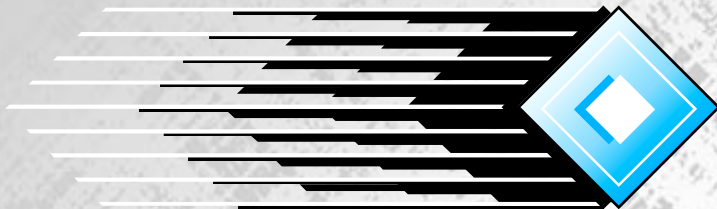


80C186EA/80C188EA  
Microprocessor  
User's Manual

intel<sup>®</sup>





|                |  |
|----------------|--|
| <b>MCS-186</b> | <b>80C186 – 80C188</b><br><b>80C186XL – 80C188XL</b><br><b>80C186EA – 80C188EA</b><br><b>80C186EB – 80C188EB</b><br><b>80C186EC – 80C188EC</b> |
|----------------|--|

|  |  |
|--|--|
| <b>Ceibo In-Circuit<br/>Emulator<br/>Supporting<br/>MCS-186:</b> | <b>DS-186</b><br><a href="http://ceibo.com/eng/products/ds186.shtml">http://ceibo.com/eng/products/ds186.shtml</a> |
|--|--|



**80C186EA/80C188EA  
Microprocessor  
User's Manual**

1995



Information in this document is provided solely to enable use of Intel products. Intel assumes no liability whatsoever, including infringement of any patent or copyright, for sale and use of Intel products except as provided in Intel's Terms and Conditions of Sale for such products.

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

Intel retains the right to make changes to these specifications at any time, without notice.

Contact your local Intel sales office or your distributor to obtain the latest specifications before placing your product order.

MDS is an ordering code only and is not used as a product name or trademark of Intel Corporation.

Intel Corporation and Intel's FASTPATH are not affiliated with Kinetics, a division of Excelan, Inc. or its FASTPATH trademark or products.

\*Other brands and names are the property of their respective owners.

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

Intel Corporation  
Literature Sales  
P.O. Box 7641  
Mt. Prospect, IL 60056-7641  
or call 1-800-879-4683



# CONTENTS

## CHAPTER 1

### INTRODUCTION

|         |   |     |
|---------|---|-----|
| 1.1     | HOW TO USE THIS MANUAL.....   | 1-2 |
| 1.2     | RELATED DOCUMENTS .....   | 1-3 |
| 1.3     | ELECTRONIC SUPPORT SYSTEMS .....  | 1-4 |
| 1.3.1   | FaxBack Service .....   | 1-4 |
| 1.3.2   | Bulletin Board System (BBS) .....   | 1-5 |
| 1.3.2.1 | How to Find <i>ApBUILDER</i> Software and Hypertext Documents<br>on the BBS ..... | 1-5 |
| 1.3.3   | CompuServe Forums .....   | 1-6 |
| 1.3.4   | World Wide Web .....  | 1-6 |
| 1.4     | TECHNICAL SUPPORT .....   | 1-6 |
| 1.5     | PRODUCT LITERATURE.....   | 1-6 |

## CHAPTER 2

### OVERVIEW OF THE 80C186 FAMILY ARCHITECTURE

|         |   |      |
|---------|---|------|
| 2.1     | ARCHITECTURAL OVERVIEW .....                            | 2-1  |
| 2.1.1   | Execution Unit .....                                    | 2-2  |
| 2.1.2   | Bus Interface Unit .....                                | 2-3  |
| 2.1.3   | General Registers .....                                 | 2-4  |
| 2.1.4   | Segment Registers .....                                 | 2-5  |
| 2.1.5   | Instruction Pointer .....                               | 2-6  |
| 2.1.6   | Flags .....   | 2-7  |
| 2.1.7   | Memory Segmentation .....                               | 2-8  |
| 2.1.8   | Logical Addresses .....                                 | 2-10 |
| 2.1.9   | Dynamically Relocatable Code .....                      | 2-13 |
| 2.1.10  | Stack Implementation .....                              | 2-15 |
| 2.1.11  | Reserved Memory and I/O Space .....                     | 2-15 |
| 2.2     | SOFTWARE OVERVIEW .....                                 | 2-17 |
| 2.2.1   | Instruction Set .....                                   | 2-17 |
| 2.2.1.1 | Data Transfer Instructions .....                        | 2-18 |
| 2.2.1.2 | Arithmetic Instructions .....                           | 2-19 |
| 2.2.1.3 | Bit Manipulation Instructions .....                     | 2-21 |
| 2.2.1.4 | String Instructions .....                               | 2-22 |
| 2.2.1.5 | Program Transfer Instructions .....                     | 2-23 |
| 2.2.1.6 | Processor Control Instructions .....                    | 2-27 |
| 2.2.2   | Addressing Modes .....                                  | 2-27 |
| 2.2.2.1 | Register and Immediate Operand Addressing Modes .....   | 2-27 |
| 2.2.2.2 | Memory Addressing Modes .....                           | 2-28 |
| 2.2.2.3 | I/O Port Addressing .....                               | 2-36 |
| 2.2.2.4 | Data Types Used in the 80C186 Modular Core Family ..... | 2-37 |

CONTENTS





**CHAPTER 4**

**PERIPHERAL CONTROL BLOCK**

4.1 PERIPHERAL CONTROL REGISTERS..... 4-1

4.2 PCB RELOCATION REGISTER..... 4-1

4.3 RESERVED LOCATIONS ..... 4-4

4.4 ACCESSING THE PERIPHERAL CONTROL BLOCK..... 4-4

    4.4.1 Bus Cycles .....4-4

    4.4.2 READY Signals and Wait States .....4-4

    4.4.3 F-Bus Operation .....4-5

        4.4.3.1 Writing the PCB Relocation Register .....4-6

        4.4.3.2 Accessing the Peripheral Control Registers .....4-6

        4.4.3.3 Accessing Reserved Locations .....4-6

4.5 SETTING THE PCB BASE LOCATION..... 4-6

    4.5.1 Considerations for the 80C187 Math Coprocessor Interface .....4-7

**CHAPTER 5**

**CLOCK GENERATION AND POWER MANAGEMENT**

5.1 CLOCK GENERATION..... 5-1

    5.1.1 Crystal Oscillator .....5-1

        5.1.1.1 Oscillator Operation .....5-2

        5.1.1.2 Selecting Crystals .....5-5

    5.1.2 Using an External Oscillator .....5-6

    5.1.3 Output from the Clock Generator .....5-6

    5.1.4 Reset and Clock Synchronization .....5-6

5.2 POWER MANAGEMENT..... 5-10

    5.2.1 Idle Mode .....5-11

        5.2.1.1 Entering Idle Mode .....5-11

        5.2.1.2 Bus Operation During Idle Mode .....5-13

        5.2.1.3 Leaving Idle Mode .....5-14

        5.2.1.4 Example Idle Mode Initialization Code .....5-15

    5.2.2 Powerdown Mode .....5-16

        5.2.2.1 Entering Powerdown Mode .....5-17

        5.2.2.2 Leaving Powerdown Mode .....5-18

    5.2.3 Power-Save Mode .....5-19

        5.2.3.1 Entering Power-Save Mode .....5-19

        5.2.3.2 Leaving Power-Save Mode .....5-21

        5.2.3.3 Example Power-Save Initialization Code .....5-21

    5.2.4 Implementing a Power Management Scheme .....5-23





## CONTENTS

### CHAPTER 6

#### CHIP-SELECT UNIT

|         |   |      |
|---------|---|------|
| 6.1     | COMMON METHODS FOR GENERATING CHIP-SELECTS..... | 6-1  |
| 6.2     | CHIP-SELECT UNIT FEATURES AND BENEFITS .....    | 6-1  |
| 6.3     | CHIP-SELECT UNIT FUNCTIONAL OVERVIEW .....      | 6-2  |
| 6.4     | PROGRAMMING .....                               | 6-6  |
| 6.4.1   | Initialization Sequence .....                   | 6-6  |
| 6.4.2   | Programming the Active Ranges .....             | 6-12 |
| 6.4.2.1 | UCS Active Range .....                          | 6-12 |
| 6.4.2.2 | LCS Active Range .....                          | 6-13 |
| 6.4.2.3 | MCS Active Range .....                          | 6-13 |
| 6.4.2.4 | PCS Active Range .....                          | 6-15 |
| 6.4.3   | Bus Wait State and Ready Control .....          | 6-15 |
| 6.4.4   | Overlapping Chip-Selects .....                  | 6-16 |
| 6.4.5   | Memory or I/O Bus Cycle Decoding .....          | 6-17 |
| 6.4.6   | Programming Considerations .....                | 6-17 |
| 6.5     | CHIP-SELECTS AND BUS HOLD.....                  | 6-18 |
| 6.6     | EXAMPLES .....                                  | 6-18 |
| 6.6.1   | Example 1: Typical System Configuration .....   | 6-18 |

### CHAPTER 7

#### REFRESH CONTROL UNIT

|         |  |      |
|---------|--|------|
| 7.1     | THE ROLE OF THE REFRESH CONTROL UNIT.....      | 7-2  |
| 7.2     | REFRESH CONTROL UNIT CAPABILITIES.....         | 7-2  |
| 7.3     | REFRESH CONTROL UNIT OPERATION.....            | 7-2  |
| 7.4     | REFRESH ADDRESSES.....                         | 7-4  |
| 7.5     | REFRESH BUS CYCLES .....                       | 7-5  |
| 7.6     | GUIDELINES FOR DESIGNING DRAM CONTROLLERS..... | 7-5  |
| 7.7     | PROGRAMMING THE REFRESH CONTROL UNIT.....      | 7-7  |
| 7.7.1   | Calculating the Refresh Interval .....         | 7-7  |
| 7.7.2   | Refresh Control Unit Registers .....           | 7-7  |
| 7.7.2.1 | Refresh Base Address Register .....            | 7-8  |
| 7.7.2.2 | Refresh Clock Interval Register .....          | 7-8  |
| 7.7.2.3 | Refresh Control Register .....                 | 7-9  |
| 7.7.3   | Programming Example .....                      | 7-10 |
| 7.8     | REFRESH OPERATION AND BUS HOLD.....            | 7-12 |







## CONTENTS

## CONTENTS

|                                  |   |       |
|----------------------------------|---|-------|
| 9.2.4                            | Pulsed and Variable Duty Cycle Output .....           | 9-14  |
| 9.2.5                            | Enabling/Disabling Counters .....                     | 9-15  |
| 9.2.6                            | Timer Interrupts .....                                | 9-16  |
| 9.2.7                            | Programming Considerations .....                      | 9-16  |
| 9.3                              | TIMING .....  | 9-16  |
| 9.3.1                            | Input Setup and Hold Timings .....                    | 9-16  |
| 9.3.2                            | Synchronization and Maximum Frequency .....           | 9-17  |
| 9.3.2.1                          | Timer/Counter Unit Application Examples .....         | 9-17  |
| 9.3.3                            | Real-Time Clock .....                                 | 9-17  |
| 9.3.4                            | Square-Wave Generator .....                           | 9-17  |
| 9.3.5                            | Digital One-Shot .....                                | 9-17  |
| <br>                             |   |       |
| <b>CHAPTER 10</b>                |   |       |
| <b>DIRECT MEMORY ACCESS UNIT</b> |   |       |
| 10.1                             | FUNCTIONAL OVERVIEW.....                              | 10-1  |
| 10.1.1                           | The DMA Transfer .....                                | 10-1  |
| 10.1.1.1                         | DMA Transfer Directions .....                         | 10-3  |
| 10.1.1.2                         | Byte and Word Transfers .....                         | 10-3  |
| 10.1.2                           | Source and Destination Pointers .....                 | 10-3  |
| 10.1.3                           | DMA Requests .....                                    | 10-3  |
| 10.1.4                           | External Requests .....                               | 10-4  |
| 10.1.4.1                         | Source Synchronization .....                          | 10-5  |
| 10.1.4.2                         | Destination Synchronization .....                     | 10-5  |
| 10.1.5                           | Internal Requests .....                               | 10-6  |
| 10.1.5.1                         | Timer 2-Initiated Transfers .....                     | 10-6  |
| 10.1.5.2                         | Unsynchronized Transfers .....                        | 10-6  |
| 10.1.6                           | DMA Transfer Counts .....                             | 10-7  |
| 10.1.7                           | Termination and Suspension of DMA Transfers .....     | 10-7  |
| 10.1.7.1                         | Termination at Terminal Count .....                   | 10-7  |
| 10.1.7.2                         | Software Termination .....                            | 10-7  |
| 10.1.7.3                         | Suspension of DMA During NMI .....                    | 10-7  |
| 10.1.7.4                         | Software Suspension .....                             | 10-7  |
| 10.1.8                           | DMA Unit Interrupts .....                             | 10-8  |
| 10.1.9                           | DMA Cycles and the BIU .....                          | 10-8  |
| 10.1.10                          | The Two-Channel DMA Unit .....                        | 10-8  |
| 10.1.10.1                        | DMA Channel Arbitration .....                         | 10-8  |
| 10.2                             | PROGRAMMING THE DMA UNIT .....                        | 10-10 |
| 10.2.1                           | DMA Channel Parameters .....                          | 10-10 |
| 10.2.1.1                         | Programming the Source and Destination Pointers ..... | 10-10 |
| 10.2.1.2                         | Selecting Byte or Word Size Transfers .....           | 10-14 |
| 10.2.1.3                         | Selecting the Source of DMA Requests .....            | 10-17 |
| 10.2.1.4                         | Arming the DMA Channel .....                          | 10-18 |
| 10.2.1.5                         | Selecting Channel Synchronization .....               | 10-18 |
| 10.2.1.6                         | Programming the Transfer Count Options .....          | 10-18 |
| 10.2.1.7                         | Generating Interrupts on Terminal Count .....         | 10-19 |





- 10.2.1.8 Setting the Relative Priority of a Channel ..... 10-19
- 10.2.2 Suspension of DMA Transfers ..... 10-20
- 10.2.3 Initializing the DMA Unit ..... 10-20
- 10.3 HARDWARE CONSIDERATIONS AND THE DMA UNIT ..... 10-20
  - 10.3.1 DRQ Pin Timing Requirements ..... 10-20
  - 10.3.2 DMA Latency ..... 10-21
  - 10.3.3 DMA Transfer Rates ..... 10-21
  - 10.3.4 Generating a DMA Acknowledge ..... 10-22
- 10.4 DMA UNIT EXAMPLES ..... 10-22

**CHAPTER 11**

**MATH COPROCESSING**

- 11.1 OVERVIEW OF MATH COPROCESSING ..... 11-1
- 11.2 AVAILABILITY OF MATH COPROCESSING ..... 11-1
- 11.3 THE 80C187 MATH COPROCESSOR..... 11-2
  - 11.3.1 80C187 Instruction Set ..... 11-2
    - 11.3.1.1 Data Transfer Instructions ..... 11-3
    - 11.3.1.2 Arithmetic Instructions ..... 11-3
    - 11.3.1.3 Comparison Instructions ..... 11-5
    - 11.3.1.4 Transcendental Instructions ..... 11-5
    - 11.3.1.5 Constant Instructions ..... 11-6
    - 11.3.1.6 Processor Control Instructions ..... 11-6
  - 11.3.2 80C187 Data Types ..... 11-7
- 11.4 MICROPROCESSOR AND COPROCESSOR OPERATION..... 11-7
  - 11.4.1 Clocking the 80C187 ..... 11-10
  - 11.4.2 Processor Bus Cycles Accessing the 80C187 ..... 11-10
  - 11.4.3 System Design Tips ..... 11-11
  - 11.4.4 Exception Trapping ..... 11-13
- 11.5 EXAMPLE MATH COPROCESSOR ROUTINES..... 11-13

**CHAPTER 12**

**ONCE MODE**

- 12.1 ENTERING/LEAVING ONCE MODE..... 12-1

**APPENDIX A**

**80C186 INSTRUCTION SET ADDITIONS AND EXTENSIONS**

- A.1 80C186 INSTRUCTION SET ADDITIONS ..... A-1
  - A.1.1 Data Transfer Instructions ..... A-1
  - A.1.2 String Instructions ..... A-2
  - A.1.3 High-Level Instructions ..... A-2
- A.2 80C186 INSTRUCTION SET ENHANCEMENTS..... A-8
  - A.2.1 Data Transfer Instructions ..... A-8
  - A.2.2 Arithmetic Instructions ..... A-9





**CONTENTS**

A.2.3 Bit Manipulation Instructions ..... A-9  
A.2.3.1 Shift Instructions ..... A-9  
A.2.3.2 Rotate Instructions ..... A-10

**APPENDIX B  
INPUT SYNCHRONIZATION**

B.1 WHY SYNCHRONIZERS ARE REQUIRED ..... B-1  
B.2 ASYNCHRONOUS PINS ..... B-2

**APPENDIX C  
INSTRUCTION SET DESCRIPTIONS**

**APPENDIX D  
INSTRUCTION SET OPCODES AND CLOCK CYCLES**

**INDEX**



FIGURES

| Figure | Page  |
|--------|---|
| 2-1    | Simplified Functional Block Diagram of the 80C186 Family CPU .....2-2 |
| 2-2    | Physical Address Generation .....2-3                                  |
| 2-3    | General Registers .....2-4  |
| 2-4    | Segment Registers .....2-6  |
| 2-5    | Processor Status Word .....2-9  |
| 2-6    | Segment Locations in Physical Memory.....2-10                         |
| 2-7    | Currently Addressable Segments .....2-11                              |
| 2-8    | Logical and Physical Address .....2-12                                |
| 2-9    | Dynamic Code Relocation .....2-14                                     |
| 2-10   | Stack Operation.....2-16  |
| 2-11   | Flag Storage Format .....2-19   |
| 2-12   | Memory Address Computation .....2-29                                  |
| 2-13   | Direct Addressing .....2-30   |
| 2-14   | Register Indirect Addressing .....2-31                                |
| 2-15   | Based Addressing .....2-31  |
| 2-16   | Accessing a Structure with Based Addressing .....2-32                 |
| 2-17   | Indexed Addressing.....2-33   |
| 2-18   | Accessing an Array with Indexed Addressing .....2-33                  |
| 2-19   | Based Index Addressing .....2-34                                      |
| 2-20   | Accessing a Stacked Array with Based Index Addressing .....2-35       |
| 2-21   | String Operand .....2-36  |
| 2-22   | I/O Port Addressing .....2-36   |
| 2-23   | 80C186 Modular Core Family Supported Data Types.....2-38              |
| 2-24   | Interrupt Control Unit .....2-39                                      |
| 2-25   | Interrupt Vector Table.....2-40                                       |
| 2-26   | Interrupt Sequence .....2-42  |
| 2-27   | Interrupt Response Factors .....2-46                                  |
| 2-28   | Simultaneous NMI and Exception .....2-47                              |
| 2-29   | Simultaneous NMI and Single Step Interrupts.....2-48                  |
| 2-30   | Simultaneous NMI, Single Step and Maskable Interrupt.....2-49         |
| 3-1    | Physical Data Bus Models .....3-2                                     |
| 3-2    | 16-Bit Data Bus Byte Transfers .....3-3                               |
| 3-3    | 16-Bit Data Bus Even Word Transfers .....3-4                          |
| 3-4    | 16-Bit Data Bus Odd Word Transfers.....3-5                            |
| 3-5    | 8-Bit Data Bus Word Transfers.....3-6                                 |
| 3-6    | Typical Bus Cycle .....3-8  |
| 3-7    | T-State Relation to CLKOUT .....3-8                                   |
| 3-8    | BIU State Diagram .....3-9  |
| 3-9    | T-State and Bus Phases .....3-10                                      |
| 3-10   | Address/Status Phase Signal Relationships .....3-11                   |
| 3-11   | Demultiplexing Address Information.....3-12                           |
| 3-12   | Data Phase Signal Relationships .....3-14                             |
| 3-13   | Typical Bus Cycle with Wait States .....3-15                          |
| 3-14   | ARDY and SRDY Pin Block Diagram .....3-15                             |

## FIGURES

| Figure |  | Page |
|--------|--|------|
| 3-15   | Generating a Normally Not-Ready Bus Signal .....       | 3-16 |
| 3-16   | Generating a Normally Ready Bus Signal .....           | 3-17 |
| 3-17   | Normally Not-Ready System Timing .....                 | 3-18 |
| 3-18   | Normally Ready System Timings .....                    | 3-19 |
| 3-19   | Typical Read Bus Cycle .....                           | 3-21 |
| 3-20   | Read-Only Device Interface .....                       | 3-22 |
| 3-21   | Typical Write Bus Cycle.....                           | 3-23 |
| 3-22   | 16-Bit Bus Read/Write Device Interface .....           | 3-24 |
| 3-23   | Interrupt Acknowledge Bus Cycle.....                   | 3-26 |
| 3-24   | Typical 82C59A Interface .....                         | 3-27 |
| 3-25   | HALT Bus Cycle .....                                   | 3-30 |
| 3-26   | Returning to HALT After a HOLD/HLDA Bus Exchange ..... | 3-31 |
| 3-27   | Returning to HALT After a Refresh Bus Cycle .....      | 3-32 |
| 3-28   | Returning to HALT After a DMA Bus Cycle .....          | 3-33 |
| 3-29   | Exiting HALT (Powerdown Mode) .....                    | 3-34 |
| 3-30   | Exiting HALT (Active/Idle Mode).....                   | 3-35 |
| 3-31   | DEN and DT/R Timing Relationships .....                | 3-36 |
| 3-32   | Buffered AD Bus System.....                            | 3-37 |
| 3-33   | Qualifying DEN with Chip-Selects .....                 | 3-38 |
| 3-34   | Queue Status Timing.....                               | 3-41 |
| 3-35   | Timing Sequence Entering HOLD .....                    | 3-42 |
| 3-36   | Refresh Request During HOLD .....                      | 3-44 |
| 3-37   | Latching HLDA .....                                    | 3-45 |
| 3-38   | Exiting HOLD.....                                      | 3-46 |
| 4-1    | PCB Relocation Register.....                           | 4-2  |
| 5-1    | Clock Generator .....                                  | 5-1  |
| 5-2    | Ideal Operation of Pierce Oscillator.....              | 5-2  |
| 5-3    | Crystal Connections to Microprocessor.....             | 5-3  |
| 5-4    | Equations for Crystal Calculations.....                | 5-4  |
| 5-5    | Simple RC Circuit for Powerup Reset .....              | 5-7  |
| 5-6    | Cold Reset Waveform .....                              | 5-8  |
| 5-7    | Warm Reset Waveform .....                              | 5-9  |
| 5-8    | Clock Synchronization at Reset.....                    | 5-10 |
| 5-9    | Power Control Register .....                           | 5-12 |
| 5-10   | Entering Idle Mode .....                               | 5-13 |
| 5-11   | HOLD/HLDA During Idle Mode.....                        | 5-14 |
| 5-12   | Entering Powerdown Mode .....                          | 5-17 |
| 5-13   | Powerdown Timer Circuit .....                          | 5-18 |
| 5-14   | Power-Save Register .....                              | 5-20 |
| 5-15   | Power-Save Clock Transition .....                      | 5-21 |
| 6-1    | Common Chip-Select Generation Methods.....             | 6-2  |
| 6-2    | Chip-Select Block Diagram.....                         | 6-3  |
| 6-3    | Chip-Select Relative Timings .....                     | 6-4  |
| 6-4    | UCS Reset Configuration .....                          | 6-5  |

FIGURES

| Figure | Page  |
|--------|---|
| 6-5    | UMCS Register Definition.....6-7                                      |
| 6-6    | LMCS Register Definition .....6-8                                     |
| 6-7    | MMCS Register Definition .....6-9                                     |
| 6-8    | PACS Register Definition .....6-10                                    |
| 6-9    | MPCS Register Definition.....6-11                                     |
| 6-10   | MCS3:0 Active Ranges .....6-14  |
| 6-11   | Wait State and Ready Control Functions .....6-16                      |
| 6-12   | Using Chip-Selects During HOLD .....6-18                              |
| 6-13   | Typical System .....6-19  |
| 7-1    | Refresh Control Unit Block Diagram.....7-1                            |
| 7-2    | Refresh Control Unit Operation Flow Chart.....7-3                     |
| 7-3    | Refresh Address Formation.....7-4                                     |
| 7-4    | Suggested DRAM Control Signal Timing Relationships.....7-6            |
| 7-5    | Formula for Calculating Refresh Interval for RFTIME Register .....7-7 |
| 7-6    | Refresh Base Address Register .....7-8                                |
| 7-7    | Refresh Clock Interval Register.....7-9                               |
| 7-8    | Refresh Control Register.....7-10                                     |
| 7-9    | Regaining Bus Control to Run a DRAM Refresh Bus Cycle.....7-13        |
| 8-1    | Interrupt Control Unit in Master Mode .....8-2                        |
| 8-2    | Using External 8259A Modules in Cascade Mode .....8-8                 |
| 8-3    | Interrupt Control Unit Latency and Response Time .....8-11            |
| 8-4    | Interrupt Control Register for Internal Sources.....8-13              |
| 8-5    | Interrupt Control Register for Noncascadable External Pins .....8-14  |
| 8-6    | Interrupt Control Register for Cascadable Interrupt Pins.....8-15     |
| 8-7    | Interrupt Request Register .....8-16                                  |
| 8-8    | Interrupt Mask Register .....8-17                                     |
| 8-9    | Priority Mask Register .....8-18                                      |
| 8-10   | In-Service Register .....8-19   |
| 8-11   | Poll Register .....8-20   |
| 8-12   | Poll Status Register .....8-21  |
| 8-13   | End-of-Interrupt Register .....8-22                                   |
| 8-14   | Interrupt Status Register .....8-23                                   |
| 8-15   | Interrupt Control Unit in Slave Mode .....8-24                        |
| 8-16   | Interrupt Sources in Slave Mode .....8-25                             |
| 8-17   | Interrupt Vector Register (Slave Mode Only).....8-27                  |
| 8-18   | End-of-Interrupt Register in Slave Mode .....8-28                     |
| 8-19   | Request, Mask, and In-Service Registers .....8-28                     |
| 8-20   | Interrupt Vectoring in Slave Mode .....8-29                           |
| 8-21   | Interrupt Response Time in Slave Mode .....8-30                       |
| 9-1    | Timer/Counter Unit Block Diagram.....9-2                              |
| 9-2    | Counter Element Multiplexing and Timer Input Synchronization.....9-3  |
| 9-3    | Timers 0 and 1 Flow Chart .....9-4                                    |
| 9-4    | Timer/Counter Unit Output Modes.....9-6                               |
| 9-5    | Timer 0 and Timer 1 Control Registers .....9-7                        |

FIGURES

| Figure | Page   |
|--------|--|
| 9-6    | Timer 2 Control Register ..... 9-9                               |
| 9-7    | Timer Count Registers..... 9-10                                  |
| 9-8    | Timer Maxcount Compare Registers..... 9-11                       |
| 9-9    | TxOUT Signal Timing ..... 9-15                                   |
| 10-1   | Typical DMA Transfer..... 10-2                                   |
| 10-2   | DMA Request Minimum Response Time ..... 10-4                     |
| 10-3   | Source-Synchronized Transfers ..... 10-5                         |
| 10-4   | Destination-Synchronized Transfers ..... 10-6                    |
| 10-5   | Two-Channel DMA Module ..... 10-9                                |
| 10-6   | Examples of DMA Priority..... 10-10                              |
| 10-7   | DMA Source Pointer (High-Order Bits)..... 10-11                  |
| 10-8   | DMA Source Pointer (Low-Order Bits) ..... 10-12                  |
| 10-9   | DMA Destination Pointer (High-Order Bits)..... 10-13             |
| 10-10  | DMA Destination Pointer (Low-Order Bits)..... 10-14              |
| 10-11  | DMA Control Register..... 10-15                                  |
| 10-12  | Transfer Count Register ..... 10-19                              |
| 11-1   | 80C187-Supported Data Types..... 11-8                            |
| 11-2   | 80C186 Modular Core Family/80C187 System Configuration..... 11-9 |
| 11-3   | 80C187 Configuration with a Partially Buffered Bus..... 11-12    |
| 11-4   | 80C187 Exception Trapping via Processor Interrupt Pin..... 11-14 |
| 12-1   | Entering/Leaving ONCE Mode ..... 12-2                            |
| A-1    | Formal Definition of ENTER ..... A-3                             |
| A-2    | Variable Access in Nested Procedures ..... A-4                   |
| A-3    | Stack Frame for Main at Level 1..... A-4                         |
| A-4    | Stack Frame for Procedure A at Level 2 ..... A-5                 |
| A-5    | Stack Frame for Procedure B at Level 3 Called from A..... A-6    |
| A-6    | Stack Frame for Procedure C at Level 3 Called from B ..... A-7   |
| B-1    | Input Synchronization Circuit..... B-1                           |





## TABLES

| Table   | Page |
|---|------|
| 1-1 Comparison of 80C186 Modular Core Family Products .....                     | 1-2  |
| 1-2 Related Documents and Software .....  | 1-3  |
| 2-1 Implicit Use of General Registers .....                                     | 2-5  |
| 2-2 Logical Address Sources .....   | 2-13 |
| 2-3 Data Transfer Instructions .....  | 2-18 |
| 2-4 Arithmetic Instructions .....   | 2-20 |
| 2-5 Arithmetic Interpretation of 8-Bit Numbers .....                            | 2-21 |
| 2-6 Bit Manipulation Instructions .....   | 2-21 |
| 2-7 String Instructions .....   | 2-22 |
| 2-8 String Instruction Register and Flag Use .....                              | 2-23 |
| 2-9 Program Transfer Instructions .....   | 2-25 |
| 2-10 Interpretation of Conditional Transfers .....                              | 2-26 |
| 2-11 Processor Control Instructions .....                                       | 2-27 |
| 2-12 Supported Data Types .....   | 2-37 |
| 3-1 Bus Cycle Types .....   | 3-12 |
| 3-2 Read Bus Cycle Types .....  | 3-20 |
| 3-3 Read Cycle Critical Timing Parameters .....                                 | 3-20 |
| 3-4 Write Bus Cycle Types .....   | 3-23 |
| 3-5 Write Cycle Critical Timing Parameters .....                                | 3-25 |
| 3-6 HALT Bus Cycle Pin States .....   | 3-29 |
| 3-7 Queue Status Signal Decoding .....  | 3-40 |
| 3-8 Signal Condition Entering HOLD .....  | 3-42 |
| 4-1 Peripheral Control Block .....  | 4-3  |
| 5-1 Suggested Values for Inductor L1 in Third Overtone Oscillator Circuit ..... | 5-4  |
| 5-2 Summary of Power Management Modes .....                                     | 5-23 |
| 6-1 Chip-Select Unit Registers .....  | 6-6  |
| 6-2 $\overline{UCS}$ Block Size and Starting Address .....                      | 6-12 |
| 6-3 $\overline{LCS}$ Active Range .....   | 6-13 |
| 6-4 $\overline{MCS}$ Active Range .....   | 6-13 |
| 6-5 $\overline{MCS}$ Block Size and Start Address Restrictions .....            | 6-14 |
| 6-6 $\overline{PCS}$ Active Range .....   | 6-15 |
| 7-1 Identification of Refresh Bus Cycles .....                                  | 7-5  |
| 8-1 Default Interrupt Priorities .....  | 8-3  |
| 8-2 Fixed Interrupt Types .....   | 8-9  |
| 8-3 Interrupt Control Unit Registers in Master Mode .....                       | 8-11 |
| 8-4 Interrupt Control Unit Register Comparison .....                            | 8-26 |
| 8-5 Slave Mode Fixed Interrupt Type Bits .....                                  | 8-26 |
| 9-1 Timer 0 and 1 Clock Sources .....   | 9-12 |
| 9-2 Timer Retriggering .....  | 9-13 |
| 11-1 80C187 Data Transfer Instructions .....                                    | 11-3 |
| 11-2 80C187 Arithmetic Instructions .....                                       | 11-4 |
| 11-3 80C187 Comparison Instructions .....                                       | 11-5 |
| 11-4 80C187 Transcendental Instructions .....                                   | 11-5 |
| 11-5 80C187 Constant Instructions .....   | 11-6 |

## TABLES

| <b>Table</b> |   | <b>Page</b> |
|--------------|---|-------------|
| 11-6         | 80C187 Processor Control Instructions.....      | 11-6        |
| 11-7         | 80C187 I/O Port Assignments.....                | 11-10       |
| C-1          | Instruction Format Variables.....               | C-1         |
| C-2          | Instruction Operands.....                       | C-2         |
| C-3          | Flag Bit Functions.....                         | C-3         |
| C-4          | Instruction Set.....                            | C-4         |
| D-1          | Operand Variables.....                          | D-1         |
| D-2          | Instruction Set Summary.....                    | D-2         |
| D-3          | Machine Instruction Decoding Guide.....         | D-9         |
| D-4          | Mnemonic Encoding Matrix.....                   | D-20        |
| D-5          | Abbreviations for Mnemonic Encoding Matrix..... | D-22        |





## EXAMPLES

| <b>Example</b> |   | <b>Page</b> |
|----------------|---|-------------|
| 5-1            | Initializing the Power Management Unit for Idle or Powerdown Mode ..... | 5-16        |
| 5-2            | Initializing the Power Management Unit for Power-Save Mode .....        | 5-22        |
| 6-1            | Initializing the Chip-Select Unit.....                                  | 6-20        |
| 7-1            | Initializing the Refresh Control Unit .....                             | 7-11        |
| 8-1            | Initializing the Interrupt Control Unit for Master Mode .....           | 8-31        |
| 9-1            | Configuring a Real-Time Clock.....                                      | 9-18        |
| 9-2            | Configuring a Square-Wave Generator.....                                | 9-21        |
| 9-3            | Configuring a Digital One-Shot.....                                     | 9-22        |
| 10-1           | Initializing the DMA Unit .....   | 10-23       |
| 10-2           | Timed DMA Transfers .....   | 10-26       |
| 11-1           | Initialization Sequence for 80C187 Math Coprocessor .....               | 11-15       |
| 11-2           | Floating Point Math Routine Using FSINCOS.....                          | 11-16       |





**1**

# **Introduction**

**|**





## 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 1980s it was clear that a replacement for the 8086/8088 was necessary.

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

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

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

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

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

INTRODUCTION







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

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

## 1.2 RELATED DOCUMENTS

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

### NOTE

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

**Table 1-2. Related Documents and Software**

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

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

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

### 1.3 ELECTRONIC SUPPORT SYSTEMS

Intel FaxBack\* service and application BBS provide up-to-date technical information. Intel also maintains several forums on CompuServe and offers a variety of information on the World Wide Web. These systems are available 24 hours a day, 7 days a week, providing technical information whenever you need it.

#### 1.3.1 FaxBack Service

FaxBack is an on-demand publishing system that sends documents to your fax machine. You can get product announcements, change notifications, product literature, device characteristics, design recommendations, and quality and reliability information from FaxBack 24 hours a day, 7 days a week.

|                  |                                   |
|------------------|-----------------------------------|
| 1-800-628-2283   | U.S. and Canada                   |
| 916-356-3105     | U.S., Canada, Japan, Asia Pacific |
| 44(0)1793-496646 | Europe                            |

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

Each document has an order number and is listed in a subject catalog. The first time you use FaxBack, you should order the appropriate subject catalogs to get a complete list of document order numbers. Catalogs are updated twice monthly. In addition, daily update catalogs list the title, status, and order number of each document that has been added, revised, or deleted during the past eight weeks. To receive the update for a subject catalog, enter the subject catalog number followed by a zero. For example, for the complete microcontroller and flash catalog, request document number 2; for the daily update to the microcontroller and flash catalog, request document number 20.

The following catalogs and information are available at the time of publication:

1. *Solutions OEM* subscription form
2. Microcontroller and flash catalog

3. Development tools catalog
4. Systems catalog
5. Multimedia catalog
6. Multibus and iRMX® software catalog and BBS file listings
7. Microprocessor, PCI, and peripheral catalog
8. Quality and reliability and change notification catalog
9. iAL (Intel Architecture Labs) technology catalog

### 1.3.2 Bulletin Board System (BBS)

The bulletin board system (BBS) lets you download files to your computer. The application BBS has the latest *ApBUILDER* software, hypertext manuals and datasheets, software drivers, firmware upgrades, application notes and utilities, and quality and reliability data.

|                  |  |
|------------------|--|
| 916-356-3600     | U.S., Canada, Japan, Asia Pacific (up to 19.2 Kbaud) |
| 916-356-7209     | U.S., Canada, Japan, Asia Pacific (2400 baud only)   |
| 44(0)1793-496340 | Europe   |

The toll-free BBS (available in the U.S. and Canada) offers lists of documents available from FaxBack, a master list of files available from the application BBS, and a BBS user BBS file listing is also available from FaxBack (catalog number 6; see page 1-4 for phone numbers and a description of the FaxBack service).

|                |                      |
|----------------|----------------------|
| 1-800-897-2536 | U.S. and Canada only |
|----------------|----------------------|

Any customer with a modem and computer can access the BBS. The system provides automatic configuration support for 1200- through 19200-baud modems. Typical modem settings are 14400 baud, no parity, 8 data bits, and 1 stop bit (14400, N, 8, 1).

To access the BBS, just dial the telephone number and respond to the system prompts. During your first session, the system asks you to register with the system operator by entering your name and location. The system operator will set up your access account within 24 hours. At that time, you can access the files on the BBS.

#### NOTE

If you encounter any difficulty accessing the high-speed modem, try the dedicated 2400-baud modem. Use these modem settings: 2400, N, 8, 1.

#### 1.3.2.1 How to Find *ApBUILDER* Software and Hypertext Documents on the BBS

The latest *ApBUILDER* files and hypertext manuals and datasheets are available first from the BBS. To access the files, complete these steps:

1. Type **F** from the BBS Main menu. The BBS displays the Intel Apps Files menu.

INTRODUCTION





**2**

**Overview of the  
80C186 Family  
Architecture**







## CHAPTER 2 OVERVIEW OF THE 80C186 FAMILY ARCHITECTURE

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

### 2.1 ARCHITECTURAL OVERVIEW

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

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

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

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

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



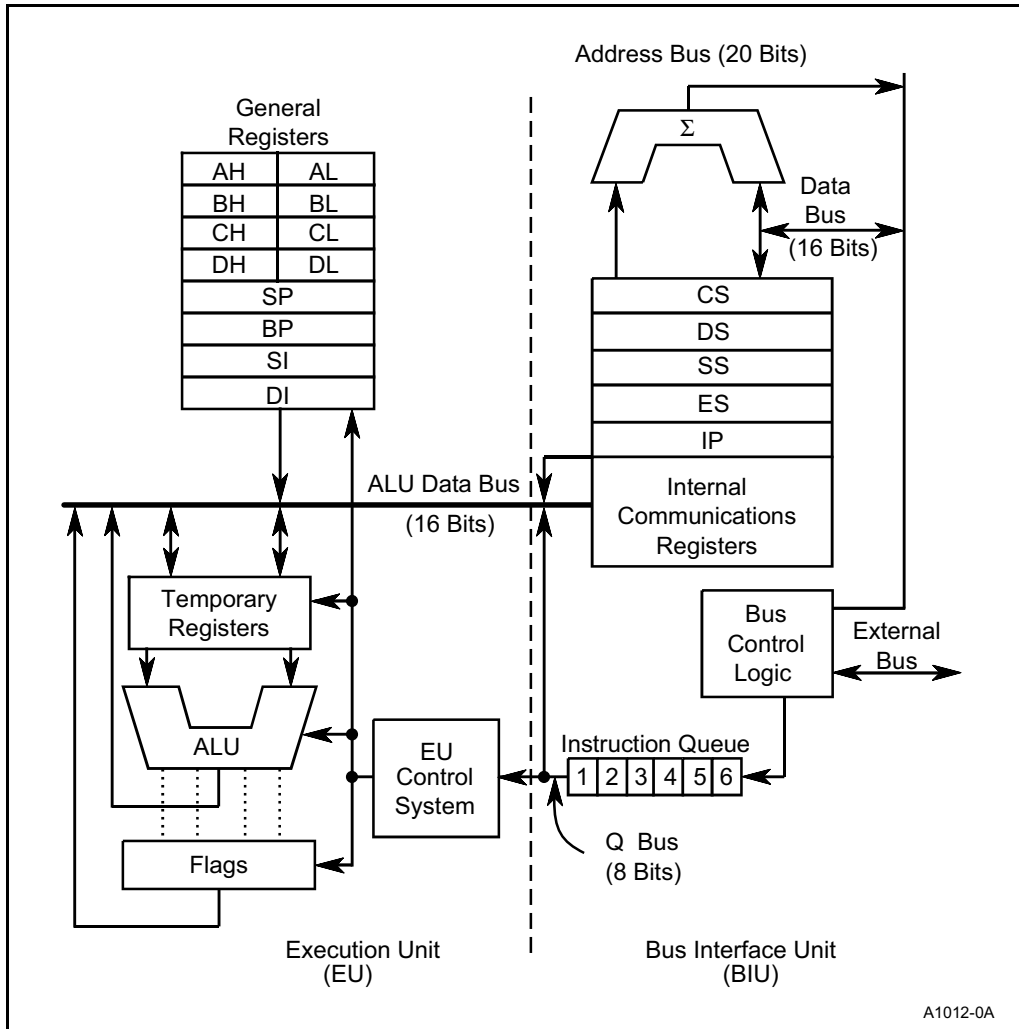
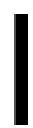


Figure 2-1. Simplified Functional Block Diagram of the 80C186 Family CPU

### 2.1.1 Execution Unit

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





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

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

### 2.1.2 Bus Interface Unit

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

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

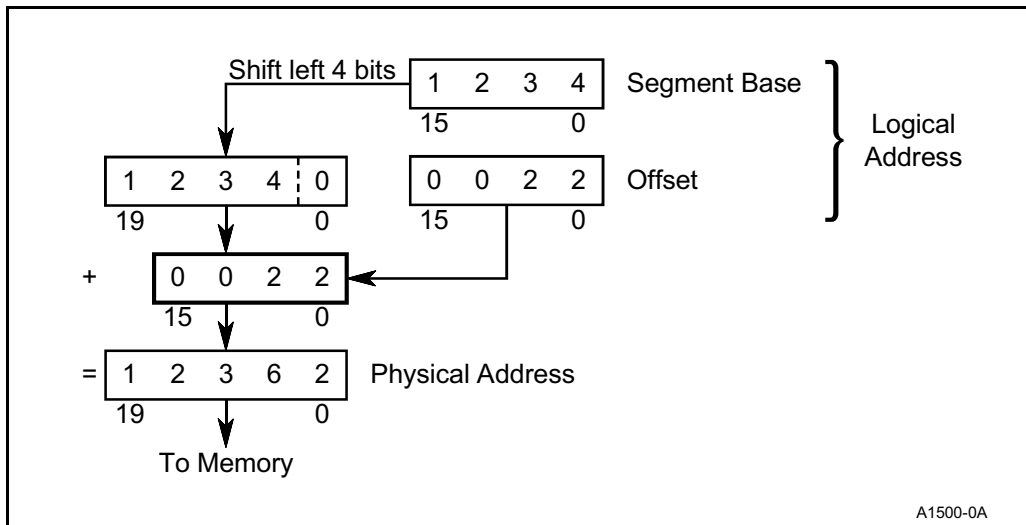


Figure 2-2. Physical Address Generation

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

### 2.1.3 General Registers

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

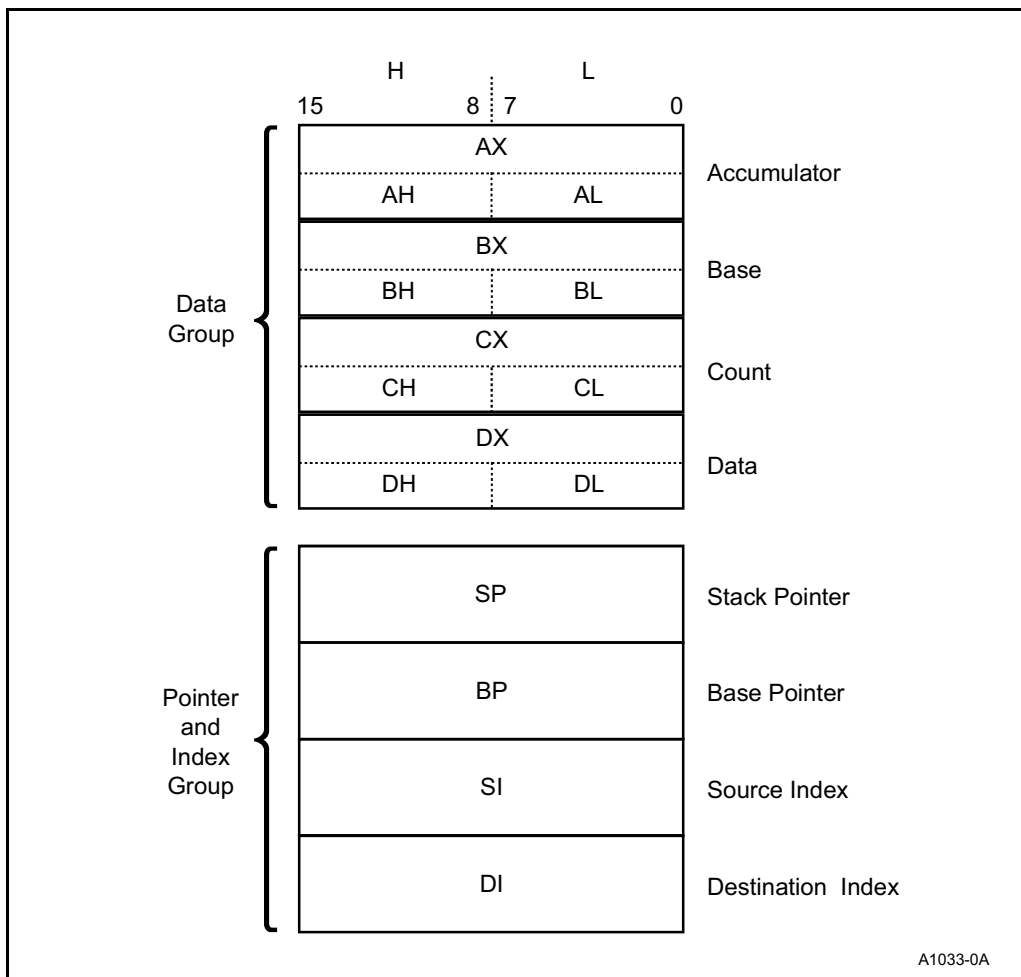


Figure 2-3. General Registers





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

**Table 2-1. Implicit Use of General Registers**

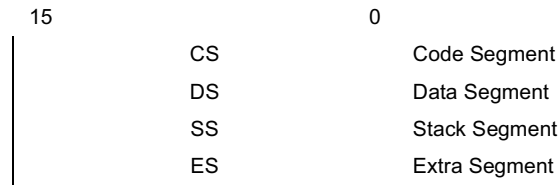
| Register | Operations  |
|----------|---|
| AX       | Word Multiply, Word Divide, Word I/O                                |
| AL       | Byte Multiply, Byte Divide, Byte I/O, Translate, Decimal Arithmetic |
| AH       | Byte Multiply, Byte Divide  |
| BX       | Translate   |
| CX       | String Operations,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

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

#### 2.1.4 Segment Registers

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





**Figure 2-4. Segment Registers**

**2.1.5 Instruction Pointer**

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

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



### 2.1.6 Flags

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

- If the Auxiliary Flag (AF) is set, there has been a carry out from the low nibble into the high nibble or a borrow from the high nibble into the low nibble of an 8-bit quantity (low-order byte of a 16-bit quantity). This flag is used by decimal arithmetic instructions.
- If the Carry Flag (CF) is set, there has been a carry out of or a borrow into the high-order bit of the instruction result (8- or 16-bit). This flag is used by instructions that add or subtract multibyte numbers. Rotate instructions can also isolate a bit in memory or a register by placing it in the Carry Flag.
- If the Overflow Flag (OF) is set, an arithmetic overflow has occurred. A significant digit has been lost because the size of the result exceeded the capacity of its destination location. An Interrupt On Overflow instruction is available that will generate an interrupt in this situation.
- If the Sign Flag (SF) is set, the high-order bit of the result is a 1. Since negative binary numbers are represented, SF indicates the sign of the result (0 = positive, 1 = negative).
- If the Parity Flag (PF) is set, the result has even parity, an even number of 1 bits. This flag can be used to check for data transmission errors.
- If the Zero Flag (ZF) is set, the result of the operation is zero.

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

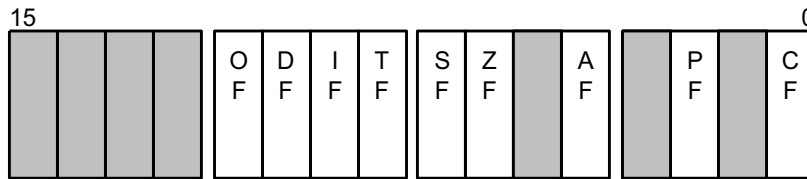
- Setting the Direction Flag (DF) causes string operations to auto-decrement. Strings are processed from high address to low address (or “right to left”). Clearing DF causes string operations to auto-increment. Strings are processed from low address to high address (or “left to right”).
- Setting the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

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

OVERVIEW OF THE 80C186 FAMILY ARCHITECTURE



**Register Name:** Processor Status Word  
**Register Mnemonic:** PSW (FLAGS)  
**Register Function:** Posts CPU status information.

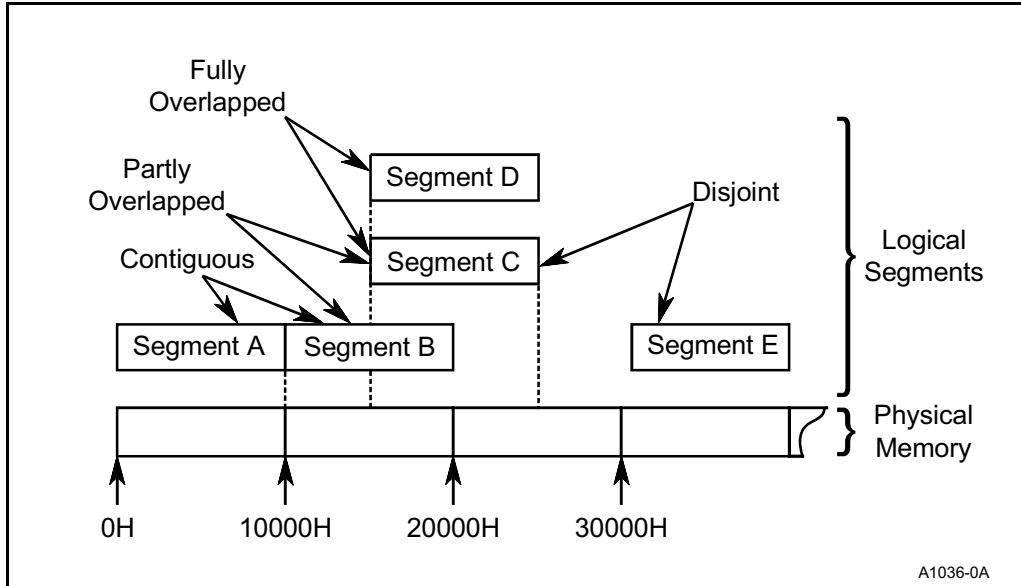


A1035-0A

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

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

Figure 2-5. Processor Status Word



**Figure 2-6. Segment Locations in Physical Memory**

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

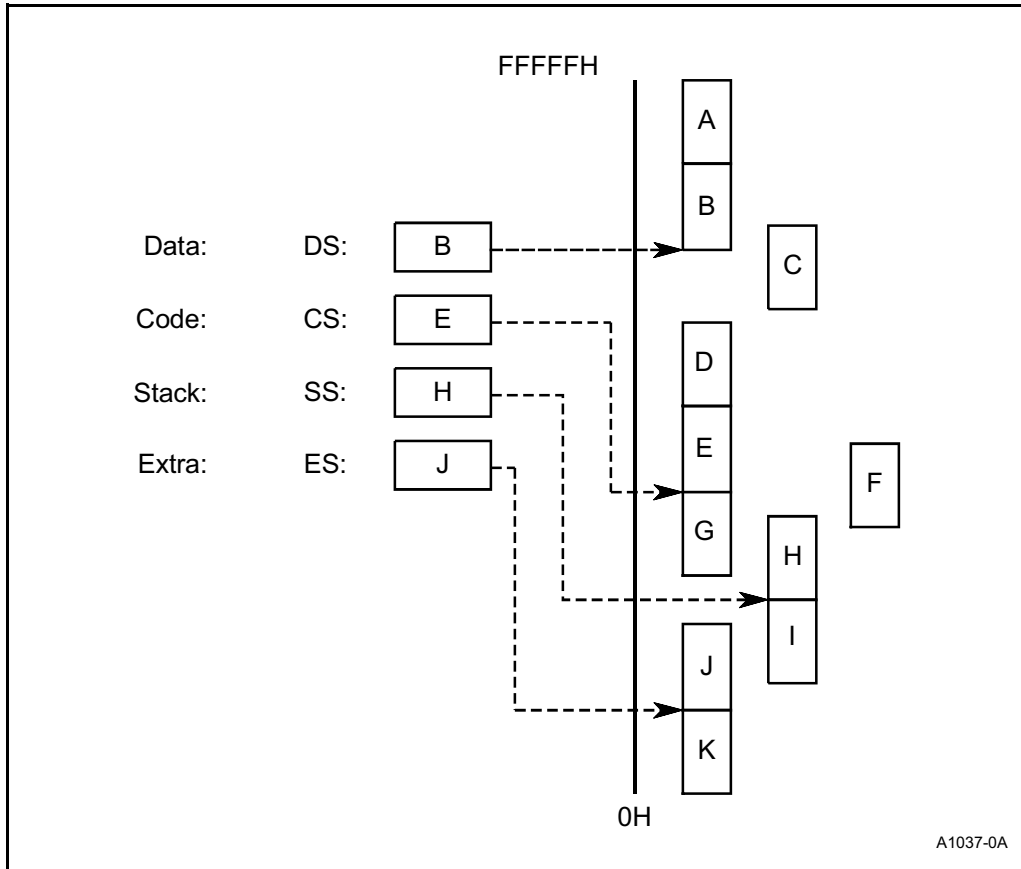
### 2.1.8 Logical Addresses

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

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







**Figure 2-7. Currently Addressable Segments**

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

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

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



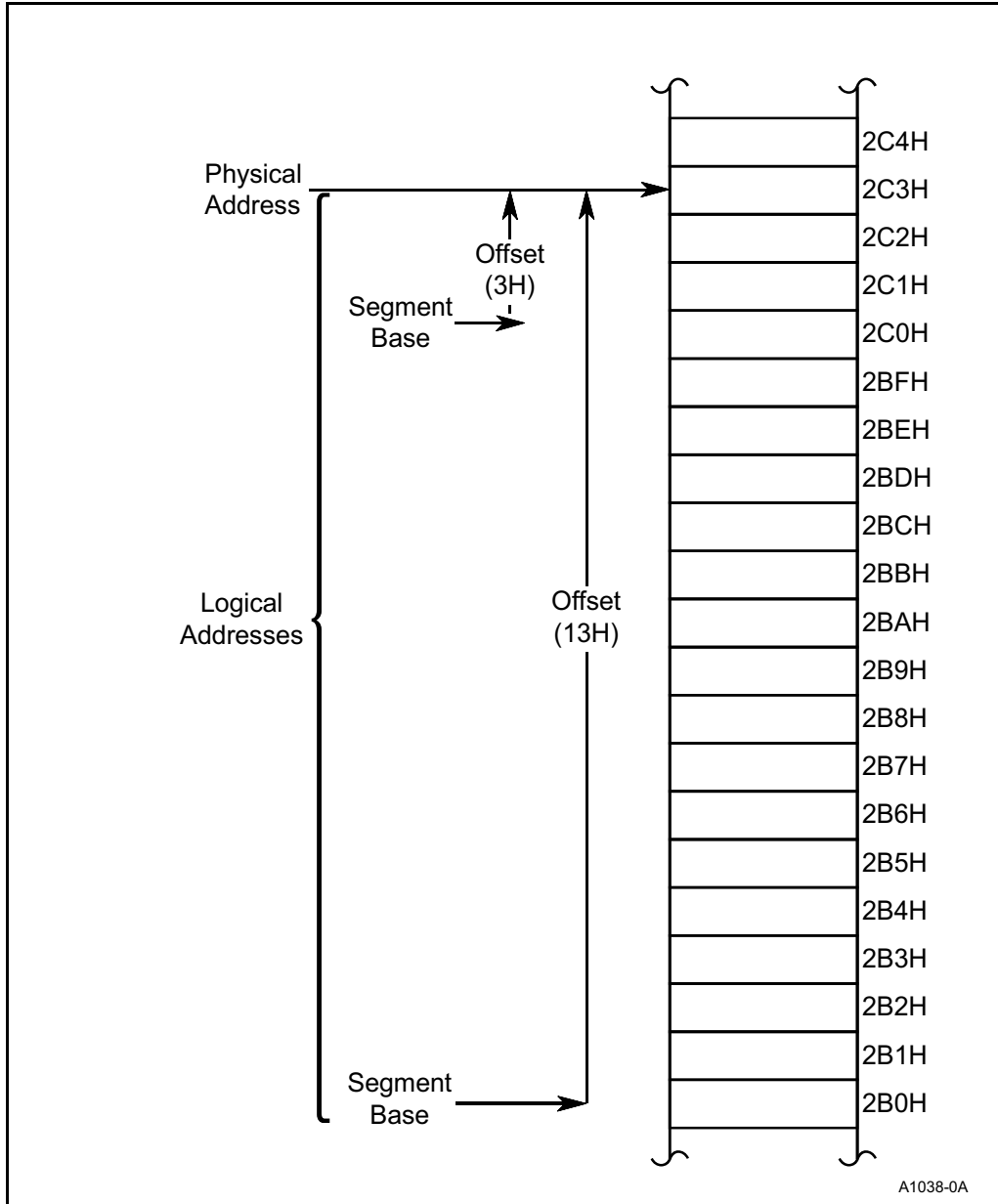


Figure 2-8. Logical and Physical Address



**Table 2-2. Logical Address Sources**

| Type of Memory Reference    | Default Segment Base | Alternate Segment Base | Offset            |
|-----------------------------|----------------------|------------------------|-------------------|
| Instruction Fetch           | CS                   | NONE                   | IP                |
| Stack Operation             | SS                   | NONE                   | SP                |
| Variable (except following) | DS                   | CS, ES, SS             | Effective Address |
| String Source               | DS                   | CS, ES, SS             | SI                |
| String Destination          | ES                   | NONE                   | DI                |
| BP Used as Base Register    | SS                   | CS, DS, ES             | Effective Address |

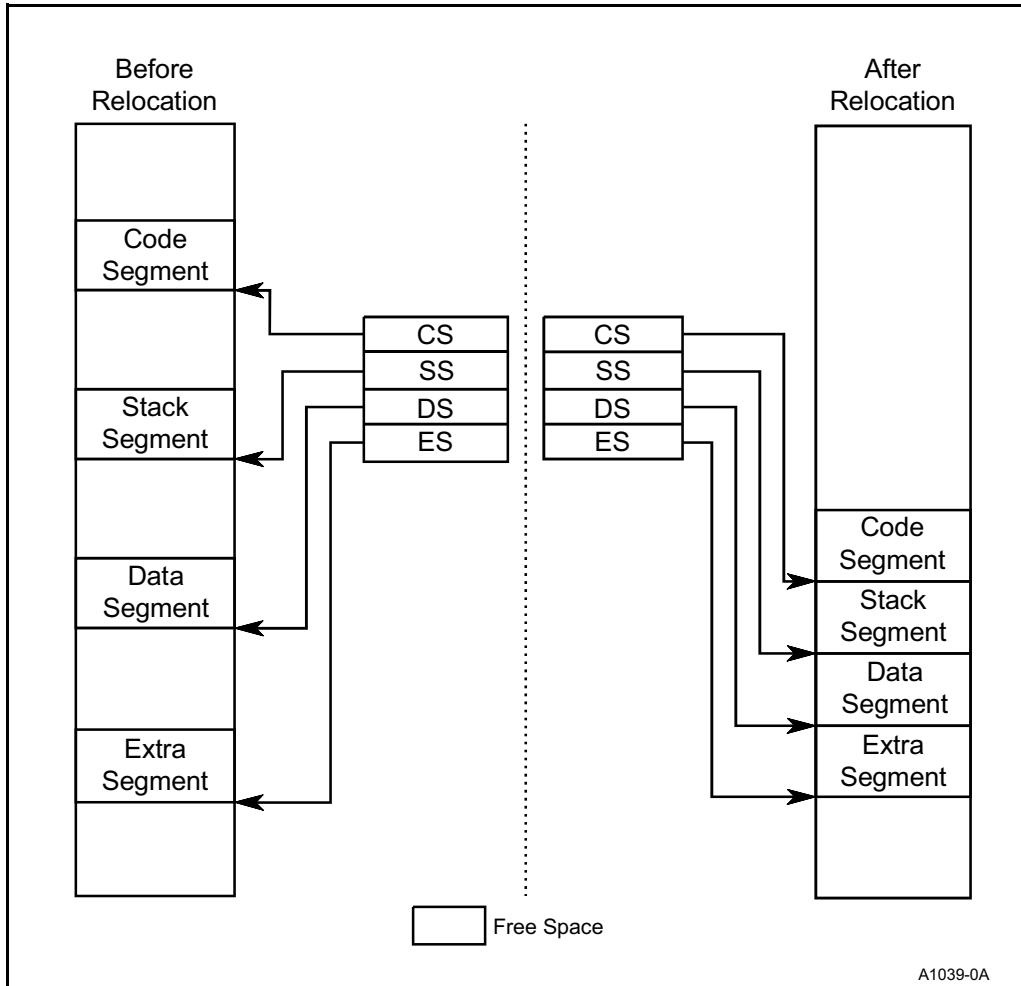
Instructions are always fetched from the current code segment. The IP register contains the instruction offset from the beginning of the segment. Stack instructions always operate on the current stack segment. The Stack Pointer (SP) register contains the offset of the top of the stack from the base of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

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

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

### 2.1.9 Dynamically Relocatable Code

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



**Figure 2-9. Dynamic Code Relocation**

To be dynamically relocatable, a program must not load or alter its segment registers and must not transfer directly to a location outside the current code segment. All program offsets must be relative to the segment registers. This allows the program to be moved anywhere in memory, provided that the segment registers are updated to point to the new base addresses.





