkway
Gaithersburg 20877
Tel: (301) 670-1600
FAX: (301) 670-0188

†Arrow Electronics, Inc.
8300 Guilford Road, #H
Columbia 21046
Tel: (301) 995-6002
FAX: (301) 995-6201

Avnet Computer
7172 Columbia Gateway Dr., #G
Columbia 21045
Tel: (301) 995-0020
FAX: (301) 995-3515

†Hamilton/Avnet Electronics
7172 Columbia Gateway Dr., #F
Columbia 21045
Tel: (301) 995-3554
FAX: (301) 995-3515

†North Atlantic Industries
Systems Division
7125 Riverwood Dr.
Columbia 21046
Tel: (301) 290-3999

†Pioneer/Technologies Group, Inc.
15810 Gaither Road
Gaithersburg 20877
Tel: (301) 921-0660
FAX: (301) 670-6746

MASSACHUSETTS

Arrow Electronics, Inc.
25 Upton Dr.
Wilmington 01887
Tel: (508) 658-0900
FAX: (508) 694-1754

Avnet Computer
10 D Centennial Drive
Peabody 01960
Tel: (508) 532-9886
FAX: (508) 532-9660

†Hamilton/Avnet Electronics
10D Centennial Drive
Peabody 01960
Tel: (508) 531-7430
FAX: (508) 532-9802

†Pioneer/Standard Electronics
44 Hartwell Avenue
Lexington 02173
Tel: (617) 861-9200
FAX: (617) 863-1547

Wyle Distribution Group
15 Third Avenue
Burlington 01803
Tel: (617) 272-7300
FAX: (617) 272-6809

MICHIGAN

†Arrow Electronics, Inc.
19880 Haggerty Road
Livonia 48152
Tel: (313) 665-4100
FAX: (313) 462-2686

Avnet Computer
2876 28th Street, S.W., #5
Grandville 49418
Tel: (616) 531-9607
FAX: (616) 531-0059

Avnet Computer
41650 Garden Road
Novi 48375
Tel: (313) 347-1820
FAX: (313) 347-4067

Hamilton/Avnet Electronics
2876 28th Street, S.W., #5
Grandville 49418
Tel: (616) 243-8805
FAX: (616) 531-0059

Hamilton/Avnet Electronics
41650 Garden Brook Rd., #100
Novi 48375
Tel: (313) 347-4270
FAX: (313) 347-4021

†Pioneer/Standard Electronics
4505 Broadmoor S.E.
Grand Rapids 49512
Tel: (616) 698-1800
FAX: (616) 698-1831

†Pioneer/Standard Electronics
13485 Stamford
Livonia 48150
Tel: (313) 525-1800
FAX: (313) 427-3720

MINNESOTA

†Arrow Electronics, Inc.
10120A West 76th Street
Eden Prairie 55344
Tel: (612) 829-5588
FAX: (612) 942-7803

Avnet Computer
10000 West 76th Street
Eden Prairie 55344
Tel: (612) 829-0025
FAX: (612) 944-2781

†Hamilton/Avnet Electronics
12400 Whitewater Drive
Minnetonka 55343
Tel: (612) 932-0600
FAX: (612) 932-0613

†Pioneer/Standard Electronics
7625 Golden Triange Dr., #G
Eden Prairie 55344
Tel: (612) 944-3355
FAX: (612) 944-3794

MISSOURI

†Arrow Electronics, Inc.
2380 Schuetz Road
St. Louis 63141
Tel: (314) 567-6888
FAX: (314) 567-1164

Avnet Computer
739 Goddard Avenue
Chesterfield 63005
Tel: (314) 537-2725
FAX: (314) 537-4248

†Hamilton/Avnet Electronics
741 Goddard
Chesterfield 63005
Tel: (314) 537-1600
FAX: (314) 537-4248

NEW HAMPSHIRE

Avnet Computer
2 Executive Park Drive
Bedford 03102
Tel: (603) 624-6630
FAX: (603) 624-2402

NEW JERSEY

†Arrow Electronics, Inc.
4 East Stow Road
Unit 11
Marlton 08053
Tel: (609) 596-8000
FAX: (609) 596-9632

†Arrow Electronics, Inc.
6 Century Drive
Parsipanny 07054
Tel: (201) 538-0900
FAX: (201) 538-4962

Avnet Computer
1-B Keystone Ave., Bldg. 36
Cherry Hill 08003
Tel: (609) 424-8961
FAX: (609) 751-2502

Avnet Computer
10 Industrial Road
Fairfield 07006
Tel: (201) 882-2879
FAX: (201) 808-9251

†Hamilton/Avnet Electronics
1 Keystone Ave., Bldg. 36
Cherry Hill 08003
Tel: (609) 424-0110
FAX: (609) 751-2552

†Hamilton/Avnet Electronics
10 Industrial
Fairfield 07006
Tel: (201) 575-3390
FAX: (201) 575-5839

†MTI Systems Sales
6 Century Drive
Parsippany 07054
Tel: (201) 539-6496
FAX: (201) 539-6430

†Pioneer/Standard Electronics
14-A Madison Rd.
Fairfield 07006
Tel: (201) 575-3510
FAX: (201) 575-3454

NEW MEXICO

Alliance Electronics Inc.
10510 Research Avenue
Albuquerque 87123
Tel: (505) 292-3360
FAX: (505) 275-6392

Avnet Computer
7801 Academy Road
Bldg. 1, Suite 204
Albuquerque 87109
Tel: (505) 828-9725
FAX: (505) 828-0360

†Hamilton/Avnet Electronics
7801 Academy Rd. N.E.
Bldg. 1, Suite 204
Albuquerque 87108
Tel: (505) 765-1500
FAX: (505) 243-1395

NEW YORK

†Arrow Electronics, Inc.
3375 Brighton Henrietta Townline Rd.
Rochester 14623
Tel: (716) 427-0300
FAX: (716) 427-0735

Arrow Electronics, Inc.
20 Oser Avenue
Hauppauge 11788
Tel: (516) 231-1000
FAX: (516) 231-1072

Avnet Computer
933 Motor Parkway
Hauppauge 11788
Tel: (516) 231-9040
FAX: (516) 434-7426

Avnet Computer
2060 Townline
Rochester 14623
Tel: (716) 272-9306
FAX: (716) 272-9685

†Hamilton/Avnet Electronics
933 Motor Parkway
Hauppauge 11788
Tel: (516) 231-9800
FAX: (516) 434-7426

†Hamilton/Avnet Electronics
2060 Townline Rd.
Rochester 14623
Tel: (716) 292-0730
FAX: (716) 292-0810

Hamilton/Avnet Electronics
103 Twin Oaks Drive
Syracuse 13120
Tel: (315) 437-2641
FAX: (315) 432-0740

MTI Systems
50 Horseblock Road
Brookhaven 11719
Tel: (516) 924-9400
FAX: (516) 924-1103

MTI Systems
1 Penn Plaza
250 W. 34th Street
New York 10119
Tel: (212) 643-1280
FAX: (212) 643-1288

Pioneer/Standard Electronics
68 Corporate Drive
Binghamton 13904
Tel: (607) 722-9300
FAX: (607) 722-9562

†Pioneer/Standard Electronics
60 Crossway Park West
Woodbury, Long Island 11797
Tel: (516) 921-8700
FAX: (516) 921-2143

†Pioneer/Standard Electronics
840 Fairport Park
Fairport 14450
Tel: (716) 381-7070
FAX: (716) 381-5955

NORTH CAROLINA

†Arrow Electronics, Inc.
5240 Greensdairy Road
Raleigh 27604
Tel: (919) 876-3132
FAX: (919) 878-9517

Avnet Computer
2725 Millbrook Rd., #123
Raleigh 27604
Tel: (919) 790-1735
FAX: (919) 872-4972

Hamilton/Avnet Electronics
5250-77 Center Dr. #350
Charlotte 28217
Tel: (704) 527-2485
FAX: (704) 527-8058

†Hamilton/Avnet Electronics
3510 Spring Forest Drive
Raleigh 27604
Tel: (919) 878-0819

Pioneer/Technologies Group, Inc.
9401 L-Southern Pine Blvd.
Charlotte 28210
Tel: (704) 527-8188
FAX: (704) 522-8564

Pioneer Technologies Group, Inc.
2810 Meridian Parkway, #148
Durham 27713
Tel: (919) 544-5400
FAX: (919) 544-5885

OHIO

Arrow Commercial System Group
284 Cramer Creek Court
Dublin 43017
Tel: (614) 889-9347
FAX: (614) 889-9680

†Arrow Electronics, Inc.
6573 Cochran Road, #E
Solon 44139
Tel: (216) 248-3990
FAX: (216) 248-1106

Arrow Electronics, Inc.
8200 Washington Village Dr.
Centerville 45458
Tel: (513) 435-5563
FAX: (513) 435-2049

†Certified VAD

NORTH AMERICAN DISTRIBUTORS (Contd.)

OHIO (Contd.)

Avnet Computer
7764 Washington Village Dr.
Dayton 45459
Tel: (513) 439-6756
FAX: (513) 439-6719

Avnet Computer
30325 Bainbridge Rd., Bldg. A
Solon 44139
Tel: (216) 349-2505
FAX: (216) 349-1894

†Hamilton/Avnet Electronics
7760 Washington Village Dr.
Dayton 45459
Tel: (513) 439-6733
FAX: (513) 439-6711

†Hamilton/Avnet Electronics
30325 Bainbridge
Solon 44139
Tel: (800) 543-2984
FAX: (216) 349-1894

Hamilton/Avnet Electronics
2600 Corp Exchange Drive, #180
Columbus 43231
Tel: (614) 882-7004
FAX: (614) 882-8650

MTI Systems Sales
23404 Commerce Park Road
Beachwood 44122
Tel: (216) 464-6688
FAX: (216) 464-3564

†Pioneer/Standard Electronics
4433 Interpoint Boulevard
Dayton 45424
Tel: (513) 236-9900
FAX: (513) 236-8133

†Pioneer/Standard Electronics
4800 E. 131st Street
Cleveland 44105
Tel: (216) 587-3600
FAX: (216) 663-1004

OKLAHOMA

Arrow Electronics, Inc.
12111 East 51st Street, #101
Tulsa 74146
Tel: (918) 252-7537
FAX: (918) 254-0917

†Hamilton/Avnet Electronics
12121 E. 51st St., Suite 102A
Tulsa 74146
Tel: (918) 664-0444
FAX: (918) 250-8763

OREGON

†Almac Electronics Corp.
1885 N.W. 169th Place
Beaverton 97006
Tel: (503) 629-8090
FAX: 503-645-0611

Avnet Computer
9409 Southwest Nimbus Ave.
Beaverton 97005
Tel: (503) 627-0900
FAX: (503) 526-6242

†Hamilton/Avnet Electronics
9409 S.W. Nimbus Ave.
Beaverton 97005
Tel: (503) 627-0201
FAX: (503) 641-4012

Wyle
9640 Sunshine Court
Bldg. G, Suite 200
Beaverton 97005
Tel: (503) 643-7900
FAX: (503) 646-5466

PENNSYLVANIA

Avnet Computer
213 Executive Drive, #320
Mars 16046
Tel: (412) 772-1888
FAX: (412) 772-1890