### 2.1.10 Stack Implementation

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

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

### 2.1.11 Reserved Memory and I/O Space

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

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



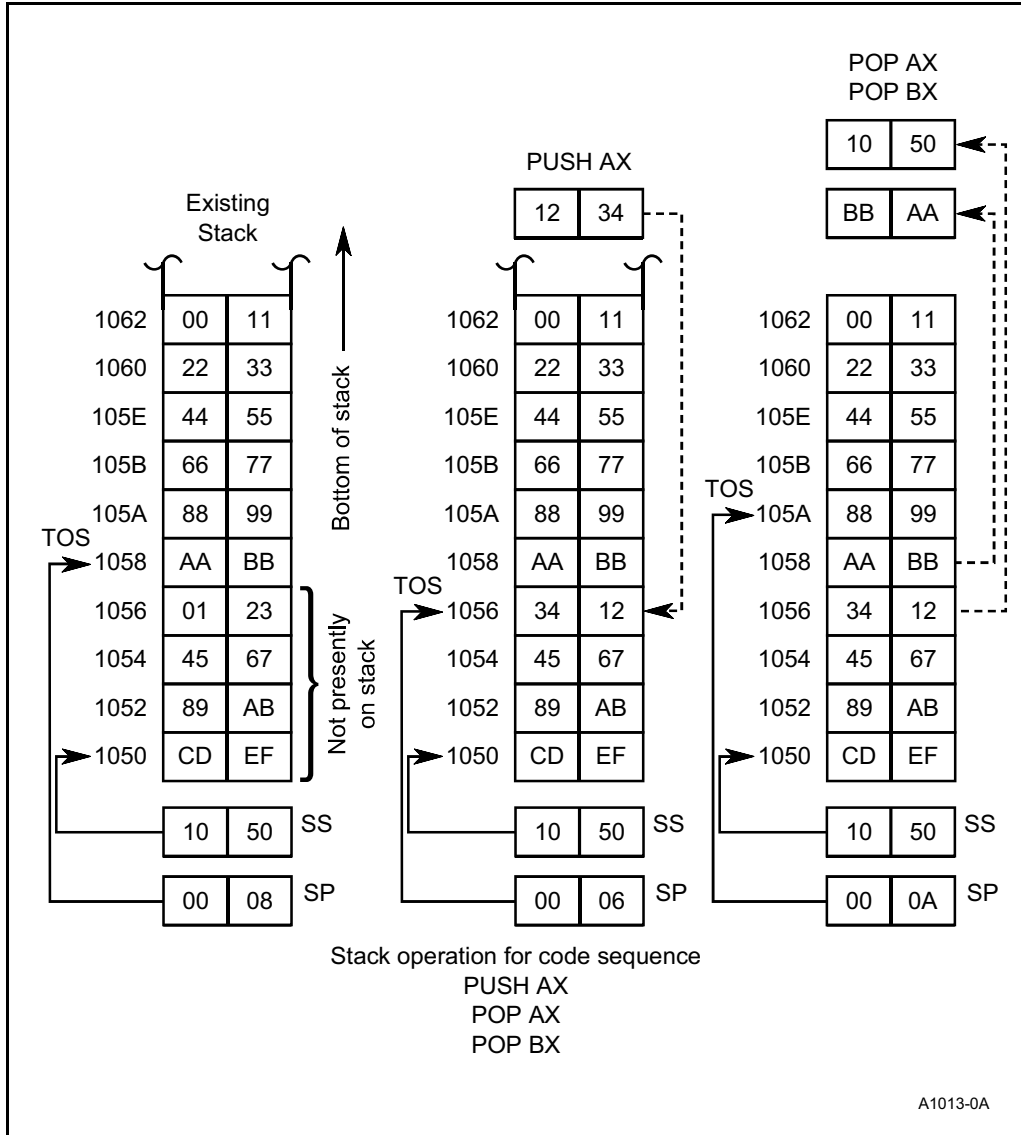


Figure 2-10. Stack Operation



## 2.2 SOFTWARE OVERVIEW

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

Software for 80C186 core family systems need not be written in assembly language. The processor provides direct hardware support for programs written in the many high-level languages available. The hardware addressing modes provide straightforward implementations of based variables, arrays, arrays of structures and other high-level language data constructs. A powerful set of memory-to-memory string operations allow efficient character data manipulation. Finally, routines with critical performance requirements can be written in assembly language and linked with high-level code.

### 2.2.1 Instruction Set

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

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

The two levels of instruction sets address two requirements: efficiency and simplicity. Approximately 300 forms of machine-level instructions make very efficient use of storage. For example, the machine instruction that increments a memory operand is three or four bytes long because the address of the operand must be encoded in the instruction. Incrementing a register, however, requires less information, so the instruction can be shorter. The 80C186 Core family has eight single-byte machine-level instructions that increment different 16-bit registers.

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



### 2.2.1.1 Data Transfer Instructions

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

**Table 2-3. Data Transfer Instructions**

| <b>General-Purpose</b>                |                             |
|---------------------------------------|-----------------------------|
| 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              |
| <b>Input/Output</b>                   |                             |
| IN                                    | Input byte or word          |
| OUT                                   | Output byte or word         |
| <b>Address Object and Stack Frame</b> |                             |
| LEA                                   | Load effective address      |
| LDS                                   | Load pointer using DS       |
| LES                                   | Load pointer using ES       |
| ENTER                                 | Build stack frame           |
| LEAVE                                 | Tear down stack frame       |
| <b>Flag Transfer</b>                  |                             |
| LAHF                                  | Load AH register from flags |
| SAHF                                  | Store AH register in flags  |
| PUSHF                                 | Push flags from stack       |
| POPF                                  | Pop flags off stack         |

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







## OVERVIEW OF THE 80C186 FAMILY ARCHITECTURE

Table 2-5 shows the interpretations of various bit patterns according to number type. Binary numbers can be 8 or 16 bits long. Decimal numbers are stored in bytes, two digits per byte for packed decimal and one digit per byte for unpacked decimal. The processor assumes that the operands in arithmetic instructions contain data that represents valid numbers for that instruction. Invalid data may produce unpredictable results. The Execution Unit analyzes the results of arithmetic instructions and adjusts status flags accordingly.

**Table 2-4. Arithmetic Instructions**

| <b>Addition</b>       |                                   |
|-----------------------|-----------------------------------|
| 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       |
| <b>Subtraction</b>    |                                   |
| 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    |
| <b>Multiplication</b> |                                   |
| MUL                   | Multiply byte or word unsigned    |
| IMUL                  | Integer multiply byte or word     |
| AAM                   | ASCII adjust for multiplication   |
| <b>Division</b>       |                                   |
| DIV                   | Divide byte or word unsigned      |
| IDIV                  | Integer divide byte or word       |
| AAD                   | ASCII adjust for division         |
| CBW                   | Convert byte to word              |
| CWD                   | Convert word to double-word       |



**Table 2-5. Arithmetic Interpretation of 8-Bit Numbers**

| Hex | Bit Pattern     | Unsigned Binary | Signed Binary | Unpacked Decimal | Packed Decimal |
|-----|-----------------|-----------------|---------------|------------------|----------------|
| 07  | 0 0 0 0 0 1 1 1 | 7               | +7            | 7                | 7              |
| 89  | 1 0 0 0 1 0 0 1 | 137             | -119          | invalid          | 89             |
| C5  | 1 1 0 0 0 1 0 1 | 197             | -59           | invalid          | invalid        |

**2.2.1.3 Bit Manipulation Instructions**

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

**Table 2-6. Bit Manipulation Instructions**

| Logicals |  |
|----------|--|
| NOT      | "Not" byte or word                         |
| AND      | "And" byte or word                         |
| OR       | "Inclusive or" byte or word                |
| XOR      | "Exclusive or" byte or word                |
| TEST     | "Test" byte or word                        |
| Shifts   |  |
| SHL/SAL  | Shift logical/arithmetic left byte or word |
| SHR      | Shift logical right byte or word           |
| SAR      | Shift arithmetic right byte or word        |
| Rotates  |  |
| ROL      | Rotate left byte or word                   |
| ROR      | Rotate right byte or word                  |
| RCL      | Rotate through carry left byte or word     |
| RCR      | Rotate through carry right byte or word    |

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

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



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

#### 2.2.1.4 String Instructions

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

**Table 2-7. String Instructions**

|            |                                 |
|------------|---------------------------------|
| REP        | Repeat                          |
| REPE/REPZ  | Repeat while equal/zero         |
| REPNE/REPZ | Repeat while not equal/not zero |
| MOVS/MOVSW | Move byte string/word string    |
| MOVS       | Move byte or word string        |
| INS        | Input byte or word string       |
| OUTS       | Output byte or word string      |
| CMPS       | Compare byte or word string     |
| SCAS       | Scan byte or word string        |
| LODS       | Load byte or word string        |
| STOS       | Store byte or word string       |

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





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

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

**Table 2-8. String Instruction Register and Flag Use**

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

### 2.2.1.5 Program Transfer Instructions

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

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



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

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

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





## OVERVIEW OF THE 80C186 FAMILY ARCHITECTURE

**Table 2-9. Program Transfer Instructions**

| <b>Conditional Transfers</b>   |                                    |
|--------------------------------|------------------------------------|
| 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                       |
| <b>Unconditional Transfers</b> |                                    |
| CALL                           | Call procedure                     |
| RET                            | Return from procedure              |
| JMP                            | Jump                               |
| <b>Iteration Control</b>       |                                    |
| LOOP                           | Loop                               |
| LOOPE/LOOPZ                    | Loop if equal/zero                 |
| LOOPNE/LOOPNZ                  | Loop if not equal/not zero         |
| JCXZ                           | Jump if register CX=0              |
| <b>Interrupts</b>              |                                    |
| INT                            | Interrupt                          |
| INTO                           | Interrupt if overflow              |
| BOUND                          | Interrupt if out of array bounds   |
| IRET                           | Interrupt return                   |



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

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

**Table 2-10. Interpretation of Conditional Transfers**

| Mnemonic | Condition Tested      | “Jump if...”               |
|----------|-----------------------|----------------------------|
| JA/JNBE  | (CF or ZF)=0          | above/not below nor equal  |
| JAE/JNB  | CF=0                  | above or equal/not below   |
| JB/JNAE  | CF=1                  | below/not above nor equal  |
| JBE/JNA  | (CF or ZF)=1          | below or equal/not above   |
| JC       | CF=1                  | carry                      |
| JE/JZ    | ZF=1                  | equal/zero                 |
| JG/JNLE  | ((SF xor OF) or ZF)=0 | greater/not less nor equal |
| JGE/JNL  | (SF xor OF)=0         | greater or equal/not less  |
| JL/JNGE  | (SF xor OF)=1         | less/not greater nor equal |
| JLE/JNG  | ((SF xor OF) or ZF)=1 | less or equal/not greater  |
| JNC      | CF=0                  | not carry                  |
| JNE/JNZ  | ZF=0                  | not equal/not zero         |
| JNO      | OF=0                  | not overflow               |
| JNP/JPO  | PF=0                  | not parity/parity odd      |
| JNS      | SF=0                  | not sign                   |
| JO       | OF=1                  | overflow                   |
| JP/JPE   | PF=1                  | parity/parity equal        |
| JS       | SF=1                  | sign                       |

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





**2.2.1.6 Processor Control Instructions**

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

**Table 2-11. Processor Control Instructions**

| Flag Operations          |  |
|--------------------------|--|
| STC                      | Set Carry flag                               |
| CLC                      | Clear Carry flag                             |
| CMC                      | Complement Carry flag                        |
| STD                      | Set Direction flag                           |
| CLD                      | Clear Direction flag                         |
| STI                      | Set Interrupt Enable flag                    |
| CLI                      | Clear Interrupt Enable flag                  |
| External Synchronization |  |
| HLT                      | Halt until interrupt or reset                |
| WAIT                     | Wait for $\overline{\text{TEST}}$ pin active |
| ESC                      | Escape to external processor                 |
| LOCK                     | Lock bus during next instruction             |
| No Operation             |  |
| NOP                      | No operation                                 |

**2.2.2 Addressing Modes**

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

**2.2.2.1 Register and Immediate Operand Addressing Modes**

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



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

#### 2.2.2.2 Memory Addressing Modes

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

The offset that the Execution Unit calculates for memory operand is called the operand effective address (EA). This address is an unsigned 16-bit number that expresses the operand's location, in bytes, from the beginning of the segment in which it resides. The Execution Unit can calculate the effective address in several ways. Information encoded in the second byte of the instruction tells the Execution Unit how to calculate the effective address of each memory operand. A compiler or assembler derives this information from the instruction written by the programmer. Assembly language programmers have access to all addressing modes.

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



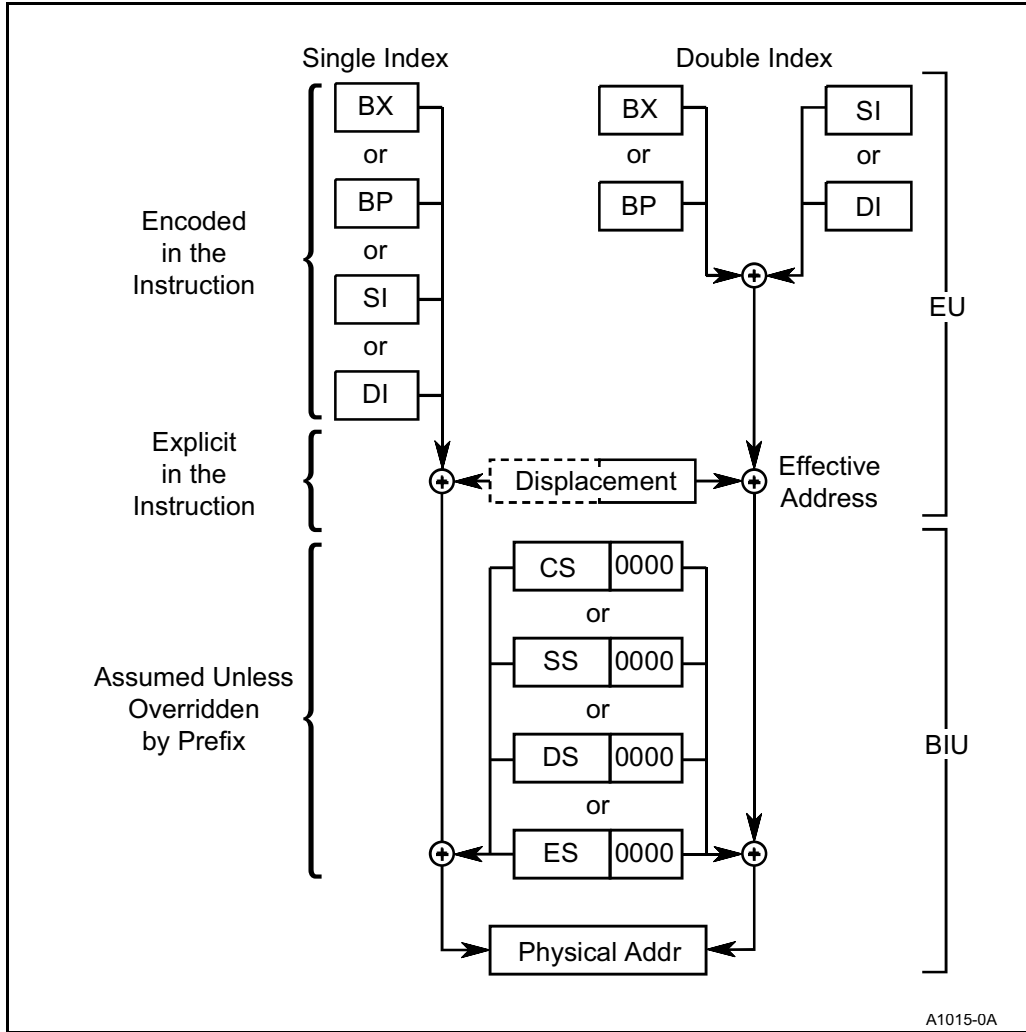


Figure 2-12. Memory Address Computation

The displacement is an 8- or 16-bit number contained in the instruction. The displacement generally is derived from the position of the operand. The programmer can modify this value or explicitly specify the displacement.

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

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

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

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

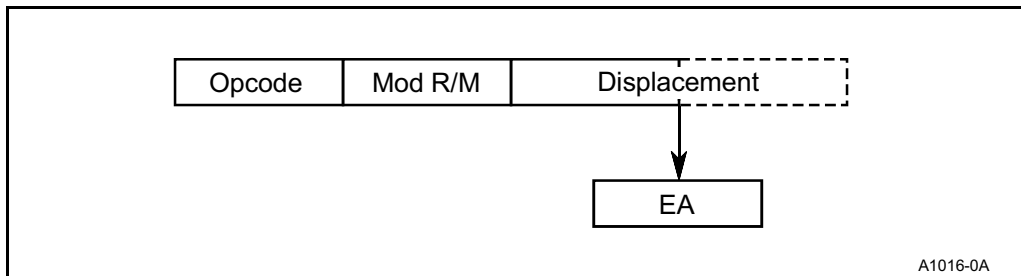
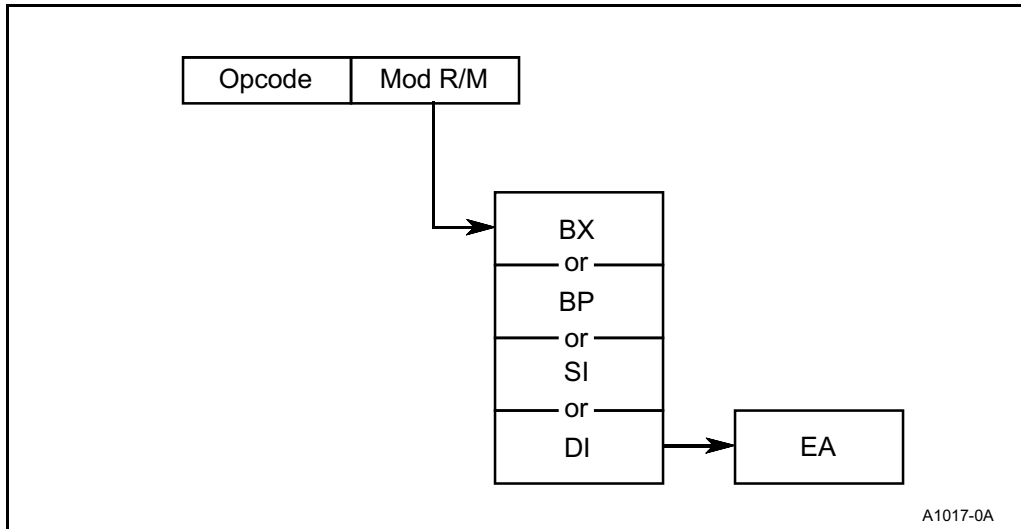
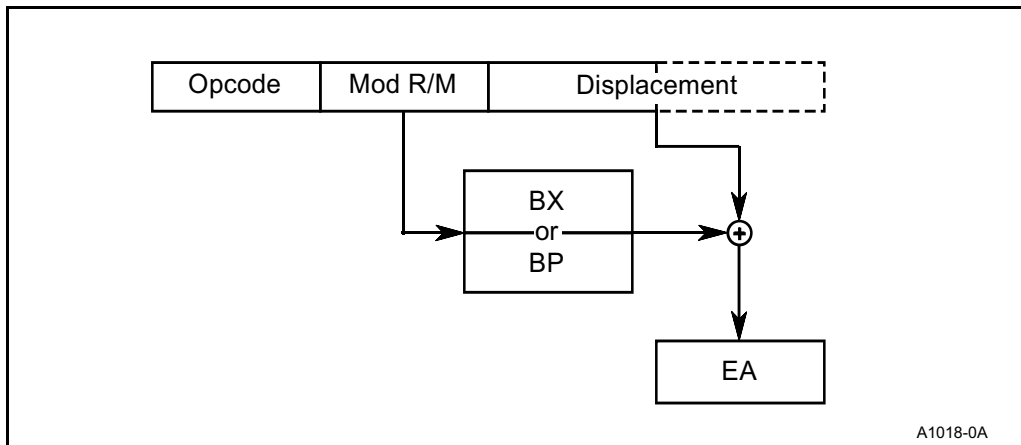


Figure 2-13. Direct Addressing





**Figure 2-14. Register Indirect Addressing**



**Figure 2-15. Based Addressing**

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



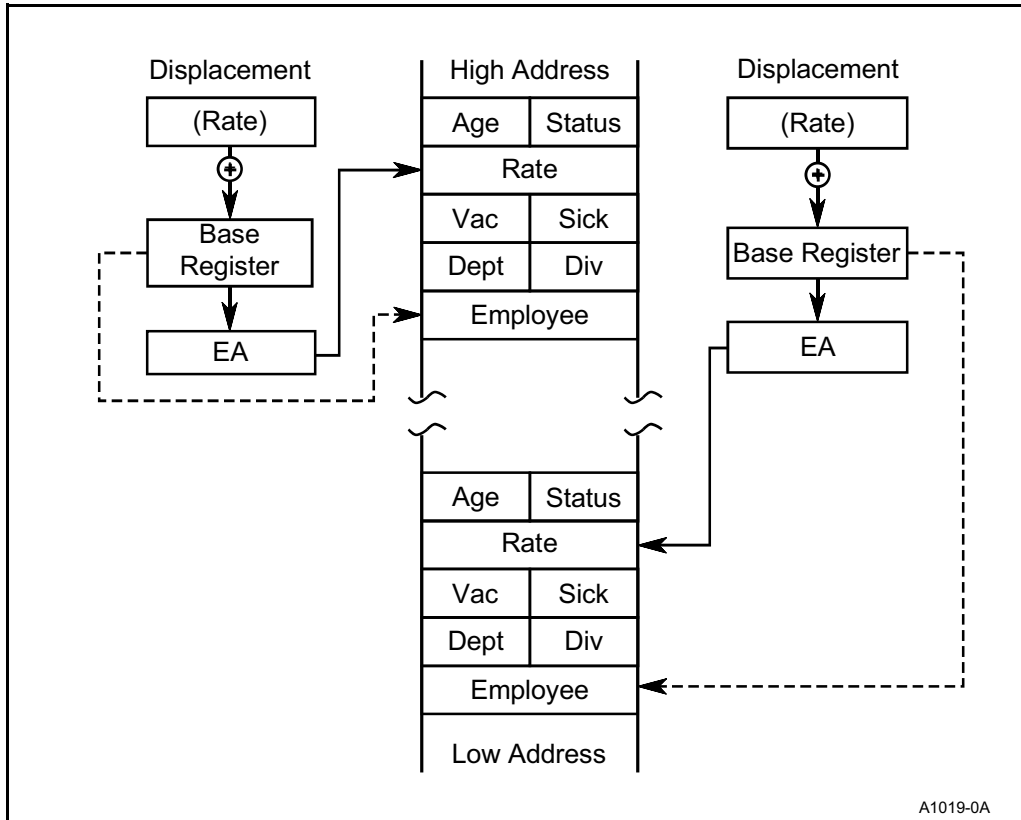


Figure 2-16. Accessing a Structure with Based Addressing

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



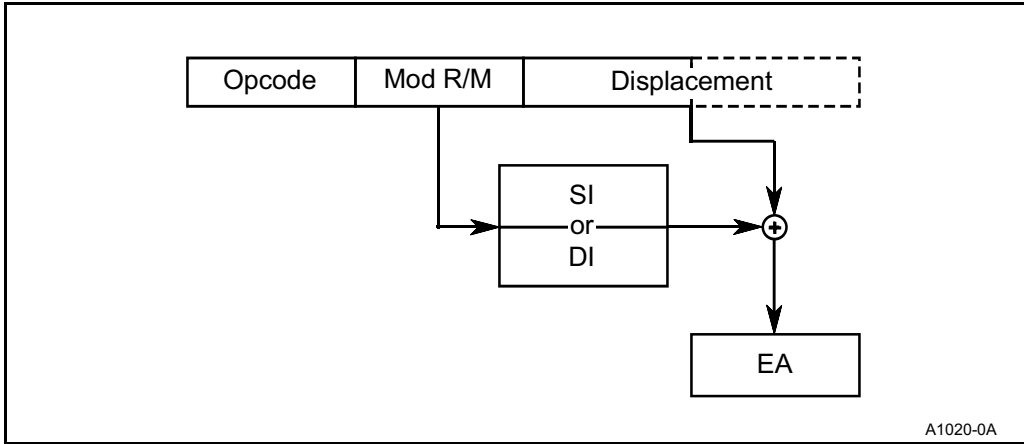


Figure 2-17. Indexed Addressing

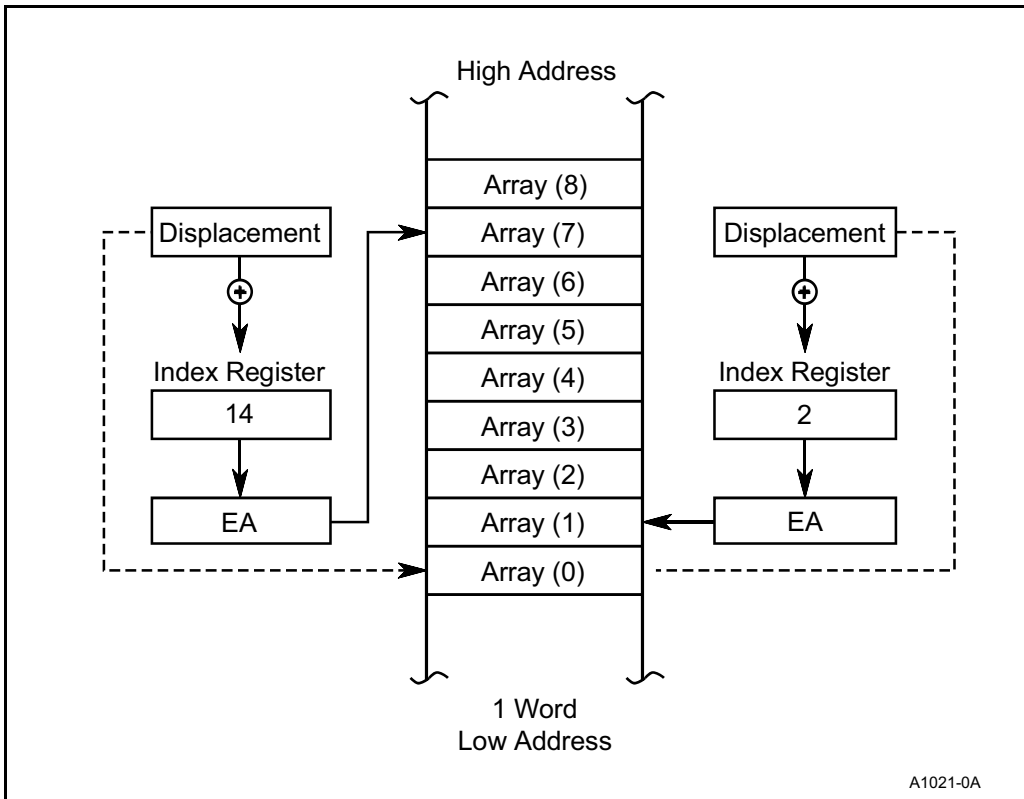
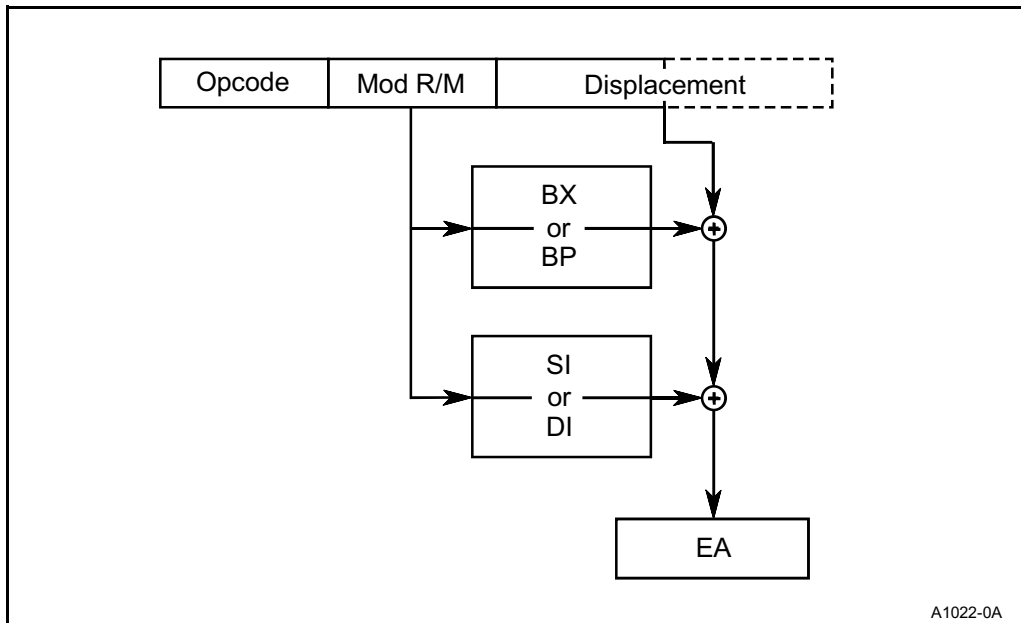


Figure 2-18. Accessing an Array with Indexed Addressing

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



**Figure 2-19. Based Index Addressing**

Based index addressing provides a convenient way for a procedure to address an array located on a stack (see Figure 2-20). The BP register can contain the offset of a reference point on the stack. This is typically the top of the stack after the procedure has saved registers and allocated local storage. The offset of the beginning of the array from the reference point can be expressed by a displacement value. The index register can be used to access individual array elements. Arrays contained in structures and matrices (two-dimensional arrays) can also be accessed with based indexed addressing.

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



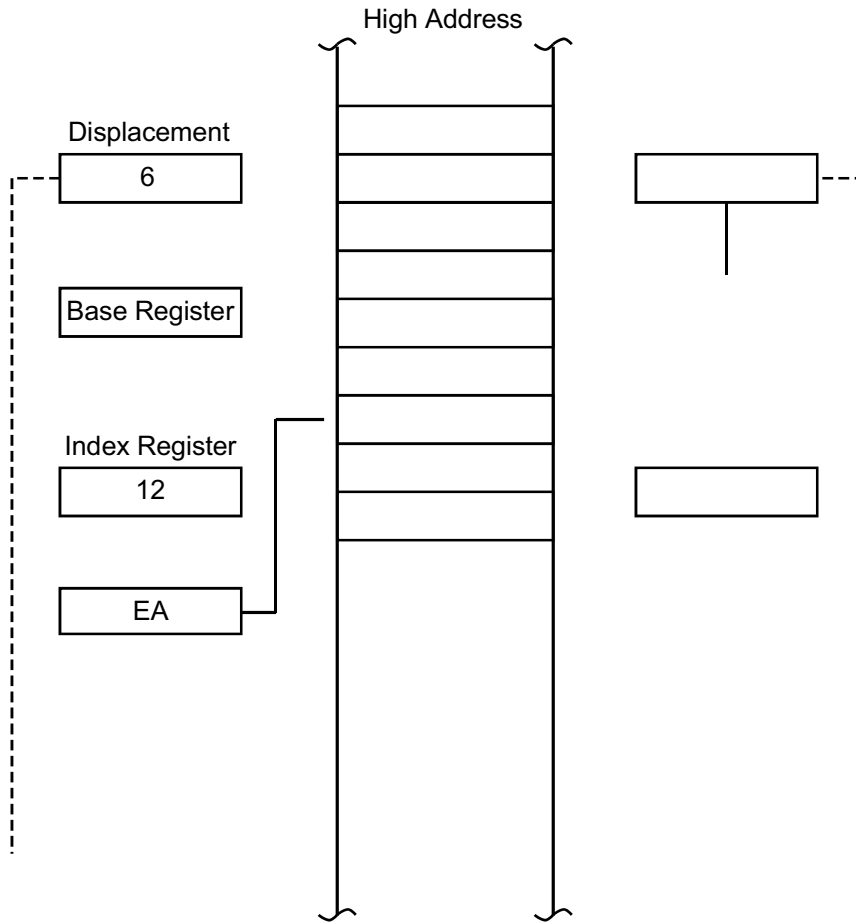


Figure 2-20. Accessing a Stacked Array with Based Index Addressing

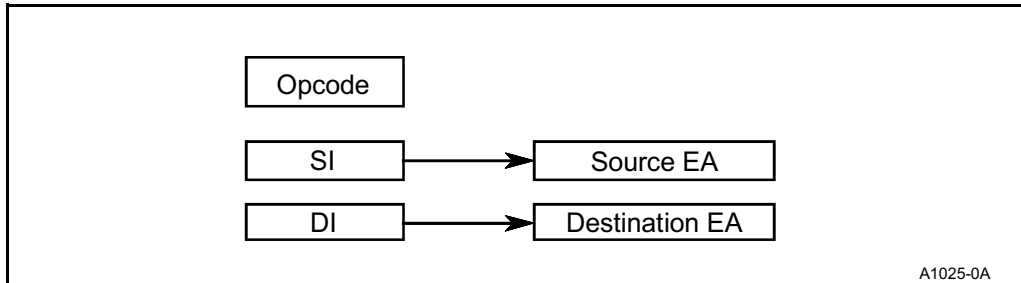


Figure 2-21. String Operand

2.2.2.3 I/O Port Addressing

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

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

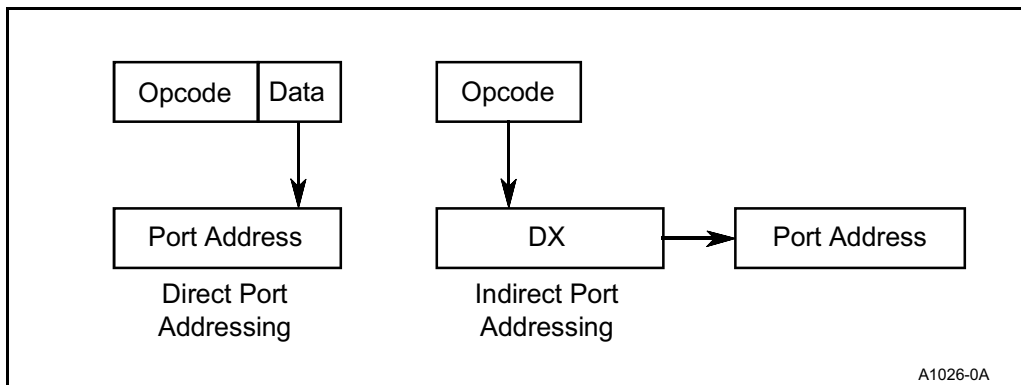


Figure 2-22. I/O Port Addressing





2.2.2.4 Data Types Used in the 80C186 Modular Core Family

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

Table 2-12. Supported Data Types

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



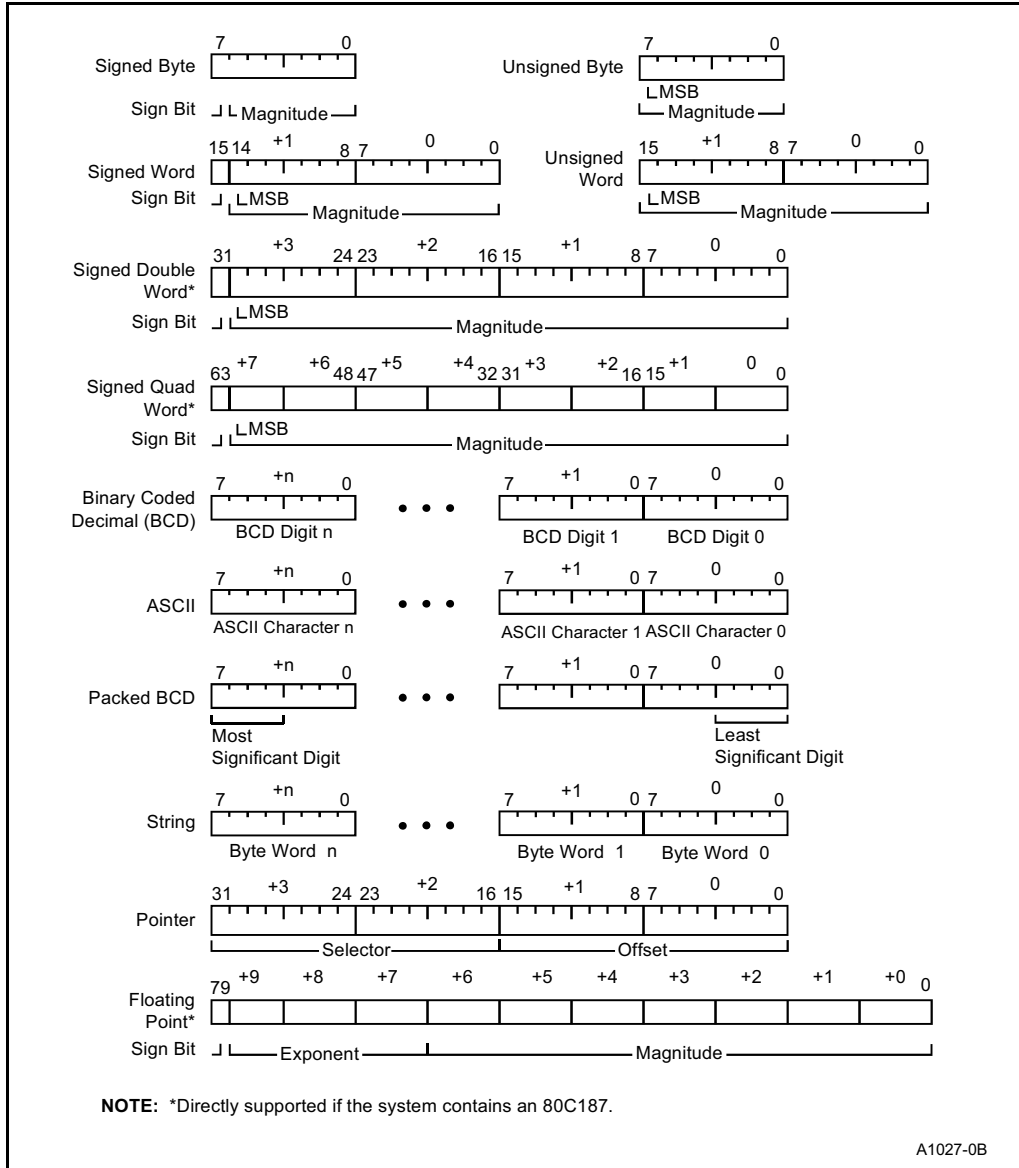


Figure 2-23. 80C186 Modular Core Family Supported Data Types



### 2.3 INTERRUPTS AND EXCEPTION HANDLING

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

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

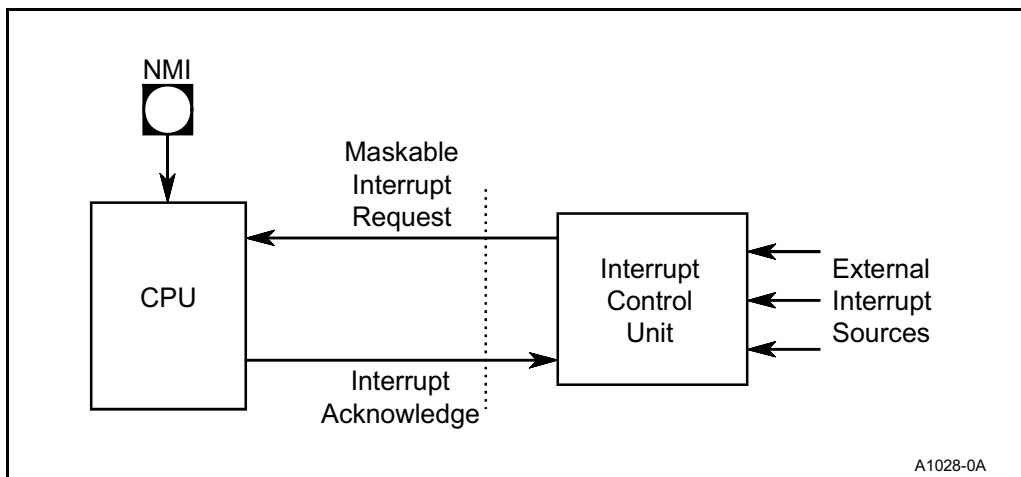
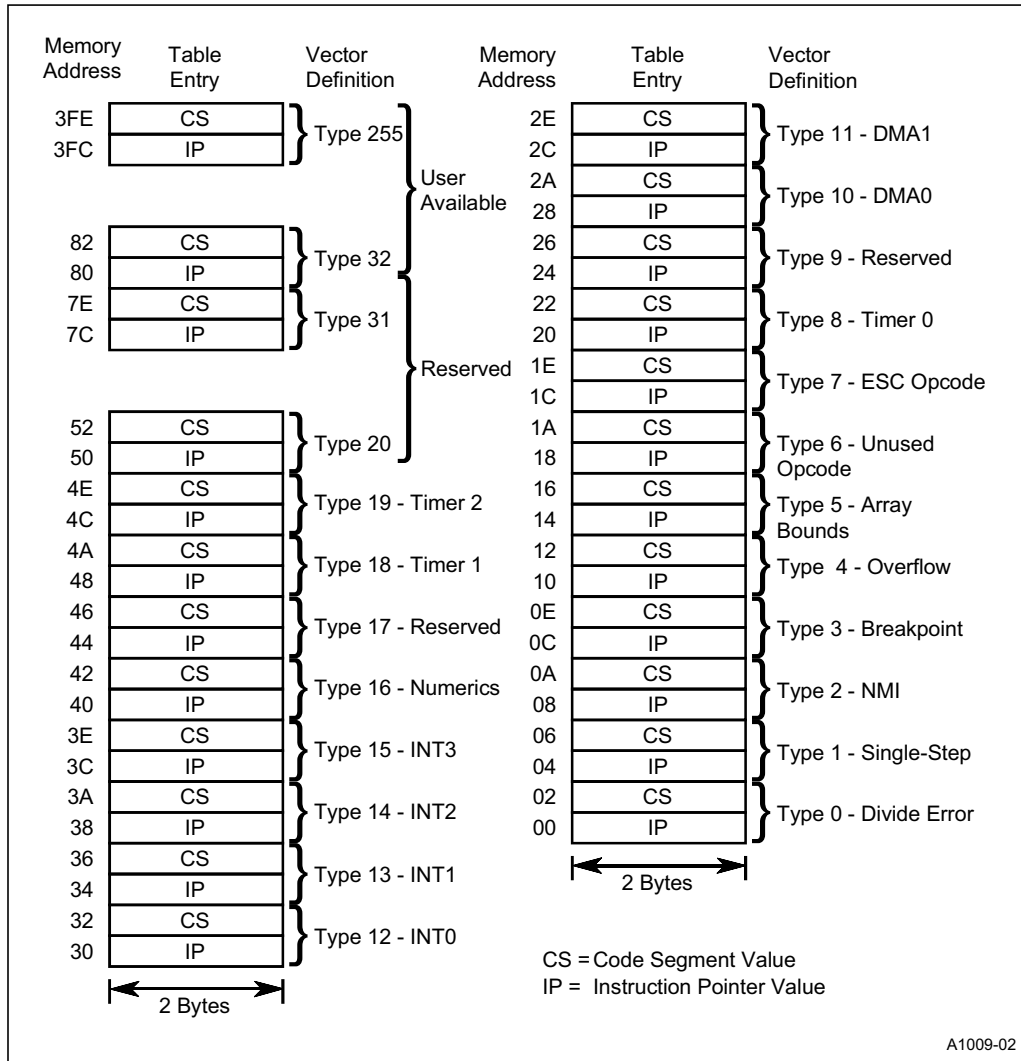


Figure 2-24. Interrupt Control Unit

#### 2.3.1 Interrupt/Exception Processing

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





**Figure 2-25. Interrupt Vector Table**

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

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





## OVERVIEW OF THE 80C186 FAMILY ARCHITECTURE

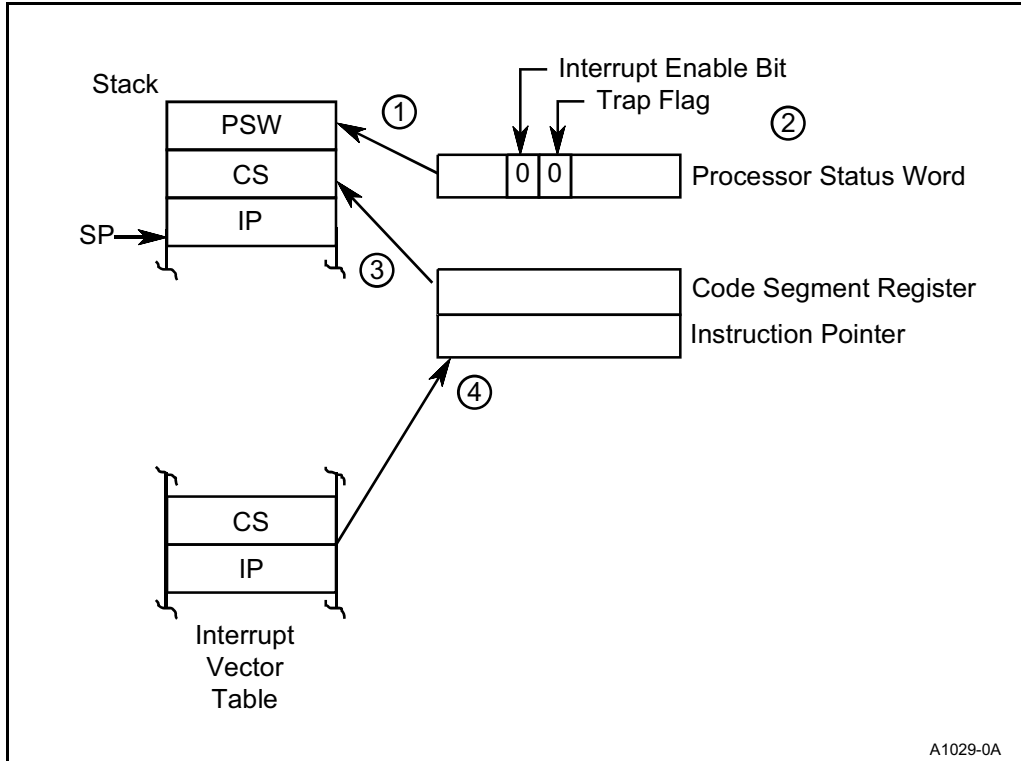


Figure 2-26. Interrupt Sequence

### 2.3.1.1 Non-Maskable Interrupts

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

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



### 2.3.1.2 Maskable Interrupts

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

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

### 2.3.1.3 Exceptions

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

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

#### Divide Error — Type 0

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

#### Single Step — Type 1

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

OVERVIEW OF THE 80C186 FAMILY ARCHITECTURE



### 2.3.2 Software Interrupts

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

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

### 2.3.3 Interrupt Latency

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

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

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

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

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

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

OVERVIEW OF THE 80C186 FAMILY ARCHITECTURE



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

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

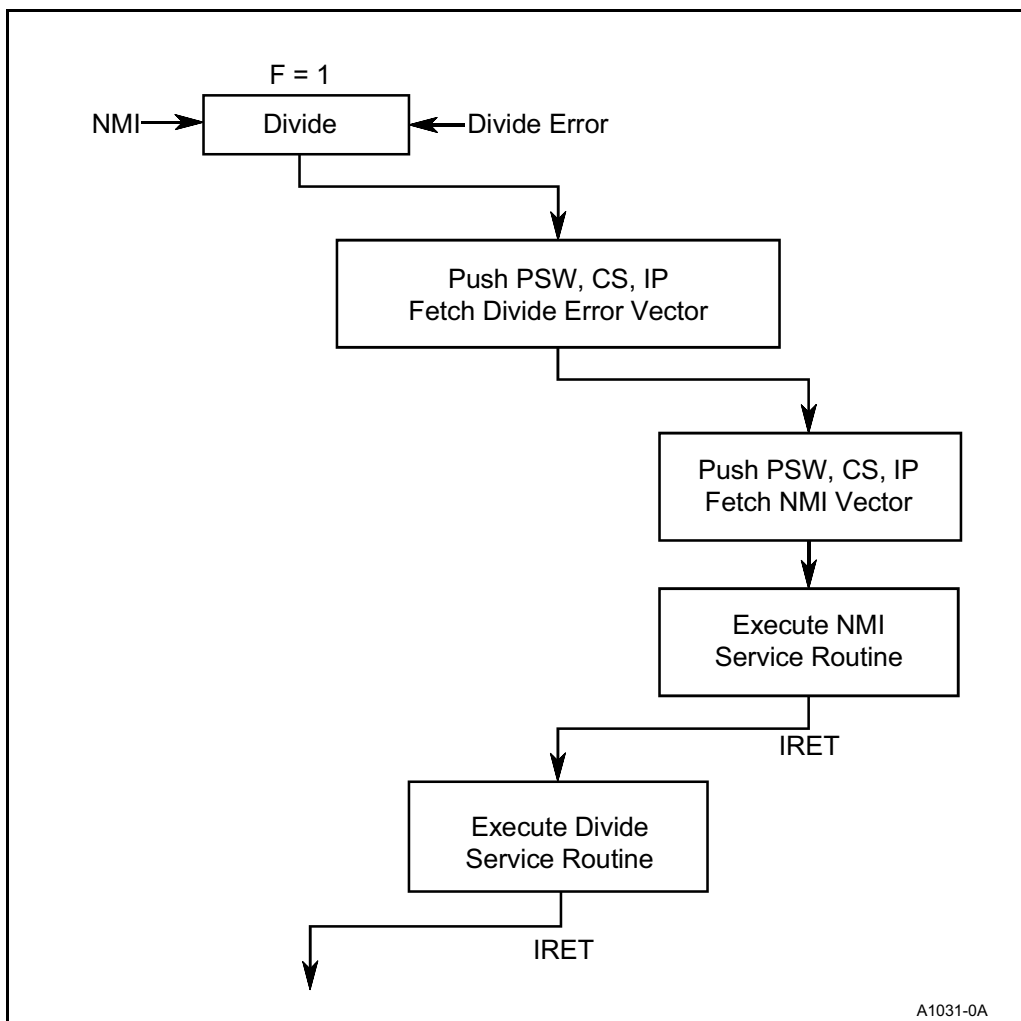
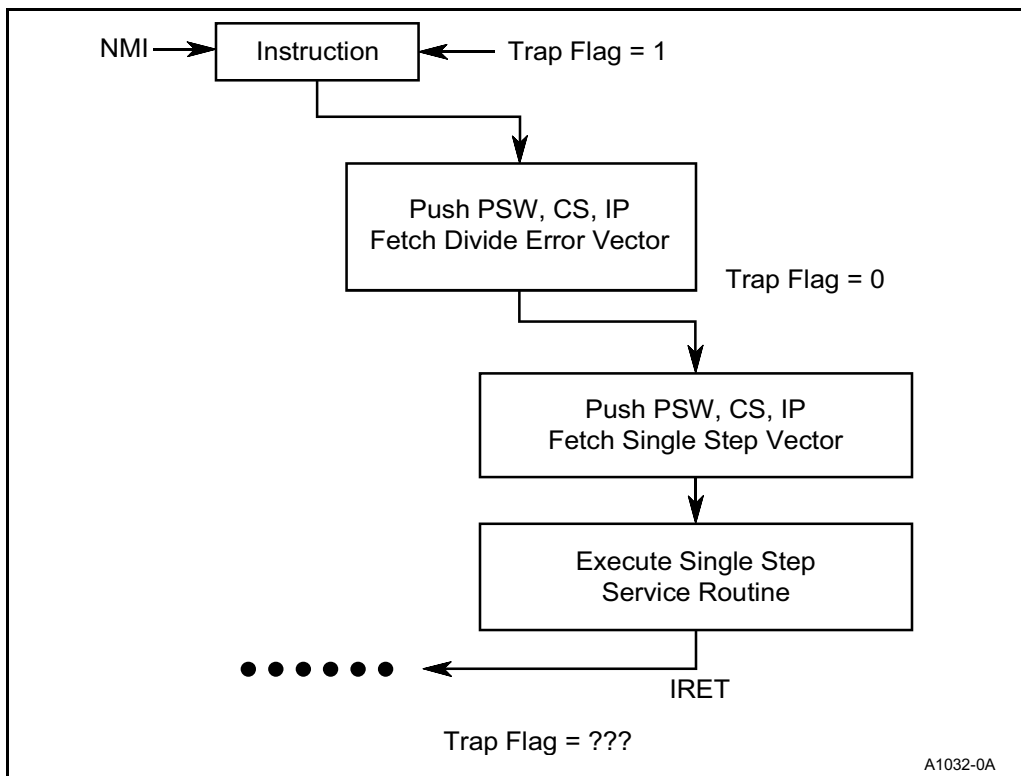


Figure 2-28. Simultaneous NMI and Exception

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

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



**Figure 2-29. Simultaneous NMI and Single Step Interrupts**

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



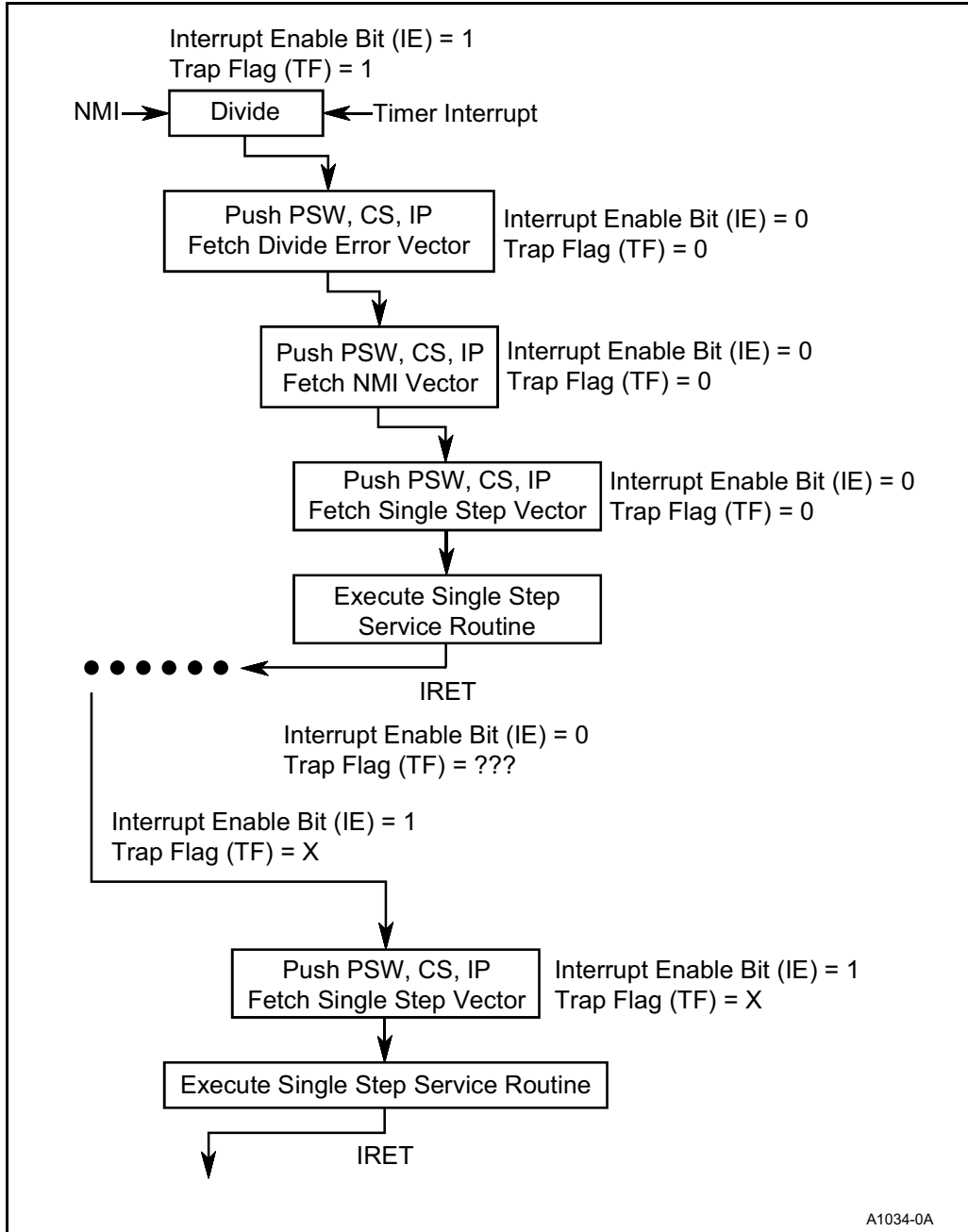


Figure 2-30. Simultaneous NMI, Single Step and Maskable Interrupt







3

## Bus Interface Unit







## CHAPTER 3 BUS INTERFACE UNIT

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

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

### 3.1 MULTIPLEXED ADDRESS AND DATA BUS

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

### 3.2 ADDRESS AND DATA BUS CONCEPTS

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

#### 3.2.1 16-Bit Data Bus

The memory address space on a 16-bit data bus is physically implemented by dividing the address space into two banks of up to 512 Kbytes each (see Figure 3-1). One bank connects to the lower half of the data bus and contains even-addressed bytes ( $A_0=0$ ). The other bank connects to the upper half of the data bus and contains odd-addressed bytes ( $A_0=1$ ). Address lines  $A_{19:1}$  select a specific byte within each bank.  $A_0$  and Byte High Enable ( $BHE$ ) determine whether one bank or both banks participate in the data transfer.



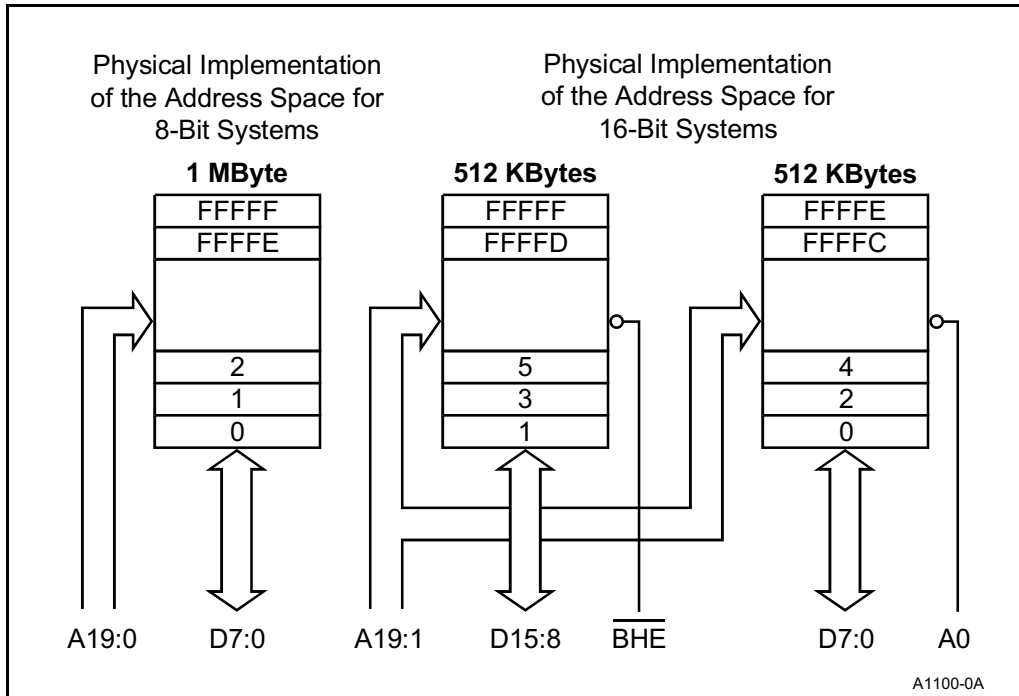


Figure 3-1. Physical Data Bus Models

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). A19:1 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.