Hamilton/Avnet Electronics
213 Executive, #320
Mars 16045
Tel: (412) 281-4152
FAX: (412) 772-1890

Pioneer/Technologies Group, Inc.
259 Kappa Drive
Pittsburgh 15238
Tel: (412) 782-2300
FAX: (412) 963-8255

†Pioneer/Technologies Group, Inc.
500 Enterprise Road
Keith Valley Business Center
Horsham 19044
Tel: (215) 674-4000
FAX: (215) 674-3107

TENNESSEE

Arrow Commercial System Group
3635 Knight Road, #7
Memphis 38118
Tel: (901) 367-0540
FAX: (901) 367-2081

TEXAS

Arrow Electronics, Inc.
3220 Commander Drive
Carrollton 75006
Tel: (214) 380-6464
FAX: (214) 248-7208

Avnet Computer
4004 Beltline, Suite 200
Dallas 75244
Tel: (214) 308-8181
FAX: (214) 308-8129

Avnet Computer
1235 North Loop West, #525
Houston 77008
Tel: (713) 867-7500
FAX: (713) 861-6851

†Hamilton/Avnet Electronics
1826-F Kramer Lane
Austin 78758
Tel: (800) 772-5668
FAX: (512) 832-4315

†Hamilton/Avnet Electronics
4004 Beltline, #200
Dallas 75244
Tel: (214) 308-8111
FAX: (214) 308-8109

†Hamilton/Avnet Electronics
1235 N. Loop West, #521
Houston 77008
Tel: (713) 240-7733
FAX: (713) 861-6541

†Pioneer/Standard Electronics
1826-D Kramer Lane
Austin 78758
Tel: (512) 835-4000
FAX: (512) 835-9829

†Pioneer/Standard Electronics
13765 Beta Road
Dallas 75244
Tel: (214) 386-7300
FAX: (214) 490-6419

†Pioneer/Standard Electronics
10530 Rockley Road, #100
Houston 77099
Tel: (713) 495-4700
FAX: (713) 495-5642

†Wyle Distribution Group
1810 Greenville Avenue
Richardson 75081
Tel: (214) 235-9953
FAX: (214) 644-5064

Wyle Distribution Group
4030 West Braker Lane, #330
Austin 78758
Tel: (512) 345-8853
FAX: (512) 345-9330

Wyle Distribution Group
11001 South Wilcrest, #100
Houston 77099
Tel: (713) 879-9953
FAX: (713) 879-6540

UTAH

Arrow Electronics, Inc.
1946 W. Parkway Blvd.
Salt Lake City 84119
Tel: (801) 973-6913

Avnet Computer
1100 E. 6600 South, #150
Salt Lake City 84121
Tel: (801) 266-1115
FAX: (801) 266-0362

Avnet Computer
17761 Northeast 78th Place
Redmond 98052
Tel: (206) 867-0160
FAX: (206) 867-0161

†Hamilton/Avnet Electronics
1100 East 6600 South, #120
Salt Lake City 84121
Tel: (801) 972-2800
FAX: (801) 263-0104

†Wyle Distribution Group
1325 West 2200 South, #E
West Valley 84119
Tel: (801) 974-9953
FAX: (801) 972-2524

WASHINGTON

†Almac Electronics Corp.
14360 S.E. Eastgate Way
Bellevue 98007
Tel: (206) 643-9992
FAX: (206) 643-9709

†Hamilton/Avnet Electronics
17761 N.E. 78th Place, #C
Redmond 98052
Tel: (206) 241-8555
FAX: (206) 241-5472

Wyle Distribution Group
15385 N.E. 90th Street
Redmond 98052
Tel: (206) 881-1150
FAX: (206) 881-1567

WISCONSIN

Arrow Electronics, Inc.
200 N. Patrick Blvd., Ste. 100
Brookfield 53005
Tel: (414) 792-0150
FAX: (414) 792-0156

Avnet Computer
20875 Crossroads Circle, #400
Waukesha 53186
Tel: (414) 784-8205
FAX: (414) 784-6006

†Hamilton/Avnet Electronics
28875 Crossroads Circle, #400
Waukesha 53186
Tel: (414) 784-4510
FAX: (414) 784-9509

Pioneer/Standard Electronics
120 Bishops Way #163
Brookfield 53005
Tel: (414) 784-3480

ALASKA

Avnet Computer
1400 West Benson Blvd.
Suite 400
Anchorage 99503
Tel: (907) 274-9899
FAX: (907) 277-2639

CANADA

ALBERTA

Avnet Computer
2816 21st Street Northeast
Calgary T2E 6Z2
Tel: (403) 291-3284
FAX: (403) 250-1591

Zentronics
6815 8th Street N.E., #100
Calgary T2E 7H
Tel: (403) 295-8838
FAX: (403) 295-8714

BRITISH COLUMBIA

†Hamilton/Avnet Electronics
8610 Commerce Court
Burnaby V5A 4N6
Tel: (604) 420-4101
FAX: (604) 420-5376

Zentronics
11400 Bridgeport Rd., #108
Richmond V6X 1T2
Tel: (604) 273-5575
FAX: (604) 273-2413

ONTARIO

Arrow Electronics, Inc.
36 Antares Dr., Unit 100
Nepean K2E 7W5
Tel: (613) 226-6903
FAX: (613) 723-2018

†Arrow Electronics, Inc.
1093 Meyerside, Unit 2
Mississauga L5T 1M4
Tel: (416) 670-7769
FAX: (416) 670-7781

Avnet Computer
Canada System Engineering Group
3688 Nashua Dr., Unit 6
Mississuaga L4V 1M5
Tel: (416) 672-8638
FAX: (416) 677-5091