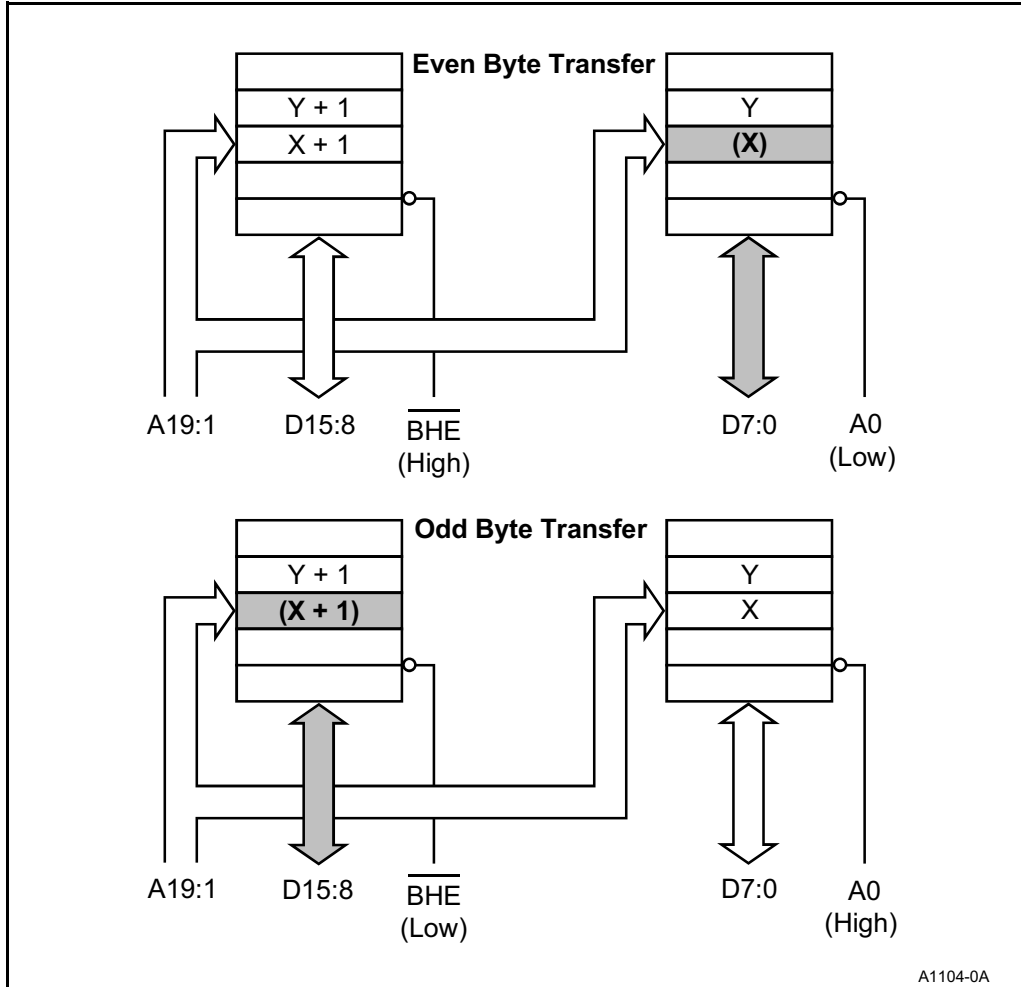
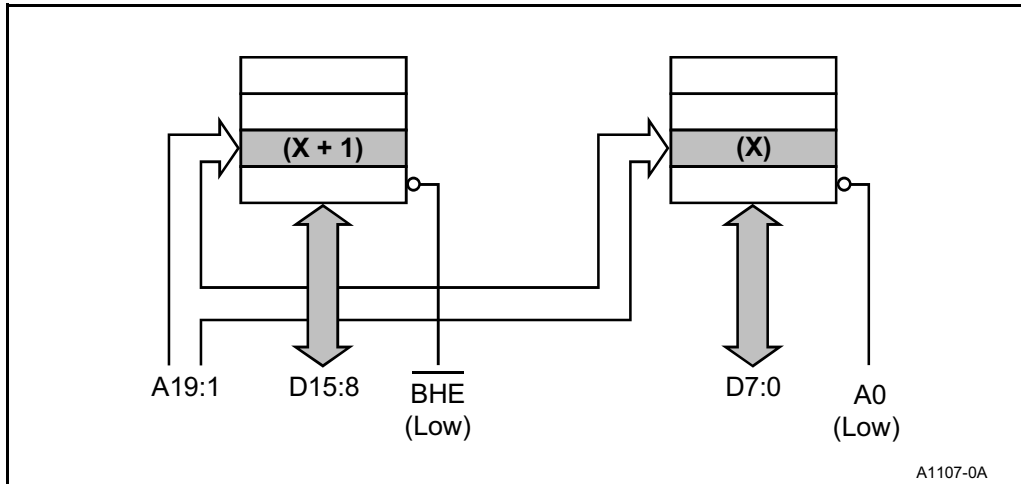


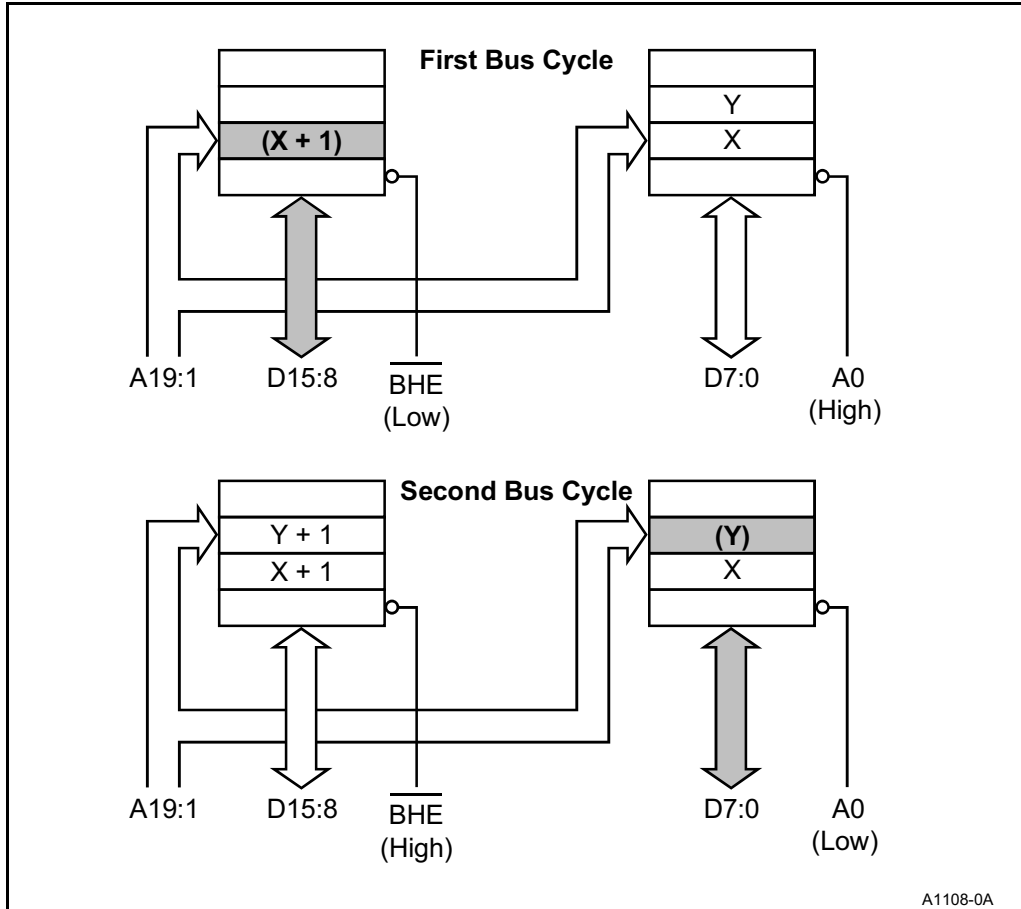
Figure 3-2. 16-Bit Data Bus Byte Transfers





**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, an external device can drive **both halves** of the bus, and the BIU automatically accesses the correct half. During the byte write operation, the BIU drives both halves of the bus. Information on the half of the bus not involved in the transfer is indeterminate. This action requires that the appropriate bank (defined by  $\overline{\text{BHE}}$  or A0 high) be disabled to prevent destroying data.



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

### 3.2.2 8-Bit Data Bus

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

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

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

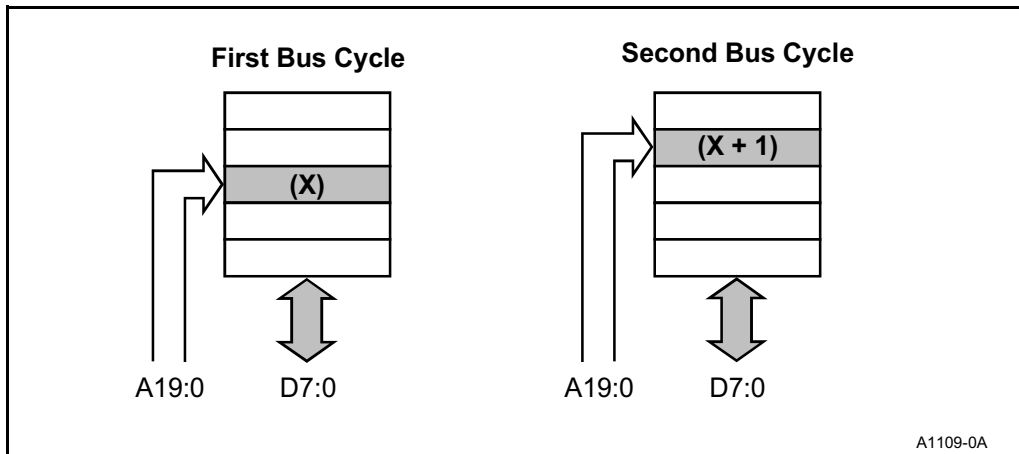


Figure 3-5. 8-Bit Data Bus Word Transfers

### 3.3 MEMORY AND I/O INTERFACES

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

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

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





### 3.3.1 16-Bit Bus Memory and I/O Requirements

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

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

### 3.3.2 8-Bit Bus Memory and I/O Requirements

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

## 3.4 BUS CYCLE OPERATION

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



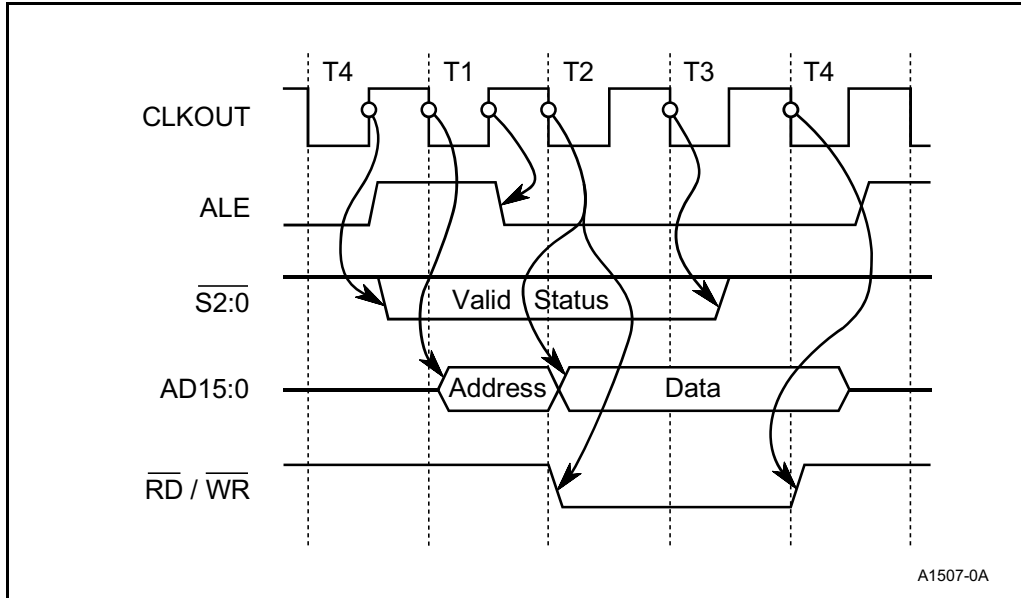


Figure 3-6. Typical Bus Cycle

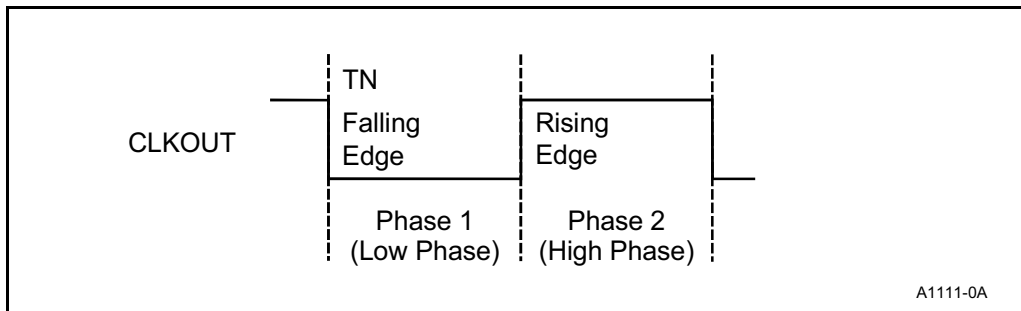


Figure 3-7. T-State Relation to CLKOUT

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

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

- Address/Status Phase
- Data Phase



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

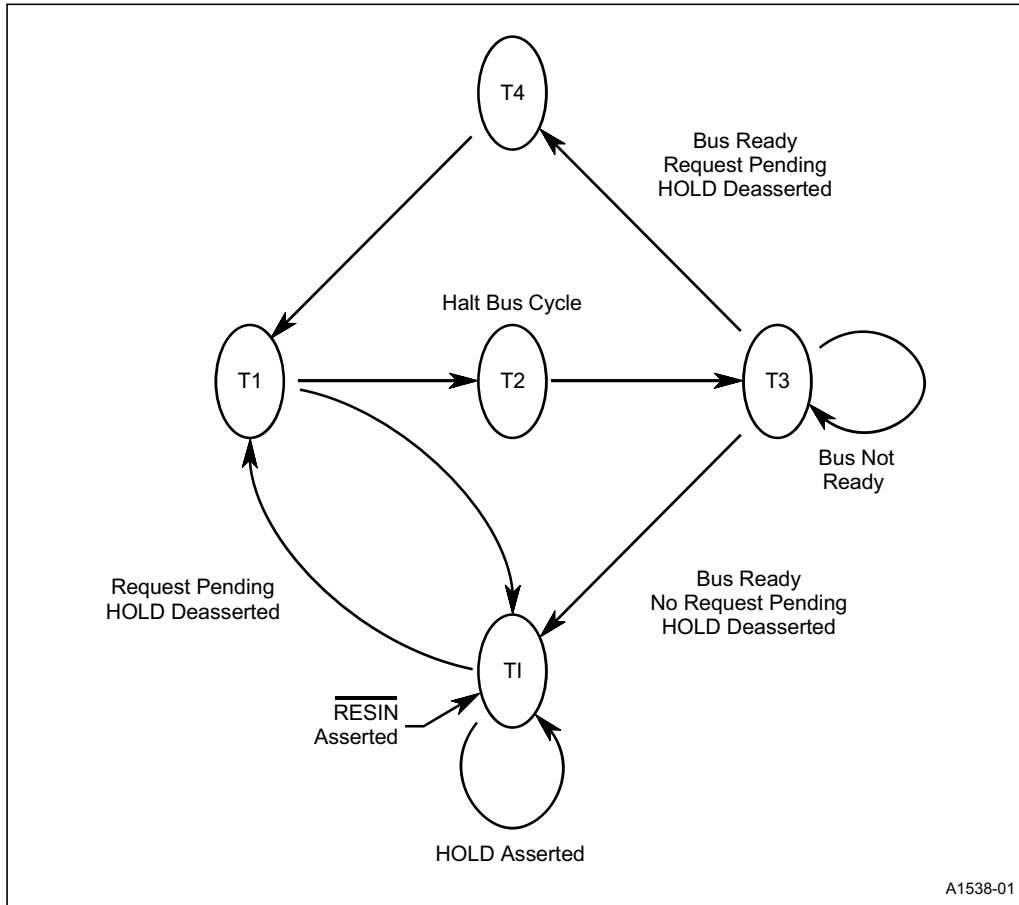


Figure 3-8. BIU State Diagram



**Figure 3-9. T-State and Bus Phases****3.4.1 Address/Status Phase**

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

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

The status signals ( $\overline{S2:0}$ ) define the type of bus cycle (Table 3-1).  $\overline{S2:0}$



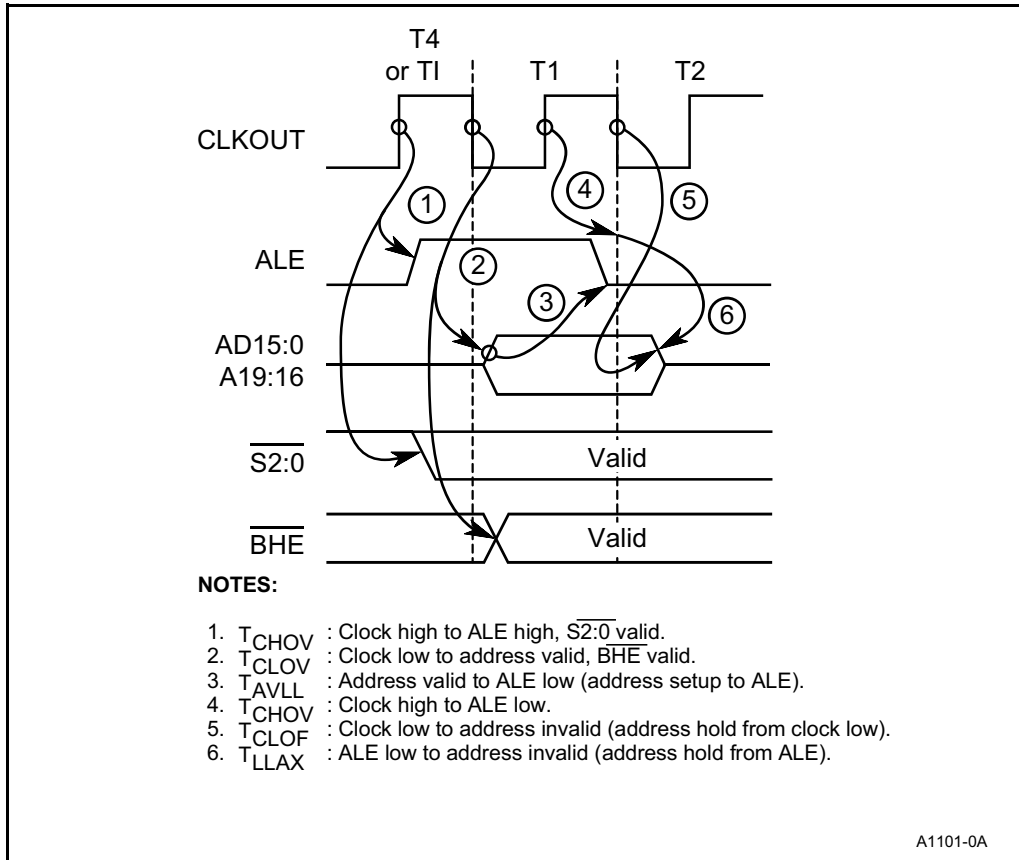


Figure 3-10. Address/Status Phase Signal Relationships

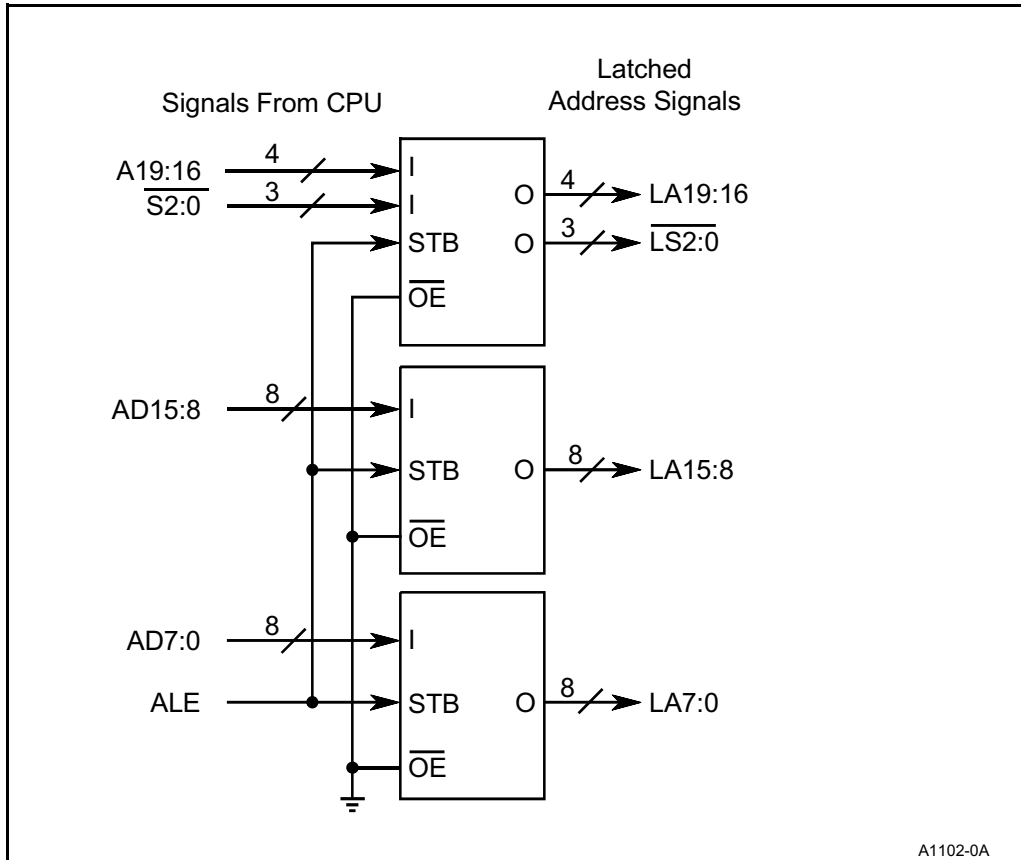


Figure 3-11. Demultiplexing Address Information

Table 3-1. Bus Cycle Types

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



### 3.4.2 Data Phase

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

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

### 3.4.3 Wait States

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

The bus ready inputs (ARDY and SRDY) and the Chip-Select Unit control bus cycle wait states. Only the bus ready inputs are described in this chapter. (See Chapter 6, “Chip-Select Unit,” for additional information.)

Figure 3-14 shows a simplified block diagram of the ARDY and SRDY inputs. Either ARDY or SRDY active signals a bus ready condition; therefore, both pins must be inactive to signal a not-ready condition. Depending on the size and characteristics of the system, ready implementation can take one of two approaches: normally not-ready or normally ready.

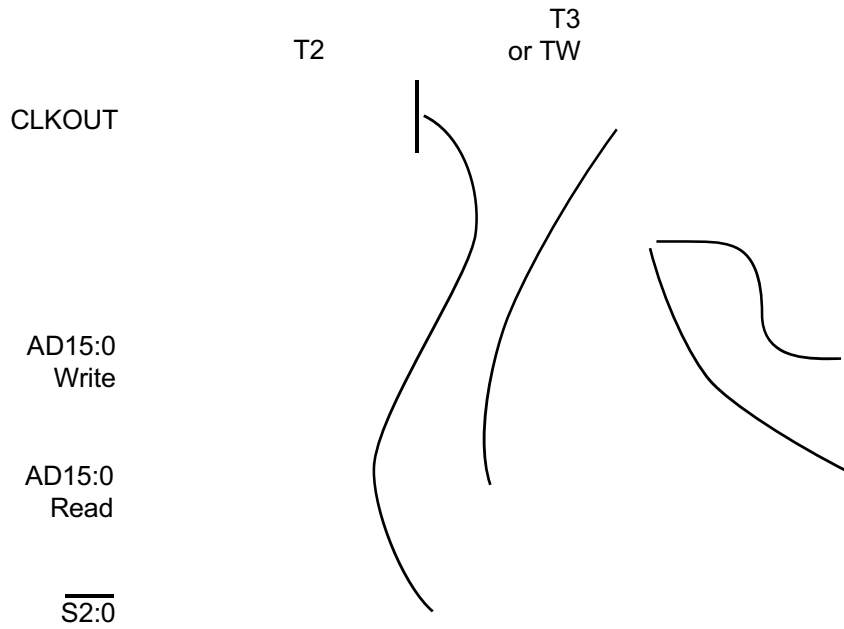


Figure 3-12. Data Phase Signal Relationships





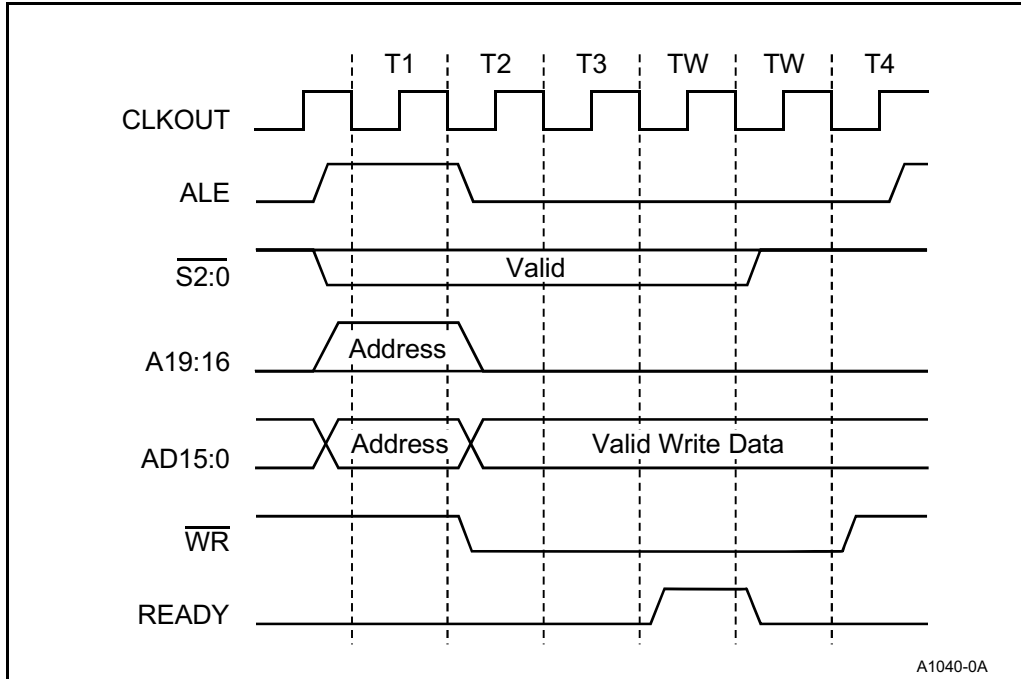


Figure 3-13. Typical Bus Cycle with Wait States

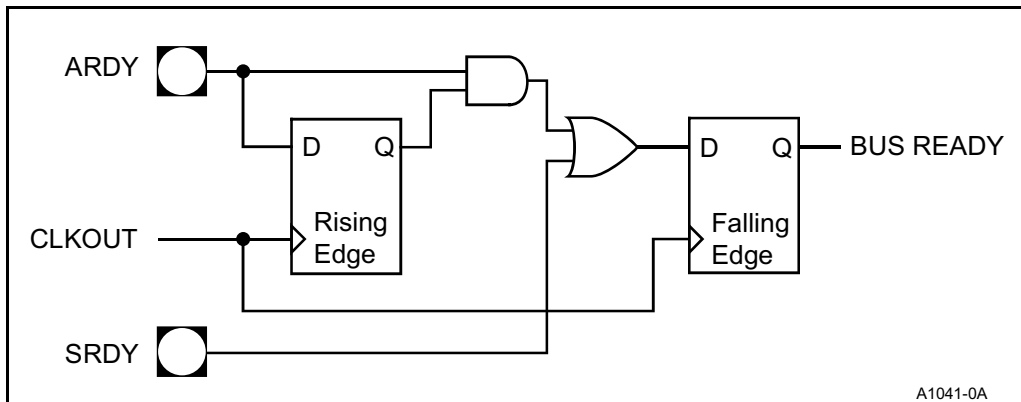


Figure 3-14. ARDY and SRDY Pin Block Diagram

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

The start of a bus cycle clears the wait state module and forces 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, but no delay through the module results in no wait states.

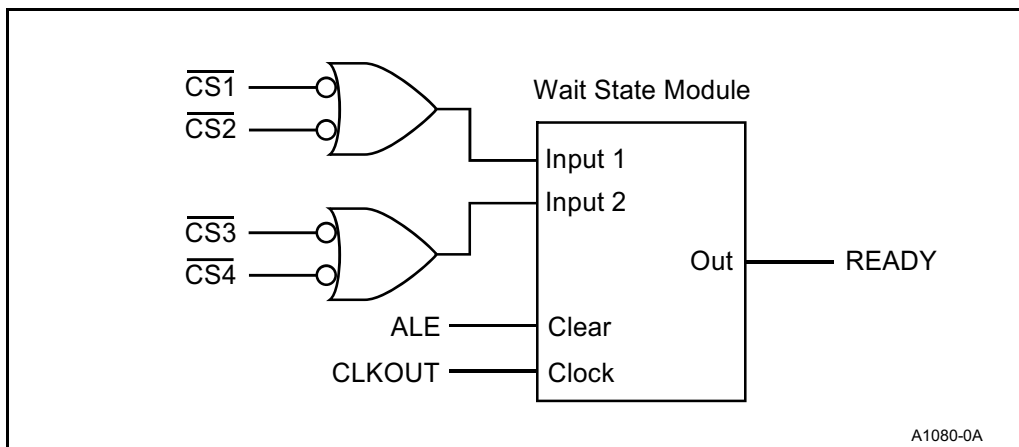
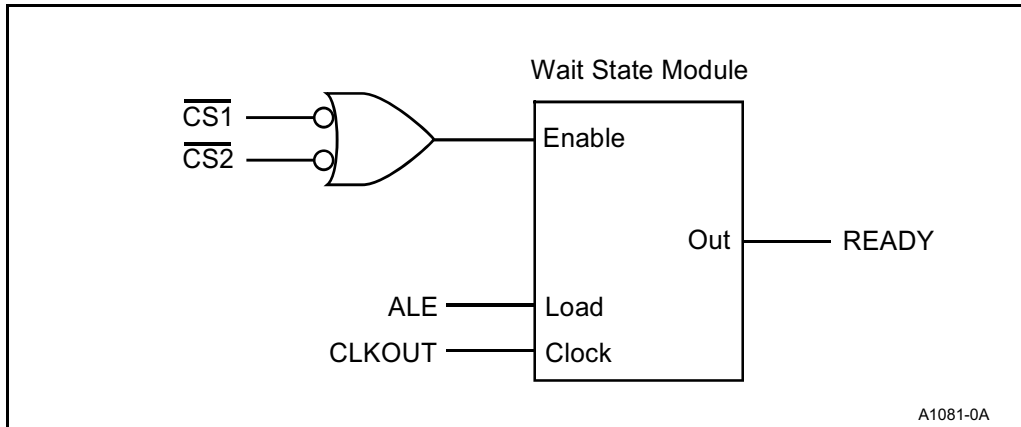


Figure 3-15. Generating a Normally Not-Ready Bus Signal

A normally ready signal remains high at all times except when the selected device needs to signal a not-ready condition. For any bus cycle, only the selected device drives the ready input (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  $\overline{CS2}$  goes low. Should either  $\overline{CS1}$  or  $\overline{CS2}$  go low, zeros are shifted out on 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, but shifting one zero through the module generates two wait states.





**Figure 3-16. Generating a Normally Ready Bus Signal**

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

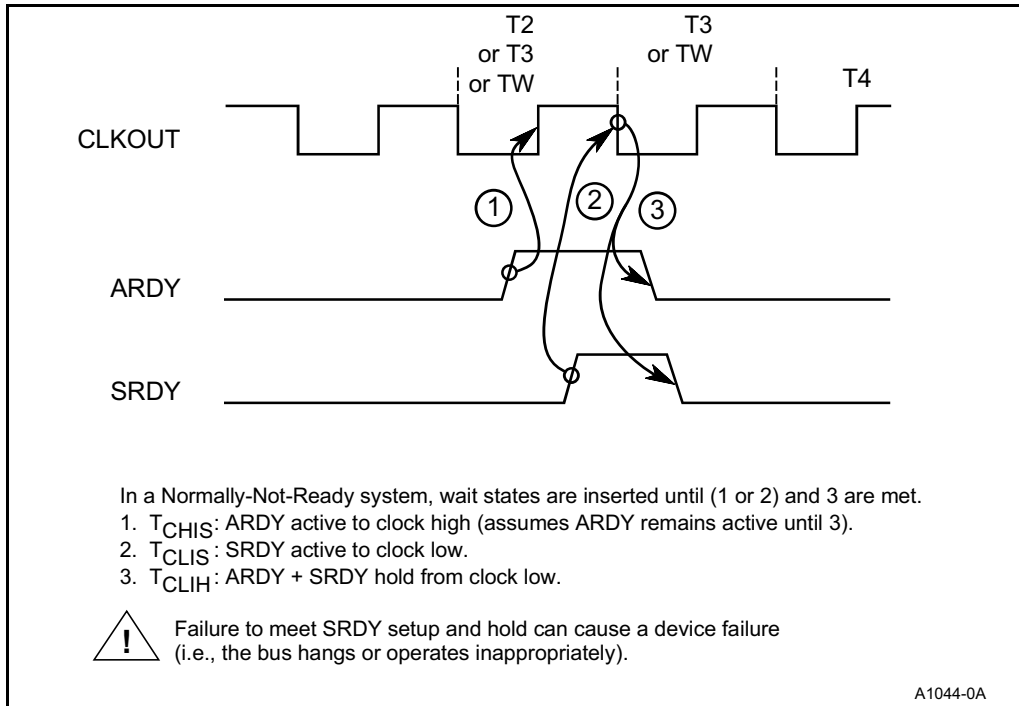
- The bus is **ready** if **both** of these two conditions are true:
  - ARDY is active prior to the phase 2 clock edge, **and**
  - ARDY remains active after the phase 1 clock edge.
- The bus is **not-ready** if **either** of these two conditions is true:
  - ARDY is inactive prior to the phase 2 clock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

A single latch captures the state of the SRDY input (see Figure 3-14 on page 3-15). SRDY must be valid by the phase 1 clock edge. The following items define the requirements of the SRDY input to meet ready or not-ready bus conditions.

- The bus is **ready** if SRDY is active prior to the phase 1 clock edge.
- The bus is **not-ready** if SRDY is inactive prior to the phase 1 clock edge.

A normally not-ready system must generate a valid ARDY input at phase 2 of T2 or a valid SRDY input at phase 1 of T3 to prevent wait states. If it cannot, then running without wait states requires a normally ready system. Figure 3-17 illustrates the timing necessary to prevent wait states in a normally not-ready system. Figure 3-17 also shows how to terminate a bus cycle with wait states in a normally not-ready system.





**Figure 3-17. Normally Not-Ready System Timing**

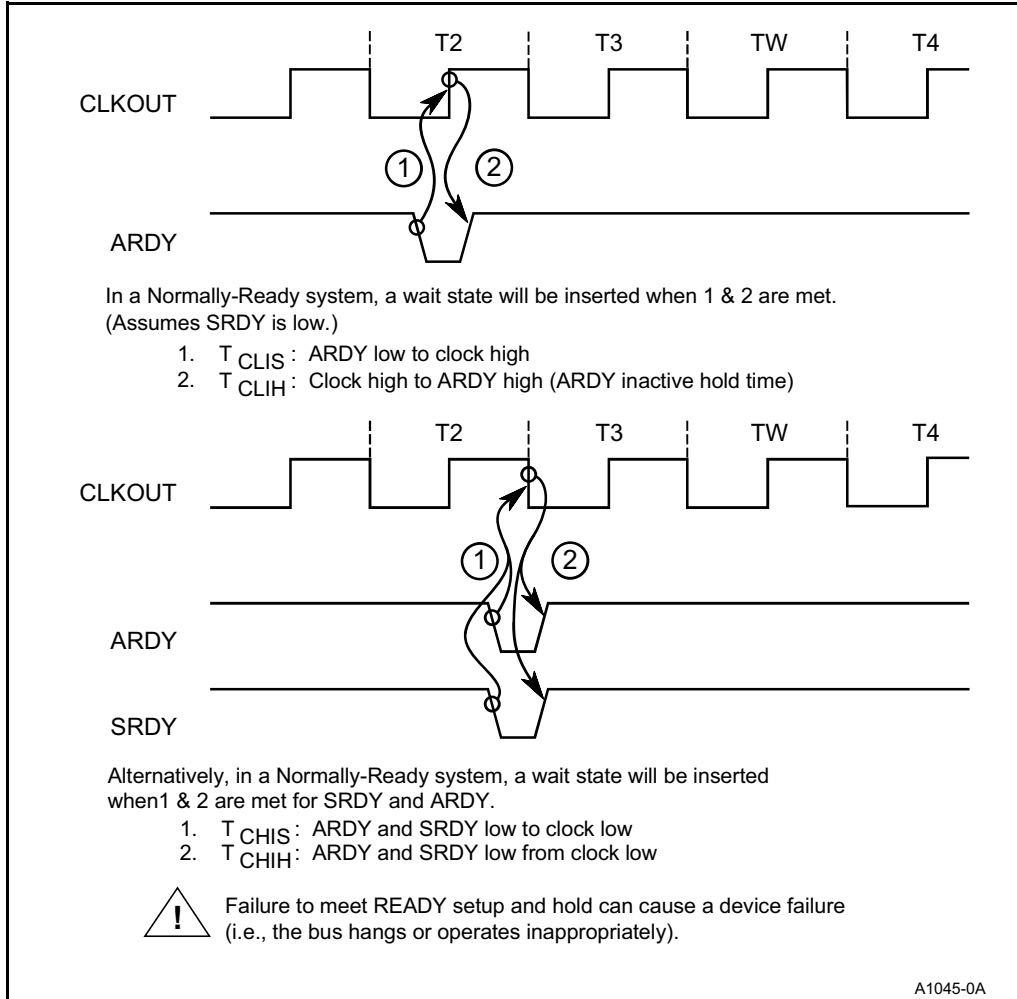
A valid not-ready input can be generated as late as phase 1 of T3 to insert wait states in a normally ready system. A normally not-ready system must run wait states if the not-ready condition cannot be met in time. Figure 3-18 illustrates the minimum and maximum timing necessary to insert wait states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

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

### 3.4.4 Idle States

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





**Figure 3-18. Normally Ready System Timings**

Conditions causing the BIU to become idle include the following.

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

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

### 3.5 BUS CYCLES

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

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

#### 3.5.1 Read Bus Cycles

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

**Table 3-2. Read Bus Cycle Types**

| Status Bit |    |    | Bus Cycle Type  |
|------------|----|----|---|
| S2         | S1 | S0 |   |
| 0          | 0  | 1  | Read I/O — Initiated by the Execution Unit for IN, OUT, INS, OUTS instructions or by the DMA Unit. A19:16 are driven to zero (see Chapter 10, "Direct Memory Access Unit"). |
| 1          | 0  | 0  | Instruction Prefetch — Initiated by the BIU. Data read from the bus fills the prefetch queue.   |
| 1          | 0  | 1  | Read Memory — A19:0 select the desired byte or word memory location.  |

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

**Table 3-3. Read Cycle Critical Timing Parameters**

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



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

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

$T_{RHAX}$  cannot be lengthened (other than by slowing the clock rate). To resolve a buffer fight condition, choose a faster device or buffer the AD bus (see “Buffering the Data Bus” on page 3-36).

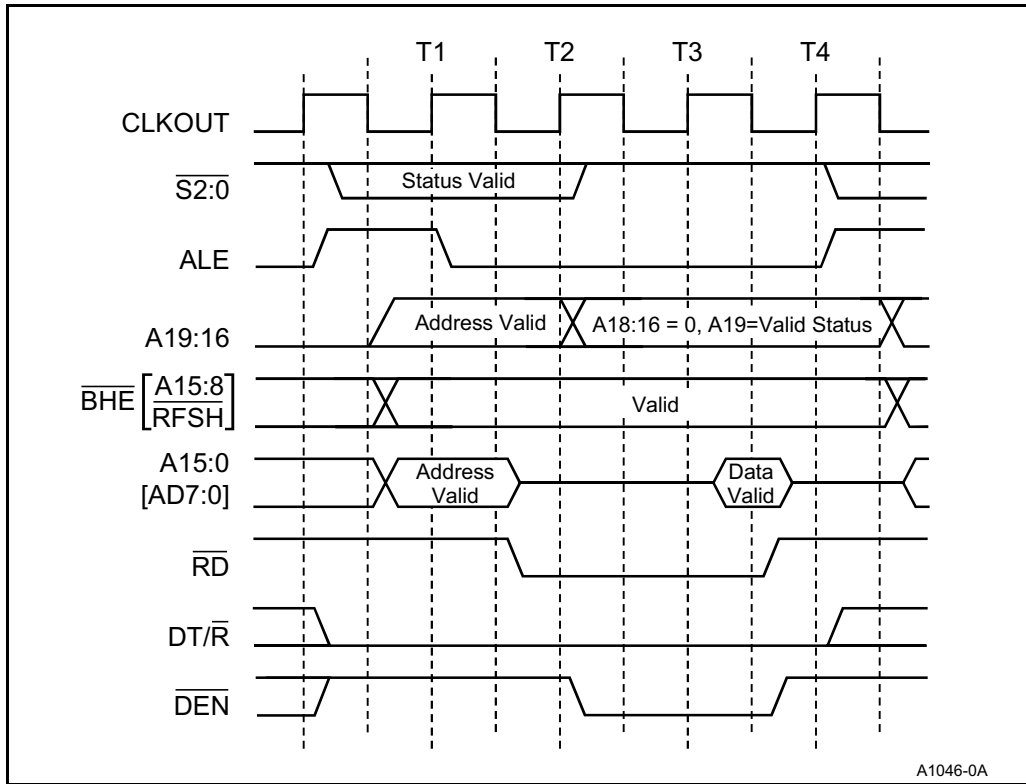


Figure 3-19. Typical Read Bus Cycle

### 3.5.1.1 Refresh Bus Cycles

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

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

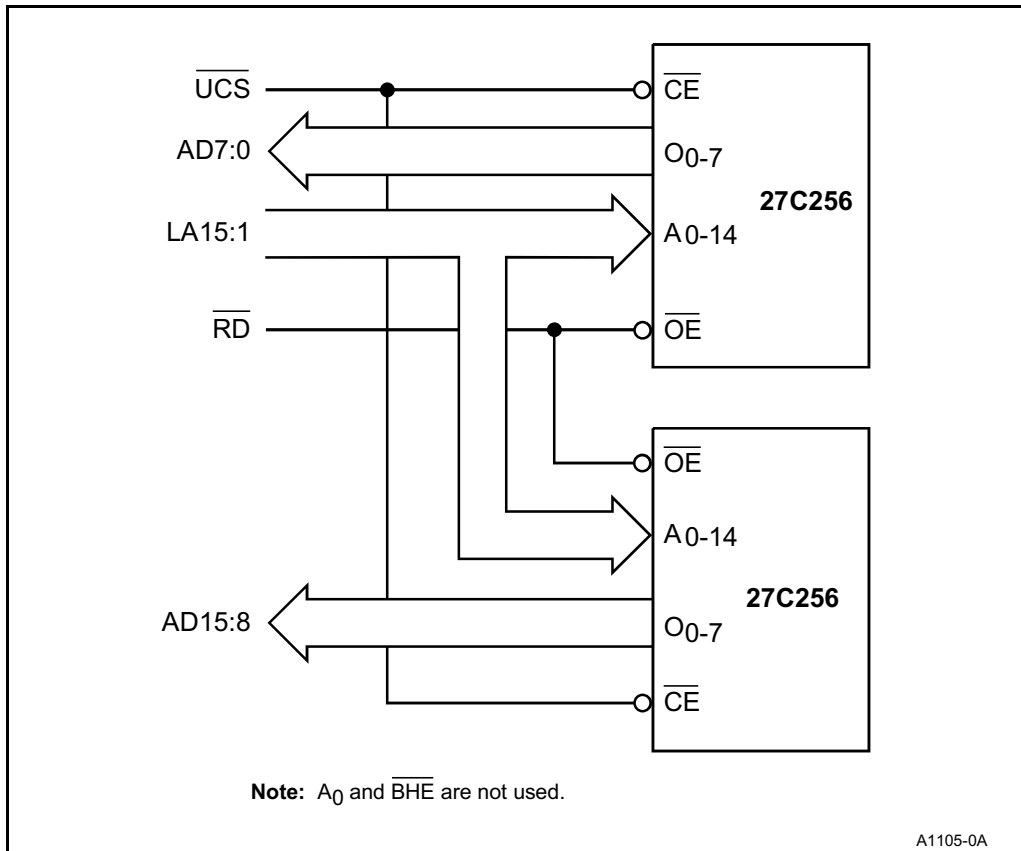


Figure 3-20. Read-Only Device Interface

### 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  $\overline{\text{S2:0}}$ . The bus cycle ends when  $\overline{\text{WR}}$  transitions high (inactive), although data remains valid for one additional clock. Table 3-4 lists the two types of write bus cycles.





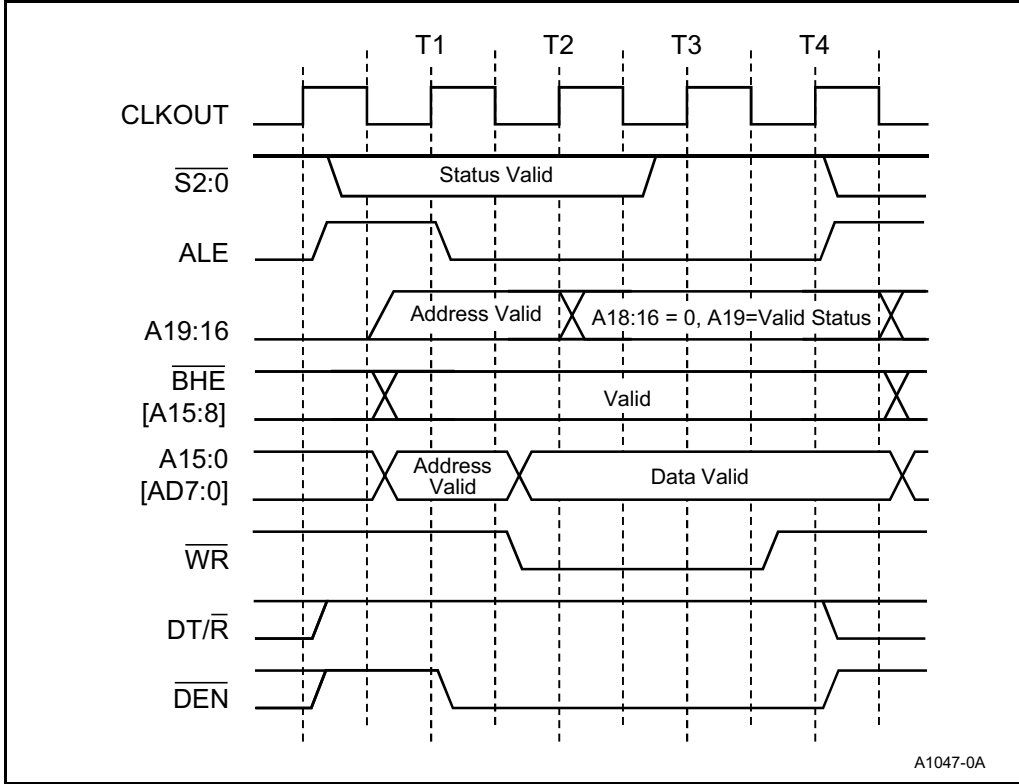


Figure 3-21. Typical Write Bus Cycle

Table 3-4. Write Bus Cycle Types

| Status Bits |    |    | Bus Cycle Type   |
|-------------|----|----|--|
| S2          | S1 | S0 |  |
| 0           | 1  | 0  | Write I/O — Initiated by executing IN, OUT, INS, OUTS instructions or by the DMA Unit. A15:0 select the desired I/O port. A19:16 are driven to zero (see Chapter 10, "Direct Memory Access 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 illustrates a typical 16-bit interface connection to a read/write device. Write bus cycles have many parameters that must be evaluated in determining the compatibility of a memory (or I/O) device. Table 3-5 lists some critical write bus cycle parameters.

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

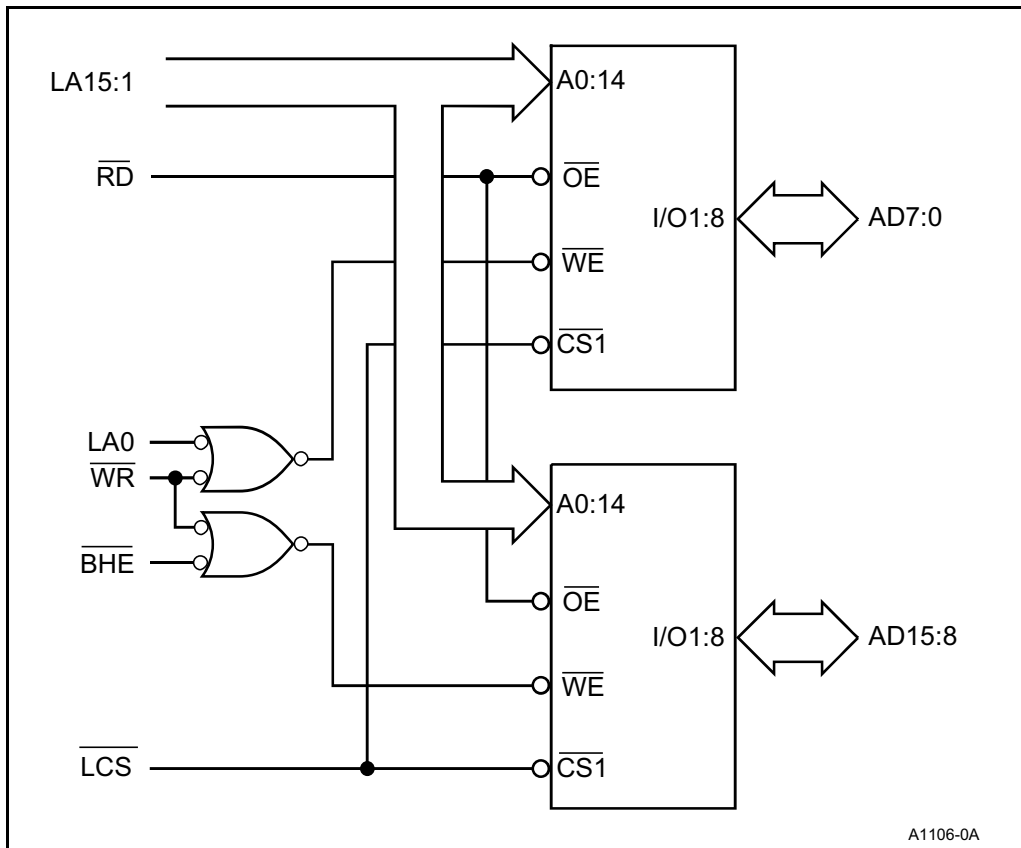


Figure 3-22. 16-Bit Bus Read/Write Device Interface



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

**Table 3-5. Write Cycle Critical Timing Parameters**

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

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

### 3.5.3 Interrupt Acknowledge Bus Cycle

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

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

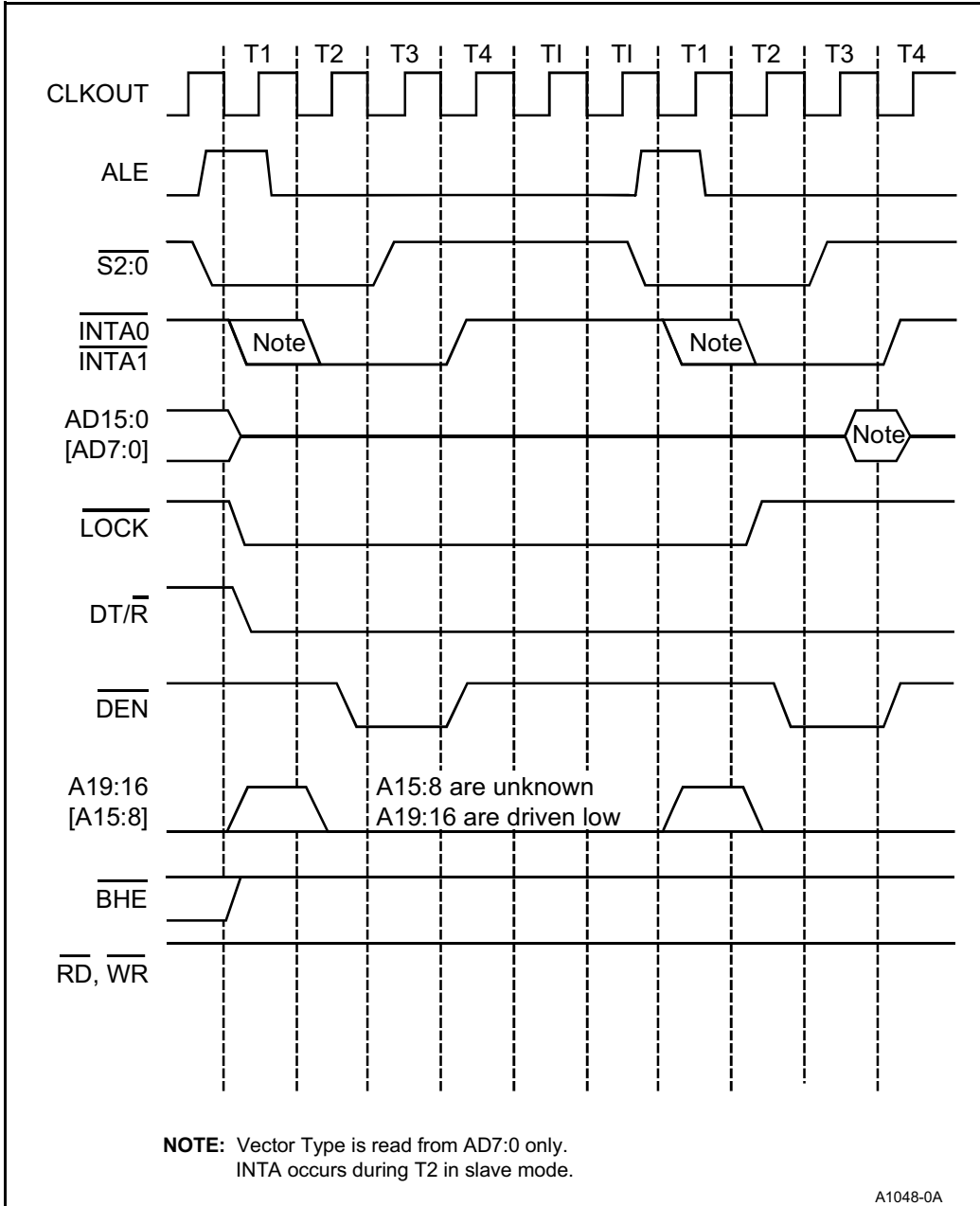


Figure 3-23. Interrupt Acknowledge Bus Cycle



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

**NOTE**

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

**3.5.3.1 System Design Considerations**

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

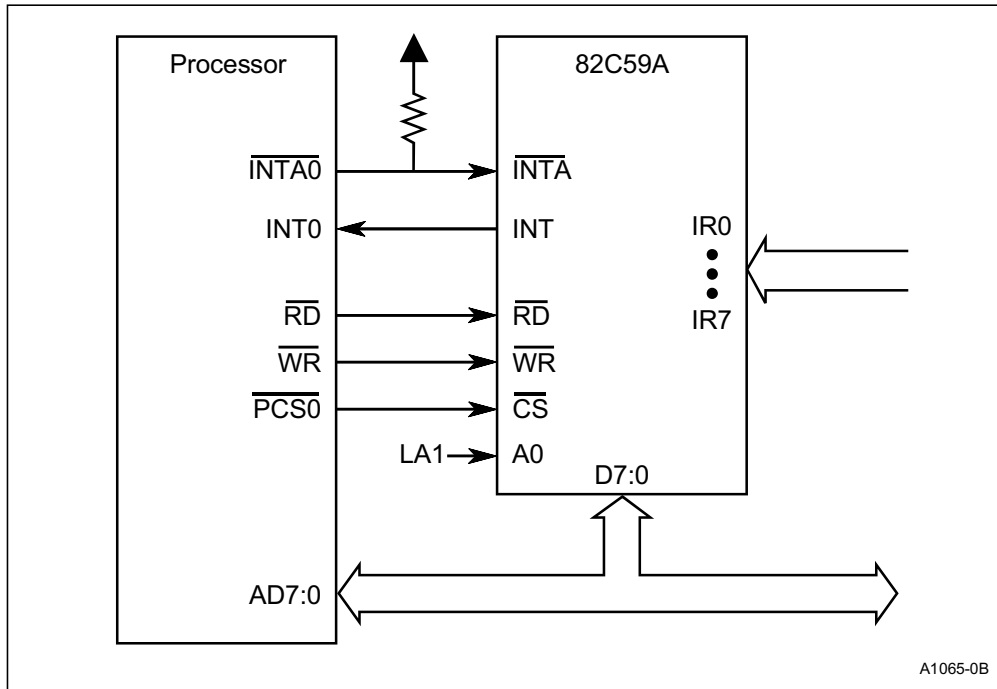


Figure 3-24. Typical 82C59A Interface

### 3.5.4 HALT Bus Cycle

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

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

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

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

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

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

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





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

**Table 3-6. HALT Bus Cycle Pin States**

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



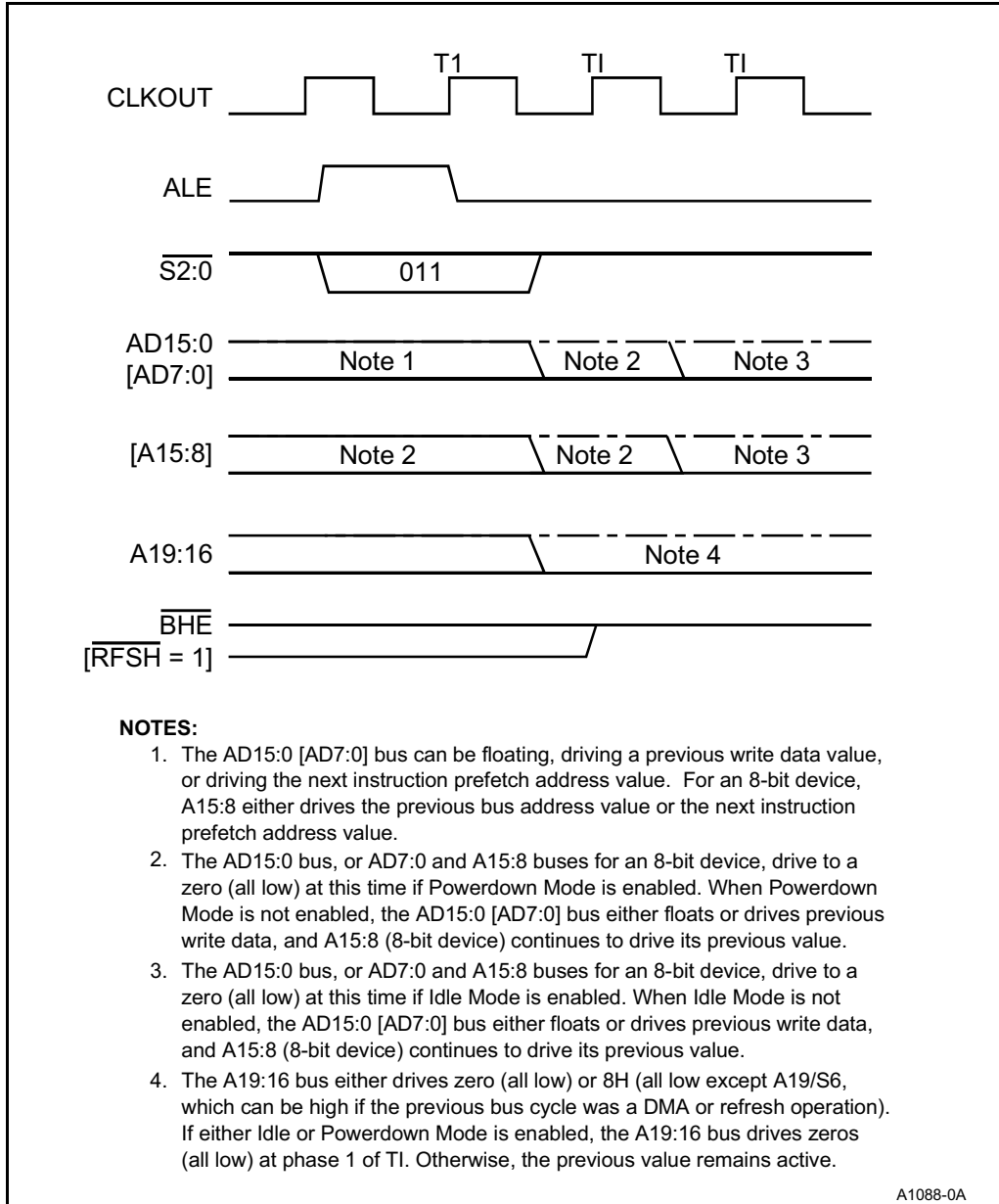


Figure 3-25. HALT Bus Cycle





### 3.5.5 Temporarily Exiting the HALT Bus State

A DMA request, refresh request or bus hold request causes the BIU to exit the HALT bus state temporarily. This can occur only when in the Active or Idle power management mode. The BIU returns to the HALT bus **state** after it completes the desired bus operation. However, the BIU **does not** execute another bus HALT **cycle** (i.e., ALE and bus cycle status are not regenerated). Figures 3-26, 3-27 and 3-28 illustrate how the BIU temporarily exits and then returns to the HALT bus state.

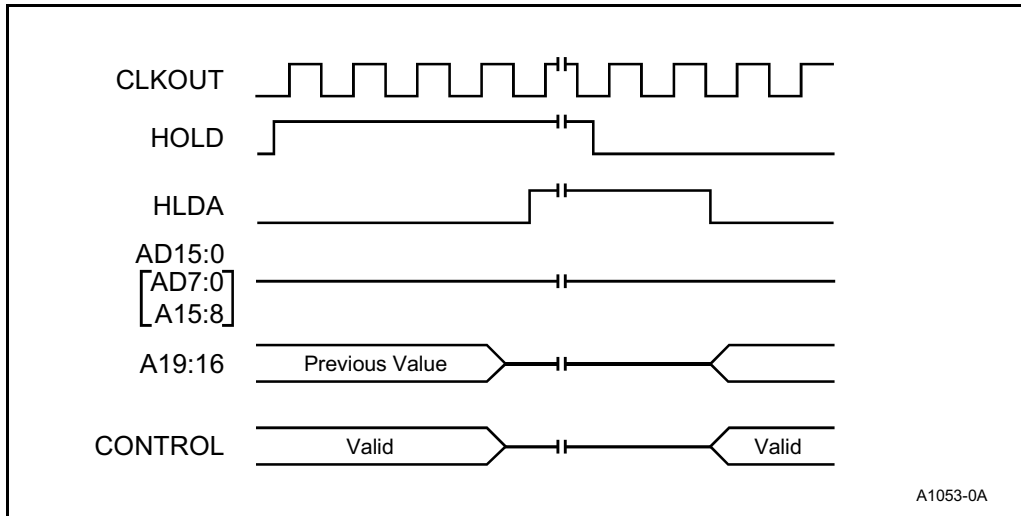


Figure 3-26. Returning to HALT After a HOLD/HLDA Bus Exchange

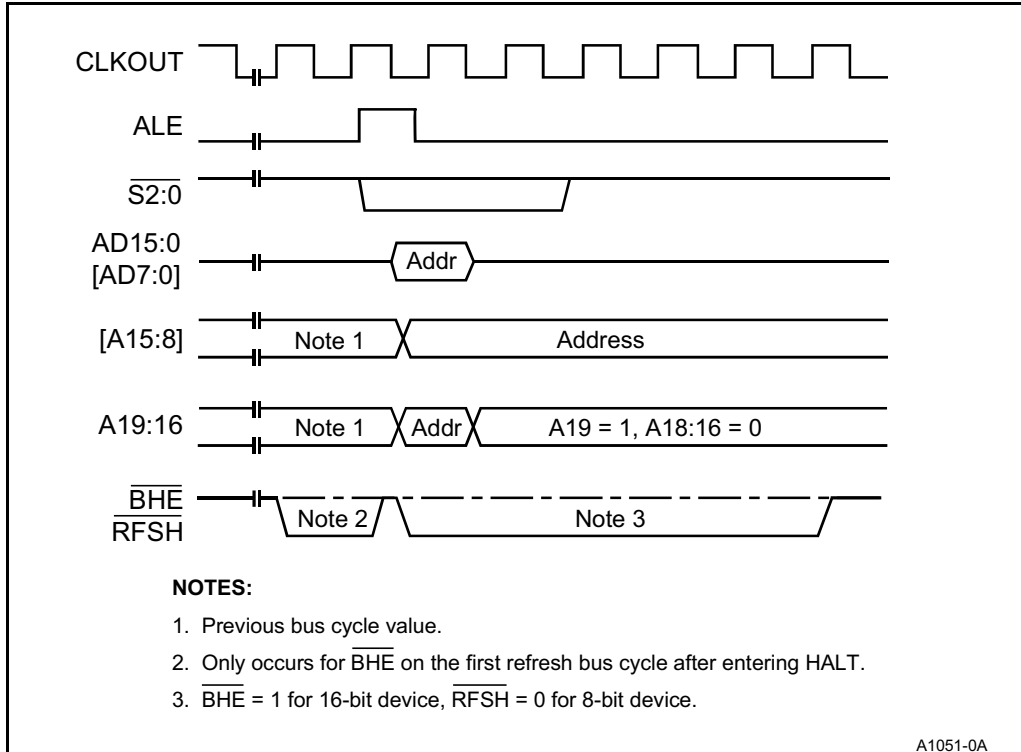


Figure 3-27. Returning to HALT After a Refresh Bus Cycle



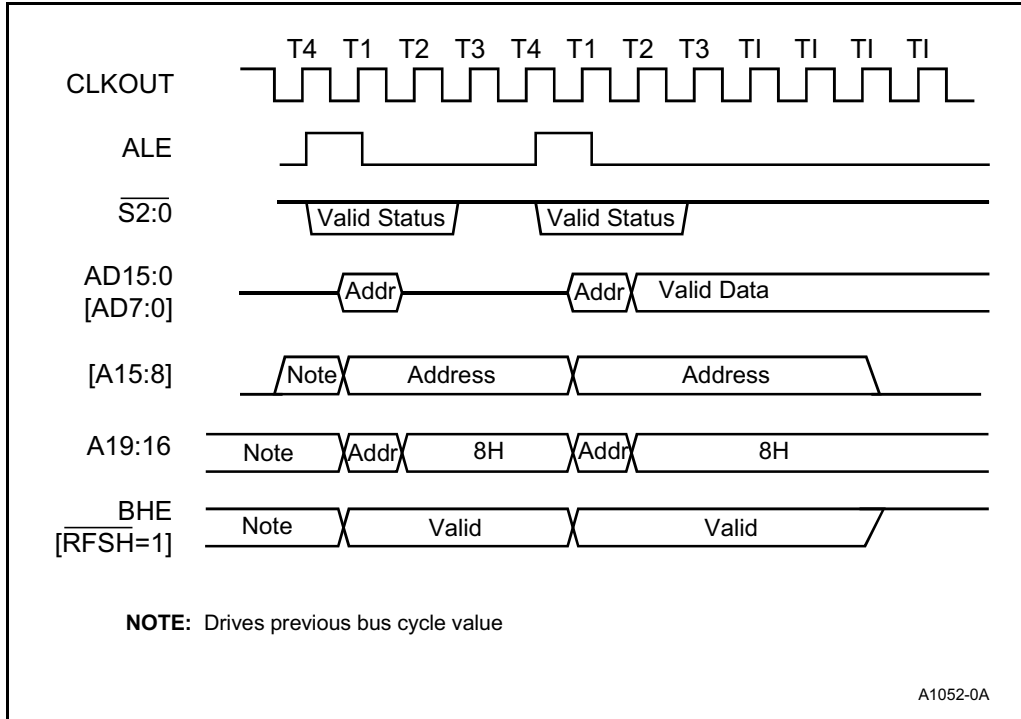


Figure 3-28. Returning to HALT After a DMA Bus Cycle

### 3.5.6 Exiting HALT

In Powerdown mode, only an NMI forces the BIU to exit the HALT bus state. In any other power management mode, either an NMI or any unmasked  $INTn$  interrupt causes the BIU to exit HALT. The first bus operations to occur after exiting HALT are read cycles to reload the CS:IP registers. Figure 3-29 and Figure 3-30 show how the HALT bus state is exited when an NMI or  $INTn$  occurs.



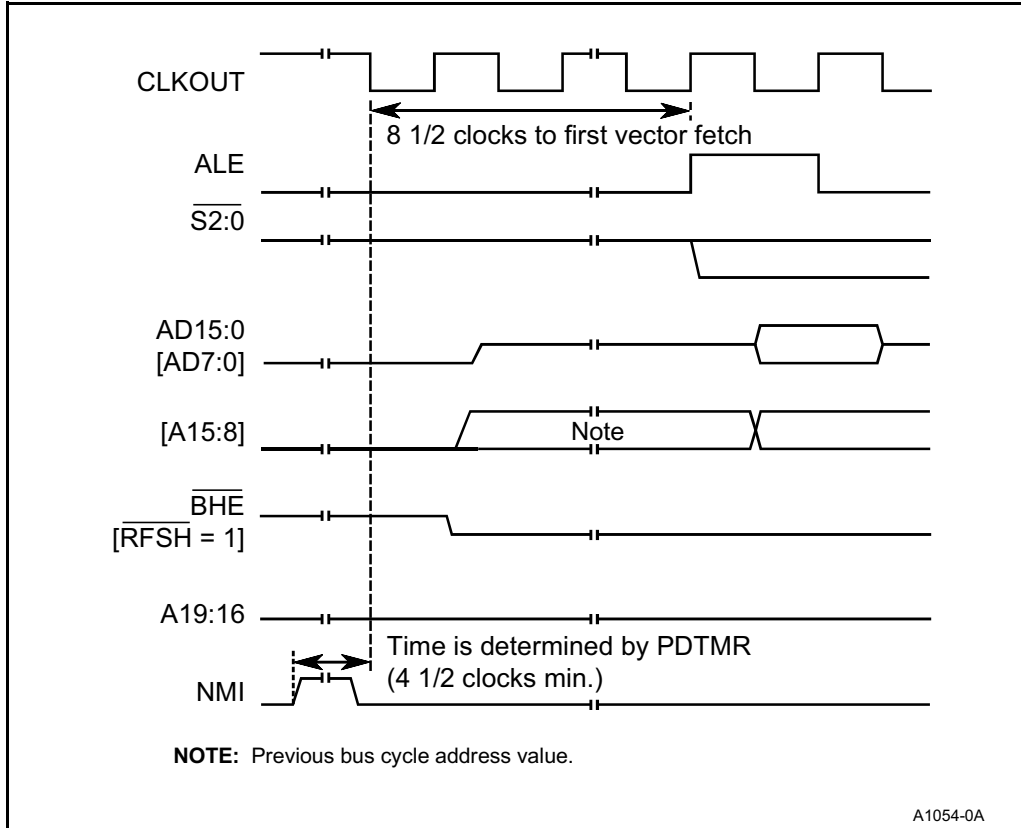


Figure 3-29. Exiting HALT (Powerdown Mode)



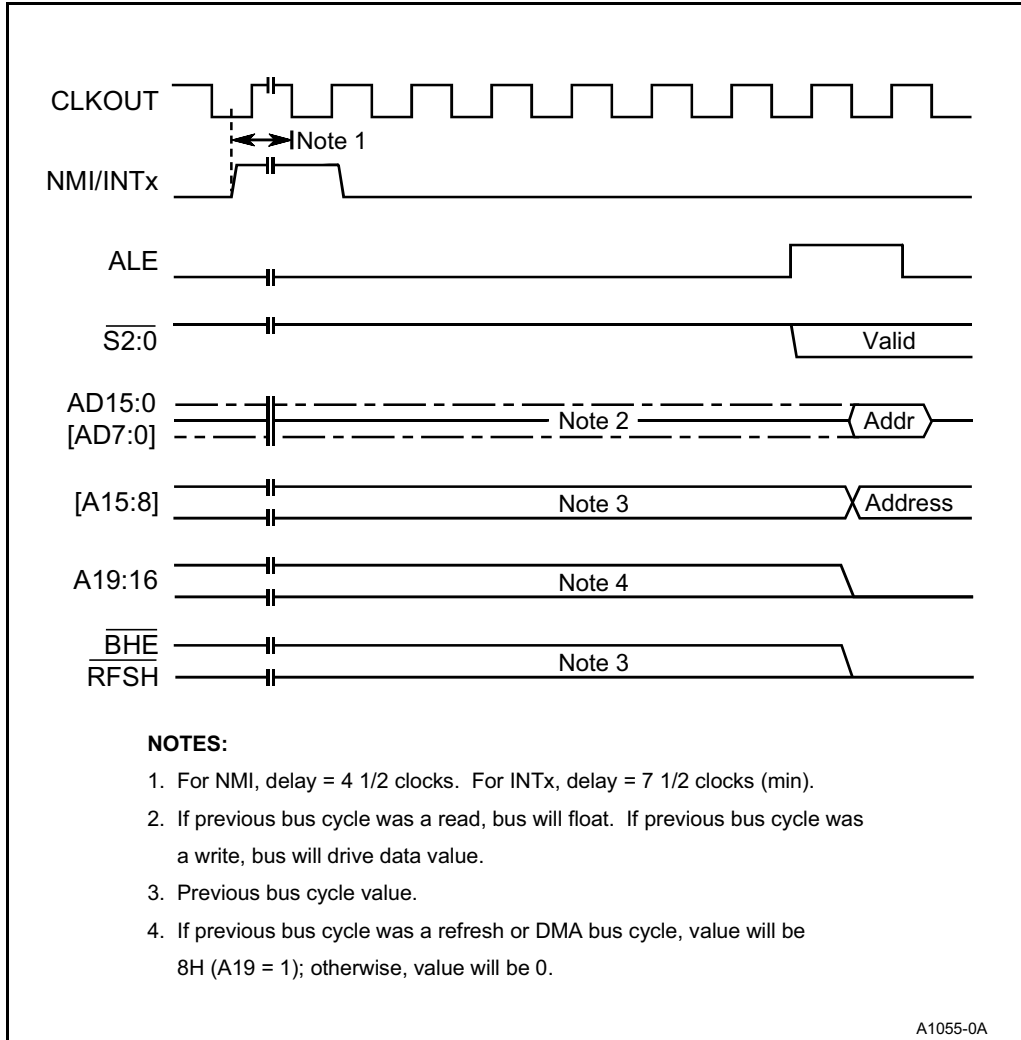


Figure 3-30. Exiting HALT (Active/Idle Mode)

### 3.6 SYSTEM DESIGN ALTERNATIVES

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

### 3.6.1 Buffering the Data Bus

The BIU generates two control signals,  $\overline{DEN}$  and  $DT/\overline{R}$ , to control bidirectional buffers or transceivers. The timing relationship of  $\overline{DEN}$  and  $DT/\overline{R}$  is shown in Figure 3-31. The following conditions require transceivers:

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

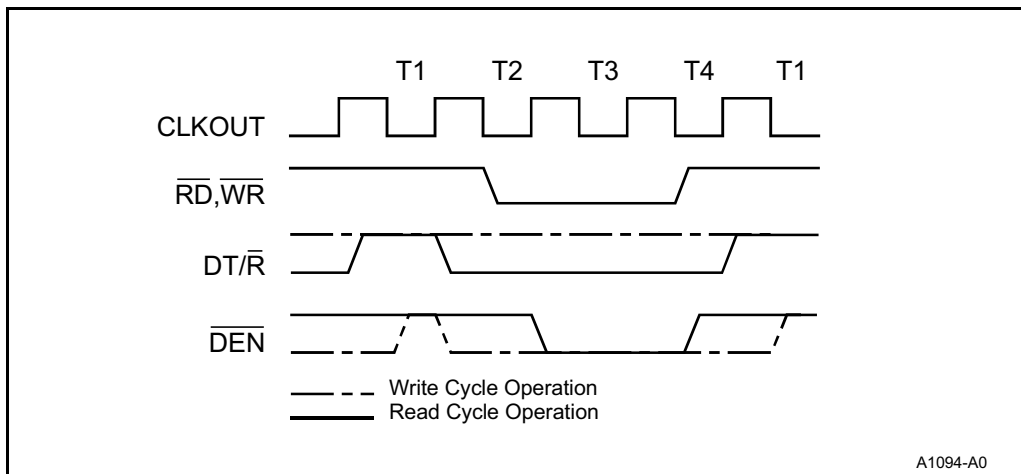


Figure 3-31.  $\overline{DEN}$  and  $DT/\overline{R}$  Timing Relationships

The circuit shown in Figure 3-32 illustrates how to use transceivers to buffer the address/data bus. The connection between the processor and the transceiver is known as the *local bus*. A connection between the transceiver and other memory or I/O devices is known as the *buffered bus*. A *fully buffered* system has **no** devices attached to the local bus. A *partially buffered* system has devices on both the local and buffered buses.



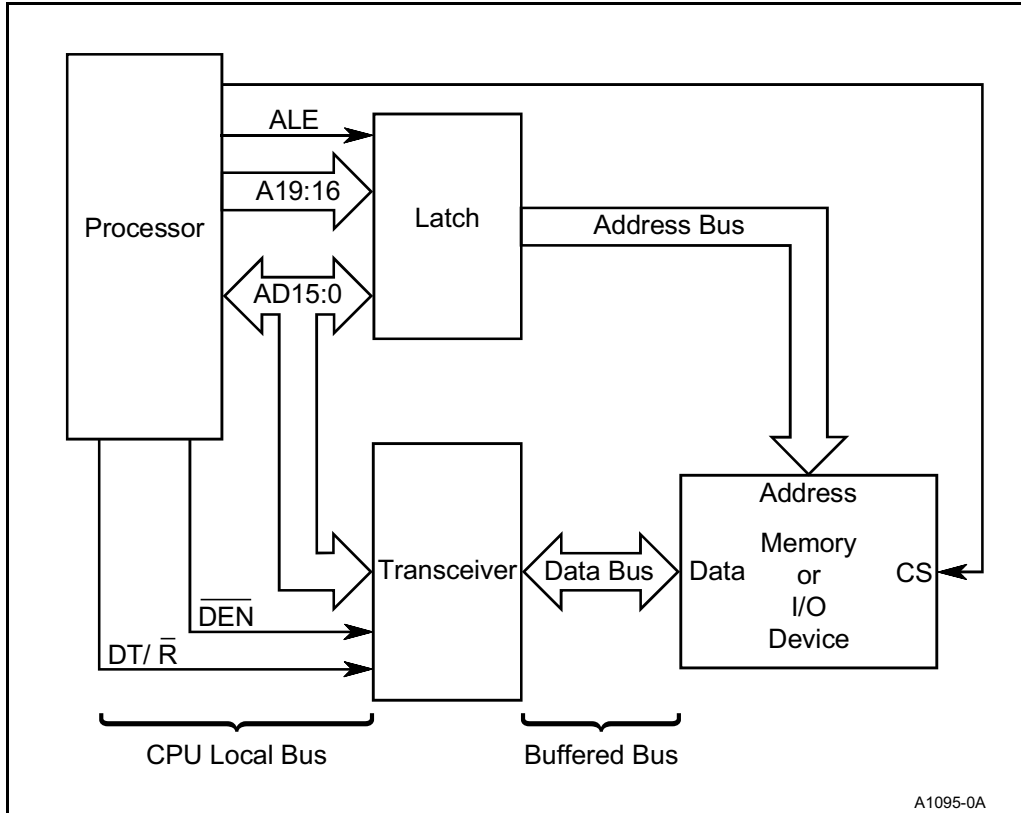


Figure 3-32. Buffered AD Bus System

In a fully buffered system,  $\overline{DEN}$  directly drives the transceiver output enable. A partially buffered system requires that  $\overline{DEN}$  be qualified with another signal to prevent the transceiver from going active for local bus accesses. Figure 3-33 illustrates how to use chip-selects to qualify  $\overline{DEN}$ .

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



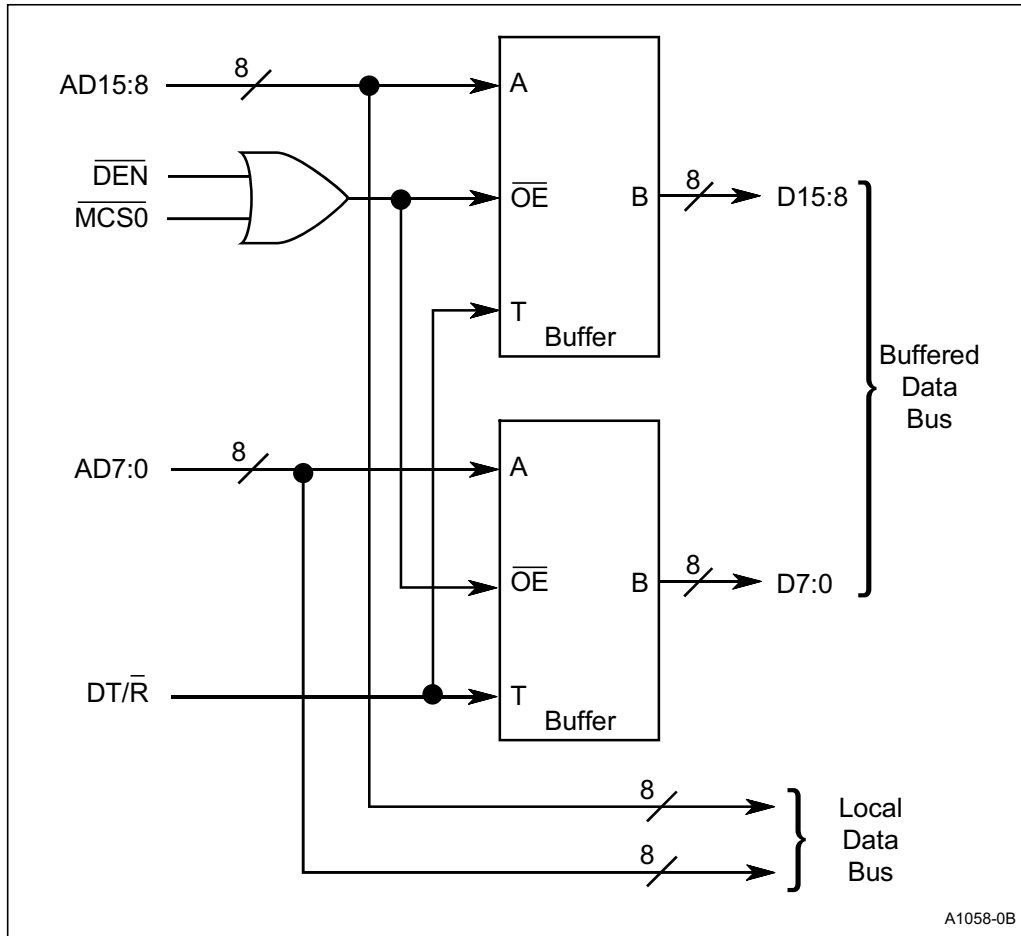


Figure 3-33. Qualifying  $\overline{DEN}$  with Chip-Selects

### 3.6.2 Synchronizing Software and Hardware Events

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

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







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

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 residing in the CPU.

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

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

Another application of LOCK for multiprocessor systems consists of a locked block move, which allows high speed message transfer from one CPU to another. During the locked CPU instruction (i.e., while LOCK is active), a bus hold, DMA or refresh request is recorded, but is not acknowledged until completion of the locked instruction. However, LOCK has no effect 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 (such as LOCK) are considered extensions of the instructions they precede. Interrupts, DMA requests and refresh requests that occur during execution of the prefix are not acknowledged until the instruction following the prefix completes (except for instructions that are servicing interrupts during their execution, such as HALT, WAIT and repeated string primitives). Note that multiple prefix bytes can precede an instruction.

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

**3.6.4 Using the Queue Status Signals**

Older-generation devices require the queue status signals to interface with an 8087 math coprocessor. Newer devices do not require these signals because they use the 80187 math coprocessor, which has an I/O port interface similar to that of a peripheral device.

The queue status signals, QS0 and QS1, indicate the state of the internal queue (Table 3-7). Since the Execution Unit can remove information from the queue on any clock boundary, the queue status pins may change state on every phase 1 clock edge (Figure 3-34). Although these signals cannot be related to any specific T-state, the relationship between the queue status signals and BIU operation always remains the same for a given instruction sequence.

QS0 and QS1 are alternate functions of ALE and  $\overline{WR}$ , respectively. To enable QS0 and QS1, you must connect the  $\overline{RD}$  pin directly to ground. In this case,  $\overline{RD}$ ,  $\overline{WR}$  and ALE are no longer available and must be generated by external hardware such as an 82C88 or a programmable logic device.

**Table 3-7. Queue Status Signal Decoding**

| QS1 | QS0 | Queue Status  |
|-----|-----|---|
| 0   | 0   | No queue operation occurred.  |
| 0   | 1   | The first byte of a new instruction was removed from the queue.   |
| 1   | 0   | The queue was reinitialized. All prefetch information was flushed; the BIU must begin prefetching new queue information.                  |
| 1   | 1   | A subsequent byte of an instruction was removed from the queue. The current instruction contains multiple opcode bytes or immediate data. |



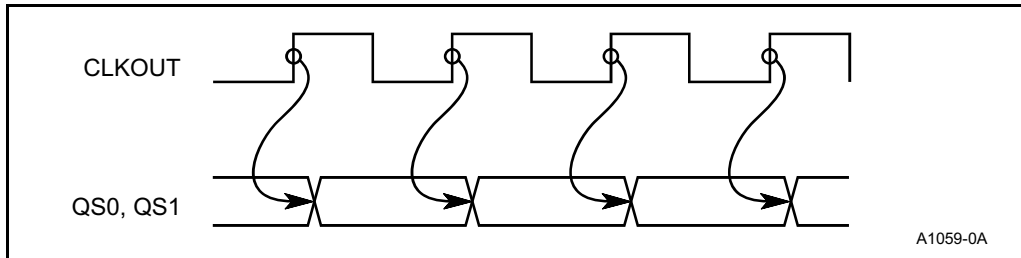


Figure 3-34. Queue Status Timing

### 3.7 MULTI-MASTER BUS SYSTEM DESIGNS

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

#### 3.7.1 Entering Bus HOLD

In responding to the hold request input, the BIU floats the entire address and data bus, and many of the control signals. Figure 3-35 illustrates the timing sequence when acknowledging the hold request. Table 3-8 lists the states of the BIU pins when HLDA is asserted. All device pins not mentioned in Table 3-8 or shown in Figure 3-35 remain either active (e.g., CLKOUT and T1OUT) or inactive (e.g., UCS and INTA). Refer to the data sheet for specific details of pin functions during a bus hold.

**Figure 3-35. Timing Sequence Entering HOLD**

**Table 3-8. Signal Condition Entering HOLD**

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

**3.7.1.1 HOLD Bus Latency**

The duration between the time that the external device asserts HOLD and the time that the BIU asserts HLDA is known as *bus latency*. In Figure 3-35, the two-clock delay between HOLD and HLDA represents the shortest bus latency. Normally this occurs only if the bus is idle or halted or if the bus hold request occurs just before the BIU begins another bus cycle.



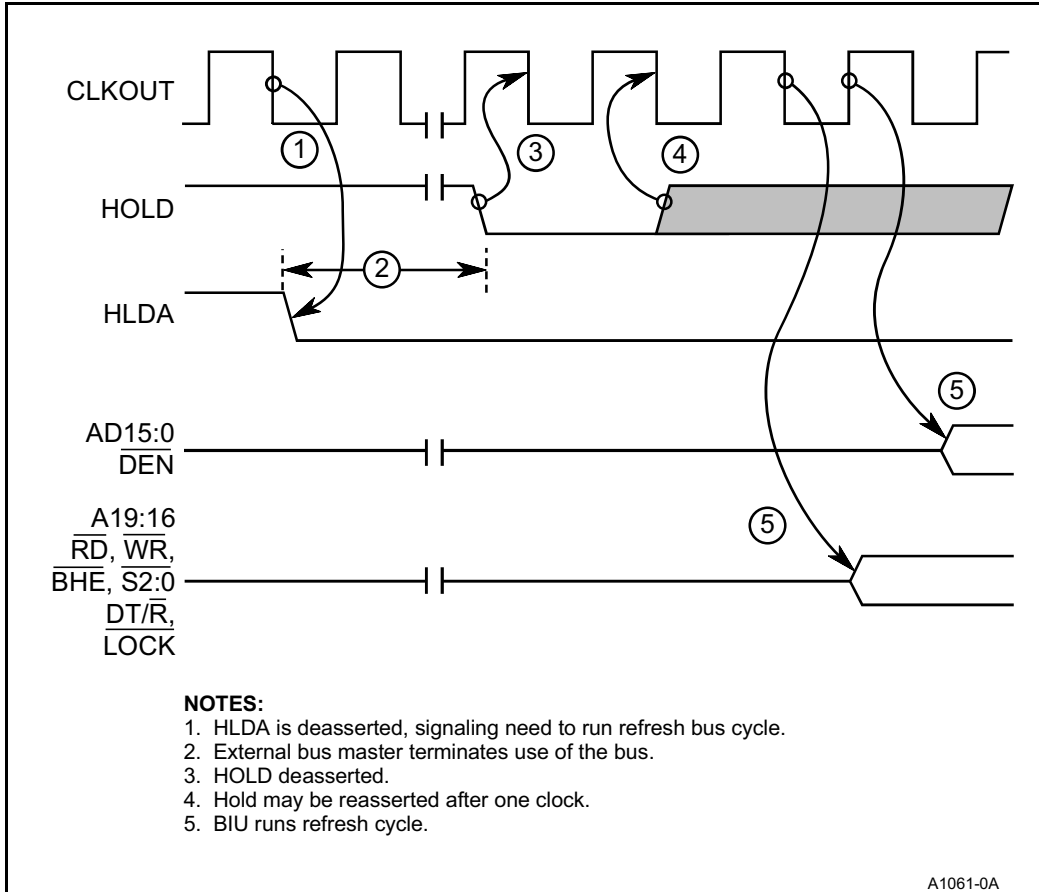
The major factors that influence bus latency are listed below (in order from longest delay to shortest delay).

1. Bus Not Ready — As long as the bus remains not ready, a bus hold request cannot be serviced.
2. Locked Bus Cycle — As long as  $\overline{\text{LOCK}}$  remains asserted, a bus hold request cannot be serviced. Performing a locked move string operation can take several thousands of clocks.
3. Completion of Current Bus Cycle — A bus hold request cannot be serviced until the current bus cycle completes. A bus hold request will not separate bus cycles required to move odd-aligned word data. Also, bus cycles with long wait states will delay the servicing of a bus hold request.
4. Interrupt Acknowledge Bus Cycle — A bus hold request is not serviced until after an  $\overline{\text{INTA}}$  bus cycle has completed. An  $\overline{\text{INTA}}$  bus cycle drives  $\overline{\text{LOCK}}$  active.
5. 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 cannot 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 HLDA has been asserted it remains asserted until HOLD is removed. However, when a refresh bus request is generated, the HLDA output is removed (driven low) to signal the need for the BIU to regain control of the local bus. The BIU does not gain control of the bus until HOLD is removed. This procedure prevents the BIU from just arbitrarily regaining control of the bus.

Figure 3-36 shows the timing associated with the occurrence of a refresh request while HLDA is active. Note that HLDA can be as short as one clock in duration. This happens when a refresh request occurs just after HLDA is granted. A refresh request has higher priority than a bus hold request; therefore, when the two occur simultaneously, the refresh request occurs before HLDA becomes active.



**Figure 3-36. Refresh Request During HOLD**

The device requesting a bus hold must be able to detect a HLDA pulse that is one clock in duration. A bus lockup (hang) condition can result if the requesting device fails to detect the short HLDA pulse and continues to wait for HLDA to be asserted while the BIU waits for HOLD to be deasserted. The circuit shown in Figure 3-37 can be used to latch HLDA.



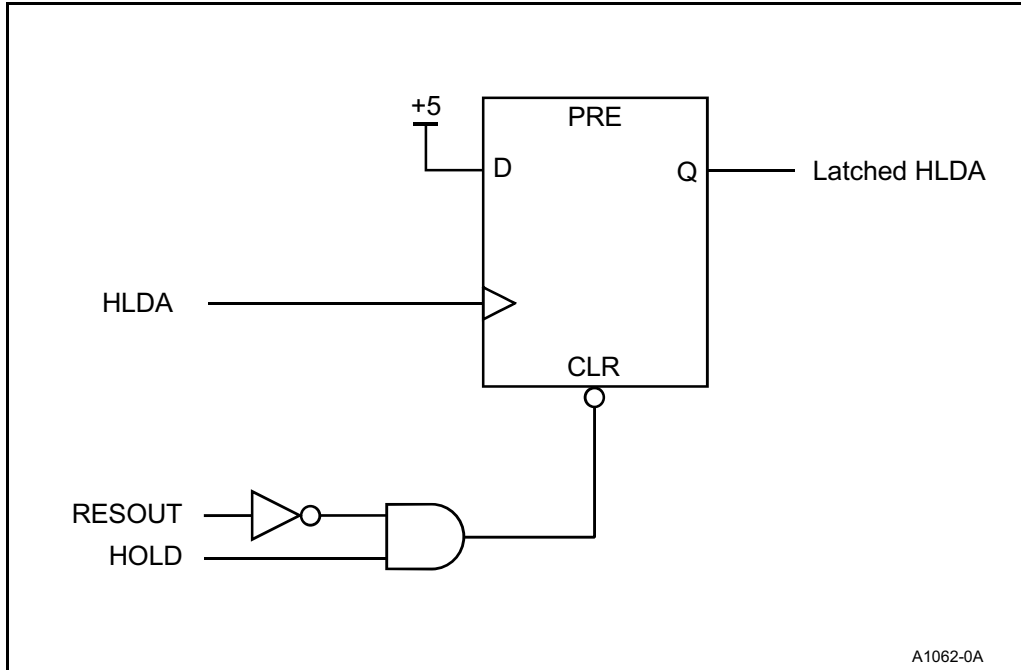


Figure 3-37. Latching HLDA

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

### 3.7.2 Exiting HOLD

Figure 3-38 shows the timing associated with exiting the bus hold state. Normally a bus operation (e.g., an instruction prefetch) occurs just after HOLD is released. However, if no bus cycle is pending when leaving a bus hold state, the bus and associated control signals remain floating, if the system is in normal operating mode. (For signal states associated with Idle and Powerdown modes, see “Temporarily Exiting the HALT Bus State” on page 3-31).



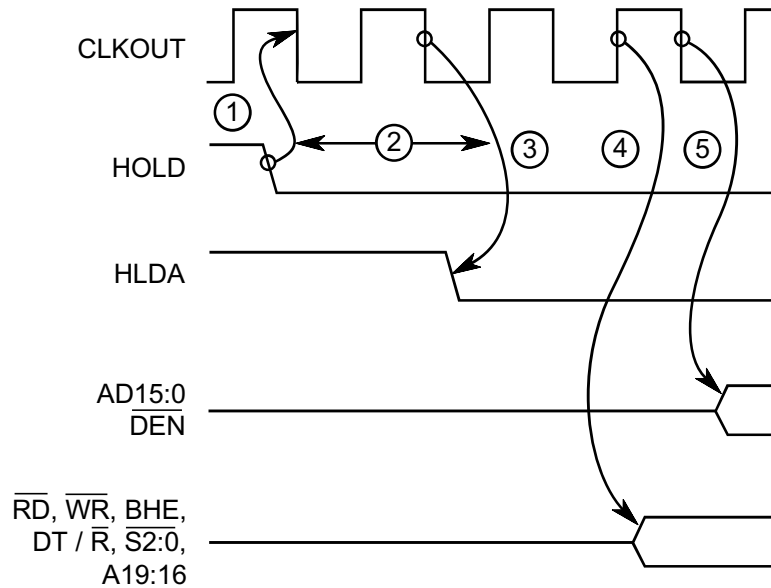


Figure 3-38. Exiting HOLD

### 3.8 BUS CYCLE PRIORITIES

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

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







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







**4**

# **Peripheral Control Block**







## CHAPTER 4 PERIPHERAL CONTROL BLOCK

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

### 4.1 PERIPHERAL CONTROL REGISTERS

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

### 4.2 PCB RELOCATION REGISTER

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

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

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

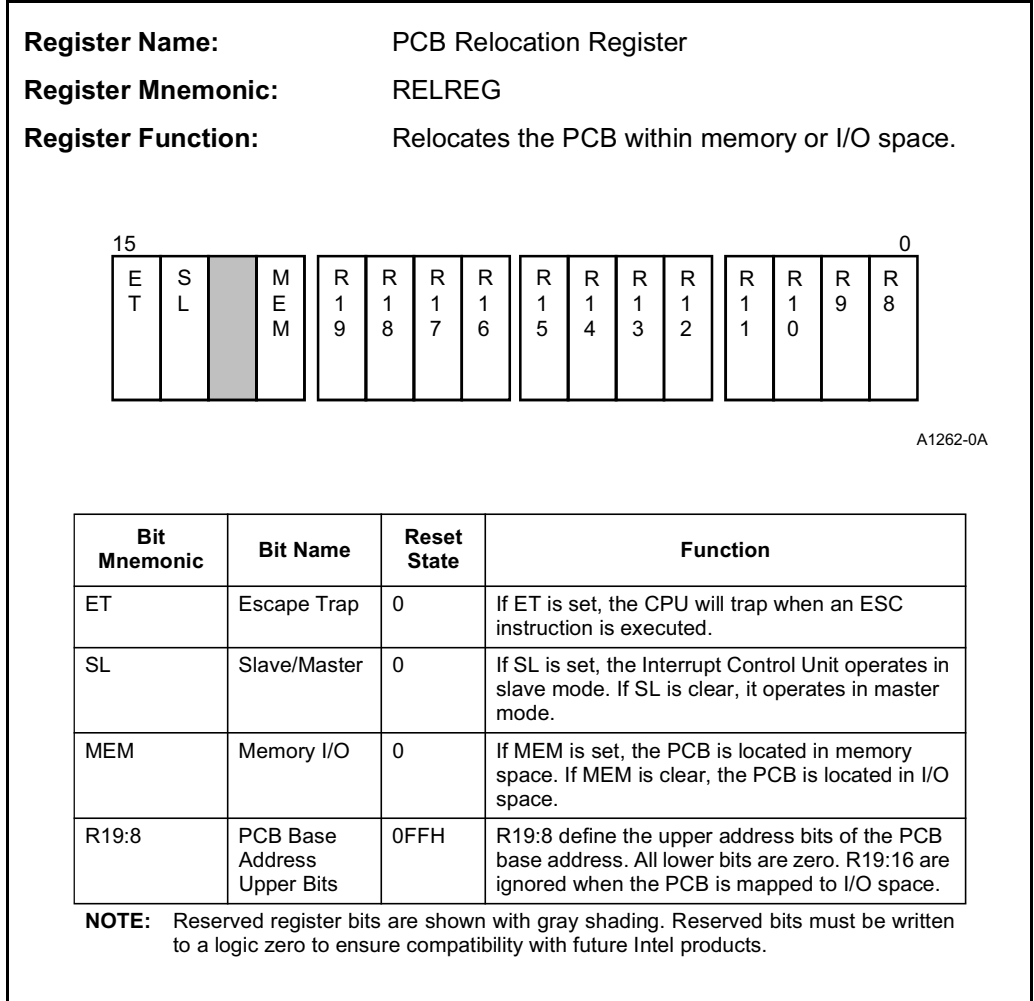


Figure 4-1. PCB Relocation Register



PERIPHERAL CONTROL BLOCK

Table 4-1. Peripheral Control Block

| 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        | Reserved |
| 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        | INSTS    | 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   |

### 4.3 RESERVED LOCATIONS

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

#### NOTE

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

### 4.4 ACCESSING THE PERIPHERAL CONTROL BLOCK

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

#### 4.4.1 Bus Cycles

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

#### 4.4.2 READY Signals and Wait States

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





### 4.4.3 F-Bus Operation

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

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

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

|                    |  |
|--------------------|--|
| <b>Word reads</b>  | Aligned word reads of the PCB work normally. Access only even-aligned words with <code>IN AX, DX</code> or <code>MOV word register, even PCB address</code> .  |
| <b>Byte reads</b>  | Byte reads of the PCB work normally. Beware of reading word-wide PCB registers that may change value between successive reads (e.g., timer count value).   |
| <b>Word writes</b> | <p>Always write even-aligned words to the PCB. Writing an odd-aligned word will give unexpected results.</p> <p>For the 80C186 Modular Core, use either</p> <ul style="list-style-type: none"> <li>– <code>OUT DX, AX</code> or</li> <li>– <code>OUT DX, AL</code> or</li> <li>– <code>MOV even PCB address, word register</code>.</li> </ul> <p>For the 80C188 Modular Core, using <code>OUT DX, AX</code> will perform an unnecessary bus cycle and is not recommended. Use either</p> <ul style="list-style-type: none"> <li>– <code>OUT DX, AL</code> or</li> <li>– <code>MOV even-aligned byte PCB address, byte register low byte</code>.</li> </ul> |
| <b>Byte writes</b> | Always use even-aligned byte writes to the PCB. Even-aligned byte writes will modify the entire word PCB location. Do <b>not</b> perform unaligned byte writes to the PCB.   |

#### 4.4.3.1 Writing the PCB Relocation Register

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

#### 4.4.3.2 Accessing the Peripheral Control Registers

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

#### 4.4.3.3 Accessing Reserved Locations

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

#### NOTE

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

### 4.5 SETTING THE PCB BASE LOCATION

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





## PERIPHERAL CONTROL BLOCK

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

### NOTE

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

### 4.5.1 Considerations for the 80C187 Math Coprocessor Interface

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

### NOTE

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

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

For details on the math coprocessor interface, see Chapter 11, "Math Coprocessing."







**5**

# **Clock Generation and Power Management**







## CHAPTER 5 CLOCK GENERATION AND POWER MANAGEMENT

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

### 5.1 CLOCK GENERATION

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

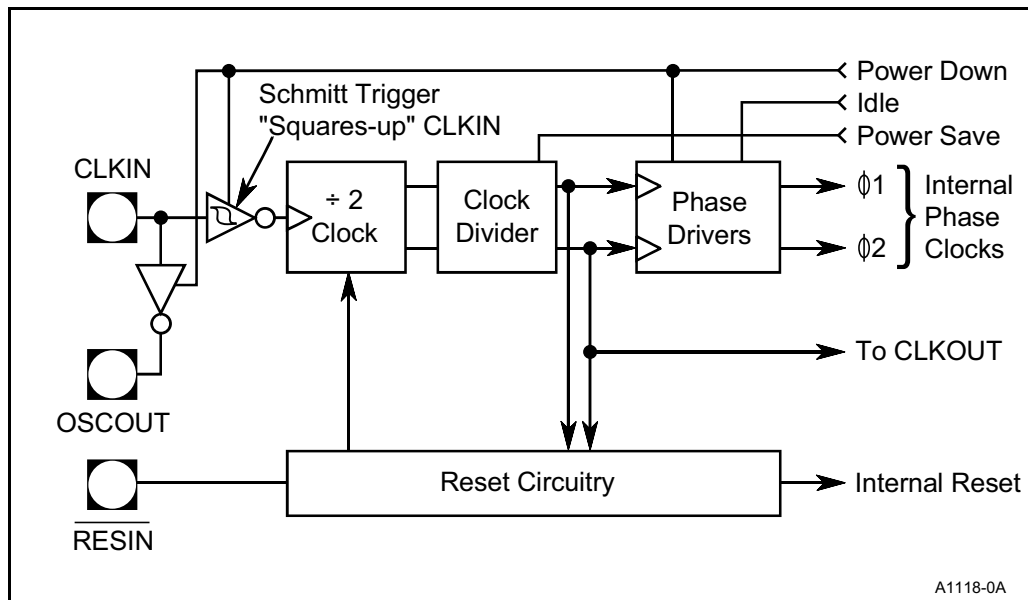


Figure 5-1. Clock Generator

#### 5.1.1 Crystal Oscillator

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

5.1.1.1 Oscillator Operation

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

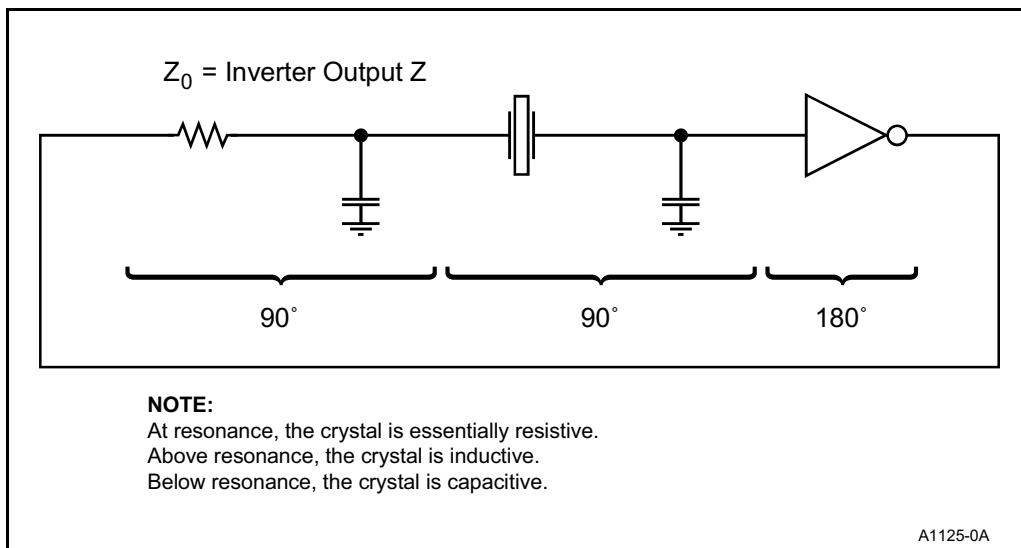


Figure 5-2. Ideal Operation of Pierce Oscillator

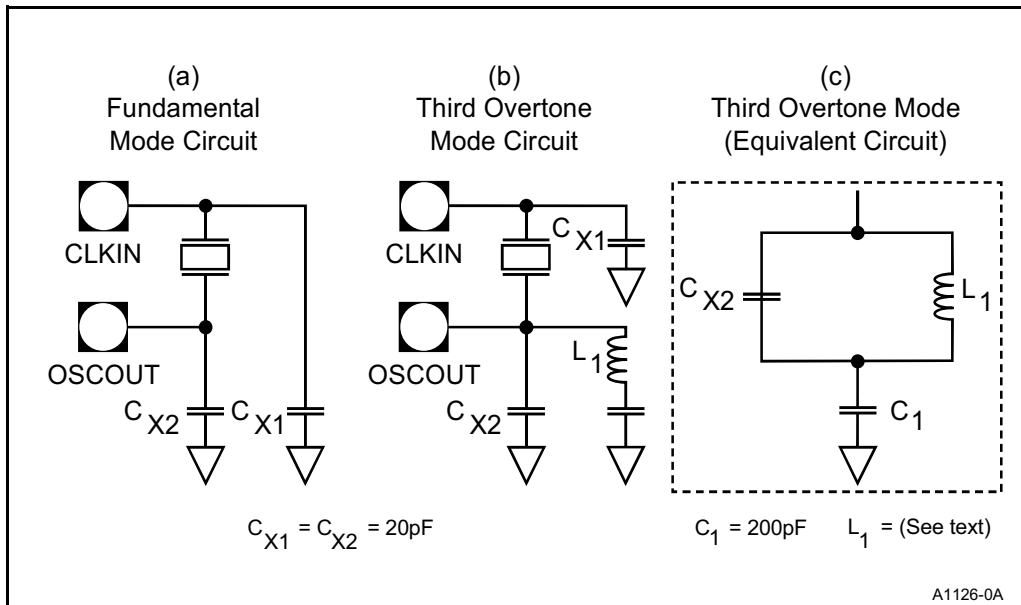
Figure 5-3 shows the actual microprocessor crystal connections. For low frequencies, crystal vendors offer fundamental mode crystals. At higher frequencies, a third overtone crystal is the only choice. The external capacitors,  $C_{X1}$  at CLKIN and  $C_{X2}$  at OSCOUT, together with stray capacitance,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





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

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



**Figure 5-3. Crystal Connections to Microprocessor**

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

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

|   |  |
|---|--|
| (a) Series or Parallel Resonant Frequency<br><br>$f = \frac{1}{2\pi\sqrt{L_1 C_1}}$ | (b) Equivalent Capacitance<br><br>$C_{eq} = \frac{\omega^2 C_1 C_{x2} L_1 - C_1 - C_{x2}}{\omega^2 C_1 L_1 - 1}$ |
|---|--|

**Figure 5-4. Equations for Crystal Calculations**

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

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

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



### 5.1.1.2 Selecting Crystals

When specifying crystals, consider these parameters:

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

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

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

### 5.1.2 Using an External Oscillator

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

### 5.1.3 Output from the Clock Generator

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

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

### 5.1.4 Reset and Clock Synchronization

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

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



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

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

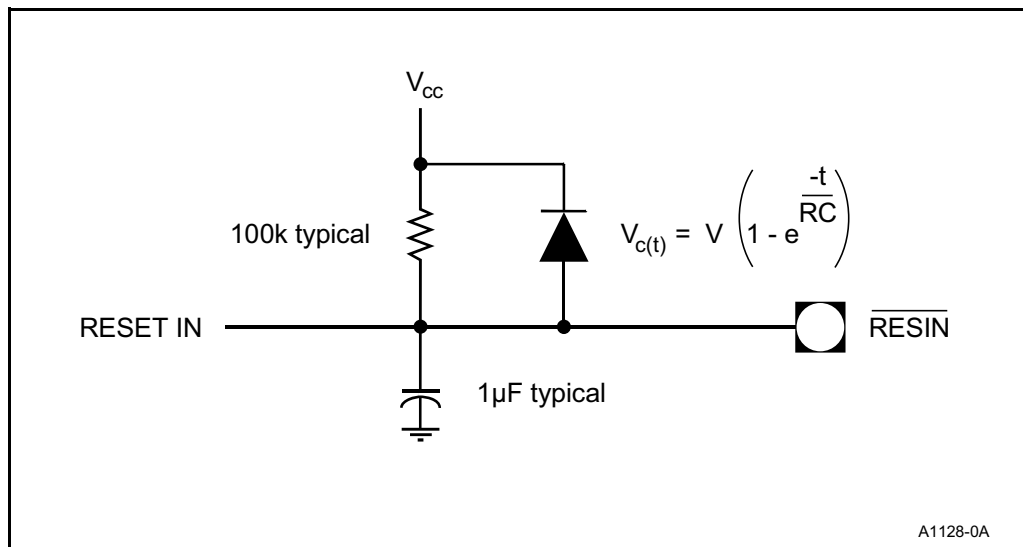


Figure 5-5. Simple RC Circuit for Powerup Reset

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



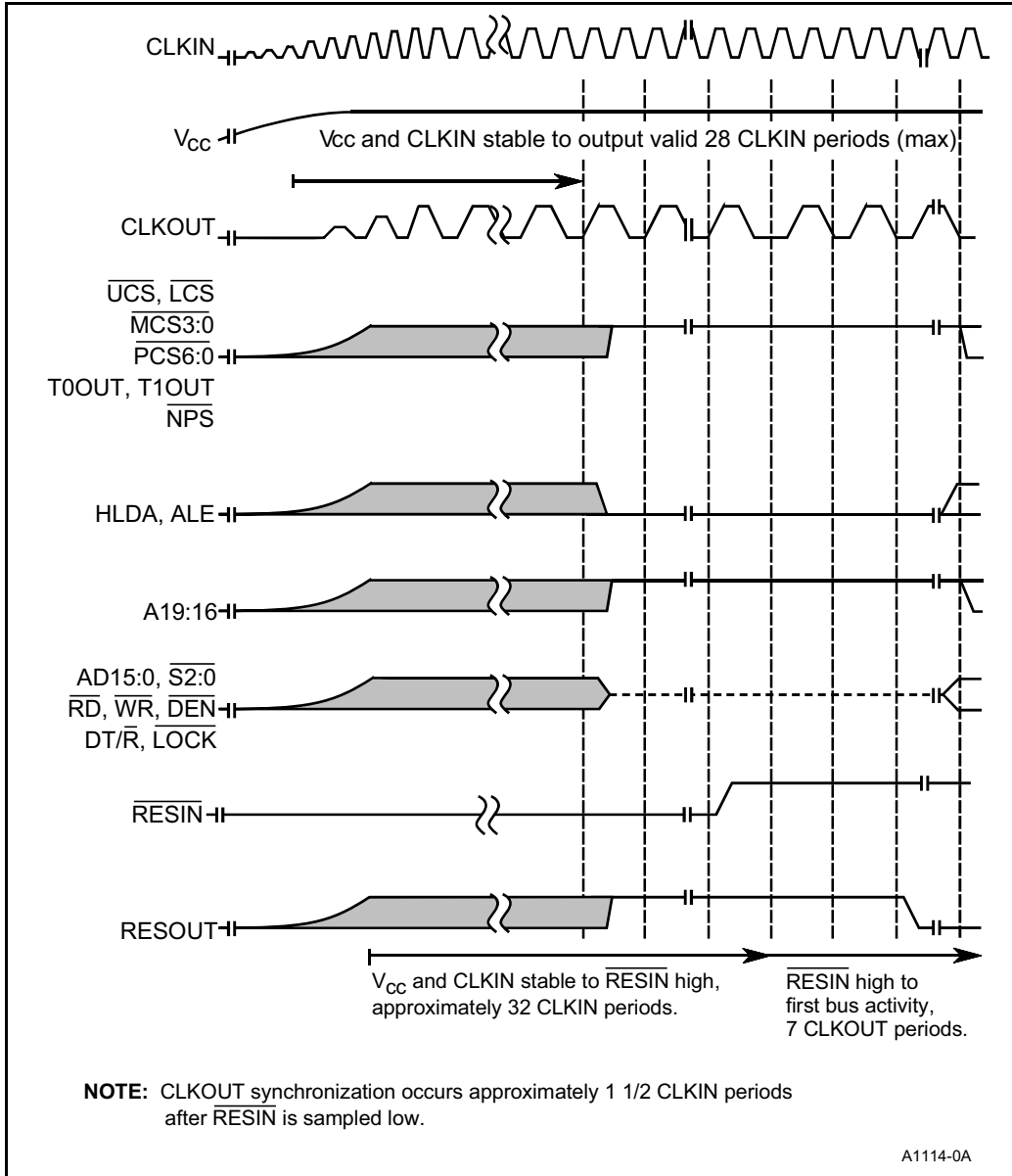


Figure 5-6. Cold Reset Waveform



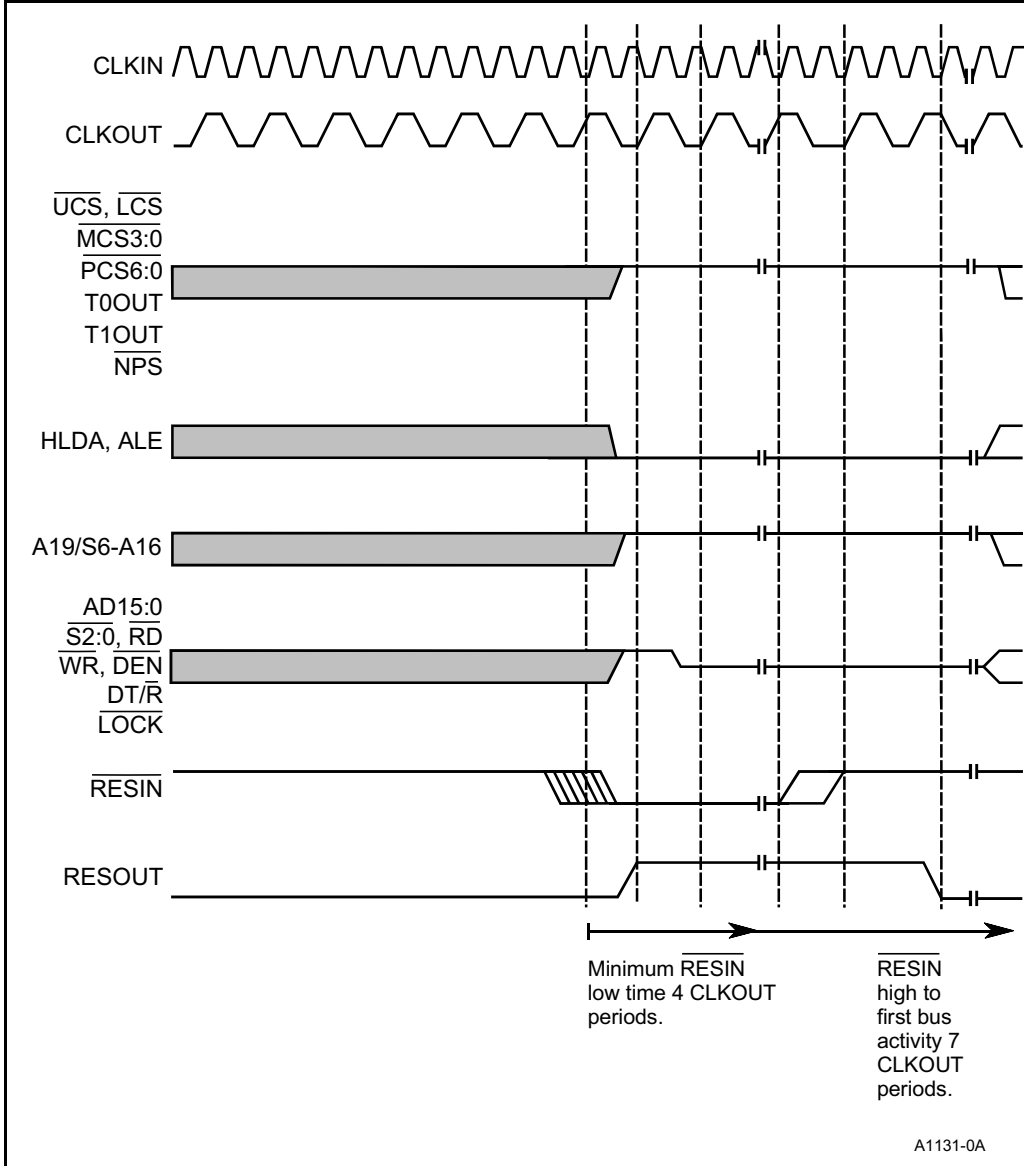


Figure 5-7. Warm Reset Waveform

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

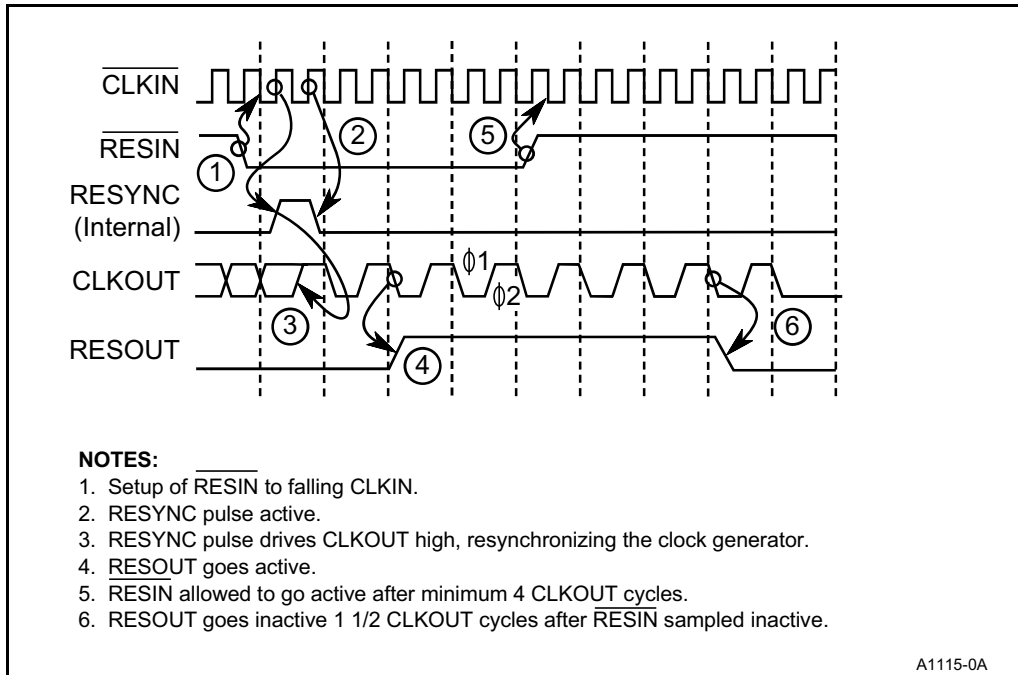


Figure 5-8. Clock Synchronization at Reset

## 5.2 POWER MANAGEMENT

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

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





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

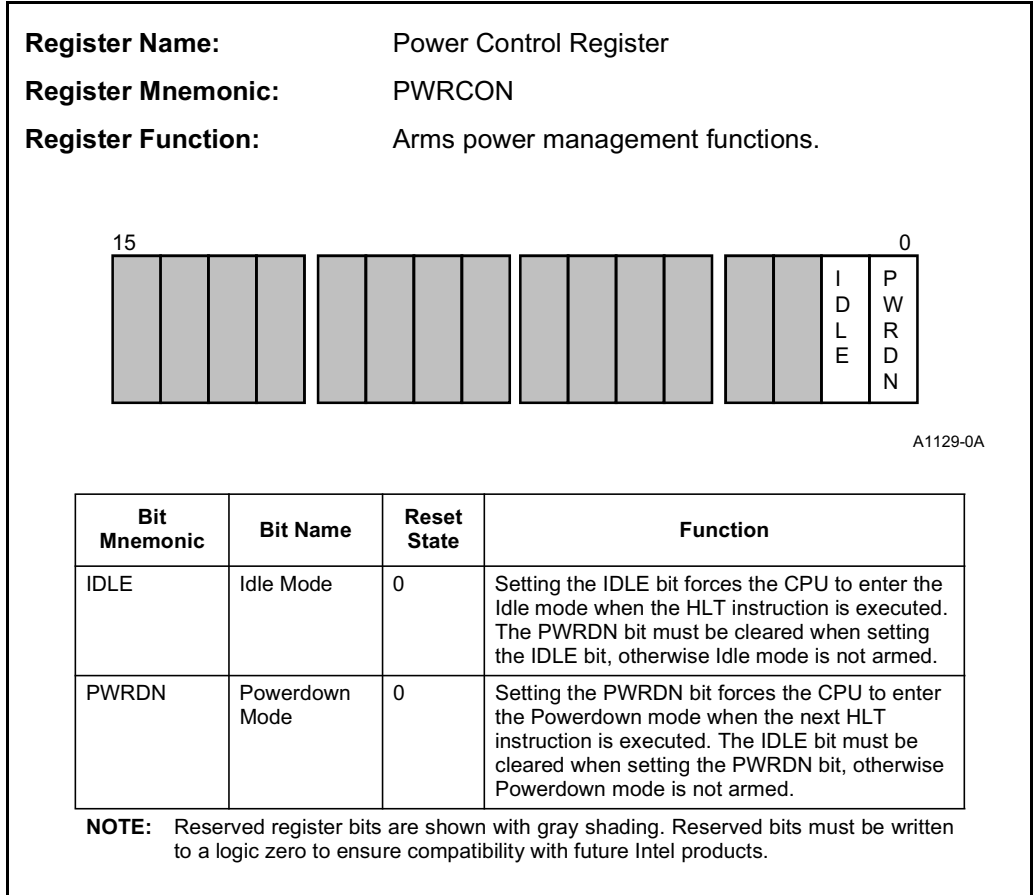
### 5.2.1 Idle Mode

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

#### 5.2.1.1 Entering Idle Mode

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





**Figure 5-9. Power Control Register**



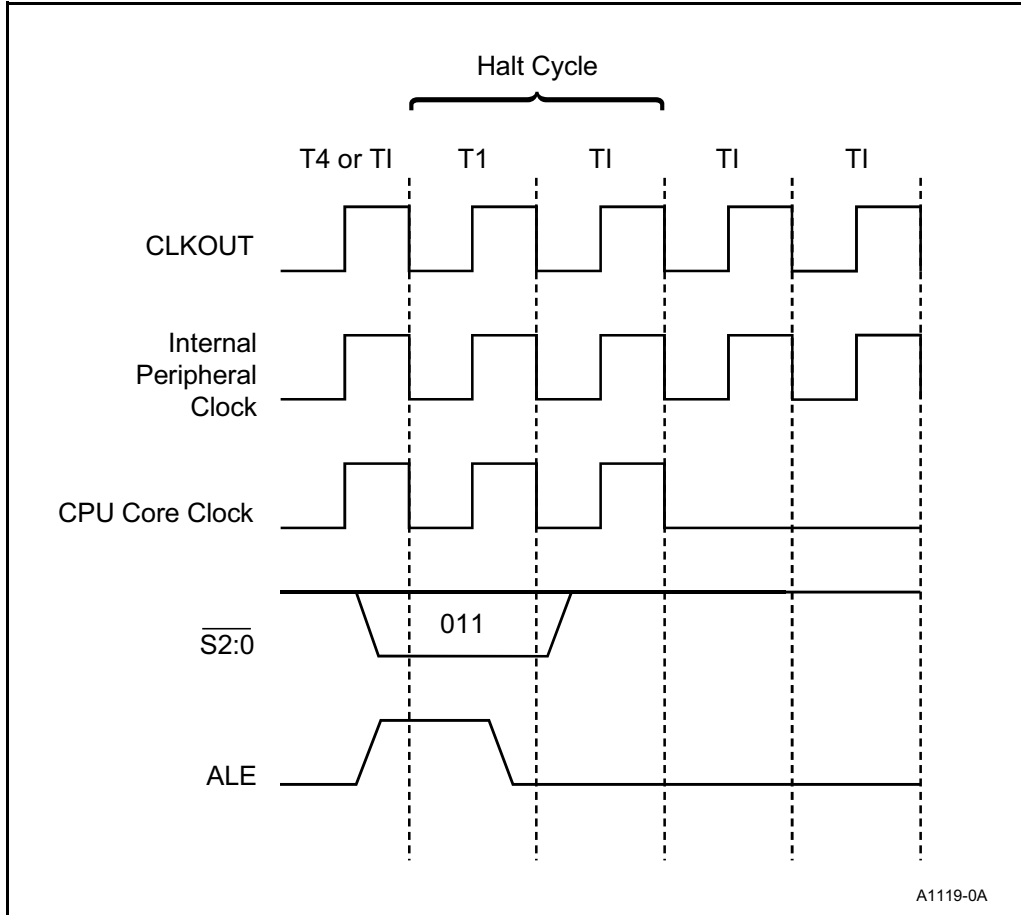


Figure 5-10. Entering Idle Mode

### 5.2.1.2 Bus Operation During Idle Mode

DMA requests, refresh requests and HOLD requests temporarily turn on the core clocks. If the processor needs to run a DMA cycle during Idle mode, the internal core clock begins to toggle on the falling CLKOUT edge three clocks after the processor samples the DMA request pin. After one idle T-state, the processor runs the DMA cycle. The BIU uses the ready, wait state generation and chip-select circuitry as necessary for DMA cycles during Idle mode. There is one idle T-state after T4 before the internal core clock shuts off again.