Avnet Computer
6845 Rexwood Road
Units 7–9
Mississuaga L4V 1M4
Tel: (416) 672-8638
FAX: (416) 672-8650

Avnet Computer
190 Colonade Road
Nepean K2E 7J5
Tel: (613) 727-7529
FAX: (613) 226-1184

†Hamilton/Avnet Electronics
6845 Rexwood Rd., Units 3–5
Mississauga L4T 1R2
Tel: (416) 677-7432
FAX: (416) 677-0940

†Hamilton/Avnet Electronics
190 Colonade Road
Nepean K2E 7J5
Tel: (613) 226-1700
FAX: (613) 226-1184

†Zentronics
1355 Meyerside Drive
Mississauga L5T 1C9
Tel: (416) 564-9600
FAX: (416) 564-3127

†Zentronics
155 Colonade Rd., South
Unit 17
Nepean K2E 7K1
Tel: (613) 226-8840
FAX: (613) 226-6352

QUEBEC

Arrow Electronics Inc.
1100 St. Regis Blvd.
Dorval H9P 2T5
Tel: (514) 421-7411
FAX: (514) 421-7430

Arrow Electronics, Inc.
500 Boul. St-Jean-Baptiste Ave.
Quebec H2E 5R9
Tel: (418) 871-7500
FAX: (418) 871-6816

Avnet Computer
2795 Rue Halpern
St. Laurent H4S 1P8
Tel: (514) 335-2483
FAX: (514) 335-2481

†Hamilton/Avnet Electronics
2795 Halpern
St. Laurent H4S 1P8
Tel: (514) 335-1000
FAX: (514) 335-2481

†Zentronics
520 McCaffrey
St. Laurent H4T 1N3
Tel: (514) 737-9700
FAX: (514) 737-5212

†Certified VAD

intel®

EUROPEAN SALES OFFICES

FINLAND

Intel Finland OY
Ruosilantie 2
00390 Helsinki
Tel: (358) 0 544 644
FAX: (358) 0 544 030

FRANCE

Intel Corporation S.A.R.L.
1, Rue Edison-BP 303
78054 St. Quentin-en-Yvelines
Cedex
Tel: (33) (1) 30 57 70 00
FAX: (33) (1) 30 64 60 32

GERMANY

Intel GmbH
Dornacher Strasse 1
8016 Feldkirchen bei Muenchen
Tel: (49) 089/90992-0
FAX: (49) 089/9043948

ISRAEL

Intel Semiconductor Ltd.
Atidim Industrial Park-Neve Sharet
P.O. Box 43202
Tel-Aviv 61430
Tel: (972) 03 498080
FAX: (972) 03 491870

ITALY

Intel Corporation Italia S.p.A.
Milanofiori Palazzo E
20094 Assago
Milano
Tel: (39) (02) 89200950
FAX: (39) (2) 3498464

NETHERLANDS

Intel Semiconductor B.V.
Postbus 84130
3009 CC Rotterdam
Tel: (31) 10 407 11 11
FAX: (31) 10 455 4688

SPAIN

Intel Iberia S.A.
Zubaran, 28
28010 Madrid
Tel: (34) 308 25 52
FAX: (34) 410 7570

SWEDEN

Intel Sweden A.B.
Dalvagen 24
171 36 Solna
Tel: (46) 8 734 01 00
FAX: (46) 8 278085

UNITED KINGDOM

Intel Corporation (U.K.) Ltd.
Pipers Way
Swindon, Wiltshire SN3 1RJ
Tel: (44) (0793) 696000
FAX: (44) (0793) 641440

EUROPEAN DISTRIBUTORS/REPRESENTATIVES

AUSTRIA

Bacher Electronics GmbH
Rotenmuehlgasse 26
A-1120 Wien
Tel: 43 222 81356460
FAX: 43 222 834276

BELGIUM

Inelco Belgium S.A.
Oorlogskruisenlaan 94
B-1120 Bruxelles
Tel: 32 2 244 2811
FAX: 32 2 216 4301

FRANCE

Almex
48, Rue de l'Aubepine
B.P. 102
92164 Antony Cedex
Tel: 33 1 4096 5400
FAX: 33 1 4666 6028

Lex Electronics
Silic 585
60 Rue des Gemeaux
94663 Rungis Cedex
Tel: 33 1 4978 4978
FAX: 33 1 4978 0596

Metrologie
Tour d'Asnieres
4, Avenue Laurent Cely
92606 Asnieres Cedex
Tel: 33 1 4790 6240
FAX: 33 1 4790 5947

Tekelec-Airtronic
Cite Des Bruyeres
Rue Carle Vernet
BP 2
92310 Sevres
Tel: 33 1 4623 2425
FAX: 33 1 4507 2191

GERMANY

E2000 Vertriebs-AG
Stahlgruberring 12
8000 Muenchen 82
Tel: 49 89 420010
FAX: 49 89 42001209

Jermyn GmbH
Im Dachsstueck 9
6250 Limburg
Tel: 49 6431 5080
FAX: 49 6431 508289

Metrologie GmbH
Steinerstrasse 15
8000 Muenchen 70
Tel: 49 89 724470
FAX: 49 89 72447111

Proelectron Vertriebs GmbH
Max-Planck-Strasse 1-3
6072 Dreieich
Tel: 49 6103 304343
FAX: 49 6103 304425

Rein Electronik GmbH
Loetscher Weg 66
4054 Nettetal 1
Tel: 49 2153 7330
FAX: 49 2153 733513

GREECE

Pouliadis Associates Corp.
5 Koumbari Street
Kolonaki Square
10674 Athens
Tel: 30 1 360 3741
FAX: 30 1 360 7501