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

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

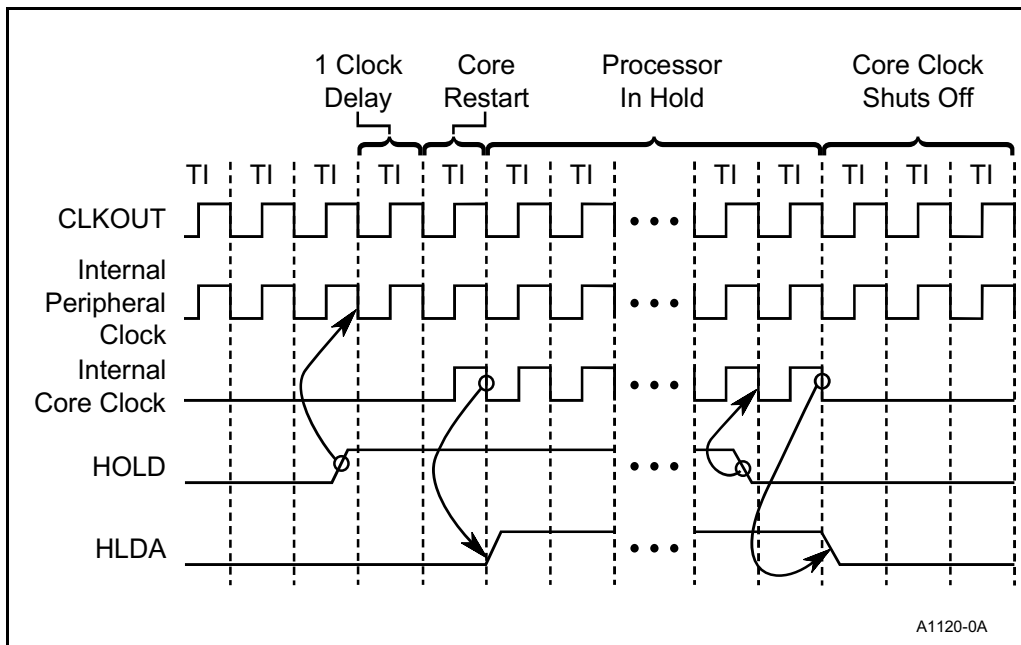


Figure 5-11. HOLD/HLDA During Idle Mode

As in Active mode, refresh requests will force the BIU to drop HLDA during bus hold. (For more information on refresh cycles during hold, see “Refresh Operation During a Bus HOLD” on page 3-43 and “Refresh Operation and Bus HOLD” on page 7-12.) Refresh requests will also correctly break into sequences of back-to-back DMA cycles.

### 5.2.1.3 Leaving Idle Mode

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





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

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

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

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

### 5.2.1.4 Example Idle Mode Initialization Code

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



```

$mod186
name          example_80C186_power_management_code

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

PWRCON equ xxxxH                                ;substitute PWRCON register
                                                ;offset

lib_80C186   segment public 'code'
             assume cs:lib_80C186
             public _power_mgt
             proc far

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

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

             _power_mgt endp
lib_80C186   ends
             end

```

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

### 5.2.2 Powerdown Mode

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

5.2.2.1 Entering Powerdown Mode

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

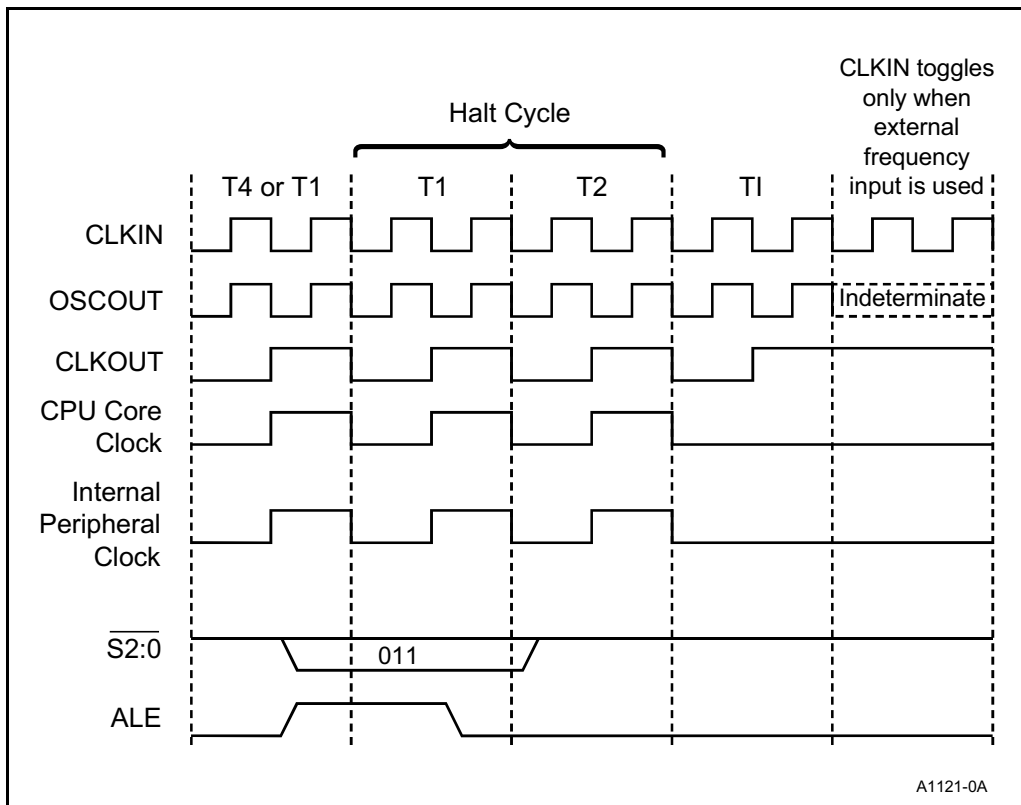


Figure 5-12. Entering Powerdown Mode

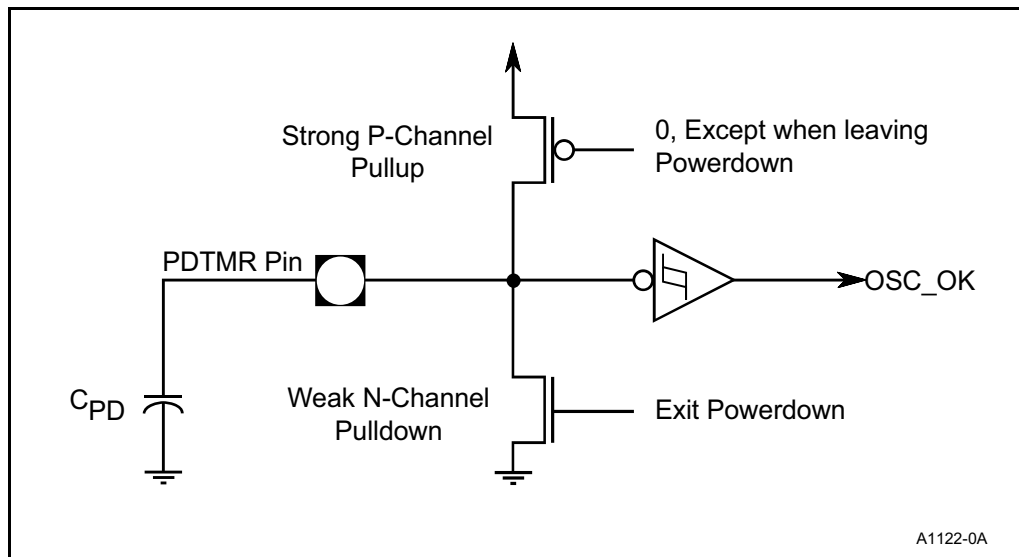
During the T2 phase of the HLT instruction, the core generates a signal called Enter\_Powerdown. Enter\_Powerdown immediately disables the internal CPU core and peripheral clocks. The processor disables the oscillator inverter during the next CLKOUT cycle. If the design uses a crystal oscillator, the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

**5.2.2.2 Leaving Powerdown Mode**

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

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

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



**Figure 5-13. Powerdown Timer Circuit**





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

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

### 5.2.3 Power-Save Mode

In addition to Idle and Powerdown modes, Power-Save mode provides another means for reducing 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 (see Figure 5-1 on page 5-1)

#### NOTE

Power-Save mode can be used to stretch bus cycles as an alternative to wait states.

Possible clock divisor settings are 1 (undivided), 4, 8 and 16. 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. Since the processor is static, a lower limit clock frequency does not apply.

The advantage of Power-Save mode over Idle and Powerdown modes is that operation of both the core and the integrated peripherals can continue. However, it may be necessary to reprogram integrated peripherals such as the Timer Counter Unit and the Refresh Control Unit to compensate for the overall reduced clock rate.

#### 5.2.3.1 Entering Power-Save Mode

The Power-Save Register (Figure 5-14) 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 T3 of the write to the Power-Save Register. CLKOUT changes simultaneously and does not glitch. Figure 5-15 illustrates the change at CLKOUT.

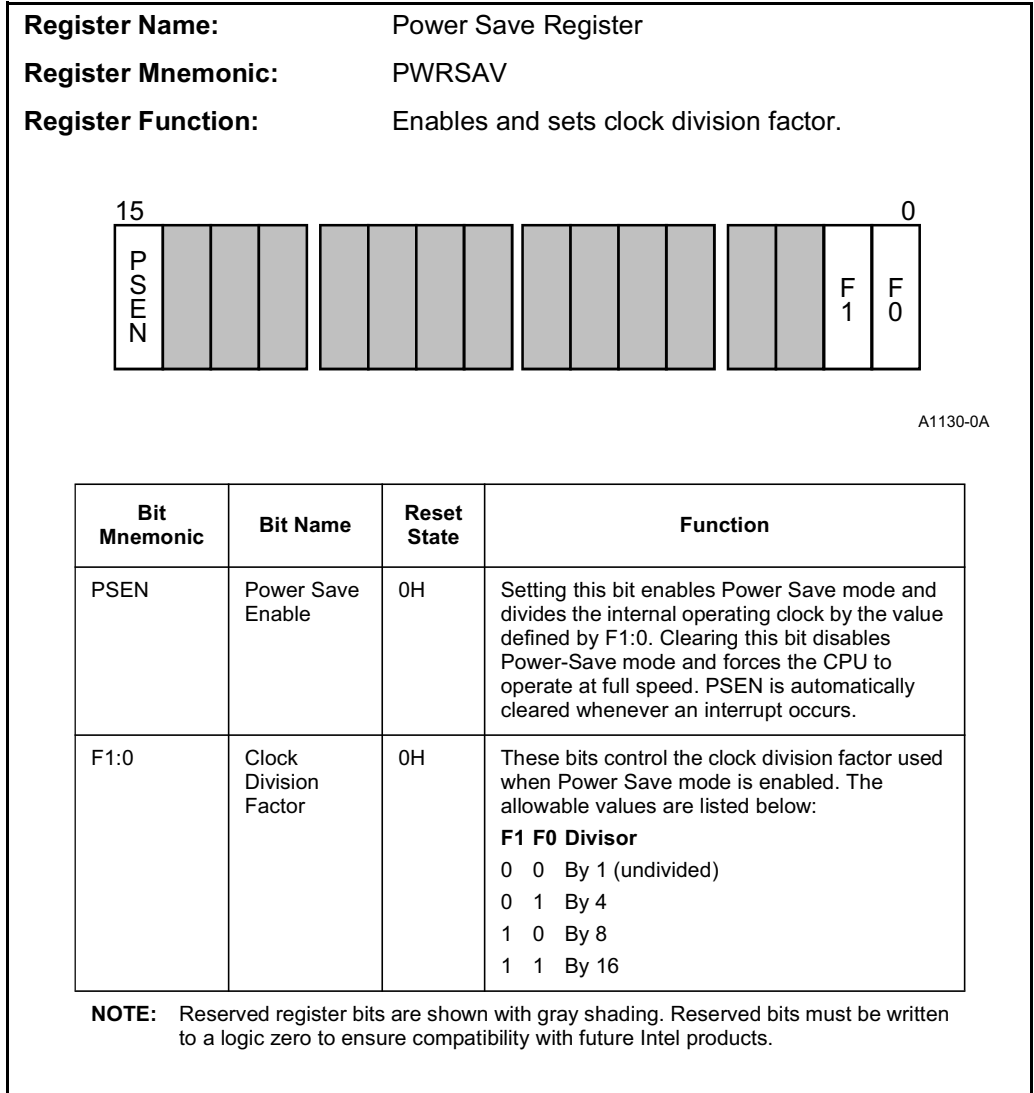


Figure 5-14. Power-Save Register



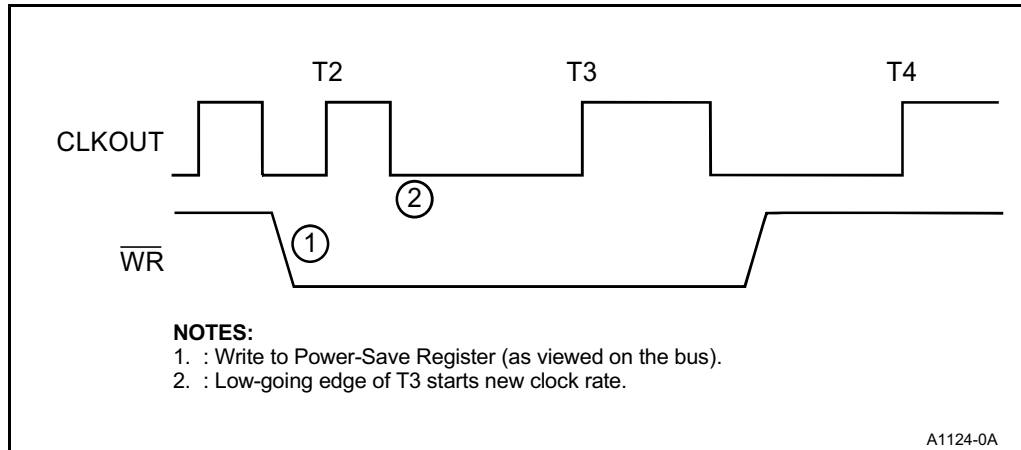


Figure 5-15. Power-Save Clock Transition

### 5.2.3.2 Leaving Power-Save Mode

Power-Save mode continues until one of three events occurs: 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 T3 edge of the write to the Power-Save Register. The same result happens from re-programming 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 T3 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.

### 5.2.3.3 Example Power-Save Initialization Code

Example 5-2 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, F1 and F2
;                  of PWRSAV.
; OUTPUTS:         None
; NOTE:           Parameters are passed on the stack as required
;                  by high-level languages

PWRSAV              equ    xxxxH                ;substitute register offset
RFTIME              equ    xxxxH                ;Power-Save Register
;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, 32, 64, 0, 0
data                ends

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

                    public _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 dx                    ;be modified

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

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

                    div FreqTable[_divisor]   ;divide refresh rate
;by _divisor
                    out dx, ax                ;set new refresh rate
                    mov dx, PWRSAV           ;select Power-Save Register
                    mov ax, _divisor          ;get divisor
                    and ax, 7                 ;mask off unwanted bits
                    or ax, PSEN               ;set enable bit
                    out dx, ax                ;divide frequency
                    pop dx                    ;restore saved registers
                    pop bx
                    pop ax
                    pop bp                    ;restore caller's bp
                    ret

                    _power_save              endp

lib_80C186          ends
end

```

**Example 5-2. Initializing the Power Management Unit for Power-Save Mode**



### 5.2.4 Implementing a Power Management Scheme

Table 5-2 summarizes the power management options available to the user. With three ways available to reduce power consumption, here are some guidelines:

- Powerdown mode reduces power consumption by several orders of magnitude. If the application goes into and out of Powerdown frequently, the power reduction can probably offset the relatively long intervals spent leaving Powerdown mode.
- If background CPU tasks are usually necessary and the overhead of reprogramming peripherals is not severe, Power-Save mode can “tune” the clock rate to the best value. Remember that current varies linearly with respect to frequency.
- Idle mode fits DMA-intensive and interrupt-intensive (as opposed to CPU-intensive) applications perfectly.

The processor can operate in Power-Save mode and Idle mode concurrently. With Idle mode alone, rated power consumption typically drops a third or more. Power-Save mode multiplies that reduction further according to the selected clock divisor.

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

**Table 5-2. Summary of Power Management Modes**

| Mode       | Relative Power  | Typical Power      | User Overhead    | Chief Advantage            |
|------------|-----------------|--------------------|------------------|----------------------------|
| Active     | Full            | 250 mW at 16 MHz   | —                | Full-speed operation       |
| Idle       | Low             | 175 mW at 16 MHz   | Low              | Peripherals are unaffected |
| Power-Save | Adjustable      | 125 mW at 16/2 MHz | Moderate to High | Code execution continues   |
| Powerdown  | Low to Moderate | Long battery life  |                  |                            |

**NOTE**

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





6

## Chip-Select Unit









## CHAPTER 6 CHIP-SELECT UNIT

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

### 6.1 COMMON METHODS FOR GENERATING CHIP-SELECTS

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

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

### 6.2 CHIP-SELECT UNIT FEATURES AND BENEFITS

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

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

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



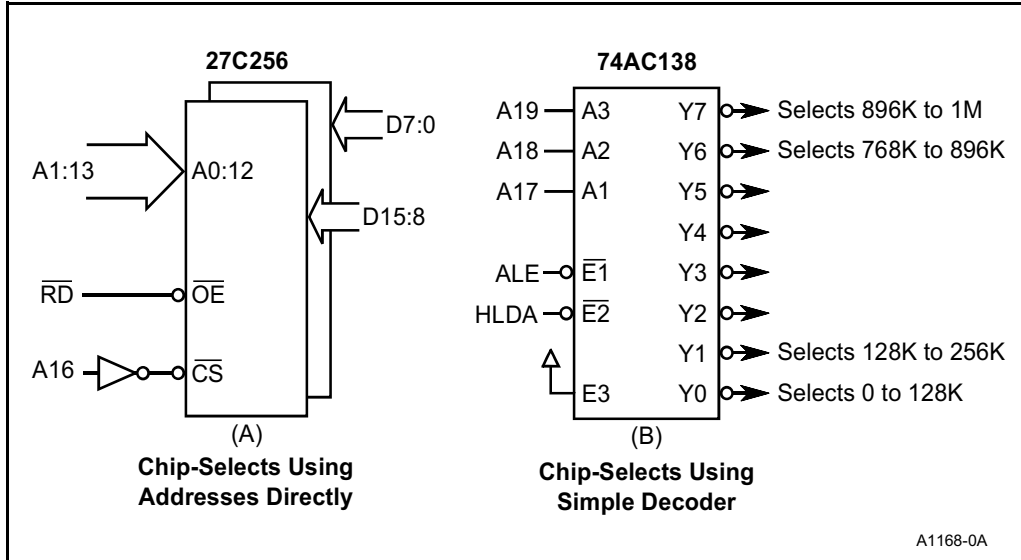


Figure 6-1. Common Chip-Select Generation Methods

### 6.3 CHIP-SELECT UNIT FUNCTIONAL OVERVIEW

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

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



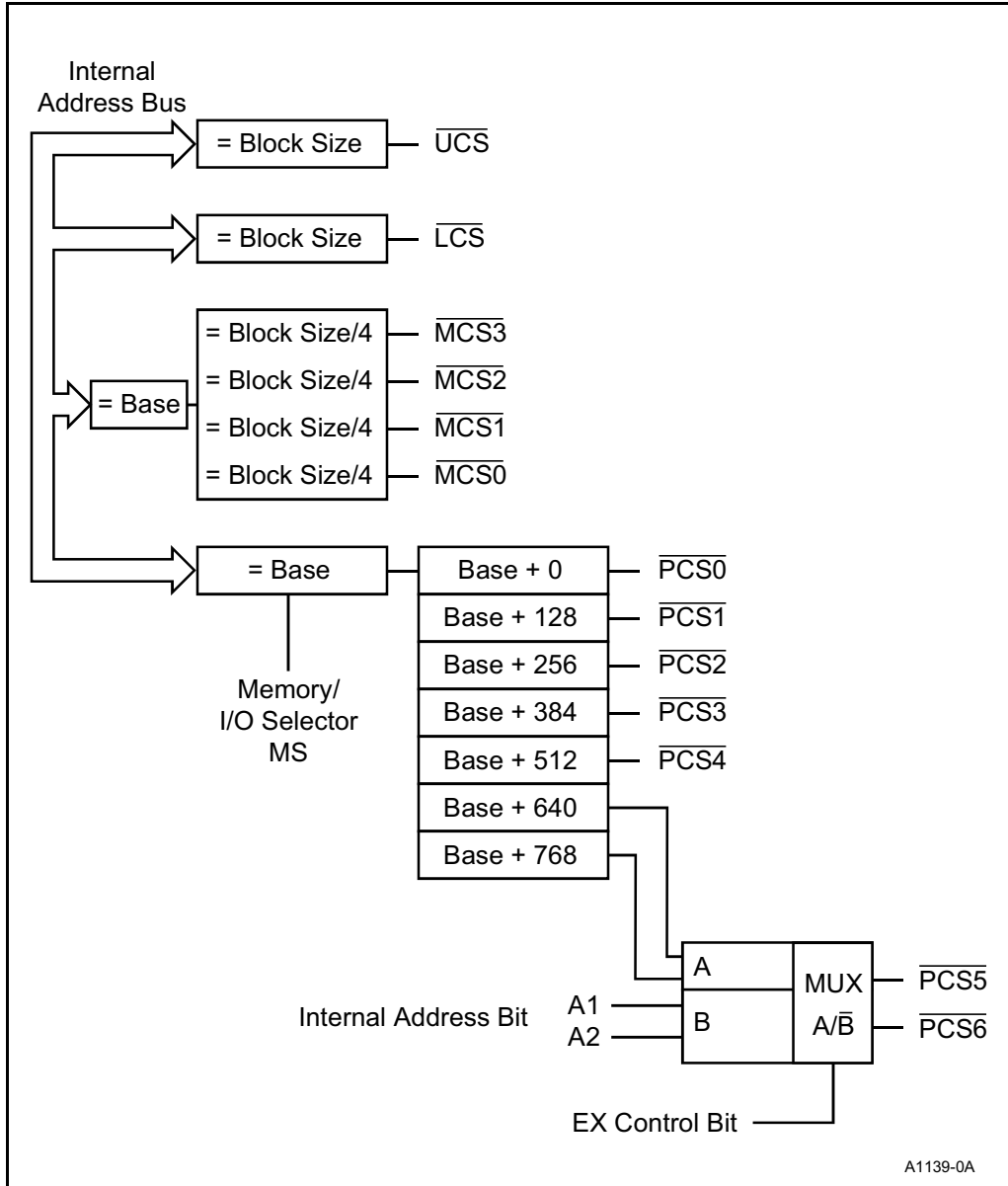
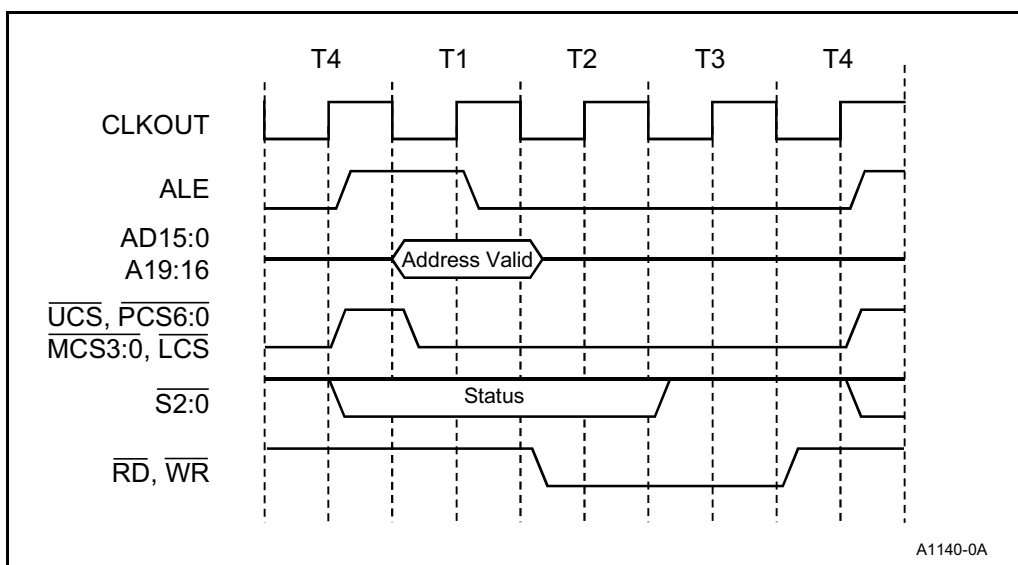


Figure 6-2. Chip-Select Block Diagram

## CHIP-SELECT UNIT

- UCS**            Mapped only to the upper memory address space; selects the BOOT memory device (EPROM or Flash memory types).
- LCS**            Mapped only to the lower memory address space; selects a static memory (SRAM) device that stores the interrupt vector table, local stack, local data, and scratch pad data.
- MCS3:0**        Mapped only to memory address space; selects additional SRAM memory, DRAM memory, or the system bus.
- PCS6:0**        Mapped to memory or I/O address space; selects peripheral devices or generates a DMA acknowledge strobe. Note that each PCS<sub>x</sub> is not individually configurable for I/O space or memory space.



**Figure 6-3. Chip-Select Relative Timings**

The  $\overline{UCS}$  chip-select always ends at address location 0FFFFH; its block size (and thus its starting address) is programmed in the UMCS register (Figure 6-5 on page 6-7). The  $\overline{LCS}$  chip-select always starts at address location 0H; its block size (and thus its ending address) is programmed in the LMCS Control register (Figure 6-6 on page 6-8). The block size can range from 1 Kbyte to 256 Kbytes for both.

The  $\overline{MCS3:0}$  chip-selects access a contiguous block of memory address space. The block size can range from 8 Kbytes to 512 Kbytes; it is programmed in the MMCS register (Figure 6-7 on page 6-9). Each chip-select goes active for one-fourth of the block. The start address is programmed in the MPCS register (Figure 6-9 on page 6-11); it must be an integer multiple of the block size. Because of the start address limitation, the  $\overline{MCS3:0}$  chip-selects cannot cover the entire memory address space between the  $\overline{LCS}$  and  $\overline{UCS}$  chip-selects.

By combining  $\overline{LCS}$ ,  $\overline{UCS}$  and  $\overline{MCS3:0}$ , you can cover up to 786 Kbytes of memory address space. Methods such as those shown in Figure 6-1 on page 6-2 can be used to decode the remaining 256 Kbytes.

The  $\overline{PCS6:0}$  chip-selects access a contiguous, 896-byte block of memory or I/O address space. Each chip-select goes active for one-seventh of the block (128 bytes). The start address is programmed in the PACS register (Figure 6-8 on page 6-10); it can begin on any 1 Kbyte boundary.

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).

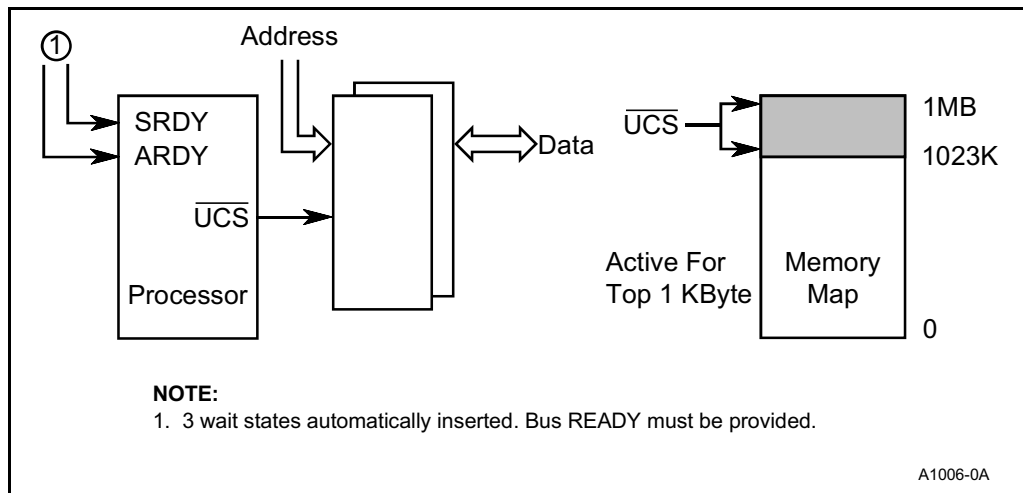


Figure 6-4.  $\overline{UCS}$  Reset Configuration

## 6.4 PROGRAMMING

Four 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 the registers and their associated programming names.

**Table 6-1. Chip-Select Unit Registers**

| Control Register Mnemonic | Alternate Register Mnemonic | Chip-Select Affected |
|---------------------------|-----------------------------|----------------------|
| UMCS                      | None                        | $\overline{UCS}$     |
| LMCS                      | None                        | $\overline{LCS}$     |
| MMCS                      | MPCS                        | $\overline{MCS3:0}$  |
| PACS                      | MPCS                        | $\overline{PCS6:0}$  |

The control registers (Figures 6-5 through 6-7) define the base address and bus ready and wait state requirements for the corresponding chip-selects. The alternate control register (Figure 6-9) defines the block size for  $\overline{MCS3:0}$ . It also selects memory or I/O space for  $\overline{PCS6:0}$ , selects the function of the  $\overline{PCS6:5}$  pins, and defines the bus ready and wait state requirements for  $\overline{PCS6:4}$ .

### 6.4.1 Initialization Sequence

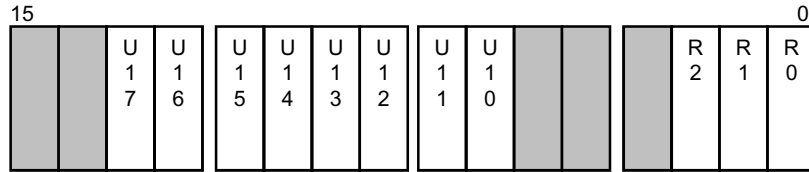
Chip-selects do not have to be initialized in any specific order. However, the following guidelines help prevent a system failure.

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

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



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



A1141-0A

| Bit Mnemonic | Bit Name          | Reset State | Function  |
|--------------|-------------------|-------------|---|
| U17:10       | Start Address     | 0FFH        | Defines the starting address for the chip-select. During memory bus cycles, U17:10 are compared with the A17:10 address bits. An equal to or greater than result enables the $\overline{UCS}$ chip-select if A19:18 are both one. Table 6-2 on page 6-12 lists the only valid programming combinations. |
| R2           | Bus Ready Disable | 0H          | When R2 is clear, bus ready must be active to complete a bus cycle. When R2 is set, R1:0 control the number of bus wait states and 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 gray shading. Reserved bits must be written to a logic zero to ensure compatibility with future Intel products. Programming U17:10 with values other than those shown in Table 6-2 on page 6-12 results in unreliable chip-select operation. Reading this register (before writing it) enables the chip-select; however, none of the programmable fields will be properly initialized.

Figure 6-5. UMCS Register Definition

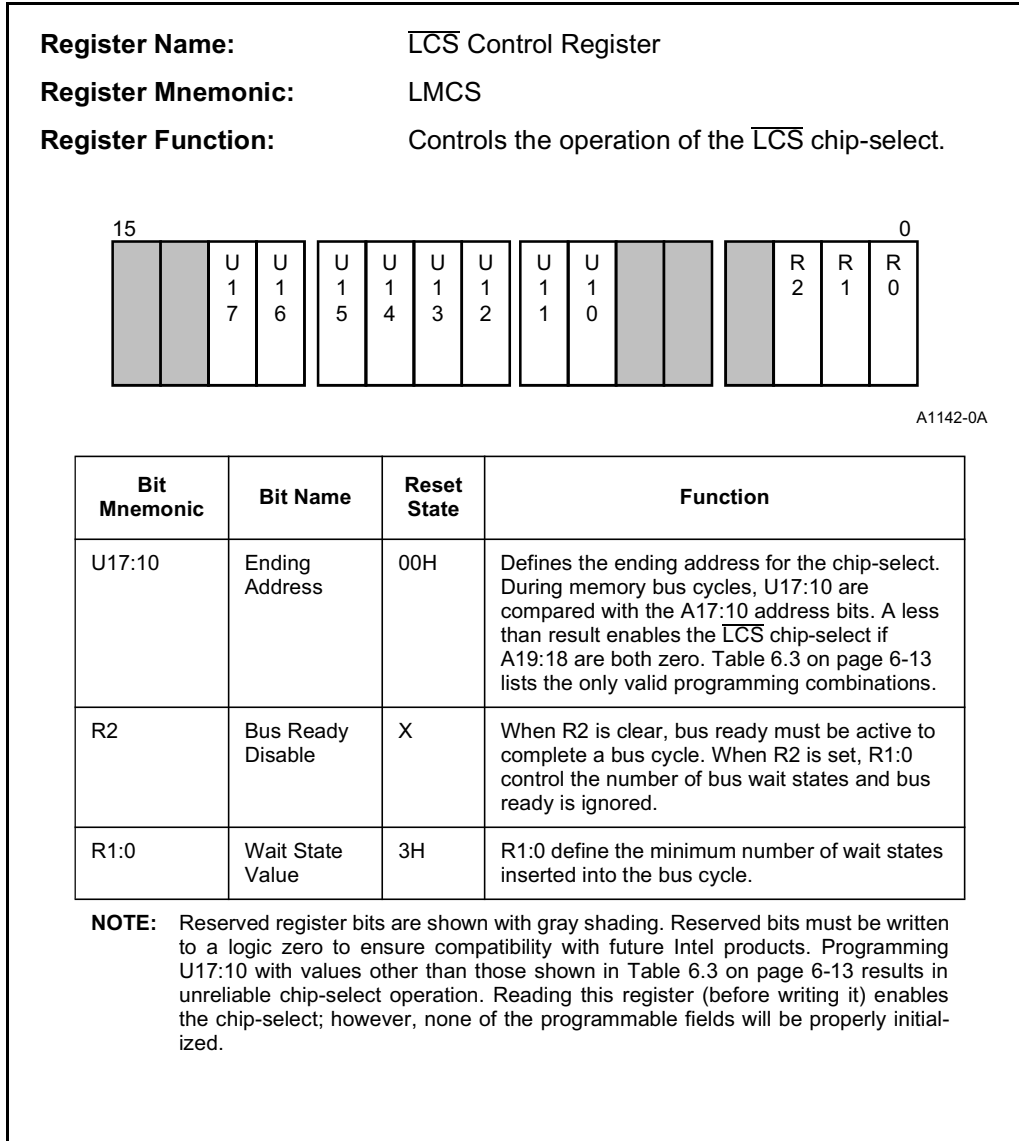
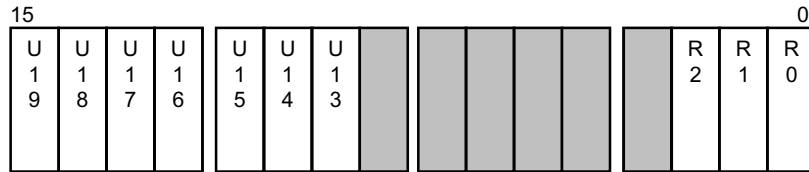


Figure 6-6. LMCS Register Definition



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



A1143-0B

| Bit Mnemonic | Bit Name          | Reset State | Function  |
|--------------|-------------------|-------------|---|
| U19:13       | Start Address     | XXH         | Defines the starting address for the block of $\overline{\text{MCS}}$ chip-selects. During memory bus cycles, U19:13 are compared with the A19:13 address bits. An equal to or greater than result enables the $\overline{\text{MCS}}$ chip-select. The starting address must be an integer multiple of the block size defined in the MPCS register. See Table 6-5 on page 6-14 for additional information. |
| R2           | Bus Ready Disable | X           | When R2 is clear, bus ready must be active to complete a bus cycle. When R2 is set, R1:0 control the number of bus wait states and 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 gray shading. Reserved bits must be written to a logic zero to ensure compatibility with future Intel products. A starting address other than an integer multiple of the block size defined in the MPCS register causes unreliable chip-select operation. (See Table 6-5 on page 6-14 for details.) Reading this register and the MPCS register (before writing them) enables the  $\overline{\text{MCS}}$  chip-selects; however, none of the programmable fields will be properly initialized.

Figure 6-7. MMCS Register Definition

CHIP-SELECT UNIT



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



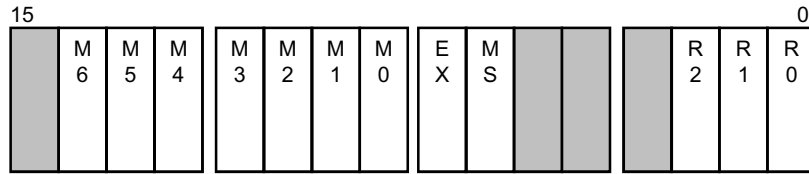
| Bit Mnemonic | Bit Name          | Reset State | Function   |
|--------------|-------------------|-------------|--|
| U19:13       | Start Address     | XXH         | Defines the starting address for the block of $\overline{\text{PCS}}$ chip-selects. During memory or I/O bus cycles, U19:13 are compared with the A19:13 address bits. An equal to or greater than result enables the $\overline{\text{PCS}}$ chip-select. U19:16 must be programmed to zero for proper I/O bus cycle operation. |
| R2           | Bus Ready Disable | X           | When R2 is clear, bus ready must be active to complete a bus cycle. When R2 is set, R1:0 control the number of bus wait states and 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 gray shading. Reserved bits must be written to a logic zero to ensure compatibility with future Intel products. U19:16 must be programmed to zero for proper I/O bus cycle operation. Reading this register and the MPCS register (before writing them) enables the  $\overline{\text{PCS}}$  chip-selects; however, none of the programmable fields will be properly initialized.

Figure 6-8. PACS Register Definition



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



A1144-0A

| Bit Mnemonic | Bit Name                                  | Reset State | Function   |
|--------------|---|-------------|--|
| M6:0         | Block Size                                | XXH         | Defines the block size for the MCS chip-selects. Table 6-5 on page 6-14 lists allowable values.  |
| EX           | Pin Selector                              | XH          | Setting EX configures $\overline{PCS6:5}$ as chip-selects. Clearing EX configures the pins as latched address bits A2:A1.  |
| MS           | Bus Cycle Selector                        | XH          | Clearing MS activates $\overline{PCS6:0}$ for I/O bus cycles. Setting MS activates $\overline{PCS6:0}$ for memory bus cycles.  |
| R2           | Bus Ready Disable for $\overline{PCS6:4}$ | X           | Applies only to $\overline{PCS6:4}$ . When R2 is clear, bus ready must be active to complete a bus cycle. When R2 is set, R1:0 control the number of bus wait states and bus ready is ignored. |
| R1:0         | Wait State Value for $\overline{PCS6:4}$  | 3H          | Apply only to $\overline{PCS6:4}$ . R1:0 define the minimum number of wait states inserted into the bus cycle.   |

**NOTE:** Reserved register bits are shown with gray shading. Reserved bits must be written to a logic zero to ensure compatibility with future Intel products. A starting address other than an integer multiple of the block size defined in this register causes unreliable chip-select operation. Reading this register and the MMCS or PACS register (before writing them) enables the associated chip-selects; however, none of the programmable fields will be properly initialized.

Figure 6-9. MPCS Register Definition

CHIP-SELECT UNIT



**6.4.2.2  $\overline{\text{LCS}}$  Active Range**

The  $\overline{\text{LCS}}$  starting address is fixed at zero in memory address space; its ending address is the programmed block size minus one. Table 6.3 defines the acceptable values for the field (U17:10) in the LMCS register that determines the  $\overline{\text{LCS}}$  block size and ending address.

**Table 6.3  $\overline{\text{LCS}}$  Active Range**

| LMCS Field U17:10 | Block Size (Kbytes) | Ending Address |
|-------------------|---------------------|----------------|
| 00H               | 1                   | 003FFH         |
| 01H               | 2                   | 007FFH         |
| 03H               | 4                   | 00FFFH         |
| 07H               | 8                   | 01FFFH         |
| 0FH               | 16                  | 03FFFH         |
| 1FH               | 32                  | 07FFFH         |
| 3FH               | 64                  | 0FFFFH         |
| 7FH               | 128                 | 1FFFFH         |
| FFH               | 256                 | 3FFFFH         |

**6.4.2.3  $\overline{\text{MCS}}$  Active Range**

The starting and ending addresses of the individual  $\overline{\text{MCS}}$  chip-selects are determined by the base address programmed in the MMCS register and the block size programmed in the MPCS register (see Table 6-4 and Figure 6-10). The base address must be an integer multiple of the block size. Table 6-5 lists the allowable block sizes and base address limitations.

**Table 6-4.  $\overline{\text{MCS}}$  Active Range**

| Chip-Select              | Active Range          |                             |
|--------------------------|-----------------------|-----------------------------|
|                          | Start Address         | Ending Address              |
| $\overline{\text{MCS0}}$ | Base                  | Base + (1/4 block size - 1) |
| $\overline{\text{MCS1}}$ | Base + 1/4 block size | Base + (1/2 block size - 1) |
| $\overline{\text{MCS2}}$ | Base + 1/2 block size | Base + (3/4 block size - 1) |
| $\overline{\text{MCS3}}$ | Base + 3/4 block size | Base + (block size - 1)     |



Table 6-5.  $\overline{MCS}$  Block Size and Start Address Restrictions

| MPCS Block Size Bits |    |    |    |    |    |    | Block Size (Kbytes) | MMCS Start Address Restrictions   |
|----------------------|----|----|----|----|----|----|---------------------|---|
| M6                   | M5 | M4 | M3 | M2 | M1 | M0 |                     |   |
| 0                    | 0  | 0  | 0  | 0  | 0  | 1  | 8                   | None  |
| 0                    | 0  | 0  | 0  | 0  | 1  | X  | 16                  | U13 must be zero.   |
| 0                    | 0  | 0  | 0  | 1  | X  | X  | 32                  | U14:13 must be zero.  |
| 0                    | 0  | 0  | 1  | X  | X  | X  | 64                  | U15:13 must be zero.  |
| 0                    | 0  | 1  | X  | X  | X  | X  | 128                 | U16:13 must be zero.  |
| 0                    | 1  | X  | X  | X  | X  | X  | 256                 | U17:13 must be zero.  |
| 1                    | X  | X  | X  | X  | X  | X  | 512                 | U18:13 must be zero.<br><b>NOTE:</b> If U19 is one, will overlap $\overline{UCS}$ . |

X= don't care, but should be 0 for future compatibility.

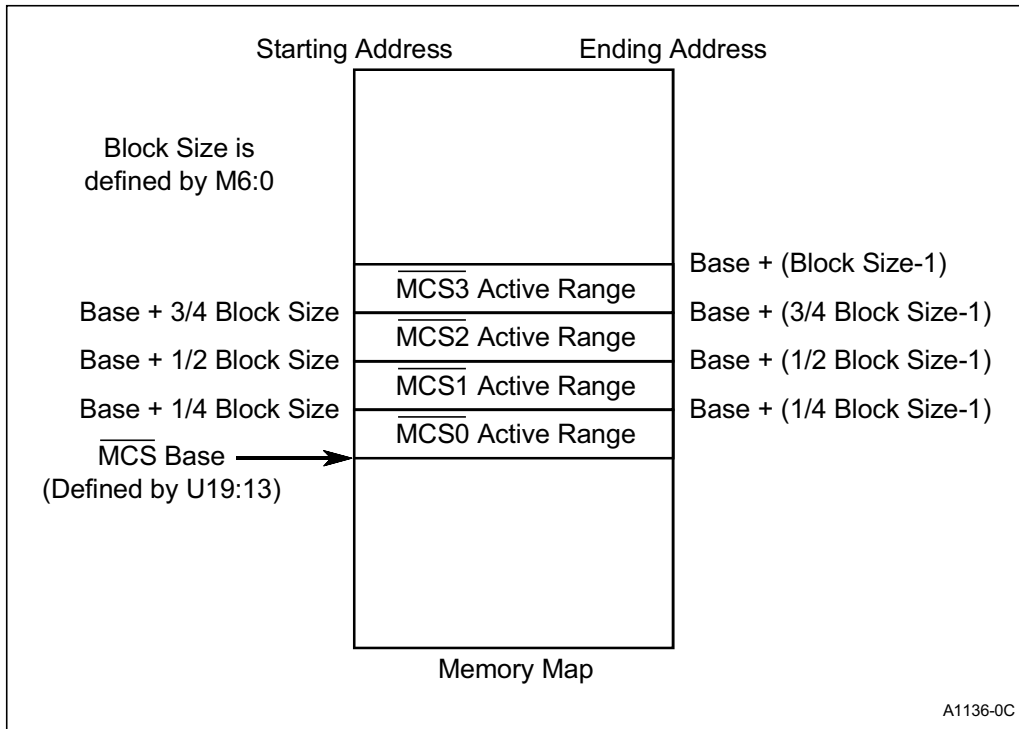


Figure 6-10.  $\overline{MCS3:0}$  Active Ranges

#### 6.4.2.4 $\overline{\text{PCS}}$ Active Range

Each  $\overline{\text{PCS}}$  chip-select starts at an offset above the base address programmed in the PACS register and is active for 128 bytes. The base address can start on any 1 Kbyte memory or I/O address location. Table 6-6 lists the active range for each  $\overline{\text{PCS}}$  chip-select.

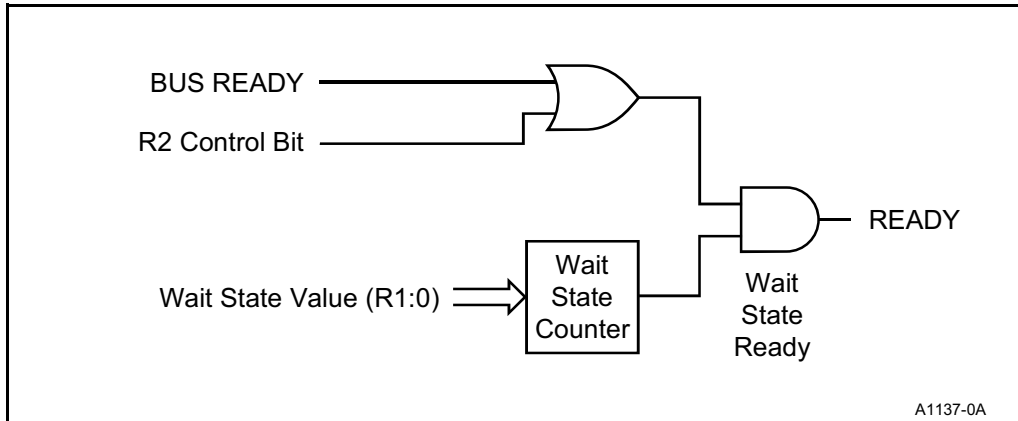
**Table 6-6.  $\overline{\text{PCS}}$  Active Range**

| Chip-Select               | Active Range      |                   |
|---------------------------|-------------------|-------------------|
|                           | Start Address     | Ending Address    |
| $\overline{\text{PCS}}_0$ | Base              | Base + 127 (7FH)  |
| $\overline{\text{PCS}}_1$ | Base + 128 (080H) | Base + 255 (0FFH) |
| $\overline{\text{PCS}}_2$ | Base + 256 (100H) | Base + 383 (17FH) |
| $\overline{\text{PCS}}_3$ | Base + 384 (180H) | Base + 511 (1FFH) |
| $\overline{\text{PCS}}_4$ | Base + 512 (200H) | Base + 639 (27FH) |
| $\overline{\text{PCS}}_5$ | Base + 640 (280H) | Base + 767 (2FFH) |
| $\overline{\text{PCS}}_6$ | Base + 768 (300H) | Base + 895 (37FH) |

#### 6.4.3 Bus Wait State and Ready Control

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

A three-bit field (R2:0) in the control registers defines the number of wait states and the ready requirements for the chip-selects. 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 R2 control bit determines whether the bus cycle completes normally (requires bus ready) or unconditionally (ignores bus ready). The R1:0 bits define the number of wait states to insert into the bus cycle. For devices requiring three or fewer wait states, you can set R2 (ignore bus ready) and program R1:0 with the number of required wait states. For devices that may require more than three wait states, you must clear R2 (require bus ready).

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

#### 6.4.4 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 overlap (i.e., if any portion of the chip-selects do not need to overlap completely to all go active). There are various reasons for overlapping chip-selects. For example, a system might have a need for overlapping a portion of read-only memory with read/write memory or copying data to two devices simultaneously.

If overlapping chip-selects do not have identical wait state and bus ready programming, the Chip-Select Unit uses the following priority scheme:

1. If any  $\overline{MCS}$  chip-select is active, it uses the R2:0 bits in the MPCS register.
2. If the  $\overline{PCS}$  chip-selects overlap  $\overline{LCS}$ , it uses the R2:0 bits in the LMCS register.
3. If the  $\overline{PCS}$  chip-selects overlap  $\overline{UCS}$ , it uses the R2:0 bits in the UMCS register.



For example, assume  $\overline{MCS3}$  overlaps  $\overline{UCS}$ .  $\overline{MCS3}$  is programmed for two wait states and requires bus ready, while  $\overline{UCS}$  is programmed for no wait states and ignores bus ready. An access to the overlapped region has two wait states and requires bus ready (the values programmed in the R2:0 bits in the MPCS register).

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

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

#### 6.4.5 Memory or I/O Bus Cycle Decoding

The  $\overline{UCS}$ ,  $\overline{LCS}$  and  $\overline{MCS}$  chip-selects activate only for memory bus cycles. The  $\overline{PCS}$  chip-selects activate for either memory or I/O bus cycles, depending on the state of the MS bit in the MPCS register (Figure 6-9 on page 6-11). Memory bus cycles consist of memory read, memory write and instruction prefetch cycles. I/O bus cycles consist of I/O read and I/O write cycles.

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

#### 6.4.6 Programming Considerations

When programming the  $\overline{PCS}$  chip-selects active for I/O bus cycles, remember that eight bytes of I/O are reserved by Intel. These eight bytes (locations 00F8H through 00FFH) control the interface to an 80C187 math coprocessor. A chip-select can overlap this reserved space provided there is no intention of using the 80C187. However, to avoid possible future compatibility issues, Intel recommends that the  $\overline{PCS}$  chip-selects not start at I/O address location 0H.

Reading or writing the chip-select registers enables the corresponding chip-select. Reading a register before writing to it enables the chip-select without initializing the programmable fields, which causes indeterminate operation. For example, reading the LMCS register enables the  $\overline{LCS}$  chip-select, but it does not ensure that  $\overline{LCS}$  is programmed correctly. Once you enable a chip-select, you cannot disable it, but you can change its operation by writing to the appropriate register.

## 6.5 CHIP-SELECTS AND BUS HOLD

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

The circuit shown in Figure 6-12 allows an external bus master to access a device during bus HOLD.

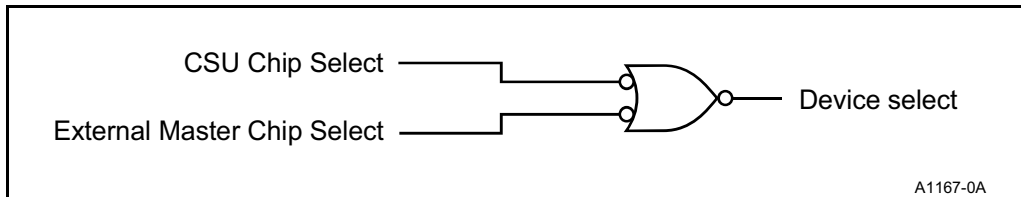


Figure 6-12. Using Chip-Selects During HOLD

## 6.6 EXAMPLES

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

### 6.6.1 Example 1: Typical System Configuration

Figure 6-13 illustrates a block diagram of a typical system design with a 128 Kbyte EPROM and a 32 Kbyte SRAM. The peripherals are mapped to I/O address space. Example 6.1 shows a program template for initializing the Chip-Select Unit.

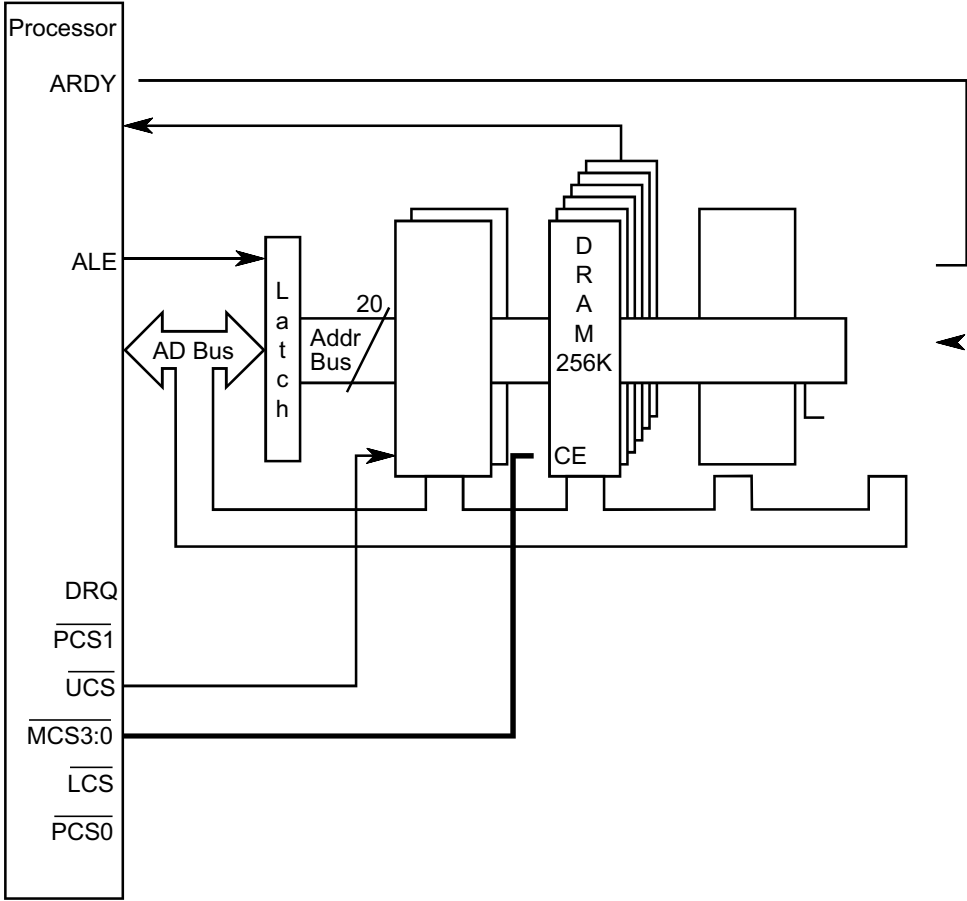


Figure 6-13. Typical System



```

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

; External reference from this module

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

; Module equates

; Configuration equates

          INTRDY EQU      0004H          ;Internal bus ready modifier
          EXTRDY EQU      0000H          ;External bus ready modifier
          IO      EQU      0080H          ;PCS Memory/IO select modifier
          ALLPCS EQU      0040H          ;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 0FFFFFF (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 1                ;Wait states

;The UMCS register values are 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)

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

          SRAM_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 equations below

          LMCS_VAL EQU  ((SRAM_SIZE - 1)SHL6) OR (0038H) OR
&          (SRAM_RDY)   OR (SRAM_WAIT)

;A DRAM interface is selected by the MCS3:0 chip-selects. The BASE value
;defines the starting address of the DRAM window. The SIZE value (along with
;the BASE ;value) defines the ending address. Zero wait state performance
;is assumed. The Refresh Control Unit uses DRAM_BASE to properly configure
;refresh operation.

```

Example 6-1. Initializing the Chip-Select Unit



```
DRAM_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 PCS# chip-selects 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)

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

IO_BASE EQU 1 ;I/O start address in Kbytes

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

PACS_VAL EQU (IO_BASE SHL 6) OR (0038H) 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
;
;Set up chip-selects.
;UCS - EPROM Select (initialized during POWER_ON code)
;LCS - SRAM Select (set to SRAM size)
;PCS - I/O Select (PCS1:0 to support floppy)
;MCS - DRAM Select (set to DRAM size)

mov dx, LMCS_REG ;set up LMCS register
mov ax, LMCS_VAL
out dx, al ;remember that byte writes are OK
```

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

```

        mov     dx, MPCS_REG           ;ready for PCS lines 4-6
        mov     ax, MPCS_VAL           ;as well as MCS programming
        out     dx, al

        mov     dx, MMCS_REG           ;set up DRAM chip-selects
        mov     ax, MMCS_VAL
        out     dx, al

        mov     dx, PACS_REG           ;set up I/O 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 system
        out     dx, al                 ;requirements
        jmp     FW_START               ;jump to initialization code
POWER_ON ENDS
;
;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. Initializing the Chip-Select Unit (Continued)



7

## Refresh Control Unit









## CHAPTER 7 REFRESH CONTROL UNIT

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

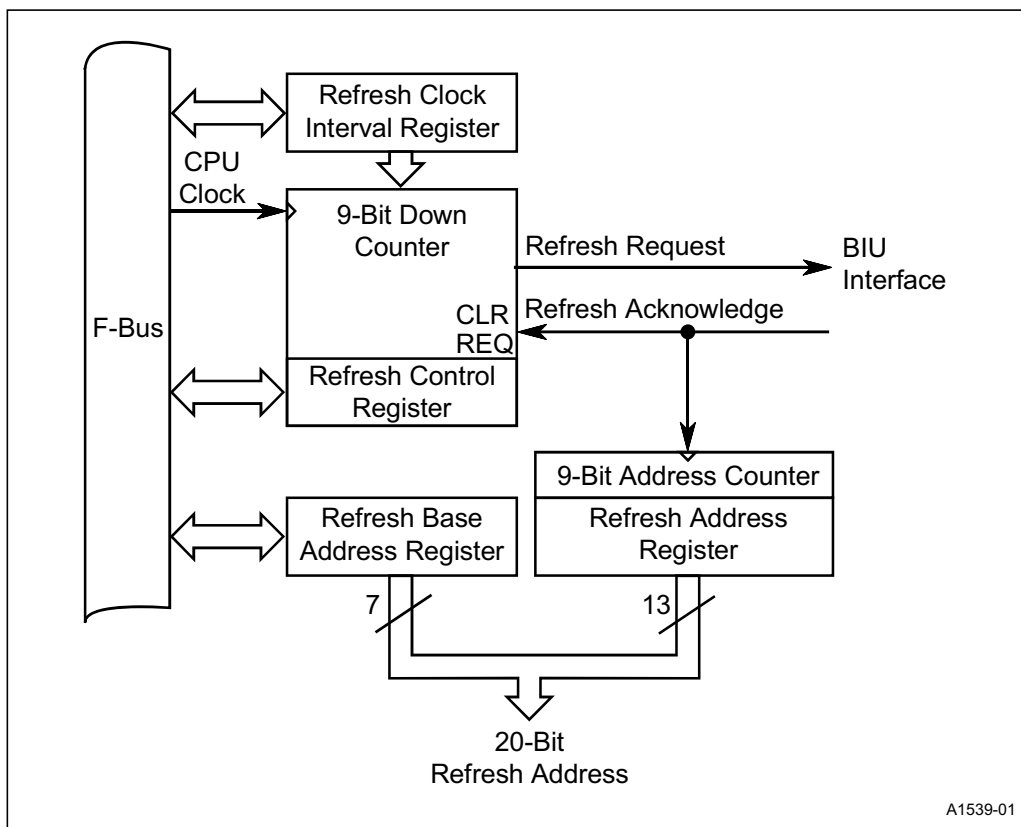


Figure 7-1. Refresh Control Unit Block Diagram

## REFRESH CONTROL UNIT

### 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 (RAS), Column Address Strobe (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 signals of pseudo-static RAMs do not need a full DRAM controller.

### 7.2 REFRESH CONTROL UNIT CAPABILITIES

A 9-bit address counter forms the refresh addresses, supporting any dynamic memory devices with up to 9 rows of memory cells (9 refresh address bits). This includes all practical DRAM sizes for the processor address space.

### 7.3 REFRESH CONTROL UNIT OPERATION

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

The nine-bit down-counter loads from the Refresh Interval Register on the falling edge of CLK-OUT. 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 “Refresh Operation and Bus HOLD” on page 7-12).