IRELAND

Micro Marketing
Tany Hall
Eglinton Terrace
Dundrum
Dublin
Tel: 0001 989 400
FAX: 0001 989 8282

ISRAEL

Eastronics Ltd.
Rozanis 11
P.O.B. 39300
Tel Baruch
Tel-Aviv 61392
Tel: 972 3 475151
FAX: 972 3 475125

ITALY

Celdis Spa
Via F.11i Gracchi 36
20092 Cinisello Balsamo
Milano
Tel: 39 2 66012003
FAX: 39 2 6182433

Intesi Div. Della Deutsche
Divisione ITT
Industries GmbH
P.I. 06550110156
Milanofiori Palazzo E5
20094 Assago (Milano)
Tel: 39 2 824701
FAX: 39 2 8242631

Lasi Elettronica S.p.A.
P.I. 00839000155
Viale Fulvio Testi, N.280
20126 Milano
Tel: 39 2 66101370
FAX: 39 2 66101385

Telcom s.r.l. – Divisione MDS
Via Trombetta
Zona Marconi
Strada Cassanese
Segrate – Milano
Tel: 39 2 2138010
FAX: 39 2 216061

NETHERLANDS

Koning en Hartman B.V.
Energieweg 1
2627 AP Delft
Tel: 31 15 609 906
FAX: 31 15 619 194

PORTUGAL

ATD Electronica LDA
Rua Dr. Faria de
Vasconcelos, 3a
1900 Lisboa
Tel: 351 1 8472200
FAX: 351 1 8472197

SPAIN

ATD Electronica SA
Avda de la Industria, 32
Nave 17, 2B
28100 Alcobendas
Madrid
Tel: 1 661 65 51
FAX: 1 661 63 00

Metrologia Iberica
Avenida de la Industria NR 32-2o
Oficina 17
28100 Alcobendas
Madrid
Tel: (1) 661 11 42
FAX: (1) 661 57 55

SCANDINAVIA

OY Fintronic AB
Heikkilantie 2a
SF-00210 Helsinki
Tel: 358 0 6926022
FAX: 358 0 6821251

ITT Multikomponent A/S
Naverland 29
DK-2600 Glostrup
Denmark
Tel: 010 45 42 451822
FAX: 010 45 42 457624

Nordisk Elektronik A/S
Postboks 122
Smedsvingen 4
N-1364 Hvalstad
Norway
Tel: 47 2 846210
FAX: 47 2 846545

Nordisk Electronik AB
Box 36
Torshamnsgatan 39
S-16493 Kista
Sweden
Tel: 46 8 7034630
FAX: 46 8 7039845

SWITZERLAND

Industrade A.G.
Hertistrasse 31
CH-8304 Wallisellen
Tel: 41 1 8328111
FAX: 41 1 8307550

TURKEY

EMPA
80050 Sishane
Refik Saydam Cad No. 89/5
Istanbul
Tel: 90 1 143 6212
FAX: 90 1 143 6547

UNITED KINGDOM

Access Elect Comp Ltd.
Jubilee House
Jubilee Road
Letchworth
Hertfordshire
SG6 1QH
Tel: 0462 480888
FAX: 0462 682467

Bytech Components Ltd.
12a Cedarwood
Chineham Business Park
Crockford Lane
Basingstoke
Hants RG12 1RW
Tel: 0256 707107
FAX: 0256 707162

Bytech Systems
Unit 3
The Western Centre
Western Road
Bracknell
Berks RG12 1RW
Tel: 0344 55333
FAX: 0344 867270

Metrologie
Rapid House
Oxford Road
High Wycombe
Bucks
Herts HP11 2EE
Tel: 0494 474147
FAX: 0494 452144

Jermyn
Vestry Estate
Otford Road
Sevenoaks
Kent TN14 5EU
Tel: 0732 450144
FAX: 0732 451251

MMD
3 Bennet Court
Bennet Road
Reading
Berkshire RG2 0QX
Tel: 0734 313232
FAX: 0734 313255

Rapid Silicon
3 Bennet Court
Bennet Road
Reading
Berks RG2 0QX
Tel: 0734 752266
FAX: 0734 312728

Metro Systems
Rapid House
Oxford Road
High Wycombe
Bucks HP11 2EE
Tel: 0494 474171
FAX: 0494 21860

YUGOSLAVIA

H.R. Microelectronics Corp.
2005 de la Cruz Blvd.
Suite 220
Santa Clara, CA 95050
U.S.A.
Tel: (408) 988-0286
FAX: (408) 988-0306

intel®

INTERNATIONAL SALES OFFICES

AUSTRALIA

Intel Australia Pty. Ltd.
Unit 13
Allambie Grove Business Park
25 Frenchs Forest Road East
Frenchs Forest, NSW, 2086
Sydney
Tel: 61-2-975-3300
FAX: 61-2-975-3375

Intel Australia Pty. Ltd.
711 High Street
1st Floor
East Kw. Vic., 3102
Melbourne
Tel: 61-3-810-2141
FAX: 61-3-819 7200

BRAZIL

Intel Semiconductores do Brazil LTDA
Avenida Paulista, 1159-CJS 404/405
01311 - Sao Paulo - S.P.
Tel: 55-11-287-5899
TLX: 11-37-557-ISDB
FAX: 55-11-287-5119

CHINA/HONG KONG

Intel PRC Corporation
15/F, Office 1, Citic Bldg.
Jian Guo Men Wai Street
Beijing, PRC
Tel: (1) 500-4850
TLX: 22947 INTEL CN
FAX: (1) 500-2953

Intel Semiconductor Ltd.*
10/F East Tower
Bond Center
Queensway, Central
Hong Kong
Tel: (852) 844-4555
FAX: (852) 868-1989

INDIA

Intel Asia Electronics, Inc.
4/2, Samrah Plaza
St. Mark's Road
Bangalore 560001
Tel: 91-812-215773
TLX: 953-845-2646 INTEL IN
FAX: 091-812-215067

JAPAN

Intel Japan K.K.
5-6 Tokodai, Tsukuba-shi
Ibaraki, 300-26
Tel: 0298-47-8511
FAX: 0298-47-8450

Intel Japan K.K.*
Hachioji ON Bldg.
4-7-14 Myojin-machi
Hachioji-shi, Tokyo 192
Tel: 0426-48-8770
FAX: 0426-48-8775

Intel Japan K.K.*
Bldg. Kumagaya
2-69 Hon-cho
Kumagaya-shi, Saitama 360
Tel: 0485-24-6871
FAX: 0485-24-7518

Intel Japan K.K.*
Kawa-asa Bldg.
2-11-5 Shin-Yokohama
Kohoku-ku, Yokohama-shi
Kanagawa, 222
Tel: 045-474-7661
FAX: 045-471-4394

Intel Japan K.K.*
Ryokuchi-Eki Bldg.
2-4-1 Terauchi
Toyonaka-shi, Osaka 560
Tel: 06-863-1091
FAX: 06-863-1084

Intel Japan K.K.
Shinmaru Bldg.
1-5-1 Marunouchi
Chiyoda-ku, Tokyo 100
Tel: 03-3201-3621
FAX: 03-3201-6850

Intel Japan K.K.
Green Bldg.
1-16-20 Nishiki
Naka-ku, Nagoya-shi
Aichi 460
Tel: 052-204-1261
FAX: 052-204-1285

KOREA

Intel Korea, Ltd.
16th Floor, Life Bldg.
61 Yoido-dong, Youngdeungpo-Ku
Seoul 150-010
Tel: (2) 784-8186
FAX: (2) 784-8096

SINGAPORE

Intel Singapore Technology, Ltd.
101 Thomson Road #08-03/06
United Square
Singapore 1130
Tel: (65) 250-7811
FAX: (65) 250-9256

TAIWAN

Intel Technology Far East Ltd.
Taiwan Branch Office
8th Floor, No. 205
Bank Tower Bldg.
Tung Hua N. Road
Taipei
Tel: 886-2-5144202
FAX: 886-2-717-2455

INTERNATIONAL DISTRIBUTORS/REPRESENTATIVES

ARGENTINA

Dafsys S.R.L.
Chacabuco, 90-6 Piso
1069-Buenos Aires
Tel. & FAX: 54.1334.1871

AUSTRALIA

Email Electronics
15-17 Hume Street
Huntingdale, 3166
Tel: 011-61-3-544-8244
TLX: AA 30895
FAX: 011-61-3-543-8179

NSD-Australia
205 Middleborough Rd.
Box Hill, Victoria 3128
Tel: 03 8900970
FAX: 03 8990819

BRAZIL

Microlinear
Largo do Arouche, 24
01219 Sao Paulo, SP
Tel: 5511-220-2215
FAX: 5511-220-5750

CHILE

Sisteco
Vecinal 40—Las Condes
Santiago
Tel: 562-234-1644
FAX: 562-233-9895

CHINA/HONG KONG

Novel Precision Machinery Co., Ltd.
Room 728 Trade Square
681 Cheung Sha Wan Road
Kowloon, Hong Kong
Tel: (852) 360-8999
TWX: 32032 NVTNL HX
FAX: (852) 725-3695

GUATEMALA

Abinitio
11 Calle 2—Zona 9
Guatemala City
Tel: 5022-32-4104
FAX: 5022-32-4123

INDIA

Micronic Devices
Arun Complex
No. 65 D.V.G. Road
Basavanagudi
Bangalore 560 004
Tel: 011-91-812-600-631
011-91-812-611-365
TLX: 9538458332 MDBG

Micronic Devices
No. 516 5th Floor
Swastik Chambers
Sion, Trombay Road
Chembur
Bombay 400 071
TLX: 9531 171447 MDEV

Micronic Devices
25/8, 1st Floor
Bada Bazaar Marg
Old Rajinder Nagar
New Delhi 110 060
Tel: 011-91-11-5723509
011-91-11-589771
TLX: 031-63253 MDND IN

Micronic Devices
6-3-348/12A Dwarakapuri Colony
Hyderabad 500 482
Tel: 011-91-842-226748

S&S Corporation
1587 Kooser Road
San Jose, CA 95118
Tel: (408) 978-6216
TLX: 820281
FAX: (408) 978-8635

JAMAICA

MC Systems
10-12 Grenada Crescent
Kingston 5
Tel: (809) 929-2638
(809) 926-0188
FAX: (809) 926-0104

JAPAN

Asahi Electronics Co. Ltd.
KMM Bldg. 2-14-1 Asano
Kokurakita-ku
Kitakyushu-shi 802
Tel: 093-511-6471
FAX: 093-551-7861

CTC Components Systems Co., Ltd.
4-8-1 Dobashi, Miyamae-ku
Kawasaki-shi, Kanagawa 213
Tel: 044-852-5121
FAX: 044-877-4268

Dia Semicon Systems, Inc.
Flower Hill Shinmachi Higashi-kan
1-23 Shinmachi, Setagaya-ku
Tokyo 154
Tel: 03-3439-1600
FAX: 03-3439-1601

Okaya Koki
2-4-18 Sakae
Naka-ku, Nagoya-shi 460
Tel: 052-204-8315
FAX: 052-204-8380