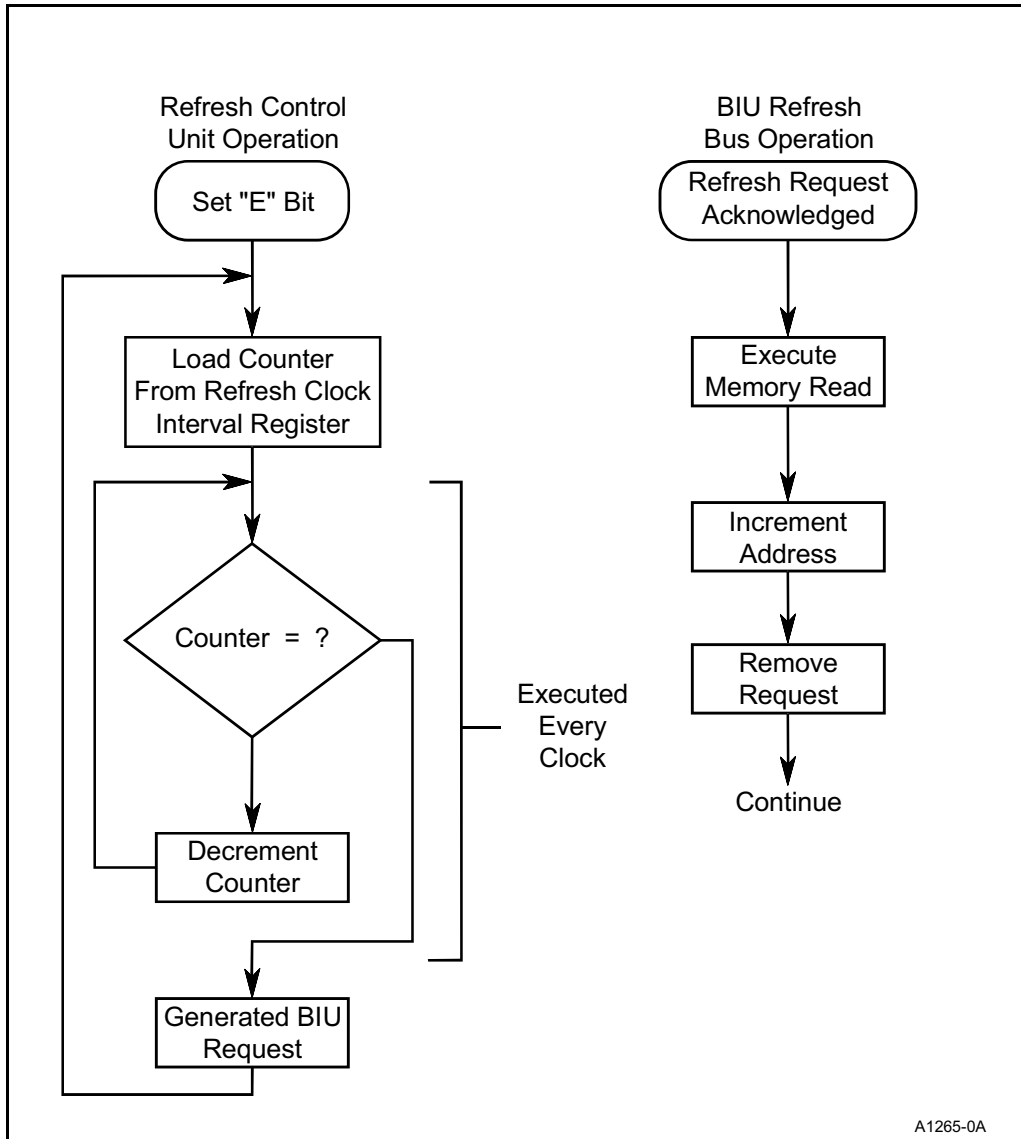


Figure 7-2. Refresh Control Unit Operation Flow Chart

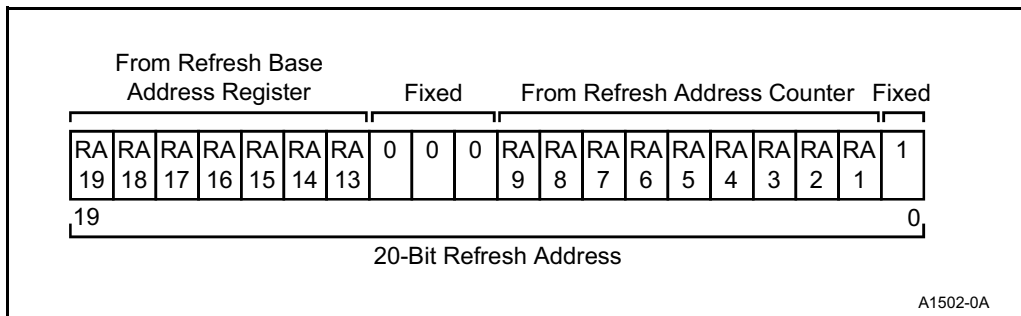
The nine-bit refresh clock counter does not wait until the BIU services the refresh request to continue counting. This operation ensures that refresh requests occur at the correct interval. Otherwise, the time between refresh requests would be a function of varying bus activity. When the BIU services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



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

### 7.4 REFRESH ADDRESSES

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



**Figure 7-3. Refresh Address Formation**

Refresh address bits RA12:10 are always zero. A linear-feedback shift counter generates address bits RA9:1 and RA0 is always one. The counter does not count linearly from 0 through 1FFH. However, the counting algorithm cycles uniquely through all possible 9-bit values. It matters only that each row of DRAM memory cells is refreshed at a specific interval. The order of the rows is unimportant.

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



### 7.5 REFRESH BUS CYCLES

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

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

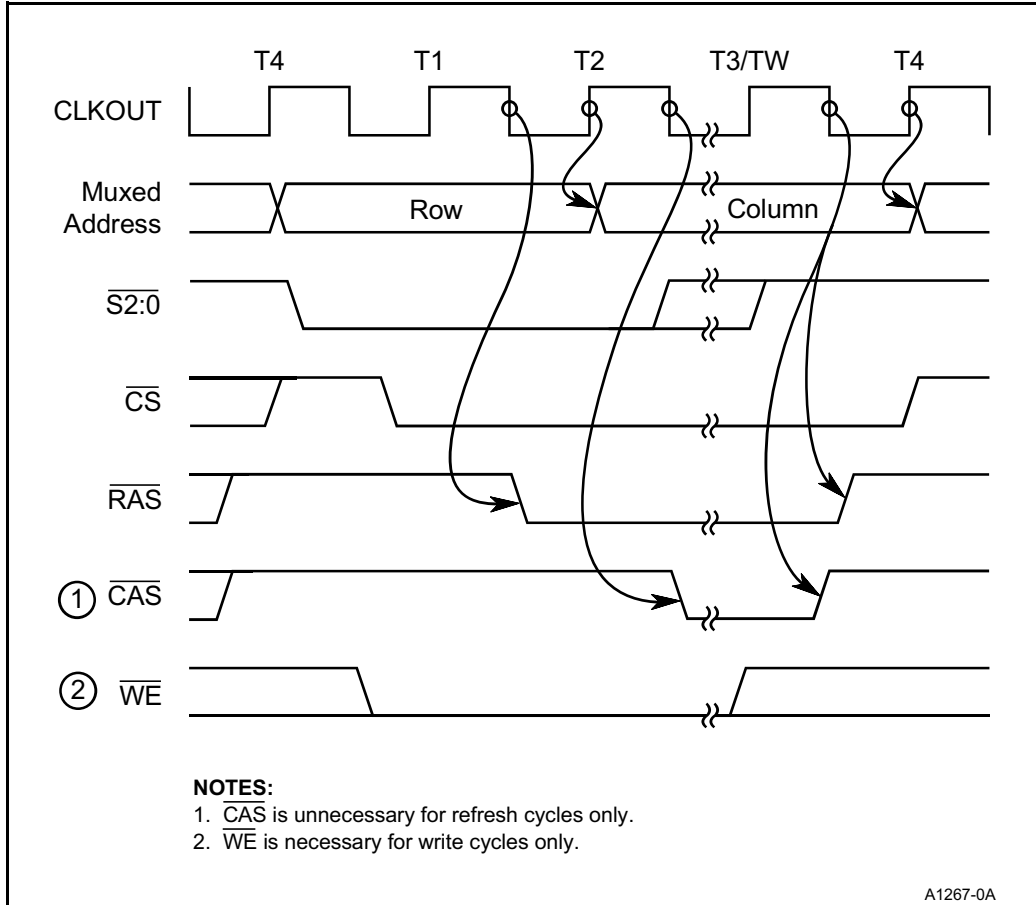
Most 80C186 Modular Core family DRAM interfaces use only this method. Others are not discussed here.

The DRAM controller uses 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). Because of the microprocessor's four-clock bus, clocking some logic elements on each CLKOUT phase is advantageous (see Figure 7-4).





**Figure 7-4. Suggested DRAM Control Signal Timing Relationships**

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

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

## 7.7 PROGRAMMING THE REFRESH CONTROL UNIT

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

### 7.7.1 Calculating the Refresh Interval

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

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

|   |  |
|---|--|
| $\frac{R_{\text{PERIOD}} \times F_{\text{CPU}}}{\text{Rows} + (\text{Rows} \times \text{Overhead}\%)} = \text{RFTIME Register Value}$ |  |
| $R_{\text{PERIOD}}$   | = Maximum refresh period specified by DRAM manufacturer (in $\mu\text{s}$ ).   |
| $F_{\text{CPU}}$  | = Operating frequency (in MHz).  |
| Rows  | = Total number of rows to be refreshed.  |
| Overhead %  | = Derating factor to compensate for missed refresh requests (typically 1–5 %). |

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

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 (Figure 7-6) programs the base (upper seven bits) of the refresh address. Seven-bit mapping places the refresh address at any 4 Kbyte boundary within the 1 Mbyte address space. When the partial refresh address from the 9-bit address counter (see Figure 7-1 and “Refresh Control Unit Capabilities” on page 7-2) passes 1FFH, the Refresh Control Unit does **not** increment the refresh base address. Setting the base address ensures that the address driven during a refresh bus cycle activates the DRAM chip select.

Figure 7-6. Refresh Base Address Register

### 7.7.2.2 Refresh Clock Interval Register

The Refresh Clock Interval Register (Figure 7-7) defines the time between refresh requests. The higher the value, the longer the time between requests. The down-counter decrements every falling CLKOUT edge, regardless of core activity. When the counter reaches one, the Refresh Control Unit generates a refresh request, and the counter reloads the value from the register. Since Power-Save mode divides the clock to the Refresh Control Unit, this register will require reprogramming if Power-Save mode is used.







**Register Name:** Refresh Clock Interval Register  
**Register Mnemonic:** RFTIME  
**Register Function:** Sets refresh rate.

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

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

**Figure 7-7. Refresh Clock Interval Register**

**7.7.2.3 Refresh Control Register**

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



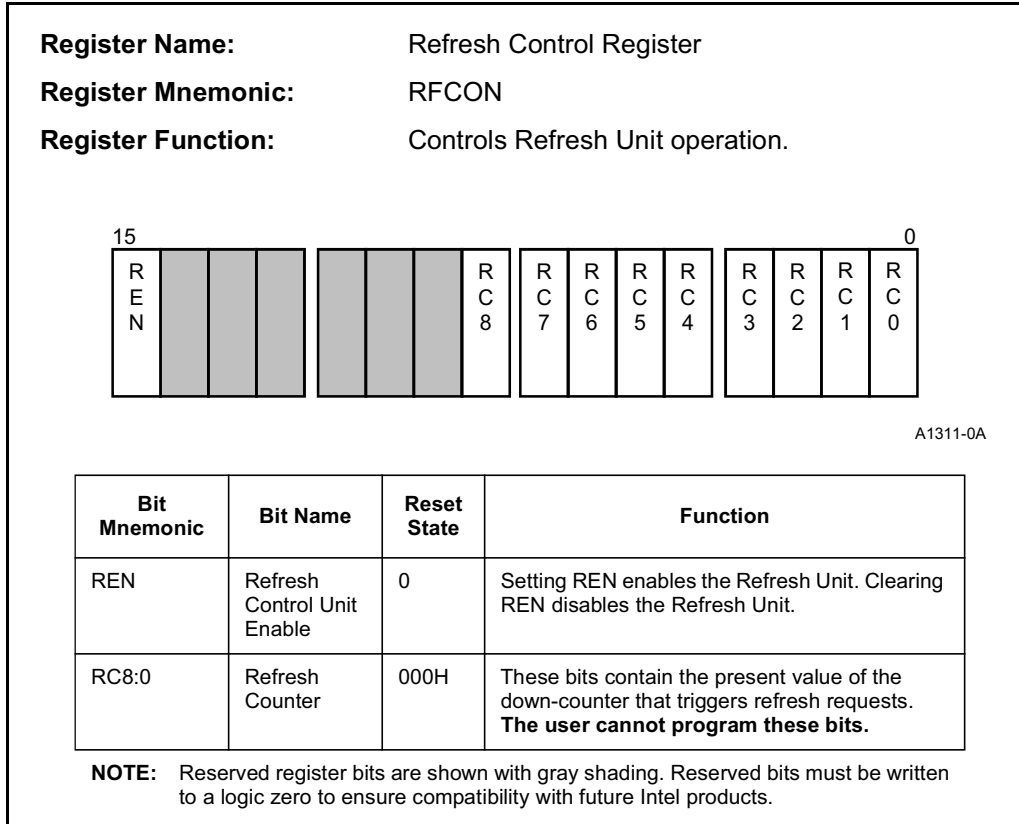


Figure 7-8. Refresh Control Register

### 7.7.3 Programming Example

Example 7-1 contains sample code to initialize the Refresh Control Unit. Example 5-2 on page 5-22 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        equ  xxxxxh                ;substitute register offset
RFTIME        equ  xxxxxh
RFCON         equ  xxxxxh
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

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

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

```

**Example 7-1. Initializing the Refresh Control Unit**