Ryoyo Electro Corp.
Konwa Bldg.
1-12-22 Tsukiji
Chuo-ku, Tokyo 104
Tel: 03-3546-5011
FAX: 03-3546-5044

KOREA

J-Tek Corporation
Dong Sung Bldg. 9/F
158-24, Samsung-Dong, Kangnam-Ku
Seoul 135-090
Tel: (822) 557-8039
FAX: (822) 557-8304

Samsung Electronics
Samsung Main Bldg.
150 Taepyung-Ro-2KA, Chung-Ku
Seoul 100-102
C.P.O. Box 8780
Tel: (822) 751-3680
TWX: KORSST K 27970
FAX: (822) 753-9065

MEXICO

PSI S.A. de C.V.
Fco. Villa esq. Ajusco s/n
Cuernavaca, MOR 62130
Tel: 52-73-13-9412
52-73-17-5340
FAX: 52-73-17-5333

NEW ZEALAND

Email Electronics
36 Olive Road
Penrose, Auckland
Tel: 011-64-9-591-155
FAX: 011-64-9-592-681

SAUDI ARABIA

AAE Systems, Inc.
642 N. Pastoria Ave.
Sunnyvale, CA 94086
U.S.A.
Tel: (408) 732-1710
FAX: (408) 732-3095
TLX: 494-3405 AAE SYS

SINGAPORE

Electronic Resources Pte, Ltd.
17 Harvey Road
#03-01 Singapore 1336
Tel: (65) 283-0888
TWX: RS 56541 ERS
FAX: (65) 289-5327

SOUTH AFRICA

Electronic Building Elements
178 Erasmus St. (off Watermeyet St.)
Meyerspark, Pretoria, 0184
Tel: 011-2712-803-7680
FAX: 011-2712-803-8294

TAIWAN

Micro Electronics Corporation
12th Floor, Section 3
285 Nanking East Road
Taipei, R.O.C.
Tel: (886) 2-7198419
FAX: (886) 2-7197916

Acer Sertek Inc.
15th Floor, Section 2
Chien Kuo North Rd.
Taipei 18479 R.O.C.
Tel: 886-2-501-0055
TWX: 23756 SERTEK
FAX: (886) 2-5012521

URUGUAY

Interfase
Zabala 1378
11000 Montevideo
Tel: 5982-96-0490
5982-96-1143
FAX: 5982-96-2965

VENEZUELA

Unixel C.A.
4 Transversal de Monte Cristo
Edf. AXXA, Piso 1, of. 1&2
Centro Empresarial Boleita
Caracas
Tel: 582-238-6082
FAX: 582-238-1816

*Field Application Location

NORTH AMERICAN SERVICE OFFICES

ALASKA

Intel Corp.
c/o TransAlaska Network
1515 Lore Rd.
Anchorage 99507
Tel: (907) 522-1776

Intel Corp.
c/o TransAlaska Data Systems
c/o GCI Operations
520 Fifth Ave., Suite 407
Fairbanks 99701
Tel: (907) 452-6264

ARIZONA

*Intel Corp.
410 North 44th Street
Suite 500
Phoenix 85008
Tel: (602) 231-0386
FAX: (602) 244-0446

*Intel Corp.
500 E. Fry Blvd., Suite M-15
Sierra Vista 85635
Tel: (602) 459-5010

ARKANSAS

Intel Corp.
c/o Federal Express
1500 West Park Drive
Little Rock 72204

CALIFORNIA

*Intel Corp.
21515 Vanowen St., Ste. 116
Canoga Park 91303
Tel: (818) 704-8500

*Intel Corp.
300 N. Continental Blvd.
Suite 100
El Segundo 90245
Tel: (213) 640-6040

*Intel Corp.
1900 Prairie City Rd.
Folsom 95630-9597
Tel: (916) 351-6143

*Intel Corp.
9665 Chesapeake Dr., Suite 325
San Diego 92123
Tel: (619) 292-8086

**Intel Corp.
400 N. Tustin Avenue
Suite 450
Santa Ana 92705
Tel: (714) 835-9642

**Intel Corp.
2700 San Tomas Exp., 1st Floor
Santa Clara 95051
Tel: (408) 970-1747

COLORADO

*Intel Corp.
600 S. Cherry St., Suite 700
Denver 80222
Tel: (303) 321-8086

CONNECTICUT

*Intel Corp.
301 Lee Farm Corporate Park
83 Wooster Heights Rd.
Danbury 06811
Tel: (203) 748-3130

FLORIDA

**Intel Corp.
800 Fairway Dr., Suite 160
Deerfield Beach 33441
Tel: (305) 421-0506
FAX: (305) 421-2444

*Intel Corp.
5850 T.G. Lee Blvd., Ste. 340
Orlando 32822
Tel: (407) 240-8000

GEORGIA

*Intel Corp.
20 Technology Park, Suite 150
Norcross 30092
Tel: (404) 449-0541

5523 Theresa Street
Columbus 31907

HAWAII

**Intel Corp.
Honolulu 96820
Tel: (808) 847-6738

ILLINOIS

**†Intel Corp.
Woodfield Corp. Center III
300 N. Martingale Rd., Ste. 400
Schaumburg 60173
Tel: (708) 605-8031

INDIANA

*Intel Corp.
8910 Purdue Rd., Ste. 350
Indianapolis 46268
Tel: (317) 875-0623

KANSAS

*Intel Corp.
10985 Cody, Suite 140
Overland Park 66210
Tel: (913) 345-2727

KENTUCKY

Intel Corp.
133 Walton Ave., Office 1A
Lexington 40508
Tel: (606) 255-2957

Intel Corp.
896 Hillcrest Road, Apt. A
Radcliff 40160 (Louisville)

LOUISIANA

Hammond 70401
(serviced from Jackson, MS)

MARYLAND

**Intel Corp.
10010 Junction Dr., Suite 200
Annapolis Junction 20701
Tel: (301) 206-2860

MASSACHUSETTS

**Intel Corp.
Westford Corp. Center
3 Carlisle Rd., 2nd Floor
Westford 01886
Tel: (508) 692-0960

MICHIGAN

*Intel Corp.
7071 Orchard Lake Rd., Ste. 100
West Bloomfield 48322
Tel: (313) 851-8905

MINNESOTA

*Intel Corp.
3500 W. 80th St., Suite 360
Bloomington 55431
Tel: (612) 835-6722

MISSISSIPPI

Intel Corp.
c/o Compu-Care
2001 Airport Road, Suite 205F
Jackson 39208
Tel: (601) 932-6275

MISSOURI

*Intel Corp.
3300 Rider Trail South
Suite 170
Earth City 63045
Tel: (314) 291-1990

Intel Corp.
Route 2, Box 221
Smithville 64089
Tel: (913) 345-2727

NEW JERSEY

**Intel Corp.
300 Sylvan Avenue
Englewood Cliffs 07632
Tel: (201) 567-0821

*Intel Corp.
Lincroft Office Center
125 Half Mile Road
Red Bank 07701
Tel: (908) 747-2233

NEW MEXICO

Intel Corp.
Rio Rancho 1
4100 Sara Road
Rio Rancho 87124-1025
(near Albuquerque)
Tel: (505) 893-7000

NEW YORK

*Intel Corp.
2950 Expressway Dr. South
Suite 130
Islandia 11722
Tel: (516) 231-3300

Intel Corp.
300 Westage Business Center
Suite 230
Fishkill 12524
Tel: (914) 897-3860

Intel Corp.
5858 East Molloy Road
Syracuse 13211
Tel: (315) 454-0576

NORTH CAROLINA

*Intel Corp.
5800 Executive Center Drive
Suite 105
Charlotte 28212
Tel: (704) 568-8966

**Intel Corp.
5540 Centerview Dr., Suite 215
Raleigh 27606
Tel: (919) 851-9537

OHIO

**Intel Corp.
3401 Park Center Dr., Ste. 220
Dayton 45414
Tel: (513) 890-5350

*Intel Corp.
25700 Science Park Dr., Ste. 100
Beachwood 44122
Tel: (216) 464-2736

OREGON

**Intel Corp.
15254 N.W. Greenbrier Pkwy.
Building B
Beaverton 97006
Tel: (503) 645-8051

PENNSYLVANIA

*†Intel Corp.
925 Harvest Drive
Suite 200
Blue Bell 19422
Tel: (215) 641-1000
1-800-468-3548
FAX: (215) 641-0785

**†Intel Corp.
400 Penn Center Blvd., Ste. 610
Pittsburgh 15235
Tel: (412) 823-4970

*Intel Corp.
1513 Cedar Cliff Dr.
Camp Hill 17011
Tel: (717) 761-0860

PUERTO RICO

Intel Corp.
South Industrial Park
P.O. Box 910
Las Piedras 00671
Tel: (809) 733-8616

TEXAS

**Intel Corp.
Westech 360, Suite 4230
8911 N. Capitol of Texas Hwy.
Austin 78752-1239
Tel : (512) 794-8086

**†Intel Corp.
12000 Ford Rd., Suite 401
Dallas 75234
Tel: (214) 241-8087

**Intel Corp.
7322 SW Freeway, Suite 1490
Houston 77074
Tel: (713) 988-8086

UTAH

Intel Corp.
428 East 6400 South
Suite 104
Murray 84107
Tel: (801) 263-8051
FAX: (801) 268-1457

VIRGINIA

*Intel Corp.
9030 Stony Point Pkwy.
Suite 360
Richmond 23235
Tel: (804) 330-9393

WASHINGTON

**Intel Corp.
155 108th Avenue N.E., Ste. 386
Bellevue 98004
Tel: (206) 453-8086

CANADA

ONTARIO

**Intel Semiconductor of Canada, Ltd.
2650 Queensview Dr., Ste. 250
Ottawa K2B 8H6
Tel: (613) 829-9714

**Intel Semiconductor of Canada, Ltd.
190 Attwell Dr., Ste. 102
Rexdale (Toronto) M9W 6H8
Tel: (416) 675-2105

QUEBEC

**Intel Semiconductor of Canada, Ltd.
1 Rue Holiday
Suite 115
Tour East
Pt. Claire H9R 5N3
Tel: (514) 694-9130
FAX: 514-694-0064

CUSTOMER TRAINING CENTERS

ARIZONA

2402 W. Beardsley Road
Phoenix 85027
Tel: (602) 869-4288
1-800-468-3548

SYSTEMS ENGINEERING OFFICES

MINNESOTA

3500 W. 80th Street
Suite 360
Bloomington 55431
Tel: (612) 835-6722

NEW YORK

2950 Expressway Dr., South
Islandia 11722
Tel: (506) 231-3300

*Carry-in locations
**Carry-in/mail-in locations

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, 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
Wiltshire, England SN3 1RJ

GERMANY
Intel GmbH
Dornacher Strasse 1
8016 Feldkirchen bei Muenchen

HONG KONG
Intel Semiconductor Ltd.
10/F East Tower
Bond Center
Queensway, Central

CANADA
Intel Semiconductor of Canada, Ltd.
190 Attwell Drive, Suite 500
Rexdale, Ontario M9W 6H8

Printed in USA/292/20K/RRD PS

ISBN 1-55512-163-2