```

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

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

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

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

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

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

        ret
_config_rcu   endp
lib_80186    ends
end

```

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

## 7.8 REFRESH OPERATION AND BUS HOLD

When another bus master controls the bus, the processor keeps  $\overline{\text{HLDA}}$  active as long as the HOLD input remains active. If the Refresh Control Unit generates a refresh request during bus hold, the processor drives the  $\overline{\text{HLDA}}$  signal inactive, indicating to the current bus master that it wishes to regain bus control (see Figure 7-9). The BIU begins a refresh bus cycle only after the alternate master removes HOLD. The user must design the system so that the processor can regain bus control. If the alternate master asserts HOLD after the processor starts the refresh cycle, the CPU will relinquish control by asserting  $\overline{\text{HLDA}}$  when the refresh cycle is complete.

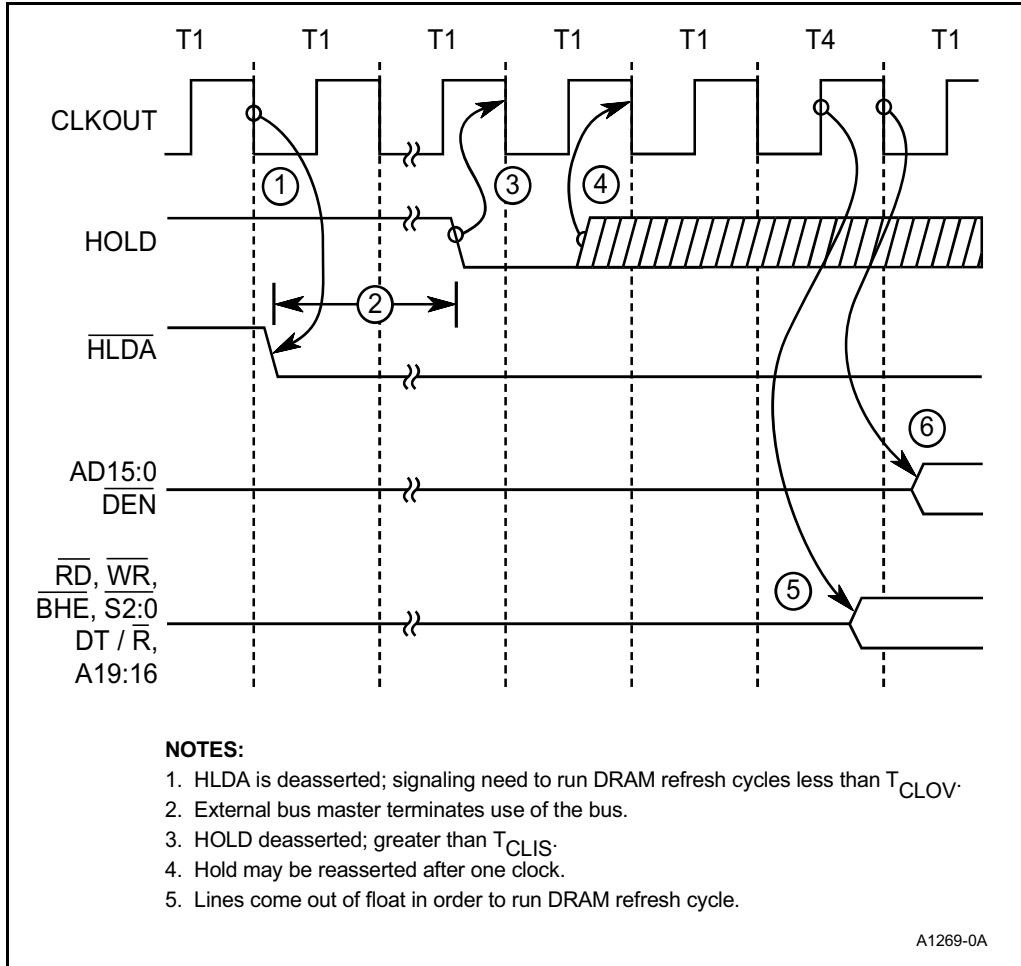


Figure 7-9. Regaining Bus Control to Run a DRAM Refresh Bus Cycle





8

# Interrupt Control Unit









## CHAPTER 8 INTERRUPT CONTROL UNIT

The 80C186 Modular Core has a single maskable interrupt input. (See “Interrupts and Exception Handling” on page 2-39.) The Interrupt Control Unit (ICU) expands the interrupt capabilities beyond a single input. To fulfill this function, the Interrupt Control Unit operates in either of two modes: Master or Slave.

In Master mode, the ICU controls the maskable interrupt input to the CPU. Interrupts can originate from the on-chip peripherals and from four external interrupt pins. The ICU synchronizes and prioritizes all interrupt sources and presents the correct interrupt type vector to the CPU. (See Figure 8-1.) Most systems use master mode.

In Slave mode, an external 8259A module controls the maskable interrupt input to the CPU and acts as the master interrupt controller. The ICU processes only those interrupts from the on-chip peripherals and acts as an interrupt input to the 8259A. (See Figure 8-15 on page 8-24.) This mode can be useful in larger system designs.

The Interrupt Control Unit has the following features:

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

### 8.1 FUNCTIONAL OVERVIEW

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

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



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

### 8.2 MASTER MODE

Figure 8-1 shows a block diagram of the Interrupt Control Unit in Master mode. In this mode, the ICU processes all interrupt requests, both external and internal. The three timer interrupt requests share a single input, while the others are supported directly.

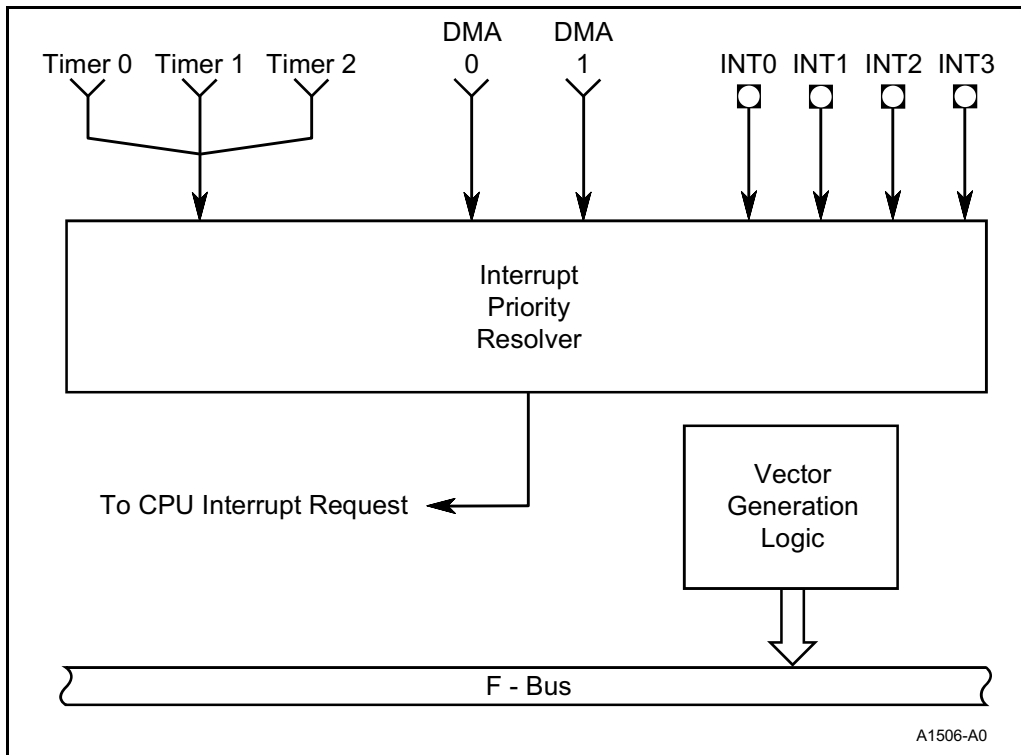


Figure 8-1. Interrupt Control Unit in Master Mode

#### 8.2.1 Generic Functions in Master Mode

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



**8.2.1.1 Interrupt Masking**

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

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

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

**8.2.1.2 Interrupt Priority**

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

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

**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                 |

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

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

### 8.2.1.3 Interrupt Nesting

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

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

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

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

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





## INTERRUPT CONTROL UNIT

## INTERRUPT CONTROL UNIT

### 8.3.2.1 Priority Resolution Example

This example illustrates priority resolution. Assume these initial conditions:

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

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

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



### 8.3.2.2 Interrupts That Share a Single Source

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

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

When the shared interrupt is acknowledged, the timer interrupt with the highest priority (see Table 8-1 on page 8-3) **at that time** is serviced, and its Interrupt Status bit is cleared.

If no other timer Interrupt Status bits are set, the shared Interrupt Request bit is also cleared. If other timer interrupts are pending, the Interrupt Request bit remains set.

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

### 8.3.3 Cascading with External 8259As

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

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

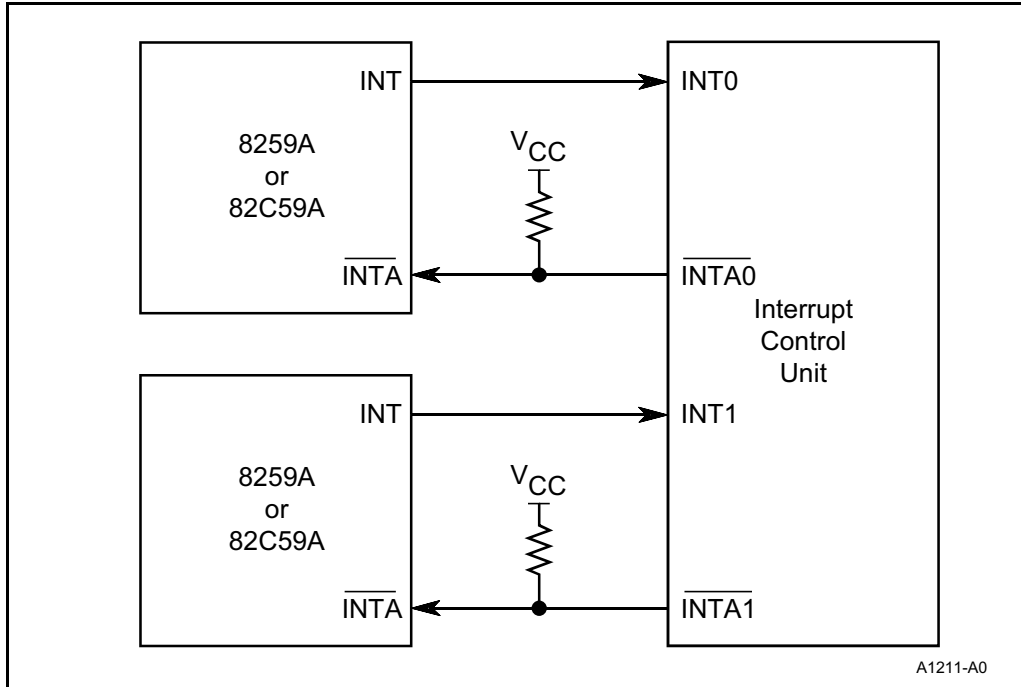


Figure 8-2. Using External 8259A Modules in Cascade Mode

8.3.3.1 Special Fully Nested Mode

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

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

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







## INTERRUPT CONTROL UNIT

### 8.3.6 Edge and Level Triggering

The external interrupts (INT3:0) can be programmed for either edge or level triggering (see “Interrupt Control Registers” on page 8-12). Both types of triggering are active high. An edge-triggered interrupt is generated by a zero-to-one transition on an external interrupt pin. The pin must remain high until after the CPU acknowledges the interrupt, then must go low to reset the edge-detection circuitry. (See the current data sheet for timing requirements.) The edge-detection circuitry must be reset to enable further interrupts to occur.

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

#### NOTE

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

### 8.3.7 Additional Latency and Response Time

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



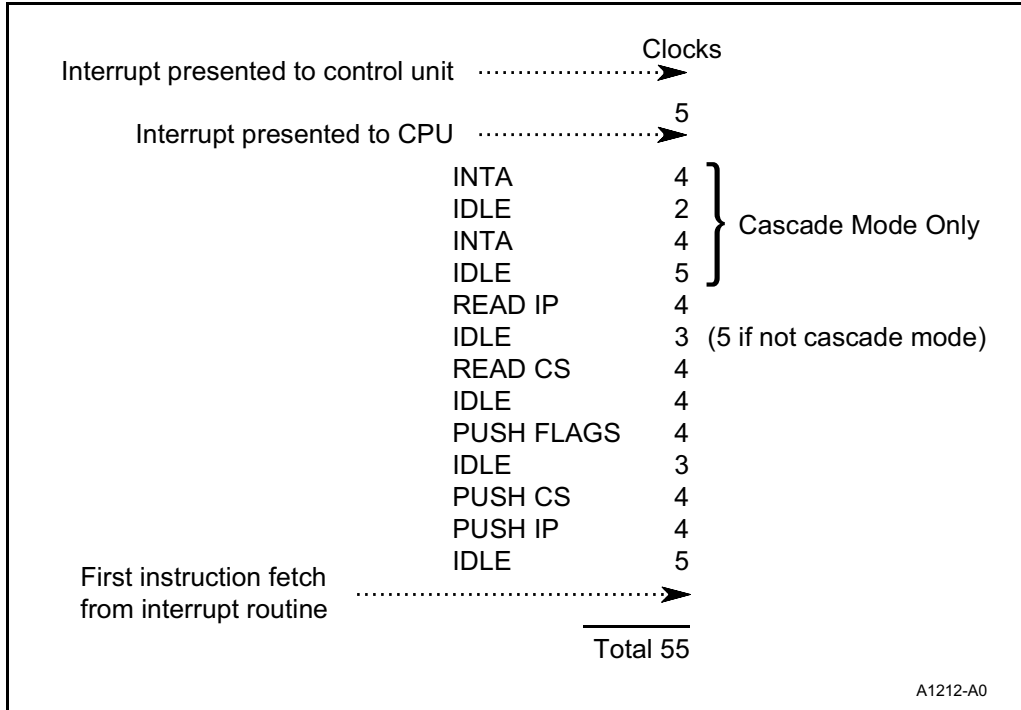


Figure 8-3. Interrupt Control Unit Latency and Response Time

### 8.4 PROGRAMMING THE INTERRUPT CONTROL UNIT

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

Table 8-3. Interrupt Control Unit Registers in Master Mode

| Register Name     | Offset Address |
|-------------------|----------------|
| INT3 Control      | 3EH            |
| INT2 Control      | 3CH            |
| INT1 Control      | 3AH            |
| INT0 Control      | 38H            |
| DMA0 Control      | 34H            |
| DMA1 Control      | 36H            |
| Timer Control     | 32H            |
| Interrupt Status  | 30H            |
| Interrupt Request | 2EH            |

**Table 8-3. Interrupt Control Unit Registers in Master Mode (Continued)**

| Register Name  | Offset Address |
|----------------|----------------|
| In-Service     | 2CH            |
| Priority Mask  | 2AH            |
| Interrupt Mask | 28H            |
| Poll Status    | 26H            |
| Poll           | 24H            |
| EOI            | 22H            |

### 8.4.1 Interrupt Control Registers

Each interrupt source has its own Interrupt Control register. The Interrupt Control register allows you to define the behavior of each interrupt source. Figure 8-4 shows the registers for the timers and DMA channels, Figure 8-5 shows the registers for INT3:2, and Figure 8-6 shows the registers for INT0 and INT1.

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

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

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



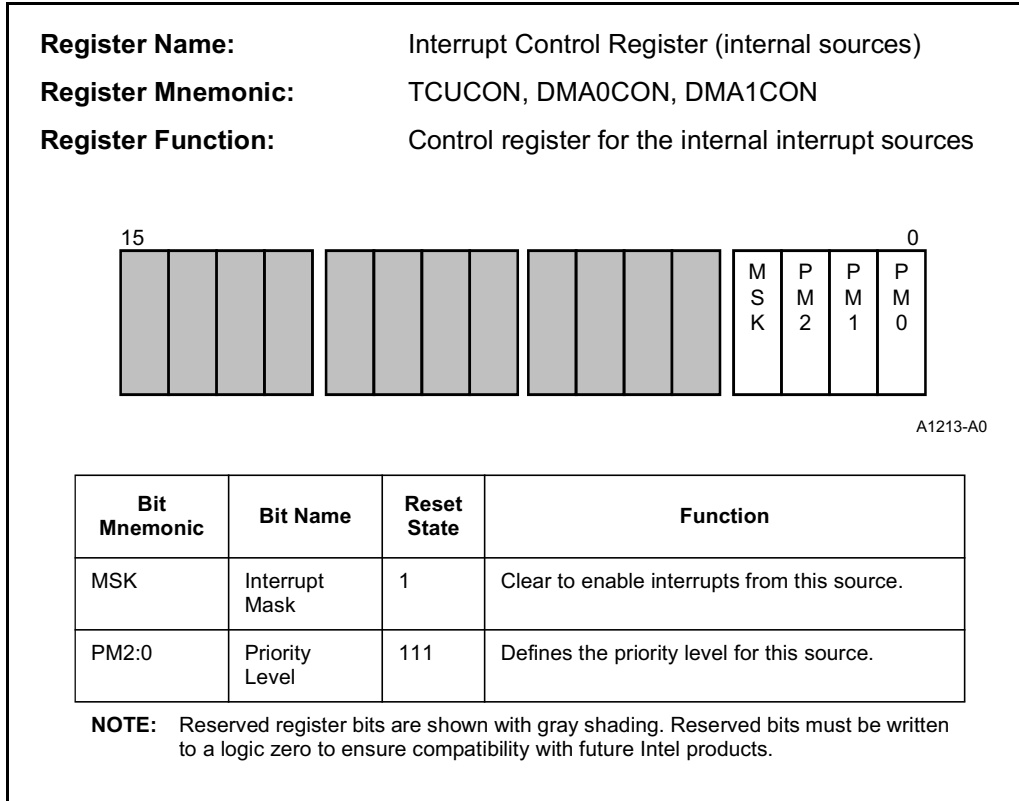
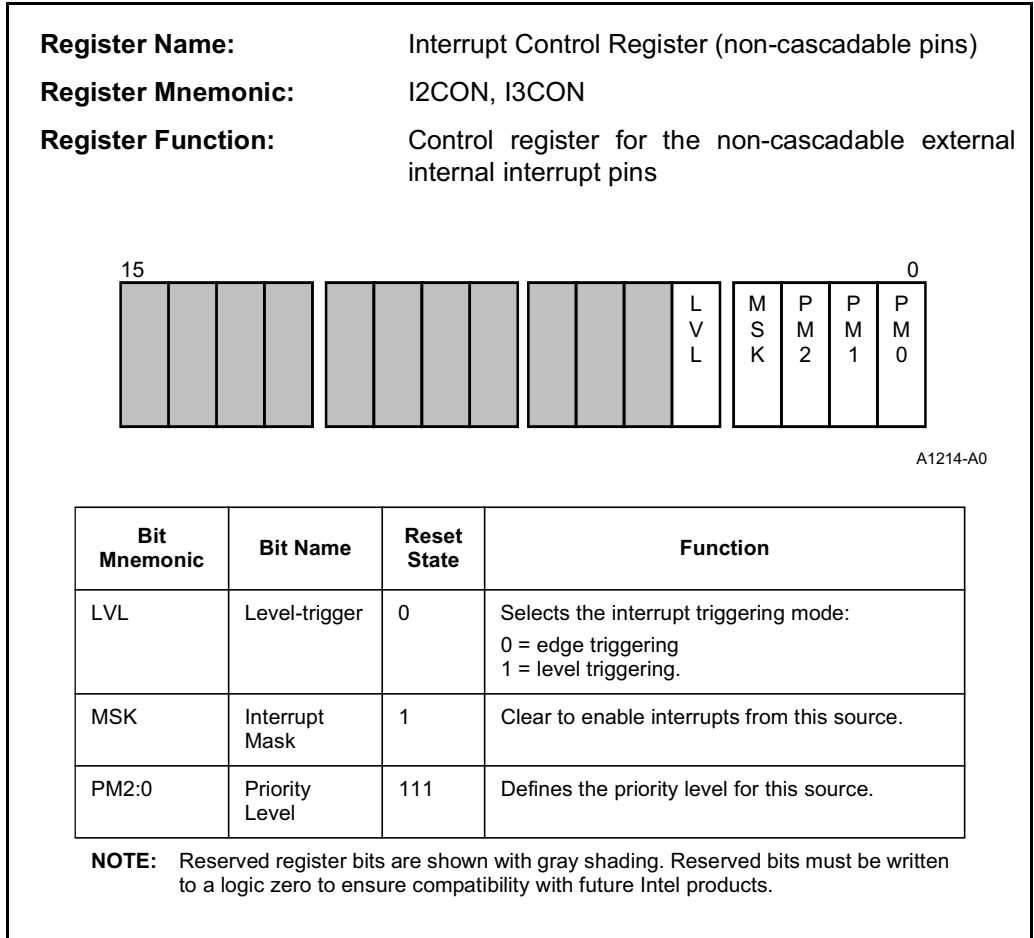


Figure 8-4. Interrupt Control Register for Internal Sources



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



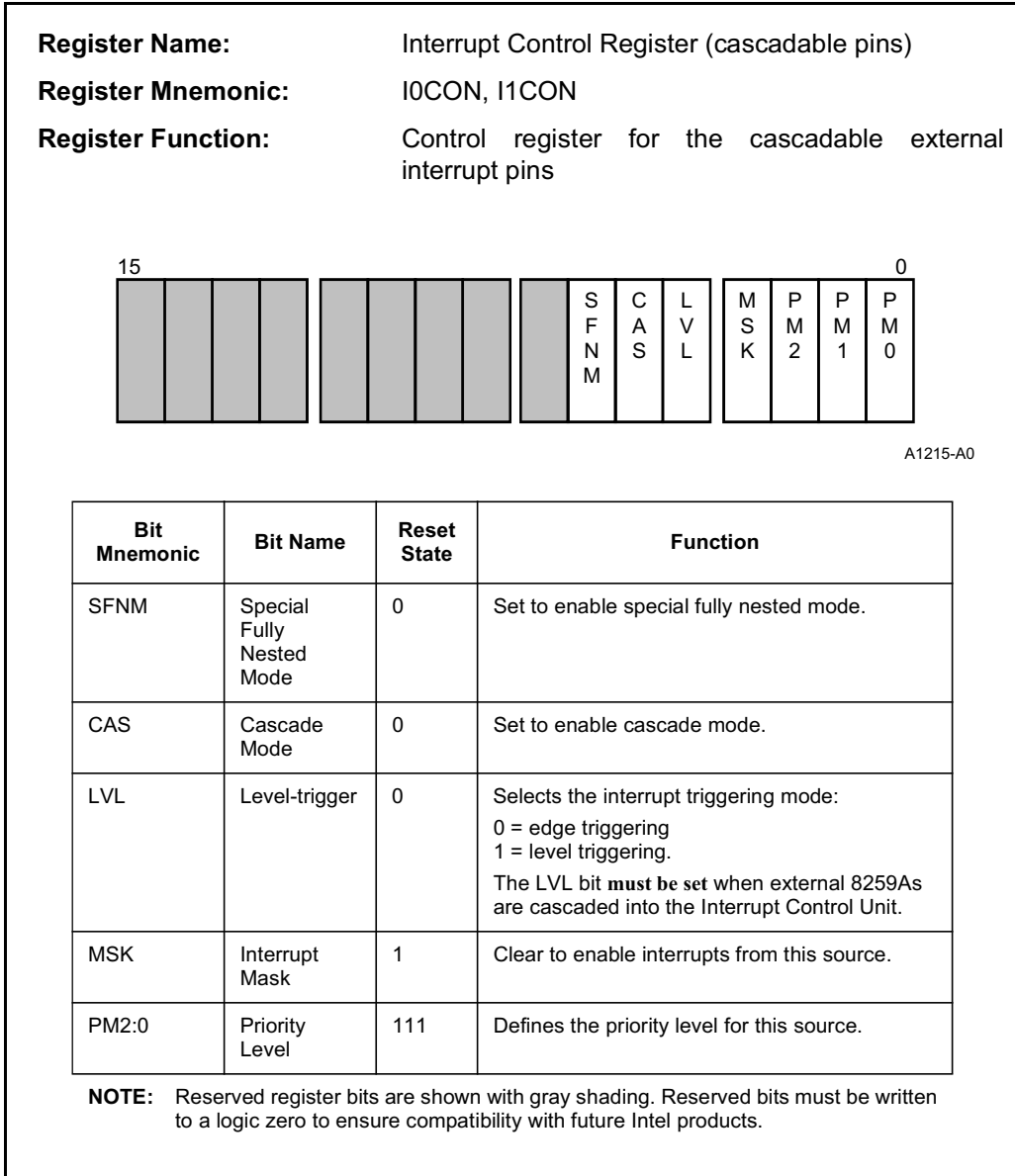


Figure 8-6. Interrupt Control Register for Cascadable Interrupt Pins

### 8.4.2 Interrupt Request Register

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

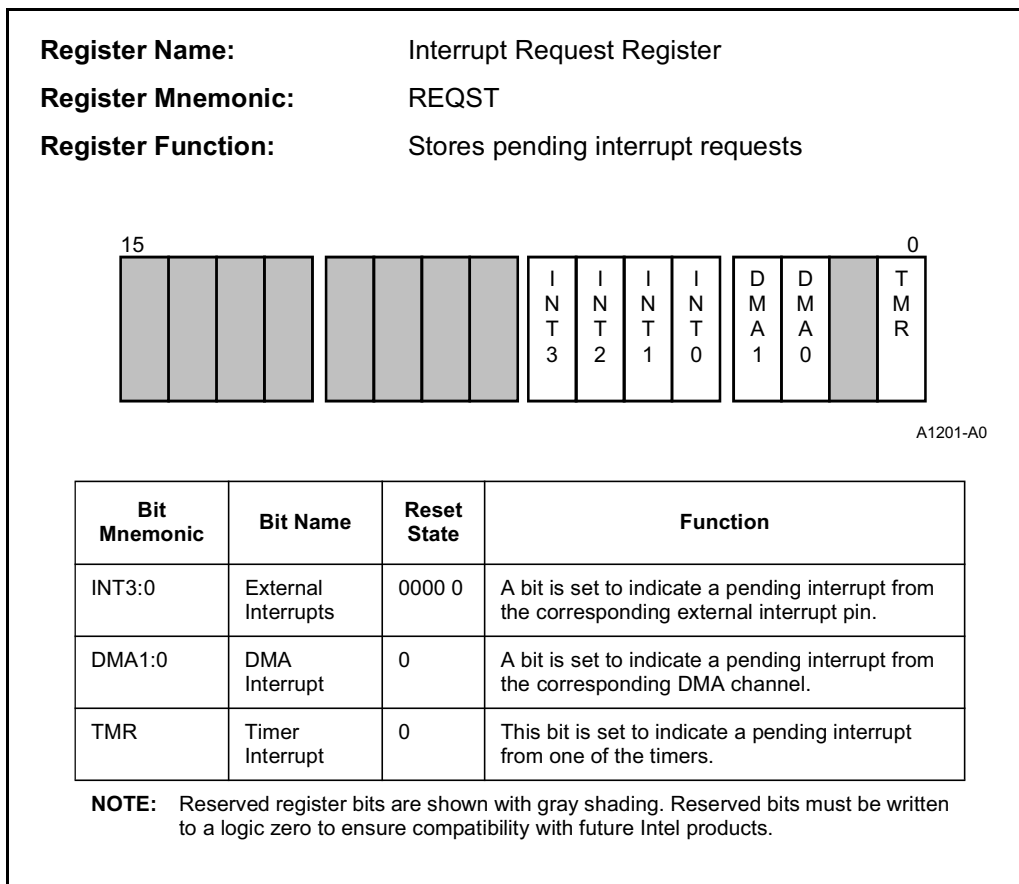


Figure 8-7. Interrupt Request Register

### 8.4.3 Interrupt Mask Register

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



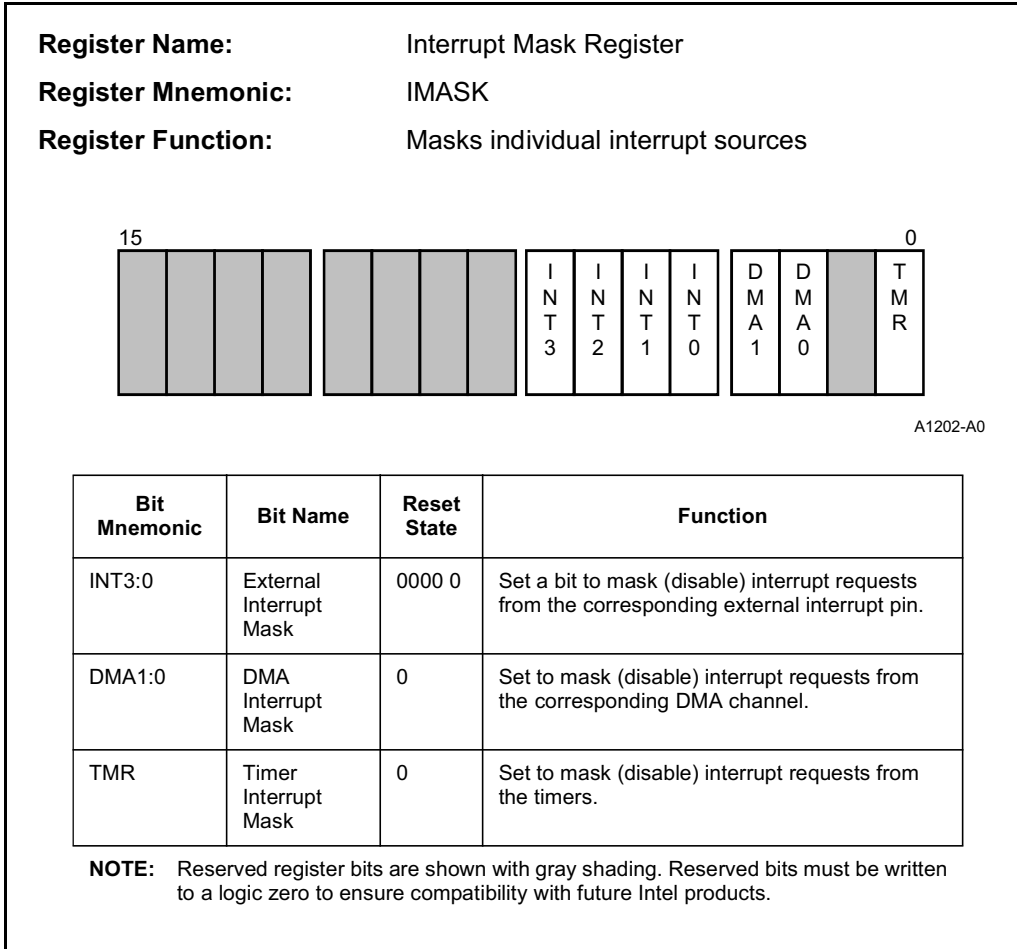


Figure 8-8. Interrupt Mask Register

### 8.4.4 Priority Mask Register

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

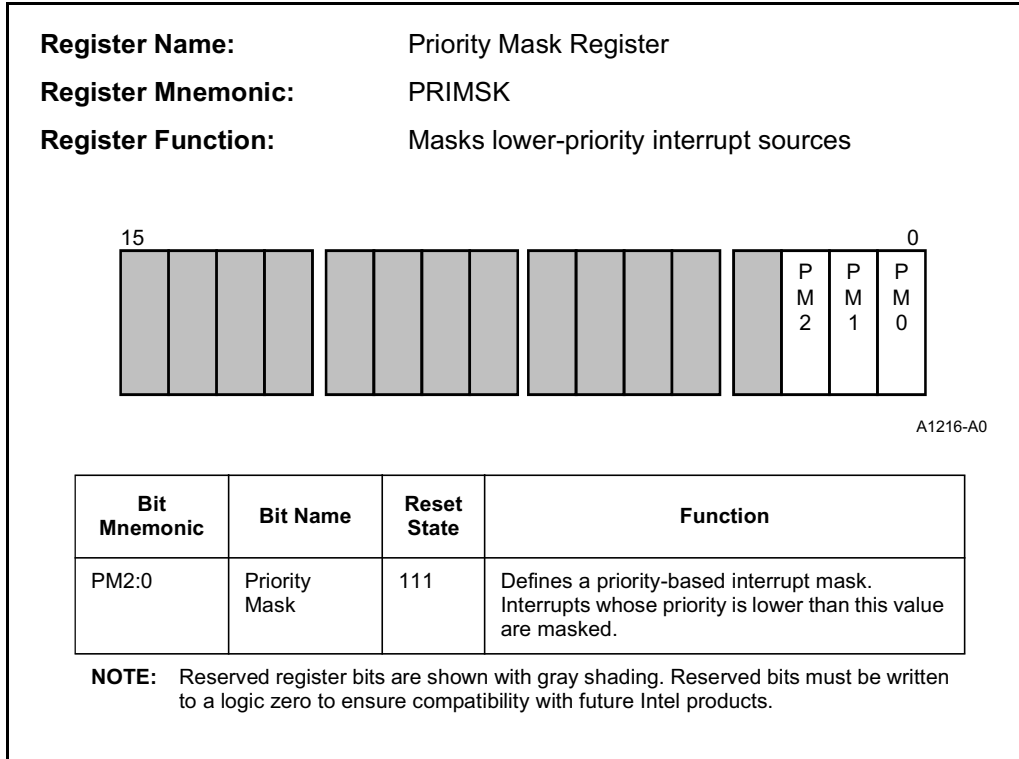


Figure 8-9. Priority Mask Register

### 8.4.5 In-Service Register

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



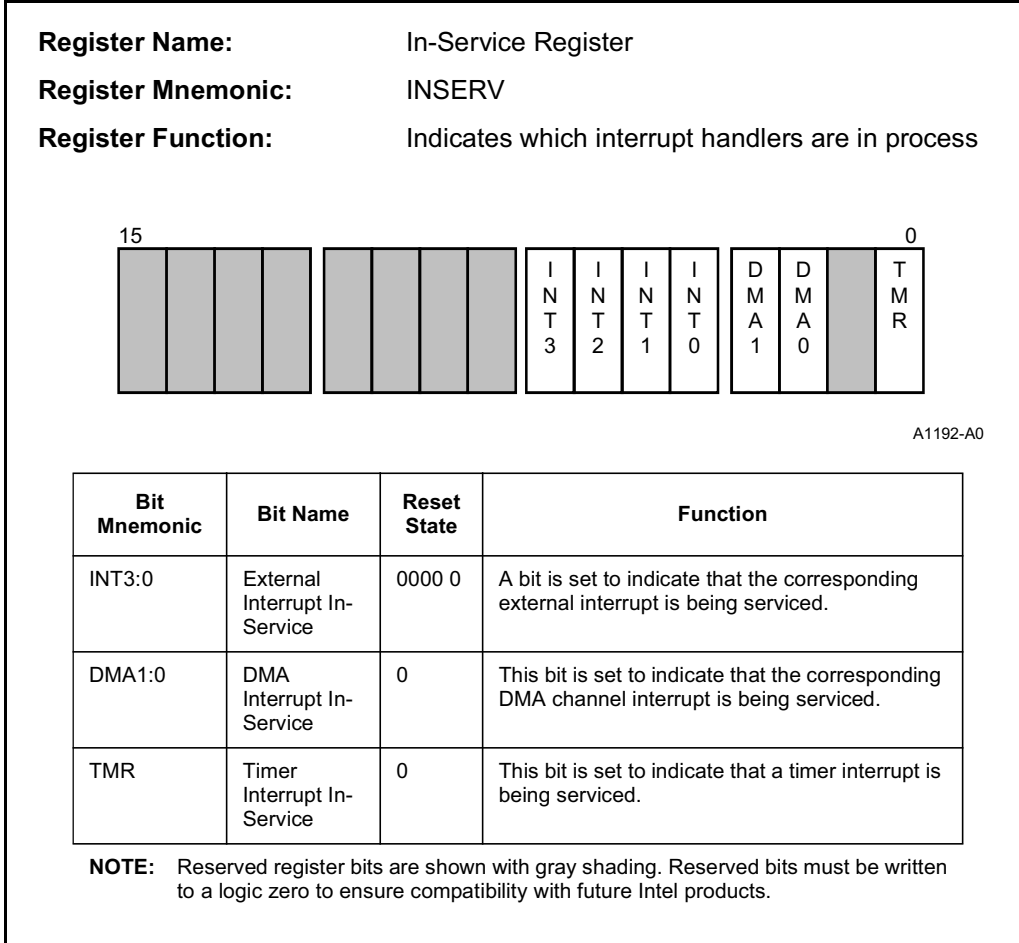


Figure 8-10. In-Service Register

### 8.4.6 Poll and Poll Status Registers

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



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

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

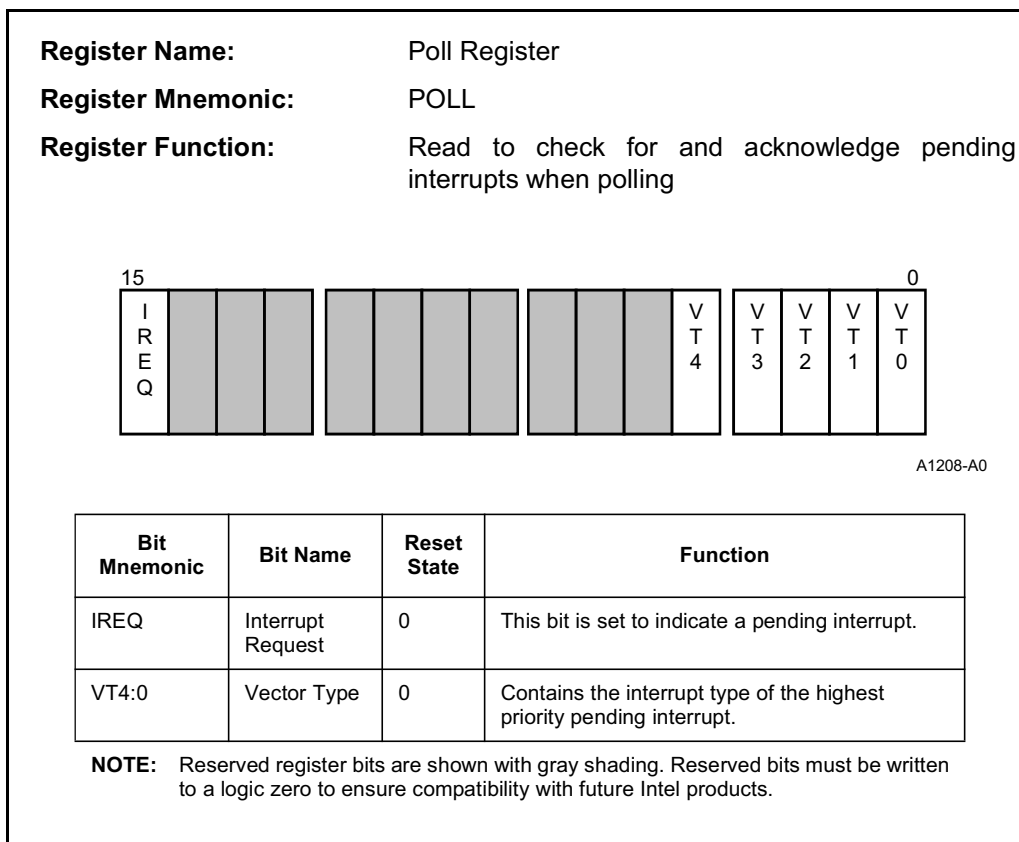


Figure 8-11. Poll Register



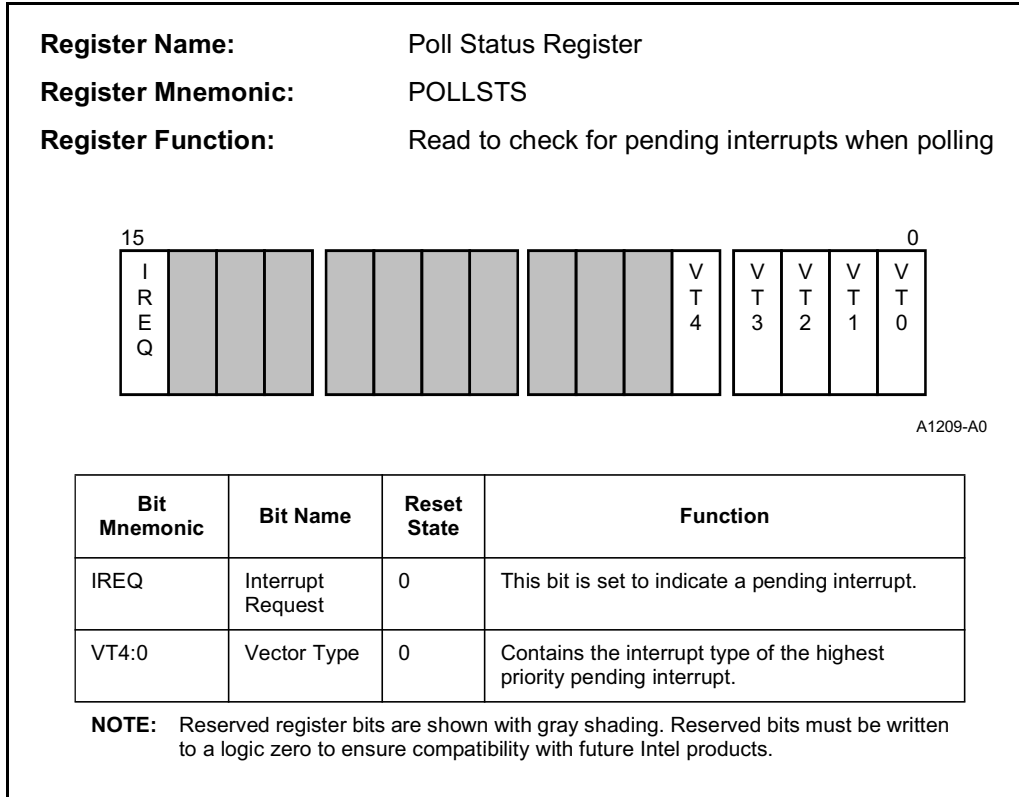


Figure 8-12. Poll Status Register

### 8.4.7 End-of-Interrupt (EOI) Register

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

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

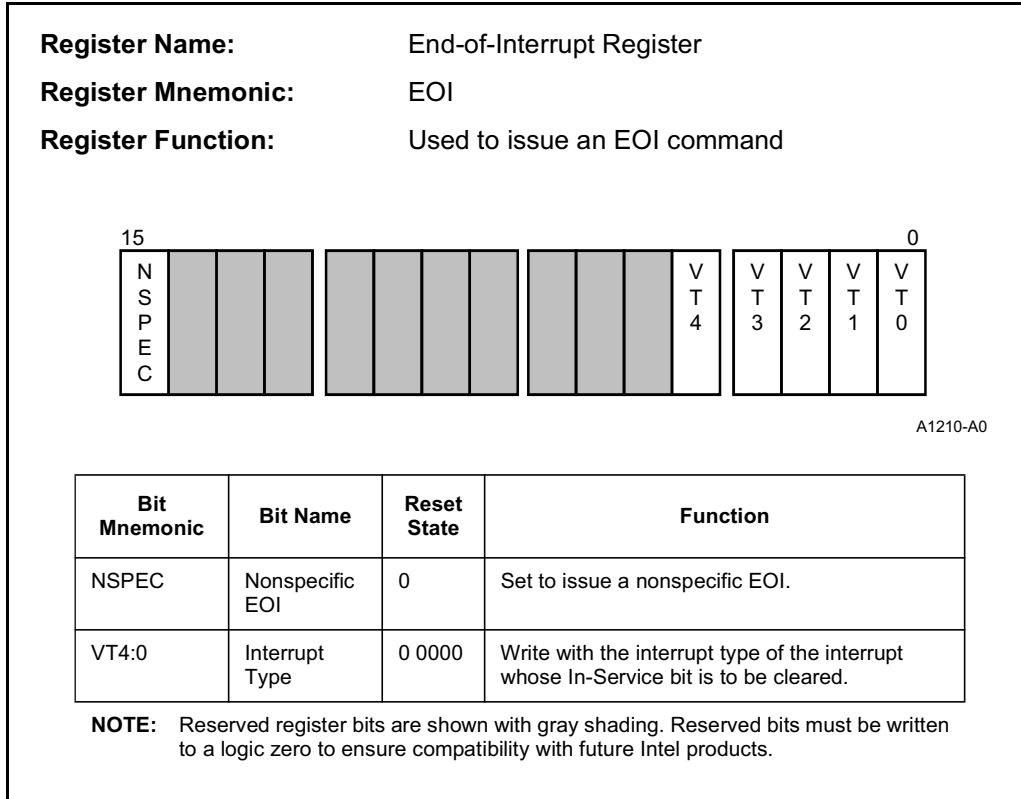


Figure 8-13. End-of-Interrupt Register

### 8.4.8 Interrupt Status Register

The Interrupt Status register (Figure 8-14) contains the DMA Halt bit and one bit for each timer interrupt. The CPU sets the DMA Halt bit to suspend DMA transfers while an NMI is processed. Software can also read and write this bit. See “Suspension of DMA Transfers” on page 10-20 for details. A timer bit is set to indicate a pending interrupt and is cleared when the interrupt request is acknowledged. Any number of bits can be set at any one time.



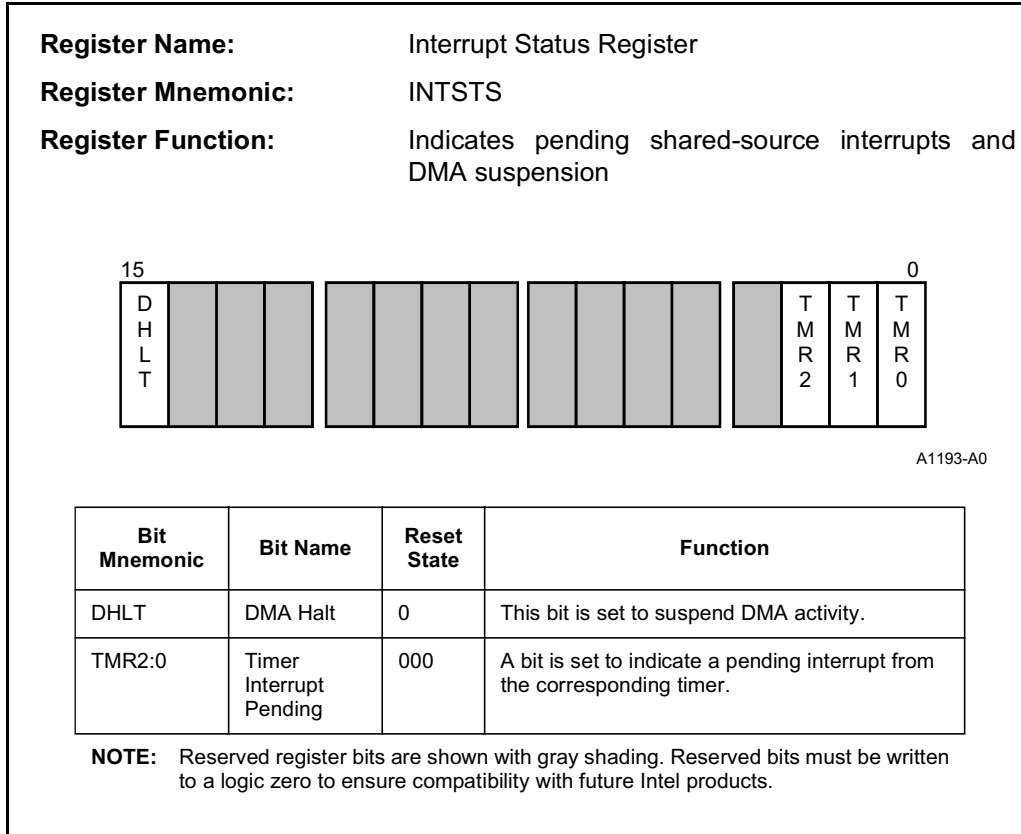


Figure 8-14. Interrupt Status Register

**NOTE**

Do not write to the DHLT bit while Timer/Counter Unit interrupts are enabled. A conflict with the internal use of the register may cause incorrect processing of timer interrupts.

The DHLT bit does not function when the interrupt controller is in slave mode.

**8.5 SLAVE MODE**

Although Master mode is the most common, Slave mode is useful in larger system designs. In Slave mode, an external 8259A module controls the interrupt input to the CPU and acts as the master interrupt controller. The Interrupt Control Unit processes only the internal interrupt requests and acts as an interrupt input to the external 8259A. In simplest terms, the Interrupt Control Unit behaves like a cascaded 8259A to the master 8259A. (See Figures 8-15 and 8-16.)

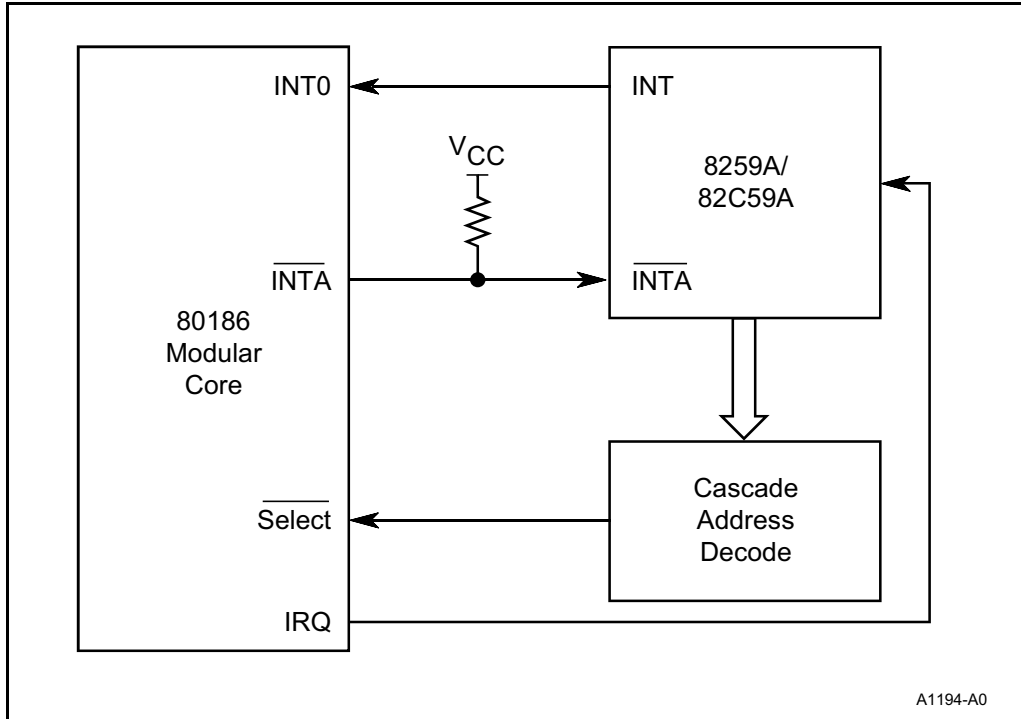
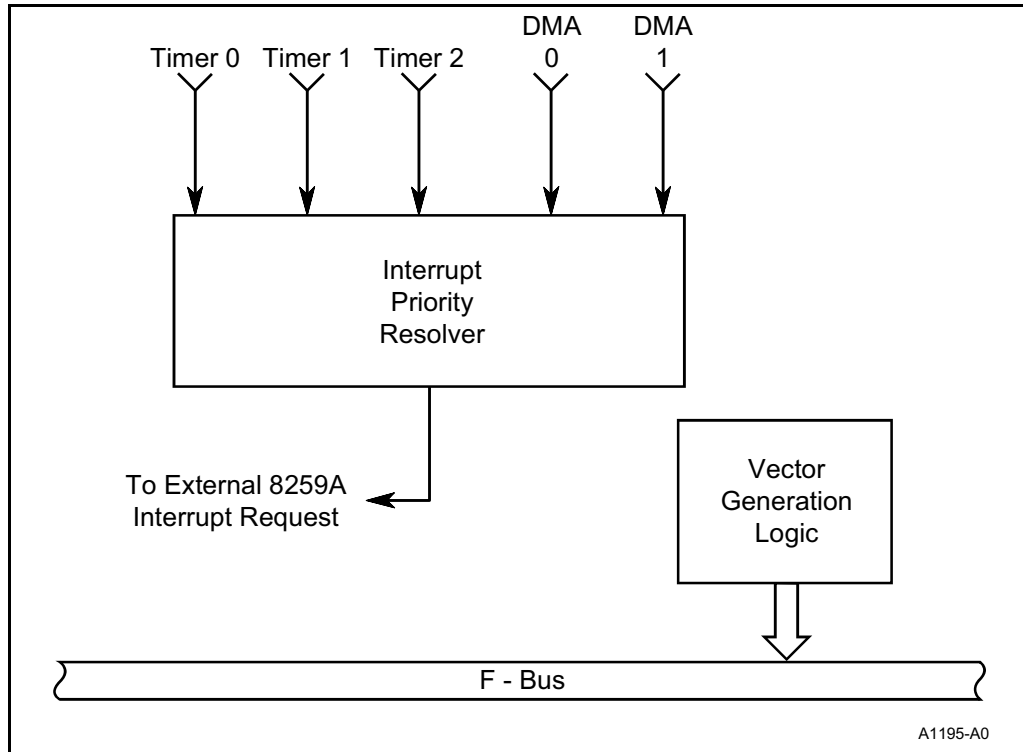


Figure 8-15. Interrupt Control Unit in Slave Mode







**Figure 8-16. Interrupt Sources in Slave Mode**

### 8.5.1 Slave Mode Programming

Some registers differ between Slave mode and Master mode. Slave mode adds the Interrupt Vector Register; it does not support the Poll, Poll Status Registers, INT3 and INT2 Control registers; and it replaces the Timer, INT1 and INT0 Control registers with individual Timer 0, Timer 1, and Timer 2 Control registers. The remaining registers retain the same functions as in Master mode; however, some bit positions change to accommodate the addition of the individual timer interrupts and the deletion of the external interrupts. Table 8-4 compares the Master and Slave mode registers and lists their PCB offset addresses.



**8.5.1.1 Interrupt Vector Register**

The Interrupt Vector Register is used only in Slave mode. In Master mode, the interrupt vector types are fixed; in Slave mode they are programmable. The Interrupt Vector Register is used to specify the five most-significant bits of the interrupt vector type. The three least-significant bits are fixed (Table 8-5).

**Table 8-4. Interrupt Control Unit Register Comparison**

| Master Mode Register Name | Slave Mode Register Name | PCB Offset Address |
|---------------------------|--------------------------|--------------------|
| INT3 Control              | (not used)               | 3EH                |
| INT2 Control              | (not used)               | 3CH                |
| INT1 Control              | Timer 2 Control          | 3AH                |
| INT0 Control              | Timer 1 Control          | 38H                |
| DMA1 Control              | DMA1 Control             | 36H                |
| DMA0 Control              | DMA0 Control             | 34H                |
| Timer Control             | Timer 0 Control          | 32H                |
| Interrupt Status          | Interrupt Status         | 30H                |
| Interrupt Request         | Interrupt Request        | 2EH                |
| In-Service                | In-Service               | 2CH                |
| Priority Mask             | Priority Mask            | 2AH                |
| Interrupt Mask            | Interrupt Mask           | 28H                |
| Poll Status               | (not used)               | 26H                |
| Poll                      | (not used)               | 24H                |
| EOI                       | EOI                      | 22H                |
| (not used)                | Interrupt Vector         | 20H                |

**Table 8-5. Slave Mode Fixed Interrupt Type Bits**

| Interrupt Source | Type Bits |   |   |
|------------------|-----------|---|---|
|                  | 2         | 1 | 0 |
| Timer 0          | 0         | 0 | 0 |
| (reserved)       | 0         | 0 | 1 |
| DMA 0            | 0         | 1 | 0 |
| DMA 1            | 0         | 1 | 1 |
| Timer 1          | 1         | 0 | 0 |
| Timer 2          | 1         | 0 | 1 |
| (reserved)       | 1         | 1 | 0 |
| (reserved)       | 1         | 1 | 1 |



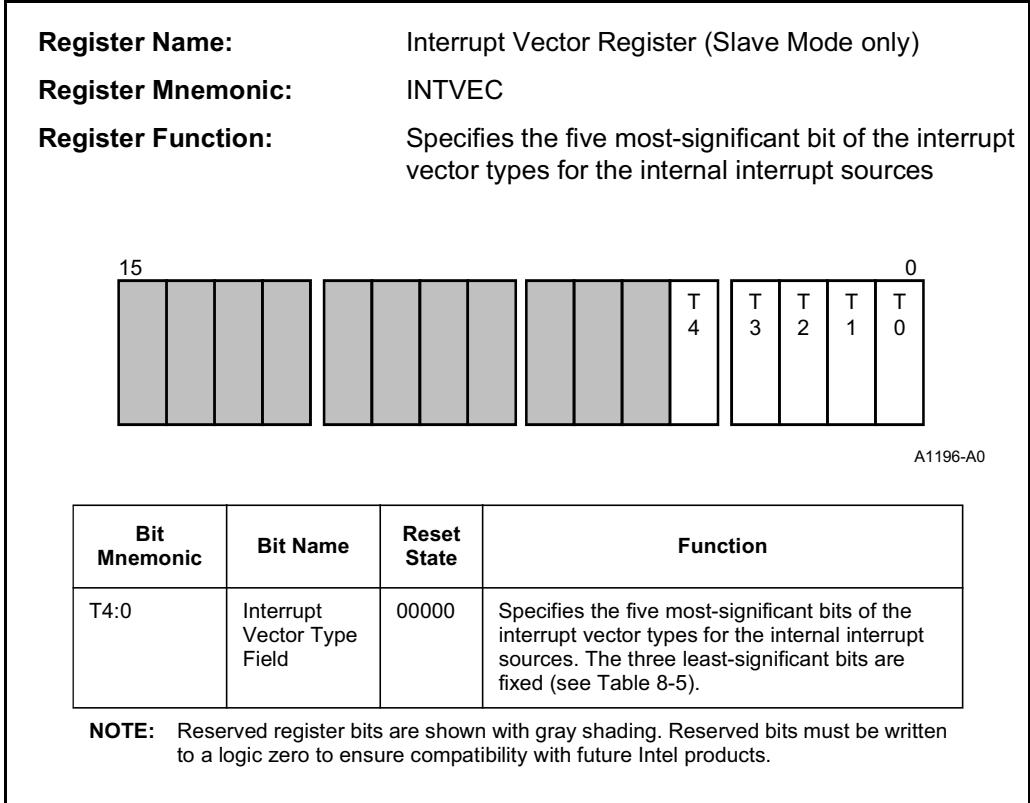


Figure 8-17. Interrupt Vector Register (Slave Mode Only)

### 8.5.1.2 End-Of-Interrupt Register

The End-of-Interrupt (EOI) register has the same function in Slave mode as in Master mode. However, non-specific EOI commands are not supported, so the NSPEC bit is omitted from the register. Only **specific** EOI commands can be issued. To clear an In-Service bit in Slave mode, write the three least-significant bits of the interrupt type (from Table 8-5) to the VT2:0 bits.

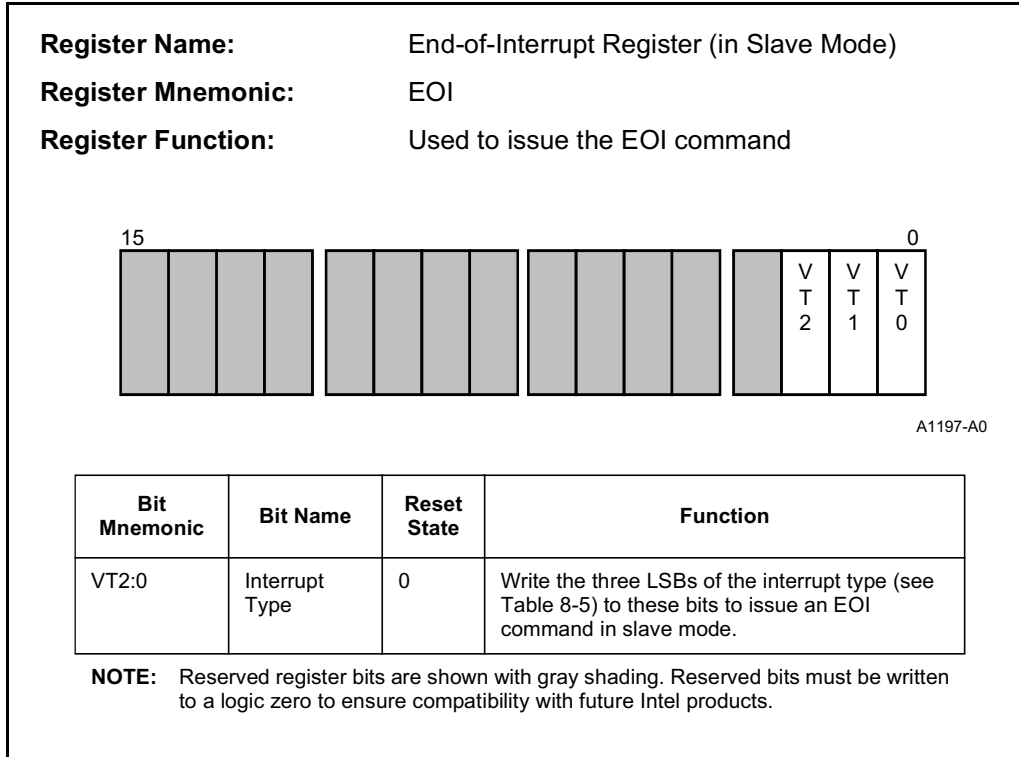


Figure 8-18. End-of-Interrupt Register in Slave Mode

8.5.1.3 Other Registers

The Priority Mask register is identical in Slave mode and Master mode. The Interrupt Request, Interrupt Mask, and In-Service registers retain the same function, but individual bits differ to accommodate the addition of the individual timer interrupts and the deletion of the external interrupts. Figure 8-19 shows the bit positions for Slave mode.

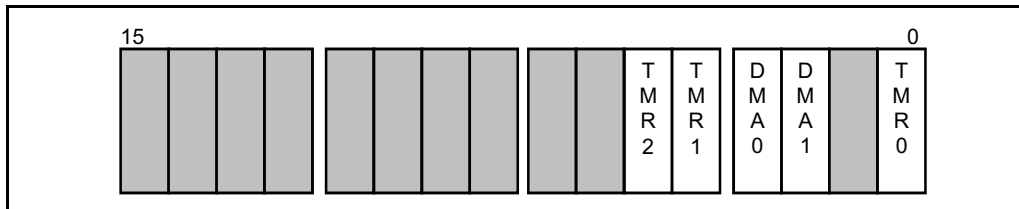


Figure 8-19. Request, Mask, and In-Service Registers



### 8.5.2 Interrupt Vectoring in Slave Mode

In Slave mode, the external 8259A module acts as the master interrupt controller. Therefore, interrupt acknowledge cycles are required for every interrupt, including those from integrated peripherals. During the first interrupt acknowledge cycle, the external 8259A determines which slave interrupt controller has the highest priority interrupt request. It then drives that slave address onto its  $\overline{\text{CAS}}2:0$  pins (Figure 8-20). External logic must decode the correct slave address from the  $\overline{\text{CAS}}2:0$  pins to drive the SELECT pin.

Figure 8-20. Interrupt Vectoring in Slave Mode

The  $\overline{\text{SELECT}}$



External interrupt acknowledge cycles must be run for every maskable interrupt. Therefore, the interrupt response time for every interrupt will be 55 clocks, as shown in Figure 8-21.

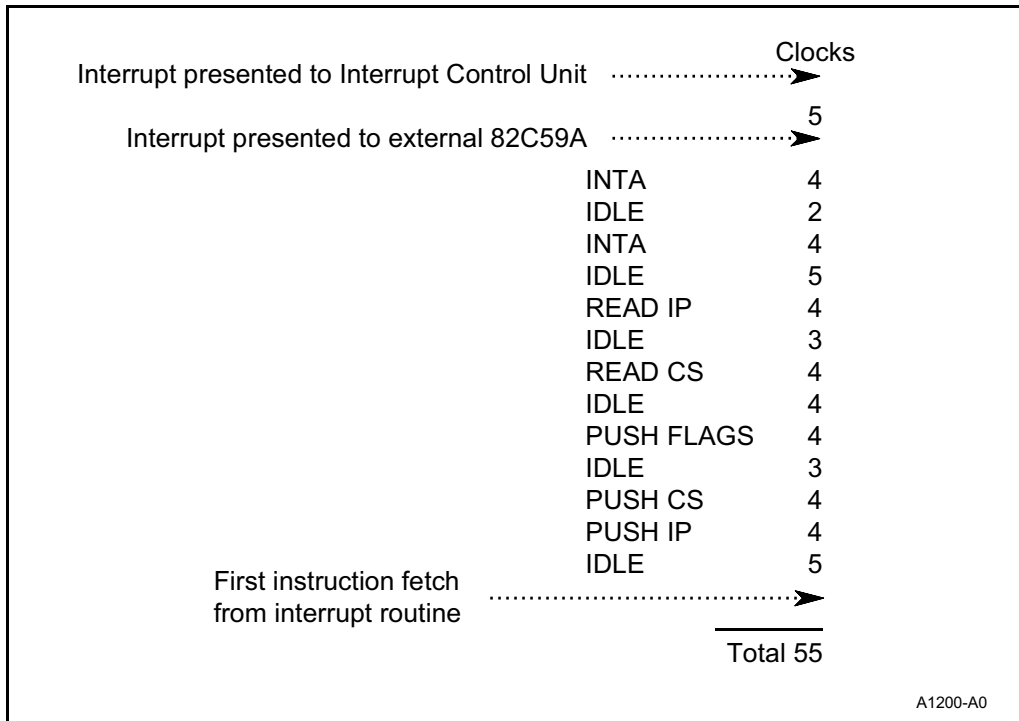


Figure 8-21. Interrupt Response Time in Slave Mode

### 8.5.3 Initializing the Interrupt Control Unit for Master Mode

Follow these steps to initialize the Interrupt Control Unit for Master mode.

1. Determine which interrupt sources you want to use.
2. Determine whether to use the default priority scheme or devise your own.
3. Program the Interrupt Control register for each interrupt source.
  - For external interrupt pins, select edge or level triggering.
  - For INT0 or INT1, enable cascade mode, special fully nested mode, or both, if you wish to use them.
  - If you are using a custom priority scheme, program the priority level for each interrupt source.
4. Program the Priority Mask with a priority mask level, if you wish to mask interrupts based on priority. (The default is level seven, which enables all interrupt levels.)



5. Set the mask bit in the Interrupt Mask register for any interrupts that you wish to disable.

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

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

**Example 8-1. Initializing the Interrupt Control Unit for Master Mode**

INTERRUPT CONTROL UNIT







9

## Timer/Counter Unit







## CHAPTER 9 TIMER/COUNTER UNIT

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

### 9.1 FUNCTIONAL OVERVIEW

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

The Timer/Counter Unit is serviced over four clock periods, one timer during each clock, with an idle clock at the end (see Figure 9-2). No connection exists between the counter element and the CPU through the register banks and the Bus Interface Unit. Timer operation and bus interface operation are asynchronous. This time-multiplexed scheme results in a delay of  $2\frac{1}{2}$  to  $6\frac{1}{2}$  CLKOUT periods from timer input to timer output.

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

Timers 0 and 1 are functionally identical. Figure 9-3 illustrates their operation. Each has a latched, synchronized input pin and a single output pin. Each timer can be clocked internally or externally. Internally, the timer can either increment at  $\frac{1}{4}$  CLKOUT frequency or be prescaled by Timer 2. A timer that is prescaled by Timer 2 increments when Timer 2 reaches its maximum count value.



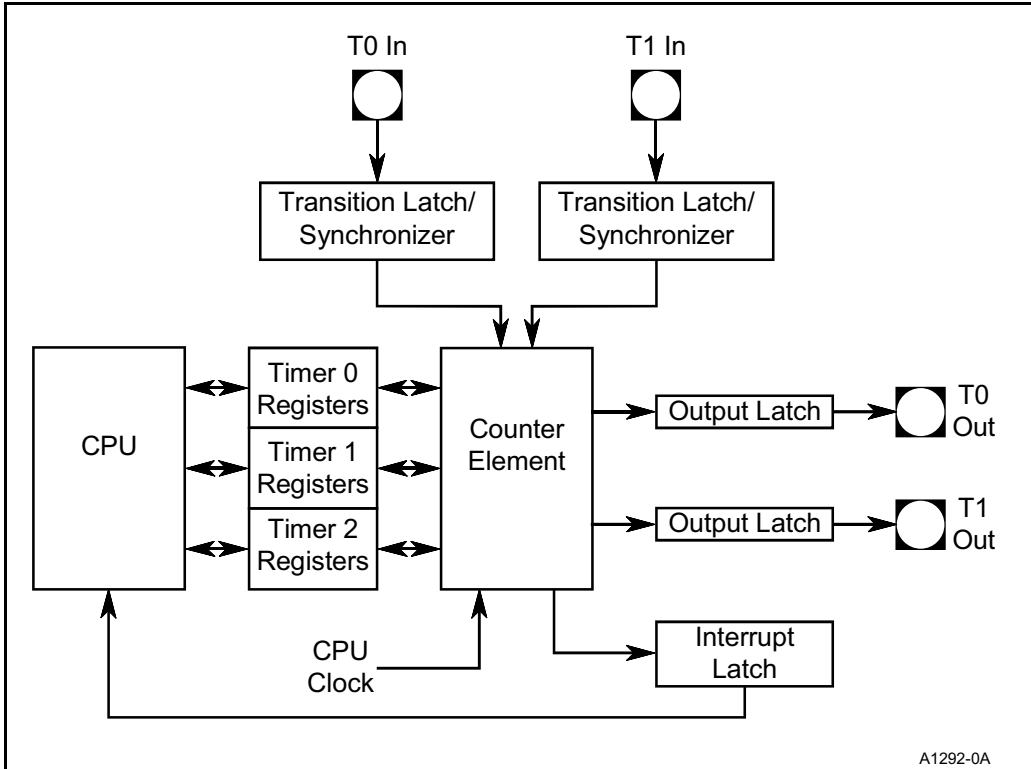


Figure 9-1. Timer/Counter Unit Block Diagram



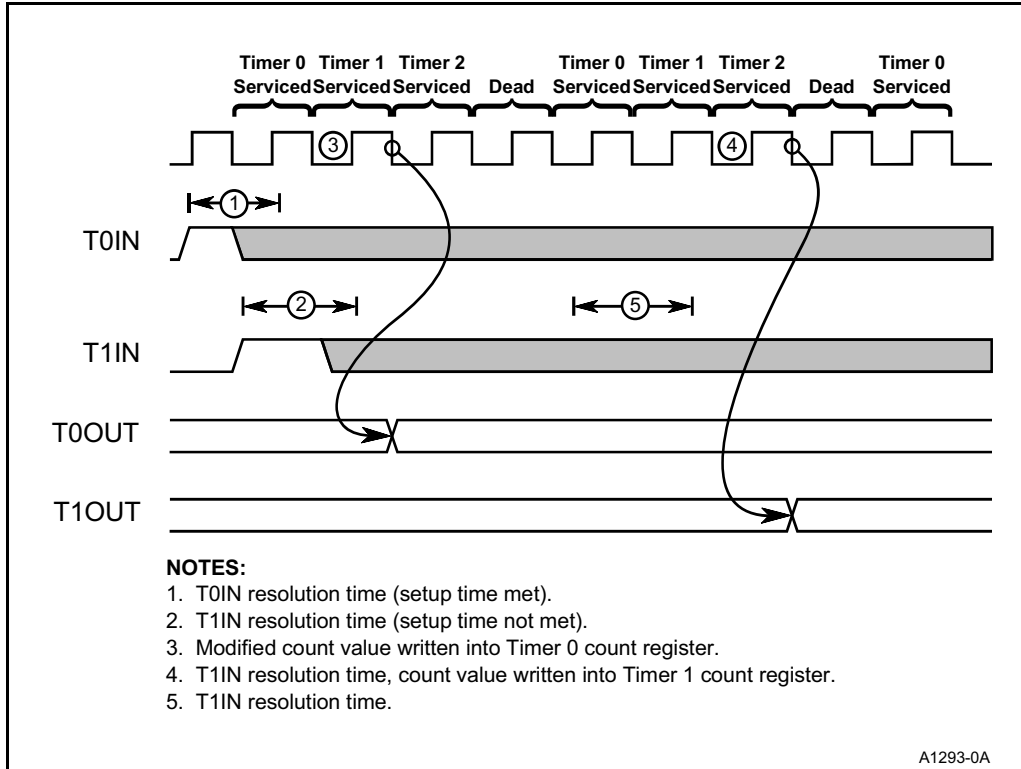


Figure 9-2. Counter Element Multiplexing and Timer Input Synchronization

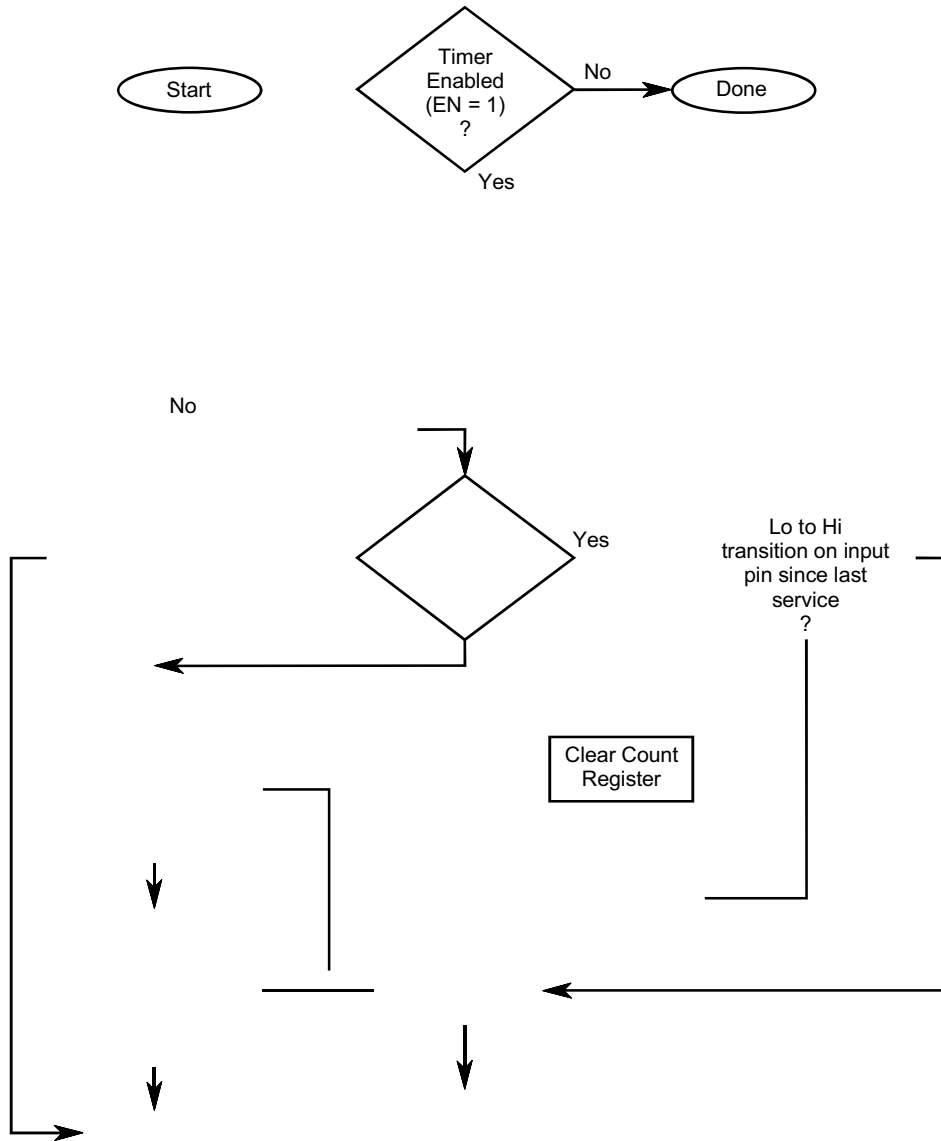
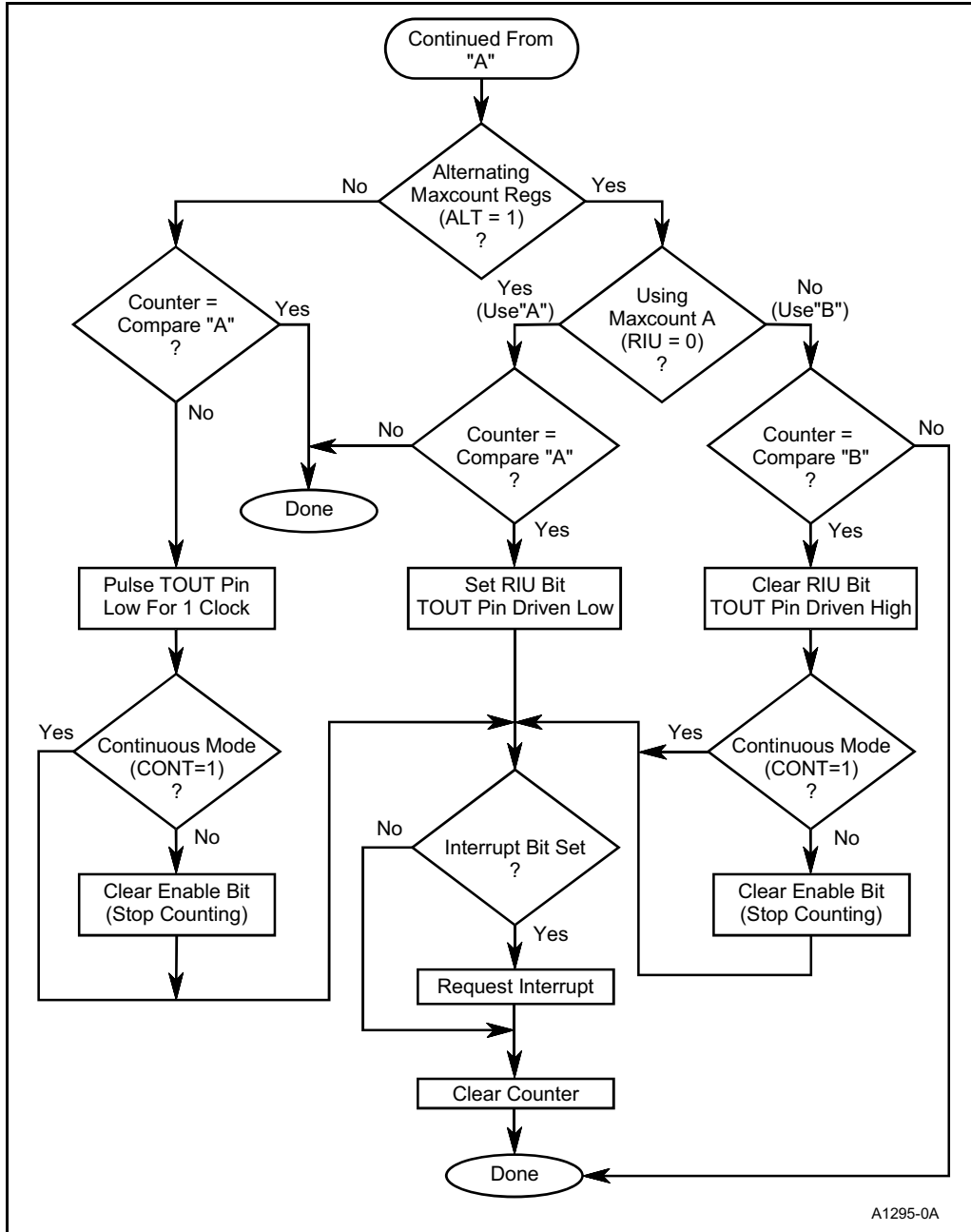


Figure 9-3. Timers 0 and 1 Flow Chart





A1295-0A

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

When configured for internal clocking, the Timer/Counter Unit uses the input pins either to enable timer counting or to retrigger the associated timer. Externally, a timer increments on low-to-high transitions on its input pin (up to  $\frac{1}{4}$  CLKOUT frequency).

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

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

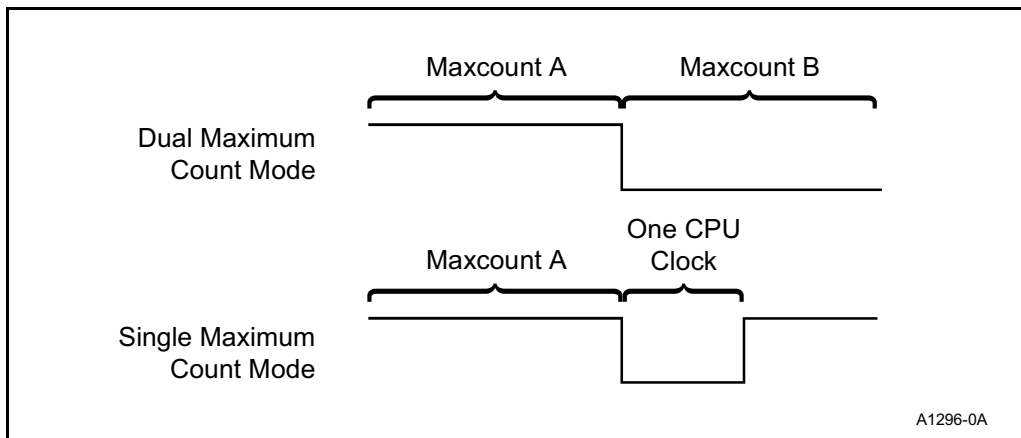


Figure 9-4. Timer/Counter Unit Output Modes

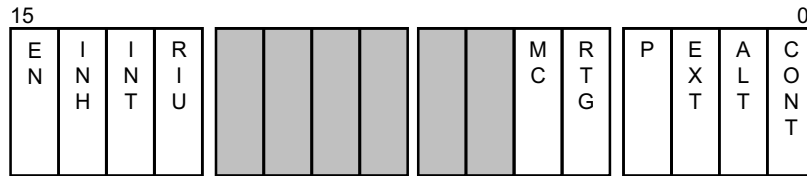
## 9.2 PROGRAMMING THE TIMER/COUNTER UNIT

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





**Register Name:** Timer 0 and 1 Control Registers  
**Register Mnemonic:** T0CON, T1CON  
**Register Function:** Defines Timer 0 and 1 operation.

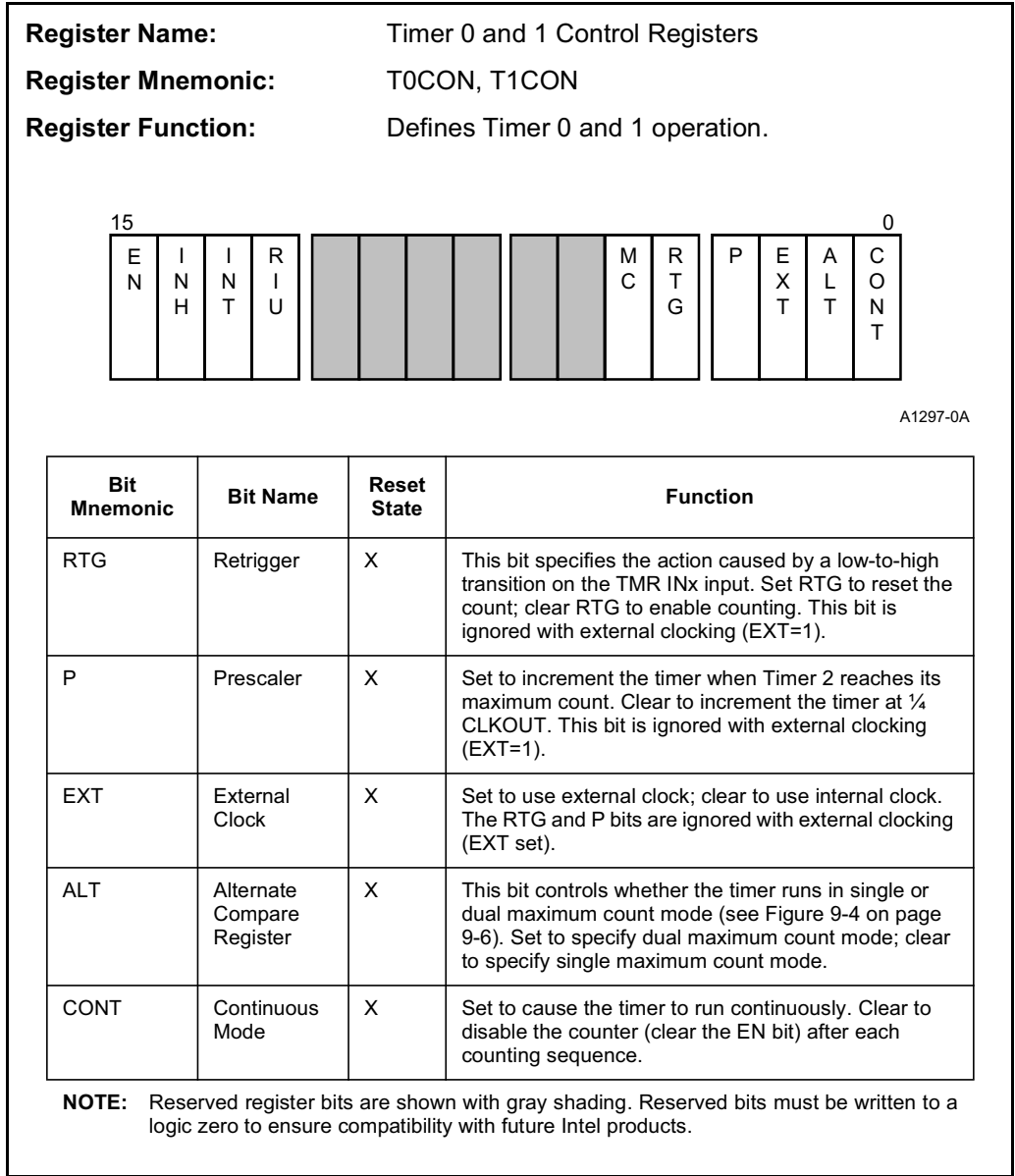


A1297-0A

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

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

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



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

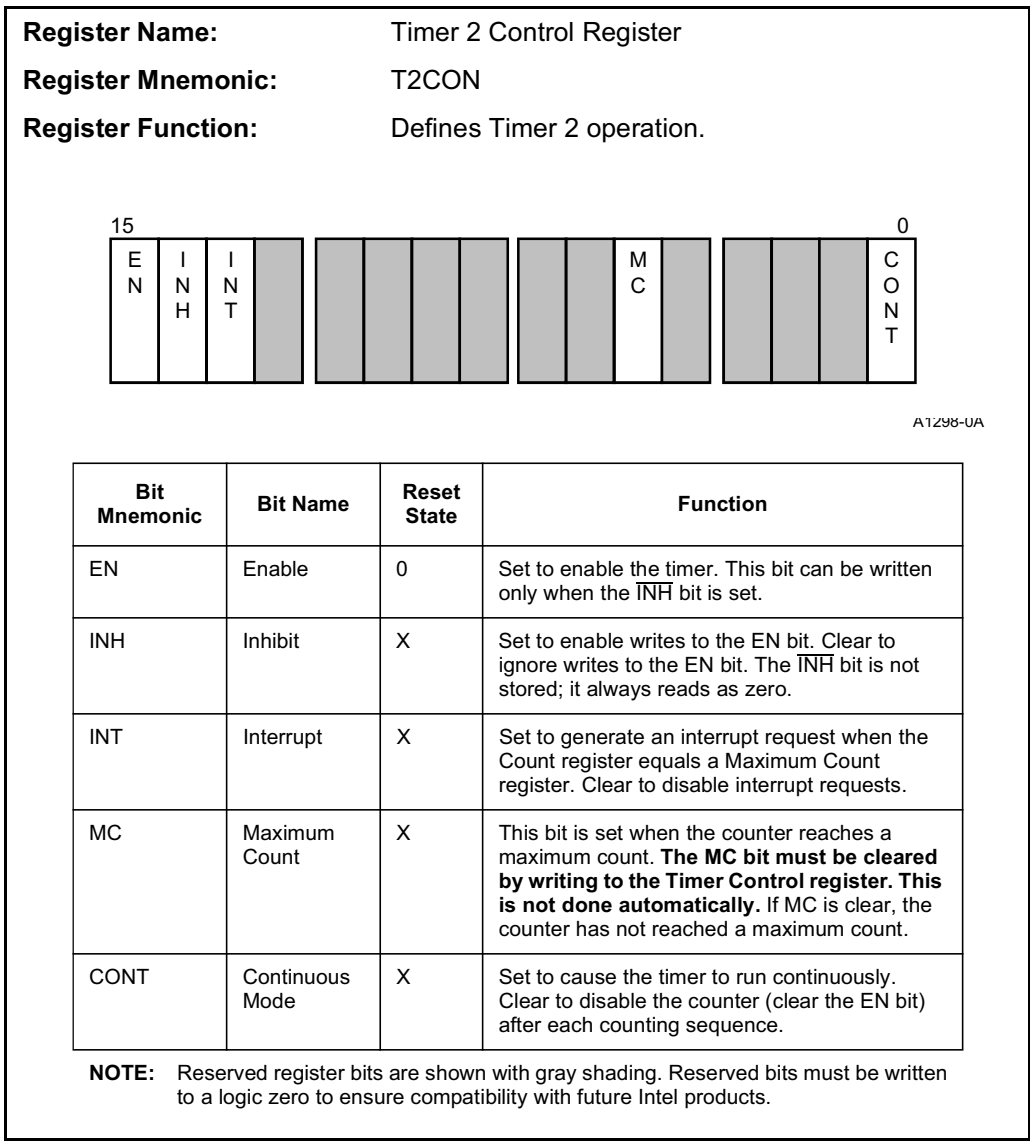
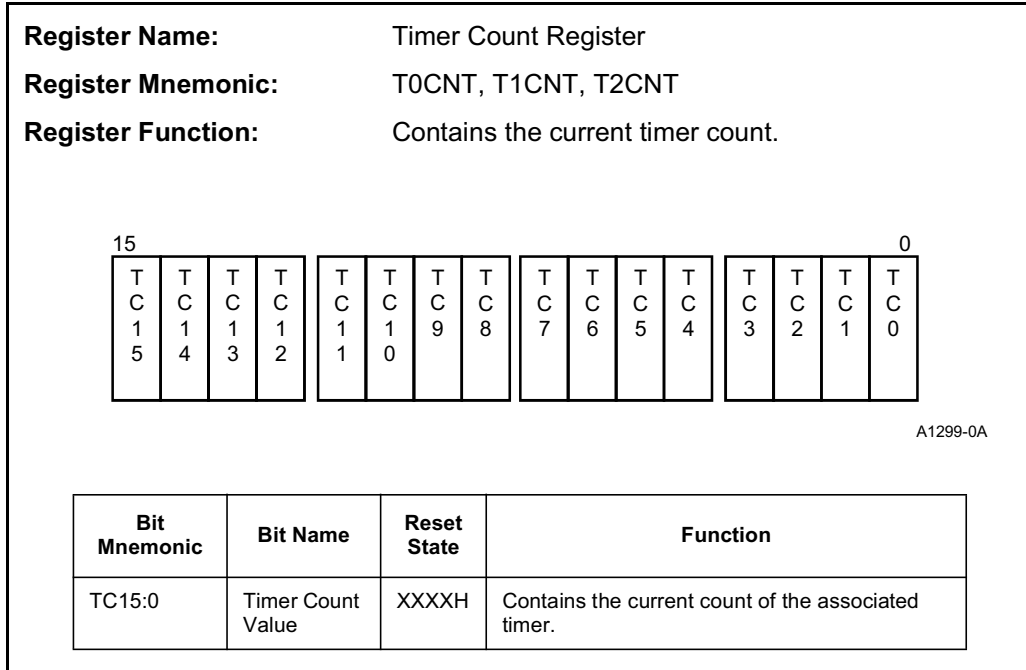


Figure 9-6. Timer 2 Control Register



**Figure 9-7. Timer Count Registers**



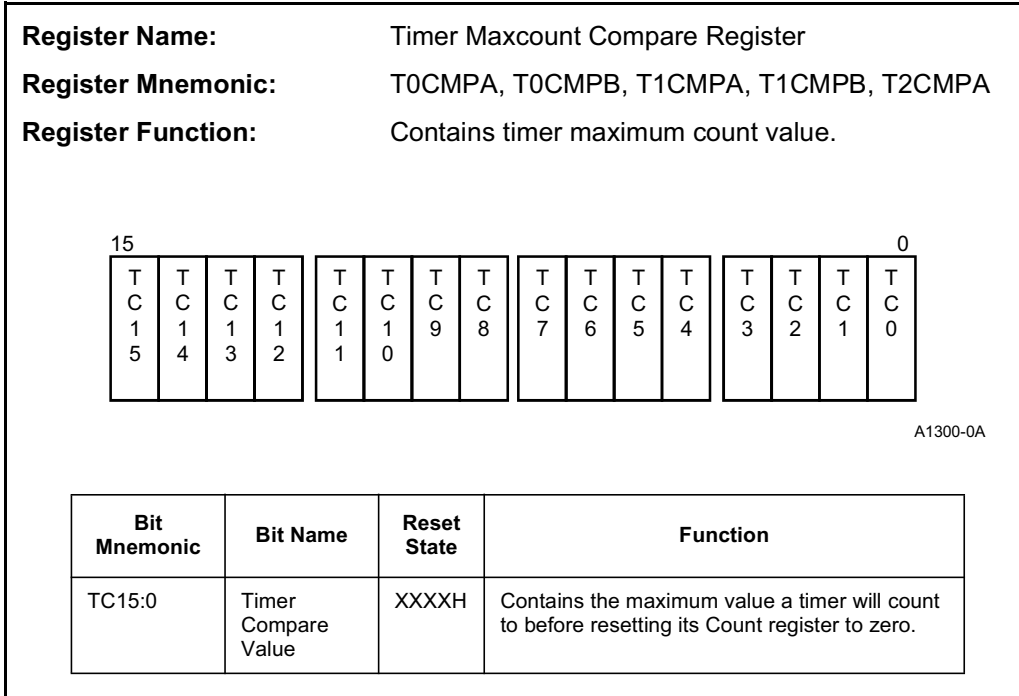


Figure 9-8. Timer Maxcount Compare Registers

### 9.2.1 Initialization Sequence

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

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



### 9.2.2 Clock Sources

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

Timers 0 and 1 can be programmed to count low-to-high transitions on their input pins as timer events by setting the External (EXT) bit in their control registers. Transitions on the external pin are synchronized to the CPU clock before being presented to the timer circuitry. The timer counts **transitions** on this pin. The input signal must go low, then high, to cause the timer to increment. The maximum count-rate for the timers is  $\frac{1}{4}$  the CPU clock rate (measured at CLKOUT) because the timers are serviced only once every four clocks.

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

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

**Table 9-1. Timer 0 and 1 Clock Sources**

| EXT | P | Clock Source  |
|-----|---|---|
| 0   | 0 | Timer clocked internally at $\frac{1}{4}$ CLKOUT frequency.       |
| 0   | 1 | Timer clocked internally, prescaled by Timer 2.                   |
| 1   | X | Timer clocked externally at up to $\frac{1}{4}$ CLKOUT frequency. |

### 9.2.3 Counting Modes

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



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

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

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

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

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

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

### 9.2.3.1 Retriggerring

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

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

**Table 9-2. Timer Retriggering**

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

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

### 9.2.4 Pulsed and Variable Duty Cycle Output

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

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

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





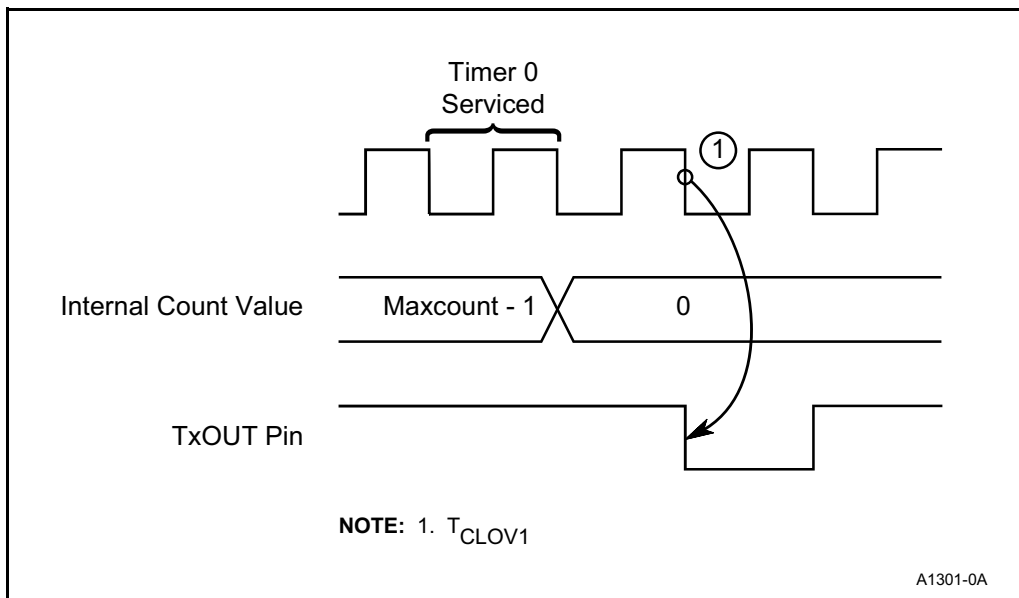


Figure 9-9. TxOUT Signal Timing

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

### 9.2.5 Enabling/Disabling Counters

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

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

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

### 9.2.6 Timer Interrupts

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

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

### 9.2.7 Programming Considerations

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

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

## 9.3 TIMING

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

### 9.3.1 Input Setup and Hold Timings

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



### 9.3.2 Synchronization and Maximum Frequency

All timer inputs are latched and synchronized with the CPU clock. Because of the internal logic required to synchronize the external signals, and the multiplexing of the counter element, the Timer/Counter Unit can operate only up to  $\frac{1}{4}$  of the CLKOUT frequency. Clocking at greater frequencies will result in missed clocks.

#### 9.3.2.1 Timer/Counter Unit Application Examples

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

### 9.3.3 Real-Time Clock

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

### 9.3.4 Square-Wave Generator

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

### 9.3.5 Digital One-Shot

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

```

$mod186
name example_80186_family_timer_code

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

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

;substitute register offsets

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

data segment public 'data'

    public _hour, _minute, _second, _msec

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

data ends

```

**Example 9-1. Configuring a Real-Time Clock**



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



```

        sti                                ;enable interrupts

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

_set_time endp

timer_2_interrupt_routine proc far

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

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

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

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

reset_hour:
        mov    _hour, 1                    ;reset hour

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

timer_2_interrupt_routine endp

lib_80186     ends
end

```

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

```

$mod186
name          example_timer1_square_wave_code

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

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

lib_80186 segment public 'code'
          assume cs:lib_80186

public   _clock
_clock  proc far

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

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

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

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

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

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

```

**Example 9-2. Configuring a Square-Wave Generator**

```

    pop    dx                ;restore saved registers
    pop    bx
    pop    ax

    pop    bp                ;restore caller's bp
    ret

    clock    endp
lib_80186   ends
end

```

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

```

$mod186
name example_timer1_1_shot_code

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

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

lib_80186   segment public 'code'
            assume cs:lib_80186

public     _one_shot
_one_shot   proc far

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

```

**Example 9-3. Configuring a Digital One-Shot**



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

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

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

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

_one_shot endp
lib_80186 ends
end
```

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





**10**

**Direct Memory  
Access Unit**





intel®

I

## DIRECT MEMORY ACCESS UNIT



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 continues 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 on page 10-2). 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.

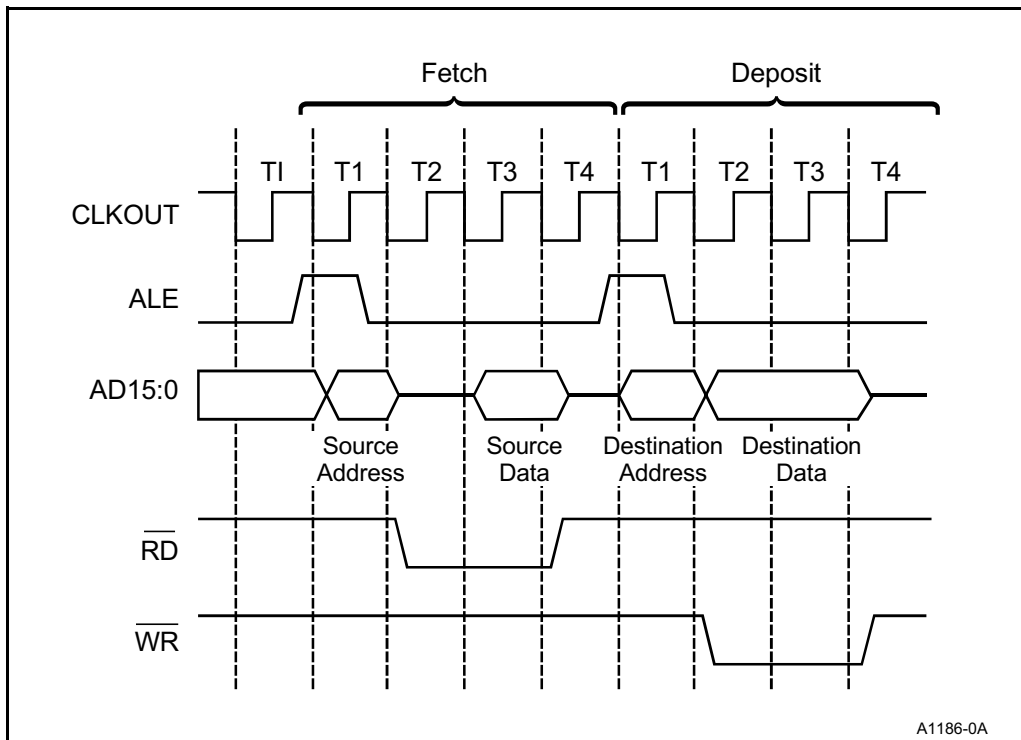


Figure 10-1. Typical DMA Transfer



#### 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 transfers. The handling of byte and word data is the same as that for normal bus cycles and is dependent upon the processor bus width. For example, odd-aligned word DMA transfers on a processor with a 16-bit bus requires two fetches and two deposits (all back-to-back). BIU bus cycles are covered in Chapter 3, “Bus Interface Unit.” 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 programmed data width (byte or word) determines the amount that a pointer is incremented or decremented. Word transfers change the pointer by two; byte transfers change the pointer by one.

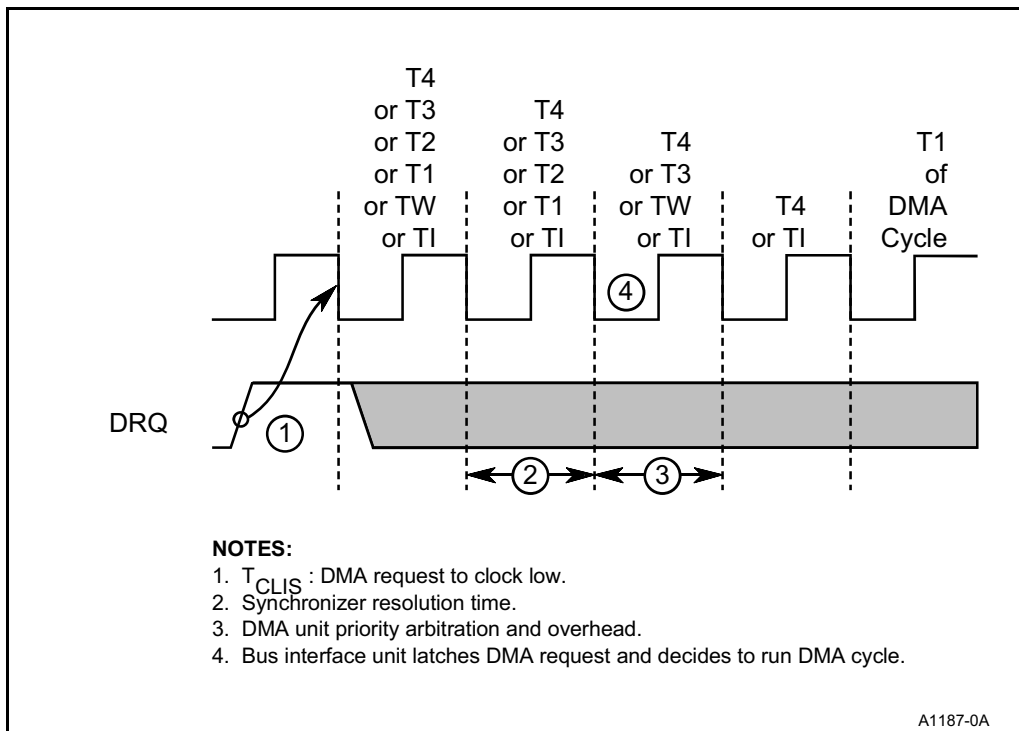
#### 10.1.3 DMA Requests

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. (See “Arming the DMA Channel” on page 10-18.)



### 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).



**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 that is **sending** data. For example, a disk controller in the process of reading data from a disk would use a source-synchronized request (data would be moving from the disk to memory). A destination-synchronized request originates from the peripheral that is **receiving** data. If a disk controller were writing data to a disk, it would use a destination-synchronized request (data would be moving from memory to the disk). The type of synchronization a channel uses is programmable. (See “Selecting Channel Synchronization” on page 10-18.)





10.1.4.1 Source Synchronization

A typical source-synchronized transfer is shown in Figure 10-3. Most DMA-driven peripherals deassert their DRQ line only after the DMA transfer has begun. The DRQ signal must be deasserted at least four clocks before the end of the DMA transfer (at the T1 state of the deposit phase) to prevent another DMA cycle from occurring. A source-synchronized transfer provides the source device at least three clock cycles from the time it is accessed (acknowledged) to deassert its request line if further transfers are not required.

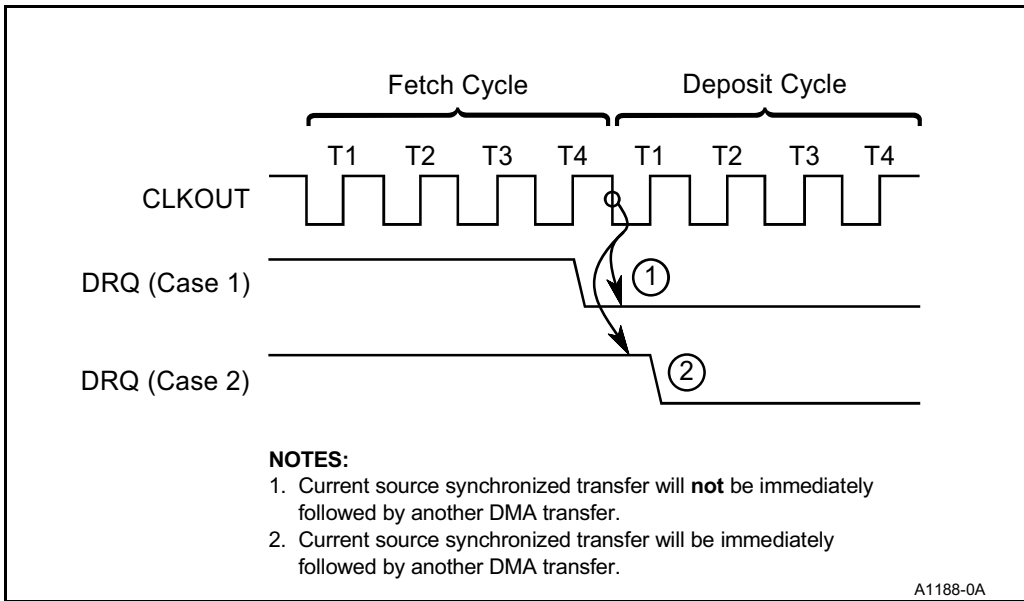


Figure 10-3. Source-Synchronized Transfers

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.



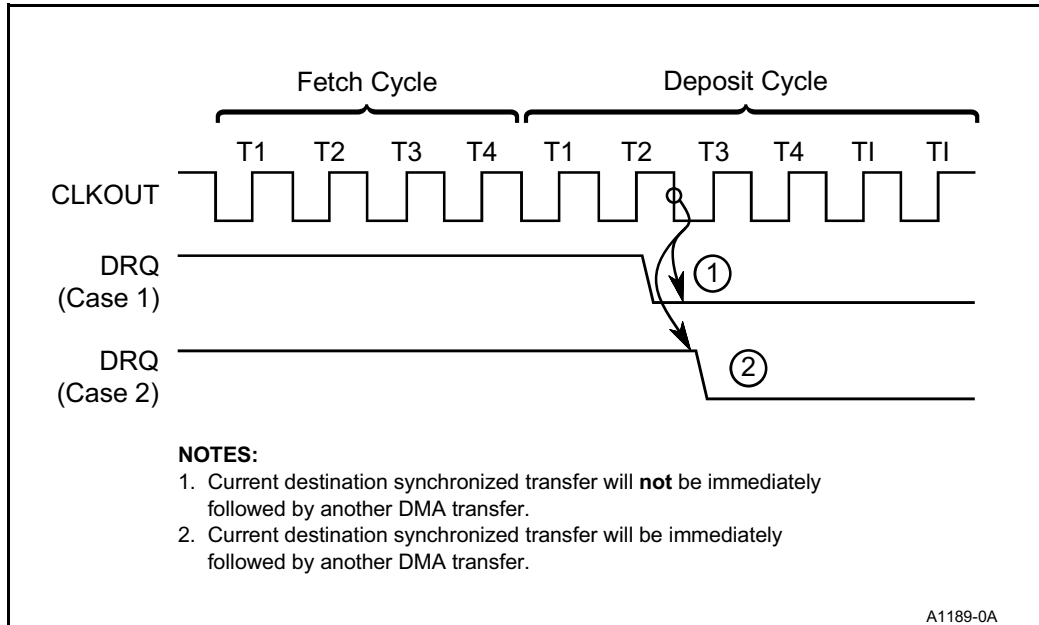


Figure 10-4. Destination-Synchronized Transfers

### 10.1.5 Internal Requests

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

#### 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-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 to the waveform data, the destination would point to the A/D converter and Timer 2 would request a transfer every 22 microseconds. (See “Timed DMA Transfers” on page 10-26.)

#### 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 transfer count reaches zero or DMA transfers are suspended by an NMI.



## DIRECT MEMORY ACCESS UNIT



## DIRECT MEMORY ACCESS UNIT

### 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, “Bus Interface Unit.”) 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 that chip-select is asserted during the cycle. The Chip-Select Unit will not recognize DMA cycles that access I/O space above 64K.

### 10.1.10 The Two-Channel DMA Unit

Two DMA channels are combined with arbitration logic to form the DMA Unit (see Figure 10-5).

#### 10.1.10.1 DMA Channel Arbitration

Within the two-channel DMA Unit, 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 performs its transfer (or transfers) first. The high priority channel continues to perform transfers as long as the following conditions are met:

- the DMA channel 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 in which 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 occurs for both channels simultaneously, channel 1 performs the first transfer. At the end of channel 1 transfer, 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 has not completed all of its transfers**. Channel 1 regains the bus at the end of channel 0 transfer. The transfers will alternate as long as both requests remain active.

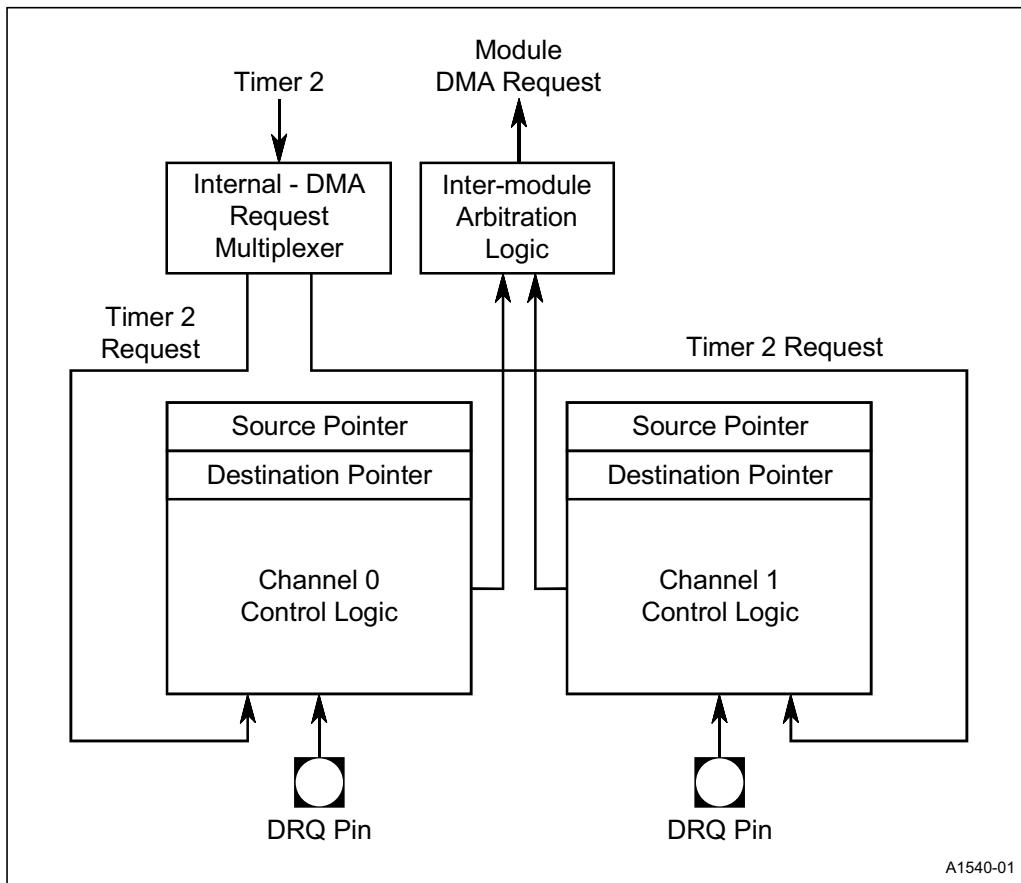


Figure 10-5. Two-Channel DMA Module

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.



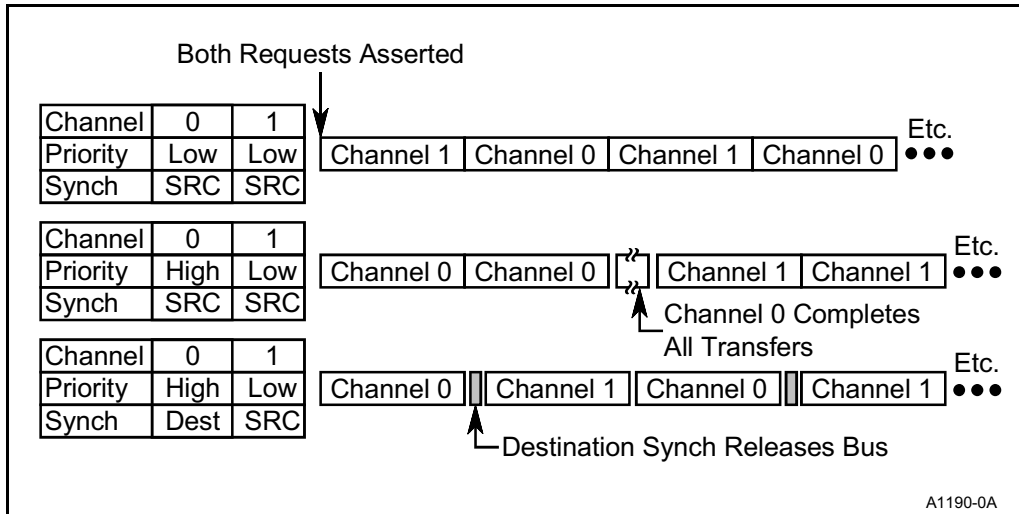


Figure 10-6. Examples of DMA Priority

**10.1.10.1.2 Rotating Priority**

Channel priority rotates when the 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. Channel 1 is reassigned high priority whenever the bus is released (that is, at the end of a destination-synchronized transfer or when DMA requests are no longer active).

**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 channel.

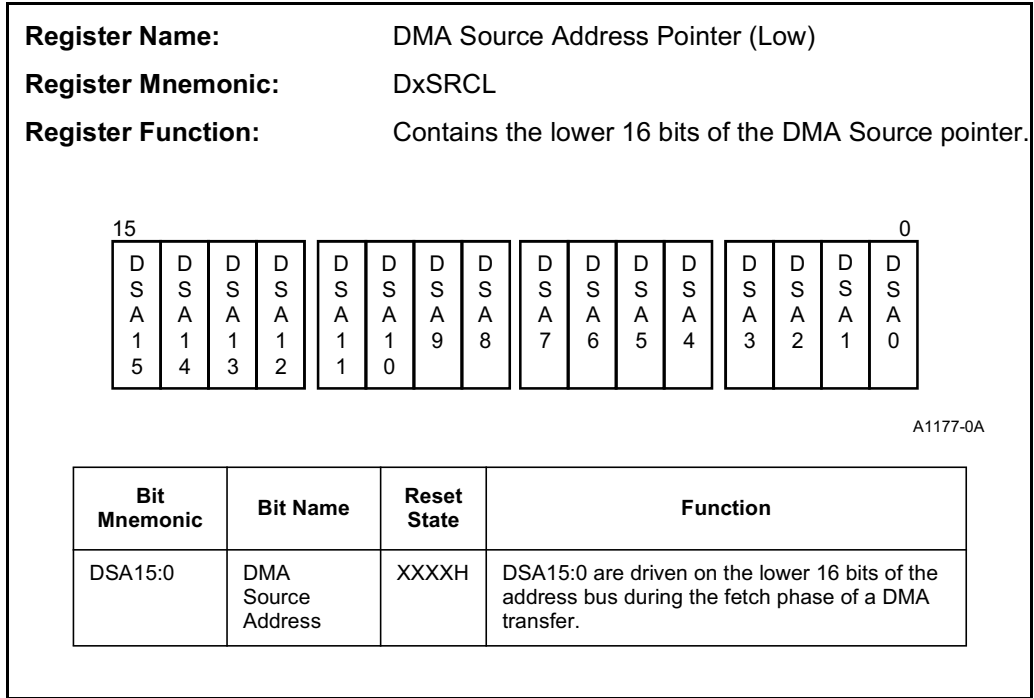
**10.2.1.1 Programming the Source and Destination Pointers**

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

- pointer address
- address space (memory or I/O)
- automatic pointer indexing (increment, decrement or no change) after transfer



## DIRECT MEMORY ACCESS UNIT



**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 10-11 on page 10-15). 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 two bits, increment and decrement, that control the indexing. If the increment and decrement bits for a pointer are programmed to the same value, then the pointer remains constant. The programmed data width (byte or word) for the channel automatically controls the amount that a pointer is incremented or decremented.





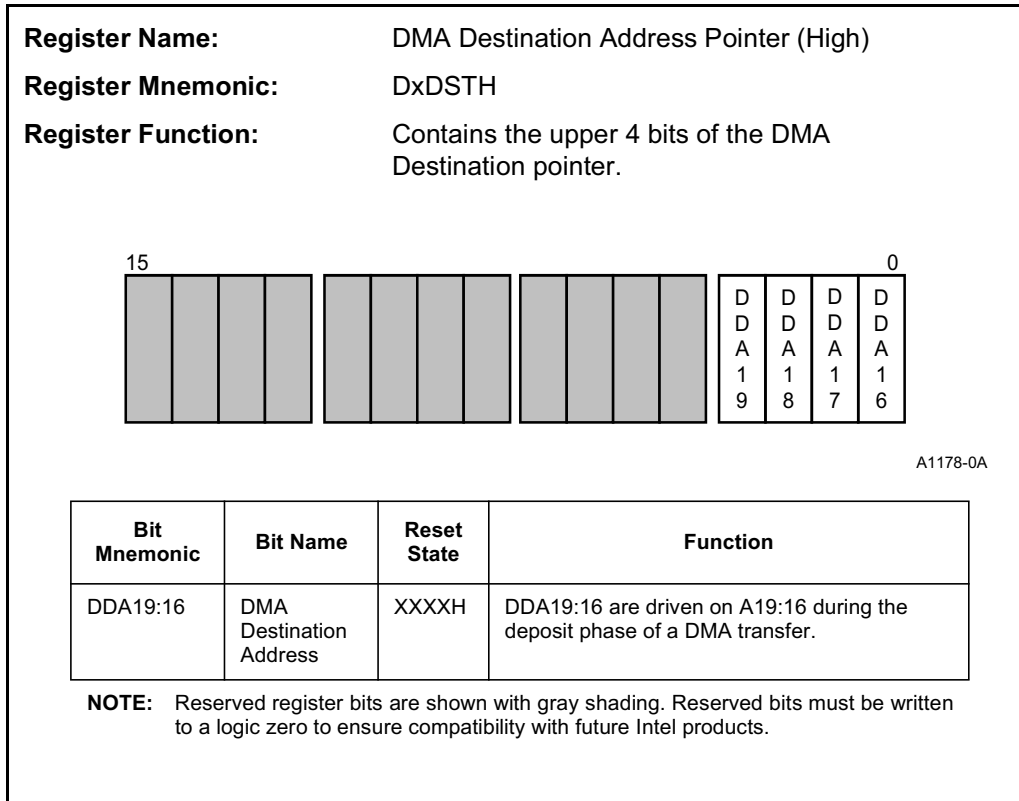
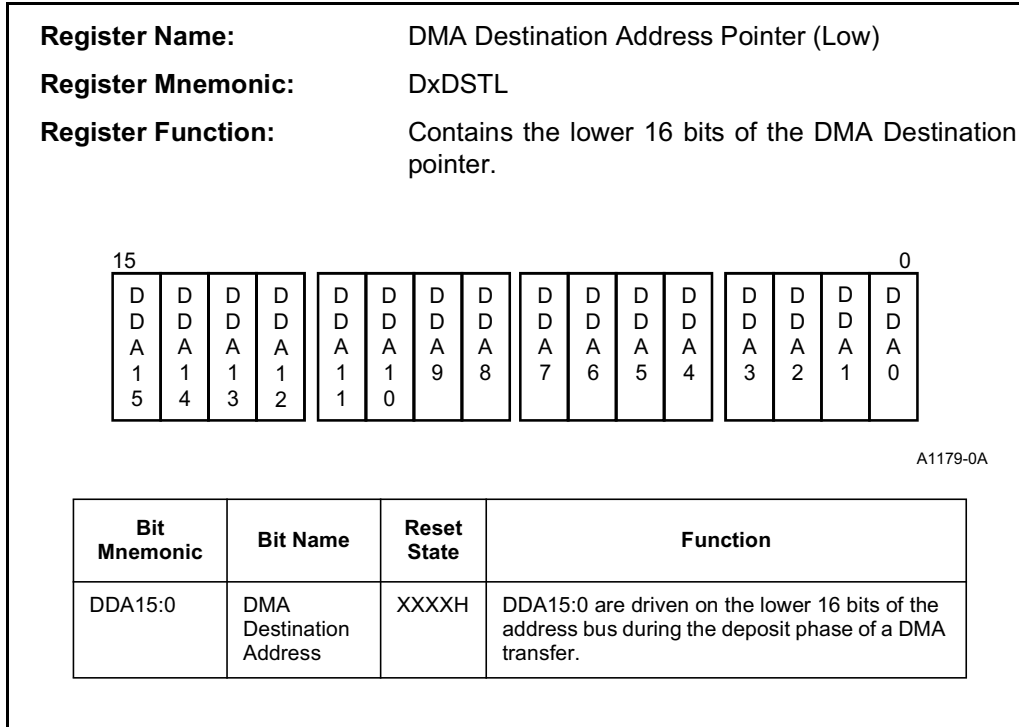


Figure 10-9. DMA Destination Pointer (High-Order Bits)



**Figure 10-10. DMA Destination Pointer (Low-Order Bits)**

**10.2.1.2 Selecting Byte or Word Size Transfers**

The WORD bit in the DMA Control Register (Figure 10-11) controls 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 transfers modify (increment or decrement) the pointer registers by two for each transfer, while byte transfers modify the pointer registers by one.



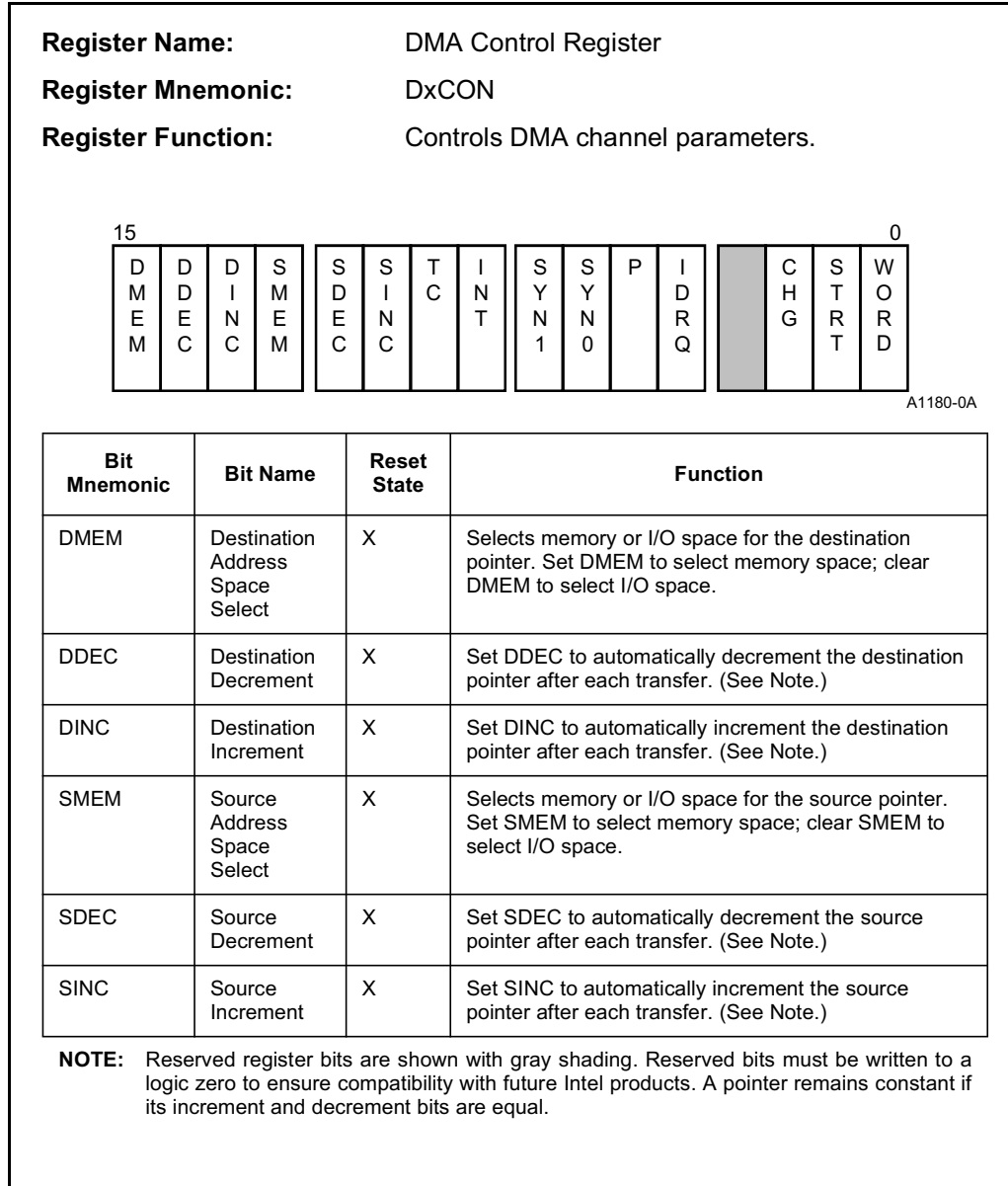
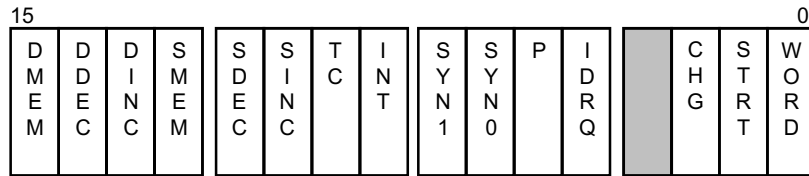


Figure 10-11. DMA Control Register

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

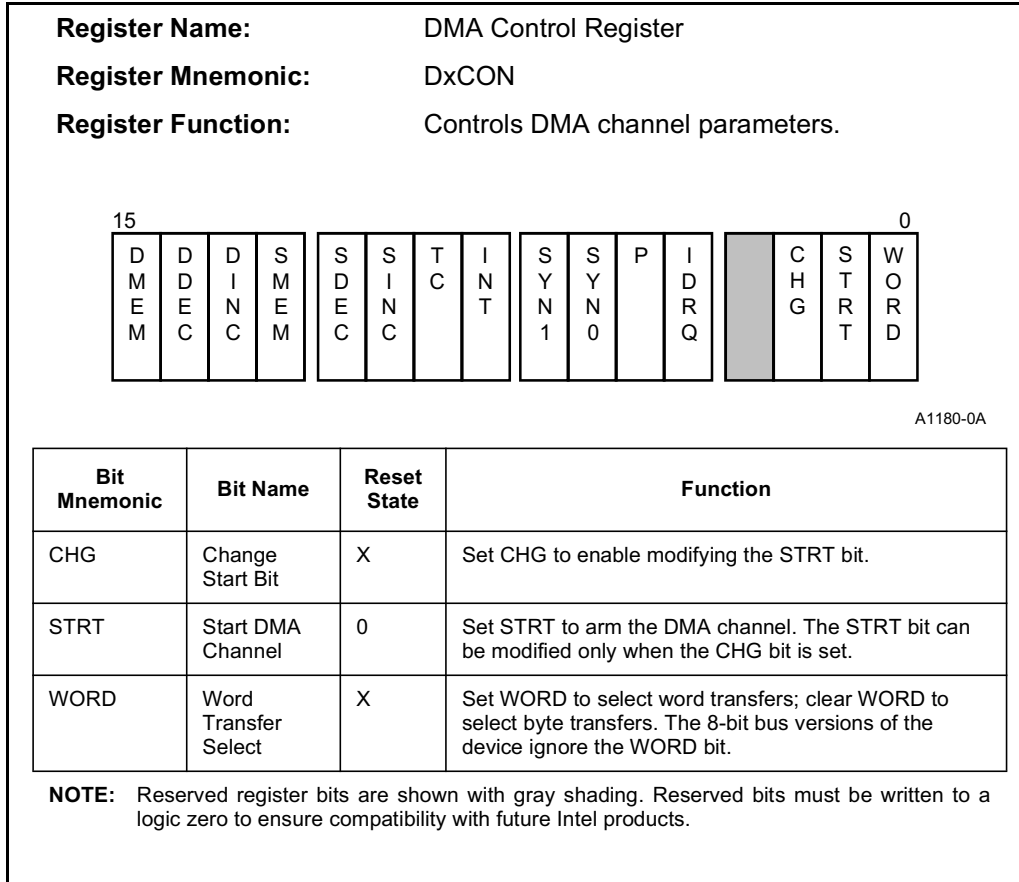


A1180-0A

| Bit Mnemonic | Bit Name                    | Reset State                  | Function  |      |      |                      |   |   |                |   |   |                     |   |   |                          |   |   |                              |
|--------------|-----------------------------|------------------------------|---|------|------|----------------------|---|---|----------------|---|---|---------------------|---|---|--------------------------|---|---|------------------------------|
| TC           | Terminal Count              | X                            | Set TC to terminate transfers on Terminal Count. This bit is ignored for unsynchronized transfers (that is, the DMA channel behaves as if TC is set, regardless of its condition).  |      |      |                      |   |   |                |   |   |                     |   |   |                          |   |   |                              |
| INT          | Interrupt                   | X                            | Set INT to generate an interrupt request on Terminal Count. The TC bit must be set to generate an interrupt.  |      |      |                      |   |   |                |   |   |                     |   |   |                          |   |   |                              |
| SYN1:0       | Synchronization Type        | XX                           | Selects channel synchronization:<br><table border="1"> <thead> <tr> <th>SYN1</th> <th>SYN0</th> <th>Synchronization Type</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>Unsynchronized</td> </tr> <tr> <td>0</td> <td>1</td> <td>Source-synchronized</td> </tr> <tr> <td>1</td> <td>0</td> <td>Destination-synchronized</td> </tr> <tr> <td>1</td> <td>1</td> <td>Reserved (do <b>not</b> use)</td> </tr> </tbody> </table> | SYN1 | SYN0 | Synchronization Type | 0 | 0 | Unsynchronized | 0 | 1 | Source-synchronized | 1 | 0 | Destination-synchronized | 1 | 1 | Reserved (do <b>not</b> use) |
| SYN1         | SYN0                        | Synchronization Type         |   |      |      |                      |   |   |                |   |   |                     |   |   |                          |   |   |                              |
| 0            | 0                           | Unsynchronized               |   |      |      |                      |   |   |                |   |   |                     |   |   |                          |   |   |                              |
| 0            | 1                           | Source-synchronized          |   |      |      |                      |   |   |                |   |   |                     |   |   |                          |   |   |                              |
| 1            | 0                           | Destination-synchronized     |   |      |      |                      |   |   |                |   |   |                     |   |   |                          |   |   |                              |
| 1            | 1                           | Reserved (do <b>not</b> use) |   |      |      |                      |   |   |                |   |   |                     |   |   |                          |   |   |                              |
| P            | Relative Priority           | X                            | Set P to select high priority for the channel; clear P to select low priority for the channel.  |      |      |                      |   |   |                |   |   |                     |   |   |                          |   |   |                              |
| IDRQ         | Internal DMA Request Select | X                            | Set IDRQ to select internal DMA requests and ignore the external DRQ pin. Clear IDRQ to select the DRQ pin as the source of DMA requests. When IDRQ is set, the channel must be configured for source-synchronized transfers (SYN1:0 = 01).   |      |      |                      |   |   |                |   |   |                     |   |   |                          |   |   |                              |

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

Figure 10-11. DMA Control Register (Continued)



**Figure 10-11. DMA Control Register (Continued)**

**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.

Internal DMA requests are selected by setting the IDRQ bit in the DMA Control Register (see Figure 10-11 on page 10-15) for the channel. The DMA channel ignores its DRQ pin when internal requests are programmed. Similarly, the DMA channel responds only 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 can recognize DMA requests. A channel is armed by setting its STRT (Start) bit in the DMA Control Register (Figure 10-11 on page 10-15). The STRT bit can be modified only if the CHG (Change Start) bit is set at the same time. The CHG bit is a safeguard to prevent accidentally arming 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 it is 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 (Figure 10-11 on page 10-15).

#### NOTE

The combination SYN1:0=11 is reserved and will result in unpredictable operation. When IDRQ is set (internal requests selected) the channel must always be programmed for source-synchronized transfers (SYN1:0=01).

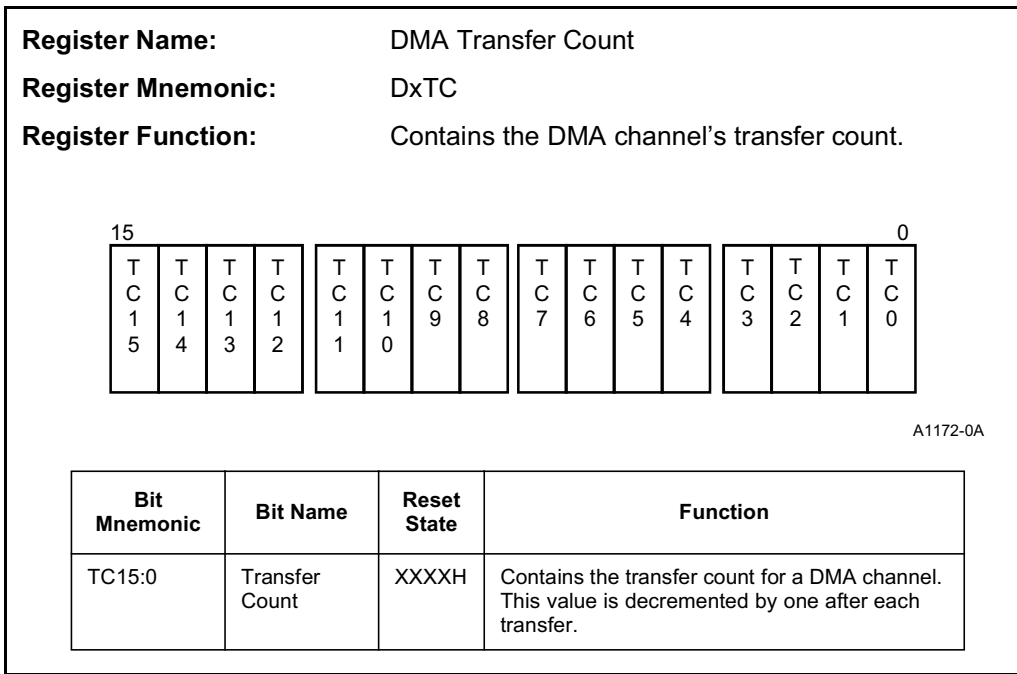
When programmed for unsynchronized transfers (SYN1:0=00), the DMA channel will begin to transfer data as soon as the STRT bit is set.

#### 10.2.1.6 Programming the Transfer Count Options

The Transfer Count Register (Figure 10-12) and the TC bit in the DMA Control Register (Figure 10-11 on page 10-15) are used to stop DMA transfers for a channel after a specified number of transfers have occurred.

The transfer count (the number of transfers desired) is written to the DMA Transfer Count Register. 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).





**Figure 10-12. Transfer Count Register**

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 (Figure 10-11 on page 10-15) must be set to generate an interrupt request.

**10.2.1.8 Setting the Relative Priority of a Channel**

The priority of a channel is controlled by the Priority bit in the DMA Control Register (Figure 10-11 on page 10-15). A channel may be assigned either high or low priority. If both channels are programmed to the same priority (i.e., both high or both low), the channels rotate priority.

### 10.2.2 Suspension of DMA Transfers

Whenever the CPU receives an NMI, all DMA activity is suspended at the end of the current transfer. The CPU suspends DMA activity by setting the DHLT bit in the Interrupt Status Register (Figure 8-14 on page 8-23). When an IRET instruction is executed, the CPU clears the DHLT bit and DMA transfers are allowed to resume. Software can read and write the DHLT bit.

#### NOTE

Do not write to the DHLT bit while Timer/Counter Unit interrupts are enabled. A conflict with the internal use of the register may cause incorrect processing of timer interrupts.

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

This section covers 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 set up a minimum of  $T_{CLIS}$  before CLKOUT falling and must be held a minimum of  $T_{CLIH}$  after CLKOUT falls. Refer to the data sheet for specific values.

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







## DIRECT MEMORY ACCESS UNIT

### 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 activate for the memory or I/O range that requires the acknowledge. The chip-select must be programmed to activate only when a DMA is in progress. Latched status line S6 can be used as a qualifier to the chip-select for situations in which the chip-select line will be active for both DMA and normal data accesses.

## 10.4 DMA UNIT EXAMPLES

Example 10-1 sets up channel 0 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.

Example 10-2 shows timed DMA transfers. A sawtooth waveform is created using DMA transfers to an A/D converter.





```
$MOD186

name      DMA_EXAMPLE_1

; This example shows code necessary to set up 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 perform 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
; 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, DOSRCL
          OUT   DX, AL                  ; AX=LOW 4 BYTES

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

; SOURCE POINTER DONE. REPEAT FOR DESTINATION.

          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, DODSTL
          OUT   DX, AX                  ; AX=LOW 4 BYTES
```

Example 10-1. Initializing the DMA Unit

## DIRECT MEMORY ACCESS UNIT



```
        MOV     DX, D0DSTH
        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, 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, AX                      ; PROGRAM LOW ADDR

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

; 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, AL                      ; 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.
```

**Example 10-1. Initializing the DMA Unit (Continued)**



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

; 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. ARM CHANNEL.

           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_1DB '80C186EC INTEGRATED PROCESSOR'
DEST_DATA_1DB  30 DUP('MITCH')                ; JUNK DATA FOR TEST

DISK_BUFF DB    512 DUP(?)

DATA_SEG  ENDS
          END  START
```

**Example 10-1. Initializing the DMA Unit (Continued)**

```

$mod186
name      DMA_EXAMPLE_1

; This example sets up the DMA Unit to perform a transfer from memory to
; I/O space every 22 uS. 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, set up 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

; 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, DOSRCL
          OUT     DX, AX                  ; AX=LOW 4 BYTES

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

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

          MOV     DX, DODSTH
          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, DOTC                ; TO GET A FULL SCALE
          OUT     DX, AX

; PROGRAM IDRQ MUX

          MOV     DX, DMAPRI
          MOV     AX, 00H                 ; TIMER2 IS IDRQ SOURCE
                                          ; MODULES HAVE EQUAL PRIORITY
          OUT     DX, AX

```

Example 10-2. Timed DMA Transfers



```
; 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_DATADB 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)**

DIRECT MEMORY ACCESS UNIT







**11**

# **Math Coprocessing**





intel®

I

The core has an Escape Trap (ET) bit in the PCB Relocation Register (Figure 4-1 on page 4-2) to control the availability of math coprocessing. If the ET bit is set, an attempted numerics execution results in a Type 7 interrupt. The 80C187 will not work with the 8-bit bus version of the processor because all 80C187 accesses must be 16-bit. The 80C188 Modular Core automatically traps ESC (numerics) opcodes to the Type 7 interrupt, regardless of Relocation Register programming.

### **11.3 THE 80C187 MATH COPROCESSOR**

The 80C187 performance is due to its 80-bit internal architecture. It contains three units: a Floating Point Unit, a Data Interface and Control Unit and a Bus Control Logic Unit. The foundation of the Floating Point Unit is an 8-element register file, which can be used either as individually addressable registers or as a register stack. The register file allows storage of intermediate results in the 80-bit format. The Floating Point Unit operates under supervision of the Data Interface and Control Unit. The Bus Control Logic Unit maintains handshaking and communications with the host microprocessor. The 80C187 has built-in exception handling.

The 80C187 executes code written for the Intel387™ DX and Intel387 SX math coprocessors. The 80C187 conforms to ANSI/IEEE Standard 754-1985.

#### **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; for example, FSQRT always takes the square root of the number in the top stack element. Other instructions allow or require the programmer to specify the operand(s) explicitly along with the instruction mnemonic. Still other instructions accept one explicit operand and one implicit operand (usually the top stack element).

As with the basic (non-numerics) instruction set, there are two types of operands for coprocessor instructions, source and destination. Instruction execution does not alter a source operand. Even when an instruction converts the source operand from one format to another (for example, real to integer), the coprocessor performs the conversion in a work area to preserve the source operand. A destination operand differs from a source operand because the 80C187 can alter the register when it receives the result of the operation. For most destination operands, the coprocessor usually replaces the destinations with results.





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**

| <b>Addition</b>       |                                | <b>Division</b>         |                                  |
|-----------------------|--------------------------------|-------------------------|----------------------------------|
| FADD                  | Add real                       | FDIV                    | Divide real                      |
| FADDP                 | Add real and pop               | FDIVP                   | Divide real and pop              |
| FIADD                 | Integer add                    | FIDIV                   | Integer divide                   |
| <b>Subtraction</b>    |                                | FDIVR                   | Divide real reversed             |
| FSUB                  | Subtract real                  | FDIVRP                  | Divide real reversed and pop     |
| FSUBP                 | Subtract real and pop          | FIDIVR                  | Integer divide reversed          |
| FISUB                 | Integer subtract               | <b>Other Operations</b> |                                  |
| FSUBR                 | Subtract real reversed         | FSQRT                   | Square root                      |
| FSUBRP                | Subtract real reversed and pop | FSCALE                  | Scale                            |
| FISUBR                | Integer subtract reversed      | FPREM                   | Partial remainder                |
| <b>Multiplication</b> |                                | FRNDINT                 | Round to integer                 |
| FMUL                  | Multiply real                  | FXTRACT                 | Extract exponent and significand |
| FMULP                 | Multiply real and pop          | FABS                    | Absolute value                   |
| FIMUL                 | Integer multiply               | 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 (see Table 11-4) perform the core calculations for common trigonometric, hyperbolic, inverse hyperbolic, logarithmic and exponential functions. Use prologue code to reduce arguments to a range accepted by the instruction. Use epilogue code to adjust the result to the range of the original arguments. The transcendentals operate on the top one or two stack elements and return their results to the stack.

**Table 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 2 bytes long, save memory space.

**Table 11-5. 80C187 Constant Instructions**

|        |                    |
|--------|--------------------|
| FLDZ   | Load + 0.1         |
| FLD1   | Load +1.0          |
| FLDPI  | Load               |
| FLDL2T | Load $\log_2 10$   |
| FLDL2E | Load $\log_2 e$    |
| FLDLG2 | Load $\log_{10} 2$ |
| FLDLN2 | Load $\log_e 2$    |

**11.3.1.6 Processor Control Instructions**

Computations do not use the processor control instructions; these instructions are available for activities at the operating system level. This group (see Table 11-6) includes initialization, exception handling and task switching instructions.

**Table 11-6. 80C187 Processor Control Instructions**

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





### 11.3.2 80C187 Data Types

The microprocessor/math coprocessor combination supports seven data types:

- ~~Word Integer~~ **Word Integer** — A signed 16-bit numeric value. All operations assume a 2 complement representation.
- **Short Integer** — A signed 32-bit numeric value (double word). All operations assume a 2 complement representation.
- **Long Integer** — A signed 64-bit numeric value (quad word). All operations assume a 2 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 (as shown in 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:  $\overline{\text{NCS}}$ ,  $\text{BUSY}$ ,  $\text{PEREQ}$  and  $\text{ERROR}$ . These pins are multiplexed with  $\overline{\text{MCS3}}$ ,  $\text{TEST}$ ,  $\overline{\text{MCS0}}$ , and  $\overline{\text{MCS1}}$ , respectively. When the processor leaves reset, the presence of the 80C187 automatically places the processor in Numerics Mode and configures the pins correctly.  $\overline{\text{MCS2}}$  retains its function as a chip-select and the processor 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.

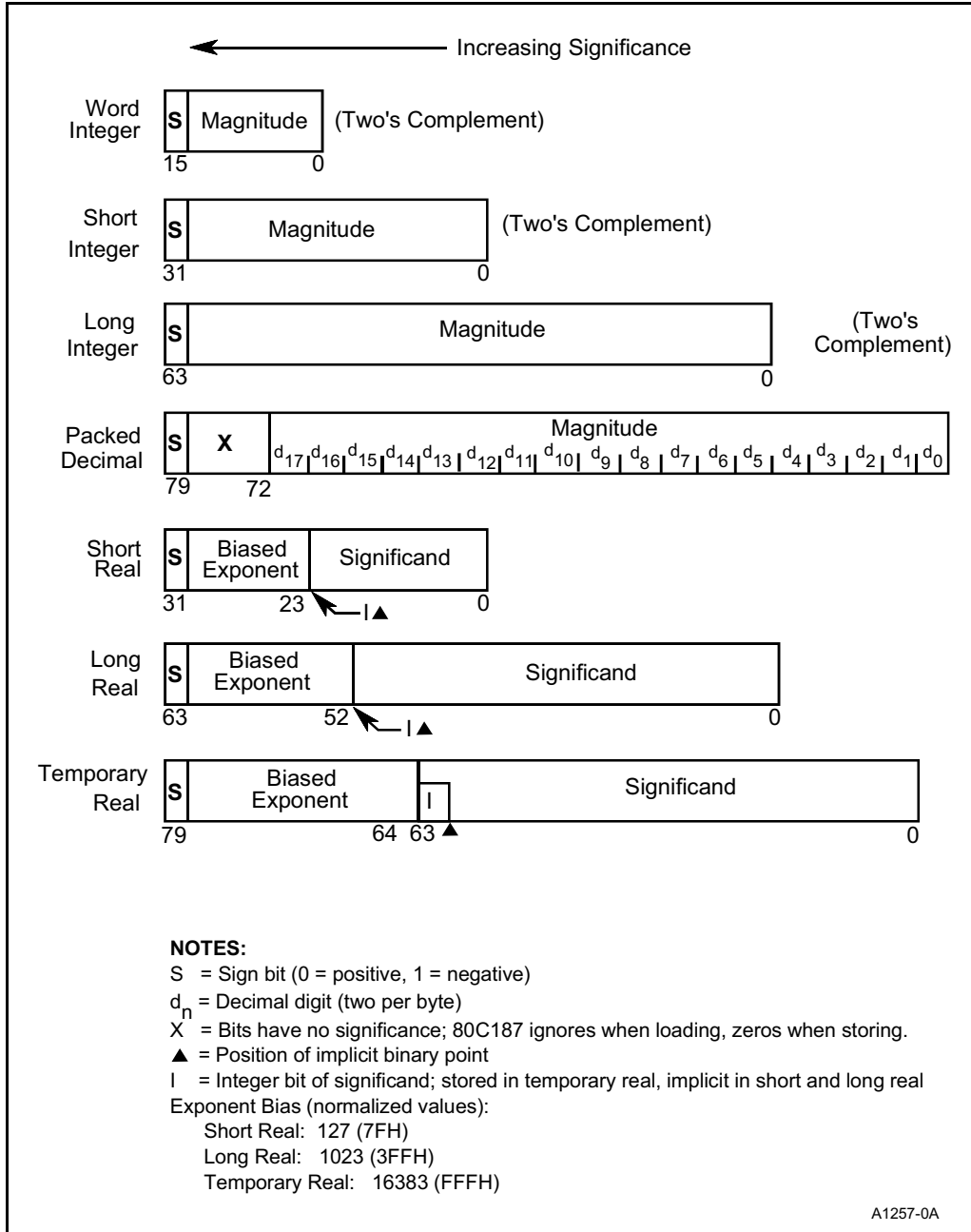


Figure 11-1. 80C187-Supported Data Types

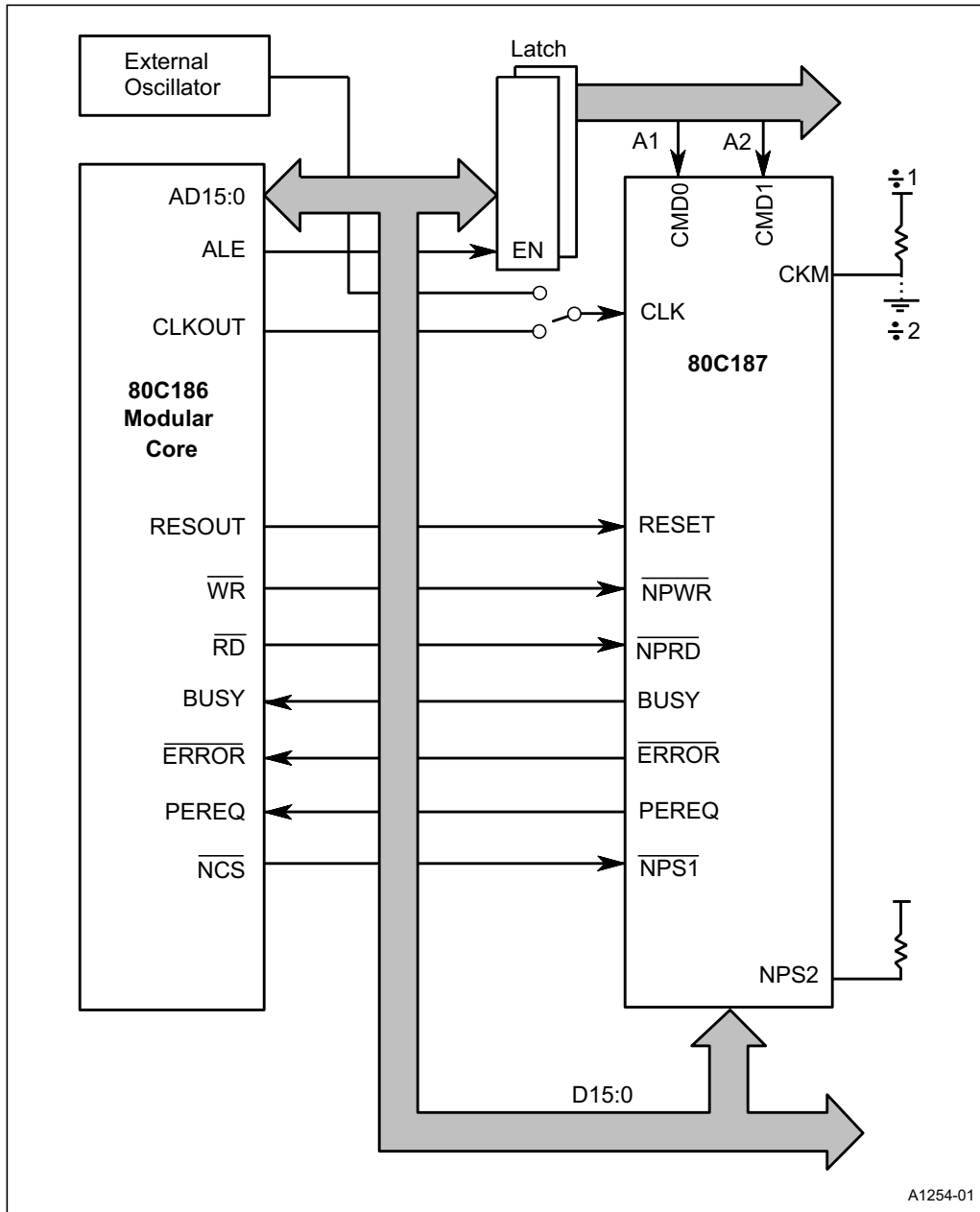


Figure 11-2. 80C186 Modular Core Family/80C187 System Configuration



### 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 that determines whether it uses the input clock directly or divided by two. Direct clocking works up to 12.5 MHz, which makes it convenient to feed the clock input from the microprocessor. Beyond 12.5 MHz, the 80C187 must use a multiply-by-two clock input up to a maximum of 32 MHz. The microprocessor and the math coprocessor have correct timing relationships, even with operation at different frequencies.

### 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 data transfer between memory and the 80C187 (via the microprocessor) takes at least 17 processor clocks.

**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 when the Escape Trap (ET) bit in the Relocation Register is a zero and the 80C187 is not present, its operation is indeterminate. Even the FINIT/FNINIT initialization instruction (used in the past to test the presence of a coprocessor) fails without the 80C187. If an application offers the 80C187 as an option, problems can be prevented in one of three ways:

- Remove all numerics (ESC) instructions, including code that checks for the presence of the 80C187.
- Use a jumper or switch setting to indicate the presence of the 80C187. The program can interrogate the jumper or switch setting and branch away from numerics instructions when the 80C187 socket is empty.
- Trick the microprocessor into predictable operation when the 80C187 socket is empty. The fix is placing pull-up or pull-down resistors on certain data and handshaking lines so the CPU reads a recognizable Opcode Status Word. This solution requires a detailed knowledge of the interface.





Bus cycles involving the 80C187 Math Coprocessor behave exactly like other I/O bus cycles with respect to bus priority. See “System Design Tips” for information on integrating 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 through hardware connected to the processor chip-select to cover the math coprocessor port addresses, its ready programming is in force and can provide bus ready for coprocessor accesses. The user must verify that there are no conflicts from other hardware connected to that chip-select pin.

A chip-select pin goes active on 80C187 accesses if you program it for a range including the math coprocessor I/O ports. The converse is not true — a non-80C187 access cannot activate  $\overline{\text{NCS}}$  (numerics coprocessor select), regardless of programming.

In a buffered system, it is customary to place the 80C187 on the local bus. Since  $\overline{\text{DTR}}$  and  $\overline{\text{DEN}}$  function normally during 80C187 transfers, you must qualify  $\overline{\text{DEN}}$  with  $\overline{\text{NCS}}$  (see Figure 11-3). Otherwise, contention between the 80C187 and the transceivers occurs on read cycles to the 80C187.

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



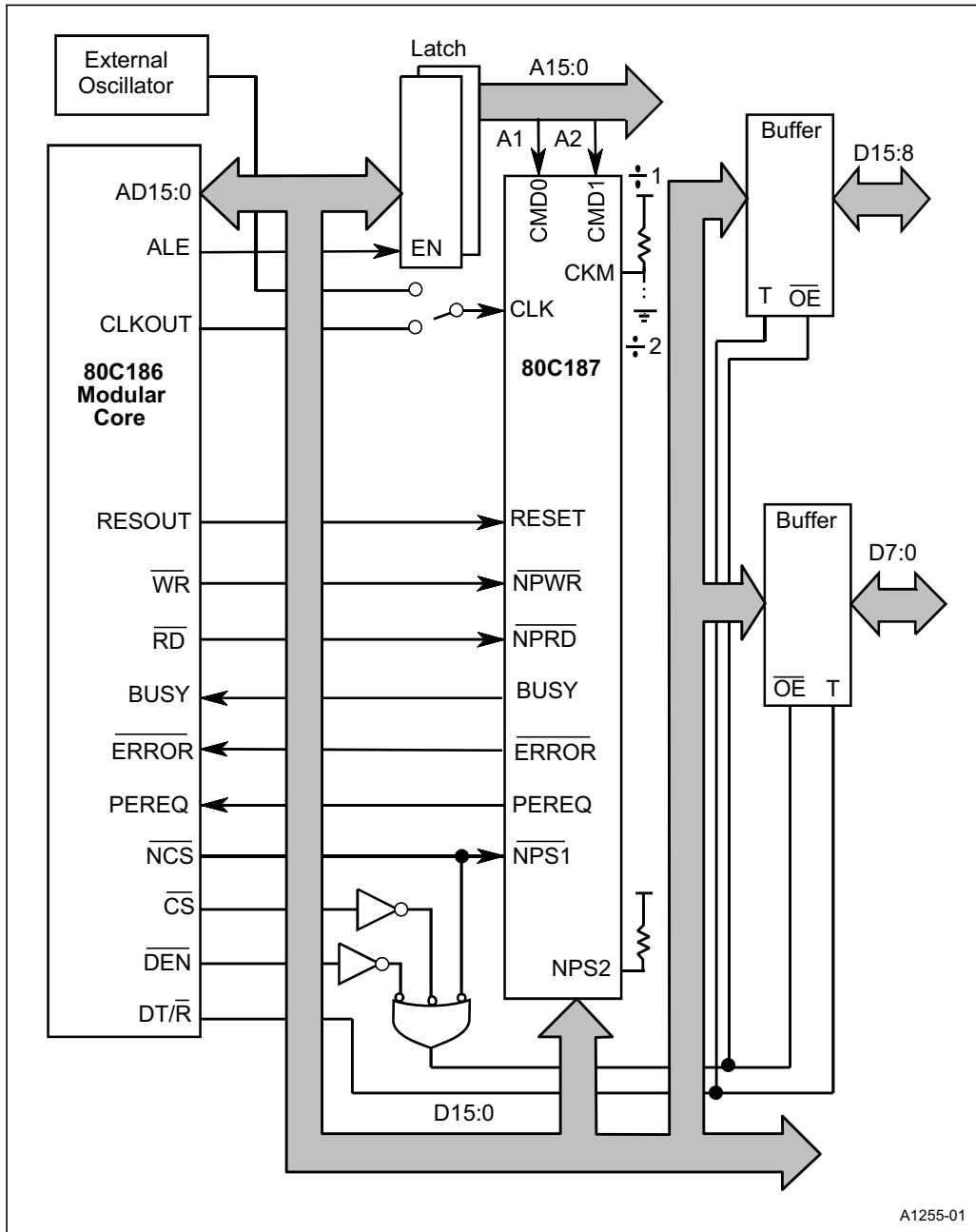


Figure 11-3. 80C187 Configuration with a Partially Buffered Bus

#### 11.4.4 Exception Trapping

The 80C187 detects six error conditions that can occur during instruction execution. The 80C187 can default fix-ups or signal exceptions to the microprocessor `_____` pin. The processor tests `ERROR` at the beginning of numeric instructions, so it traps an exception on the **next** attempted numeric instruction after it occurs. When `ERROR` tests active, the processor executes a Type 16 interrupt.

There is no automatic exception-trapping on the last numeric instruction of a series. If the last numeric instruction writes an invalid result to memory, subsequent non-numeric 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 `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 `ERROR`. The latch gets cleared during the exception-handler routine. Use an additional flip-flop to latch `PEREQ` to maintain the correct hand-shaking 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.

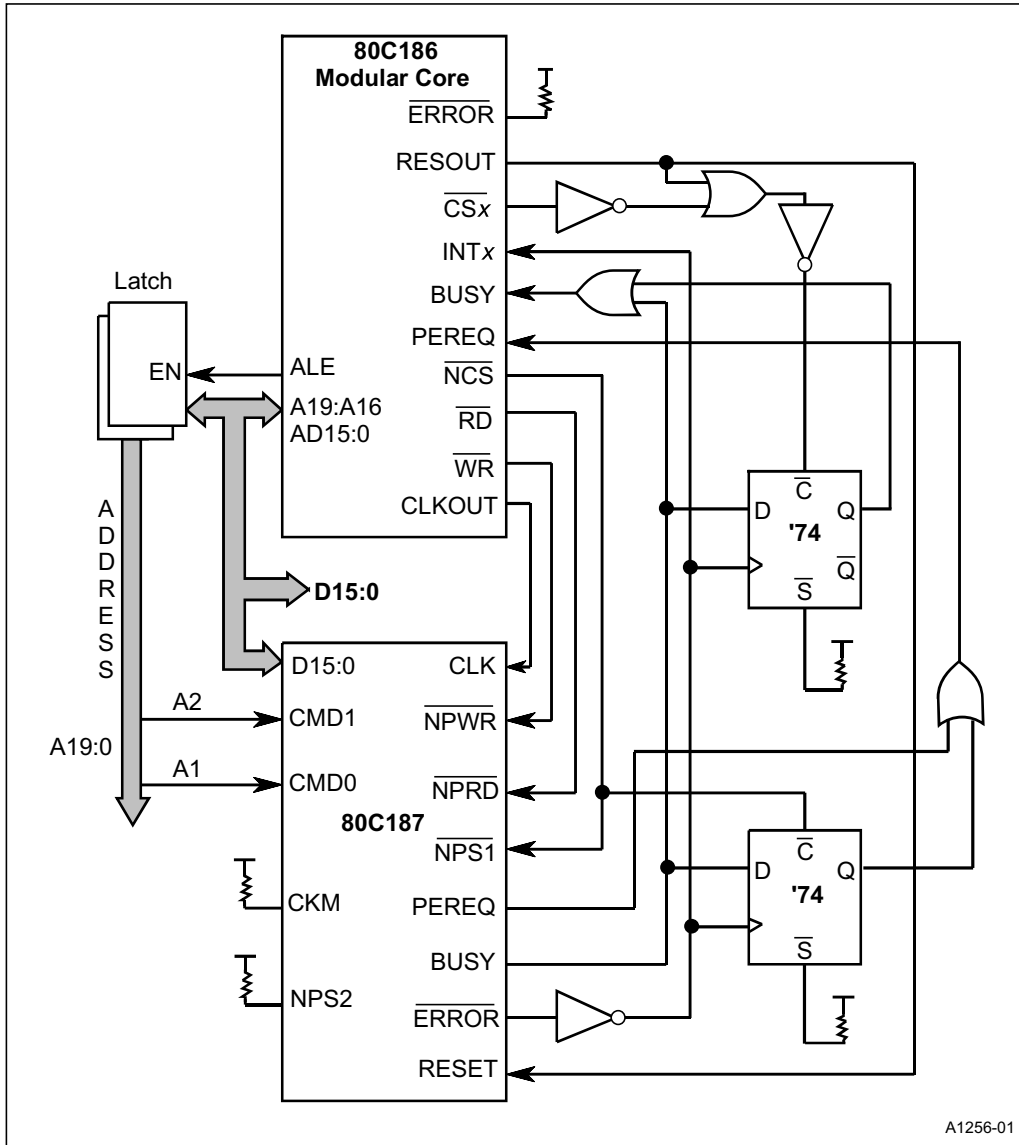


Figure 11-4. 80C187 Exception Trapping via Processor Interrupt Pin





```

$mod186
name    example_80C187_init
;
;FUNCTION:    This function initializes the 80C187 numerics coprocessor.
;
;SYNTAX:     extern unsigned char far 187_init(void);
;
;INPUTS:     None
;
;OUTPUTS:    unsigned char - 0000h -> False -> coprocessor not initialized
;                ffffh -> True  -> coprocessor initialized
;
;NOTE:       Parameters are passed on the stack as required by
;            high-level languages.
;
lib_80186   segment public 'code'
            assume cs:lib_80186

            public  _187_init

_187_initproc far

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

            cli                      ;disable maskable interrupts

            fninit                    ;init 80C187 processor
            fnstcw [bp-2]            ;get current control word

            sti                      ;enable interrupts

            mov    ax, [bp-2]
            and    ax, 0300h          ;mask off unwanted control bits
            cmp    ax, 0300h          ;PC bits = 11
            je     Ok                 ;yes: processor ok
            xor    ax, ax             ;return false (80C187 not ok)
            pop    bp                ;restore caller's bp
            ret

Ok:         and    [bp-2], 0fffeh      ;unmask possible exceptions
            fldcw  [bp-2]

            mov    ax, 0ffffh         ;return true (80C187 ok)
            pop    bp                ;restore caller's bp
            ret

_187_initendp
lib_80186ends
end

```

**Example 11-1. Initialization Sequence for 80C187 Math Coprocessor**

```

$mod186
$modc187

name    example_80C187_proc

;DESCRIPTION:  This code section uses the 80C187 FSINCOS transcendental
;              instruction to convert the locus of a point from polar
;              to Cartesian coordinates.
;
;VARIABLES:    The variables consist of the radius, r, and the angle, theta.
;              Both are expressed as 32-bit reals and 0 <= theta <= pi/4.
;
;RESULTS:      The results of the computation are the coordinates x and y
;              expressed as 32-bit reals.
;
;NOTES:        This routine is coded for Intel ASM86. It is not set up as an
;              HLL-callable routine.
;
;              This code assumes that the 80C187 has already been initialized.
;
    assume cs:code, ds:data

    data    segment at 0100h
            r        dd x.xxxx        ;substitute real operand
            theta    dd x.xxxx        ;substitute real operand
            x        dd ?
            y        dd ?
    data    ends

    code    segment at 0080h

convert    proc far
    mov     ax, data
    mov     ds, ax

            fld     r                    ;load radius
            fld     theta                ;load angle
            fsincos                ;st=cos, st(1)=sin
            fmul    st, st(2)           ;compute x
            fstp   x                    ;store to memory and pop
            fmul    st, st(1)          ;compute y
            fstp   y                    ;store to memory and pop

convert    endp

code    ends
end

```

**Example 11-2. Floating Point Math Routine Using FSINCOS**

intel<sup>®</sup>

12

**ONCE Mode**

|





## CHAPTER 12 ONCE MODE

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

ONCE Mode electrically isolates the device from the rest of the board logic. This isolation allows a bed-of-nails tester to drive the device pins directly for more accurate and thorough testing. An in-circuit emulation probe uses ONCE Mode to isolate a surface-mounted device from board logic and essentially “take over” operation of the board (without removing the soldered device from the board).

### 12.1 ENTERING/LEAVING ONCE MODE

Forcing  $\overline{UCS}$  and  $\overline{LCS}$  low while  $\overline{RESIN}$  is asserted (low) enables ONCE Mode (see Figure 12-1). Maintaining  $\overline{UCS}$  and  $\overline{LCS}$  and  $\overline{RESIN}$  low continues to keep ONCE Mode active. Returning  $\overline{UCS}$  and  $\overline{LCS}$  high exits ONCE Mode.

However, it is possible to keep ONCE Mode always active by deasserting  $\overline{RESIN}$  while keeping  $\overline{UCS}$  and  $\overline{LCS}$  low. Removing  $\overline{RESIN}$  “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  $\overline{RESIN}$  is driven high. Asserting  $\overline{RESIN}$  exits ONCE Mode, assuming  $\overline{UCS}$  and  $\overline{LCS}$  do not also remain low (see Figure 12-1).



ONCE MODE

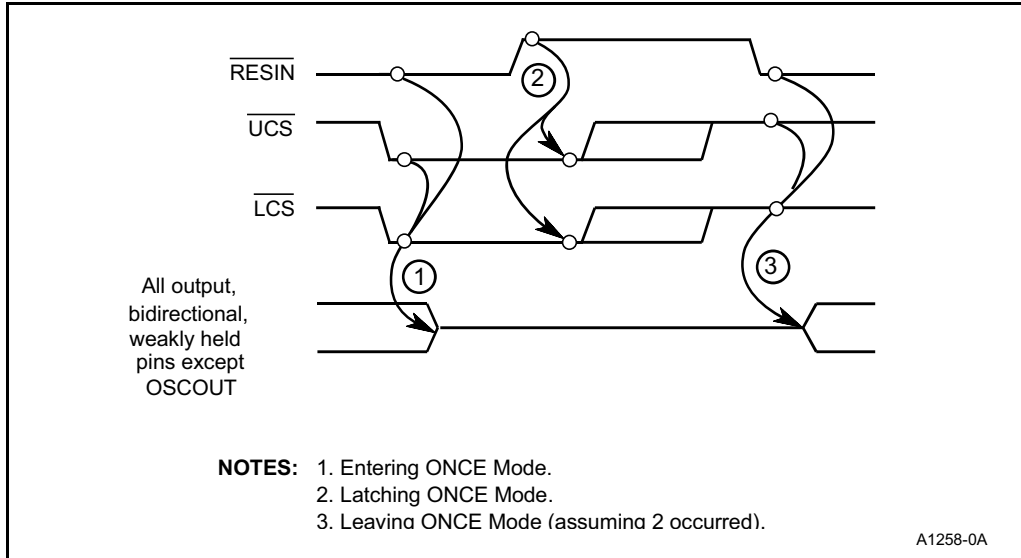


Figure 12-1. Entering/Leaving ONCE Mode





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

### A.1 80C186 INSTRUCTION SET ADDITIONS

This section describes the seven instructions that were added to the base 8086/8088 instruction set to make the instruction set for the 80C186 Modular Core family. These instructions did not exist in the 8086/8088 instruction set.

- Data transfer instructions
  - PUSHA
  - POPA
- String instructions
  - INS
  - OUTS
- High-level instructions
  - ENTER
  - LEAVE
  - BOUND

#### A.1.1 Data Transfer Instructions

##### PUSHA/POPA

PUSHA (push all) and POPA (pop all) allow all general-purpose registers to be stacked and unstacked. The PUSH instruction pushes all CPU registers (except as noted below) onto the stack. The POP instruction pops all registers pushed by PUSH 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 POP is executed, the Stack Pointer value is popped, but ignored. Note that this instruction does not save segment registers (CS, DS, SS, ES), the Instruction Pointer (IP), the Processor Status Word or any integrated peripheral registers.



### A.1.2 String Instructions

**INS** *source\_string, port*

INS (in string) performs block input from an I/O port to memory. The port address is placed in the DX register. The memory address is placed in the DI register. This instruction uses the ES segment register (which cannot be overridden). After the data transfer takes place, the pointer register (DI) increments or decrements, depending on the value of the Direction Flag (DF). The pointer register changes by one for byte transfers or by two for word transfers.

**OUTS** *port, destination\_string*

OUTS (out string) performs block output from memory to an I/O port. The port address is placed in the DX register. The memory address is placed in the SI register. This instruction uses the DS segment register, but this may be changed with a segment override instruction. After the data transfer takes place, the pointer register (SI) increments or decrements, depending on the value of the Direction Flag (DF). The pointer register changes by one for byte transfers or by two for word transfers.

### A.1.3 High-Level Instructions

**ENTER** *size, level*

ENTER creates the stack frame required by most block-structured high-level languages. The first parameter, *size*, specifies the number of bytes of dynamic storage to be allocated for the procedure being entered (16-bit value). The second parameter, *level*, is the lexical nesting level of the procedure (8-bit value). Note that the higher the lexical nesting level, the lower the procedure is in the nesting hierarchy.

The lexical nesting level determines the number of pointers to higher level stack frames copied into the current stack frame. This list of pointers is called the *display*. The first word of the display points to the previous stack frame. The display allows access to variables of higher level (lower lexical nesting level) procedures.

After ENTER creates a display for the current procedure, it allocates dynamic storage space. The Stack Pointer decrements by the number of bytes specified by *size*. All PUSH and POP operations in the procedure use this value of the Stack Pointer as a base.

Two forms of ENTER exist: non-nested and nested. A lexical nesting level of 0 specifies the non-nested form. In this situation, BP is pushed, then the Stack Pointer is copied to BP and decremented by the size of the frame. If the lexical nesting level is greater than 0, the nested form is used. Figure A-1 gives the formal definition of ENTER.



```
The following listing gives the formal definition of the
ENTER instruction for all cases.
LEVEL denotes the value of the second operand.

Push BP
Set a temporary value FRAME_PTR: = SP
If LEVEL > 0 then
    Repeat (LEVEL - 1) times:
        BP:=BP - 2
        Push the word pointed to by BP
    End Repeat
    Push FRAME_PTR
End if
BP:=FRAME_PTR
SP:=SP - first operand
```

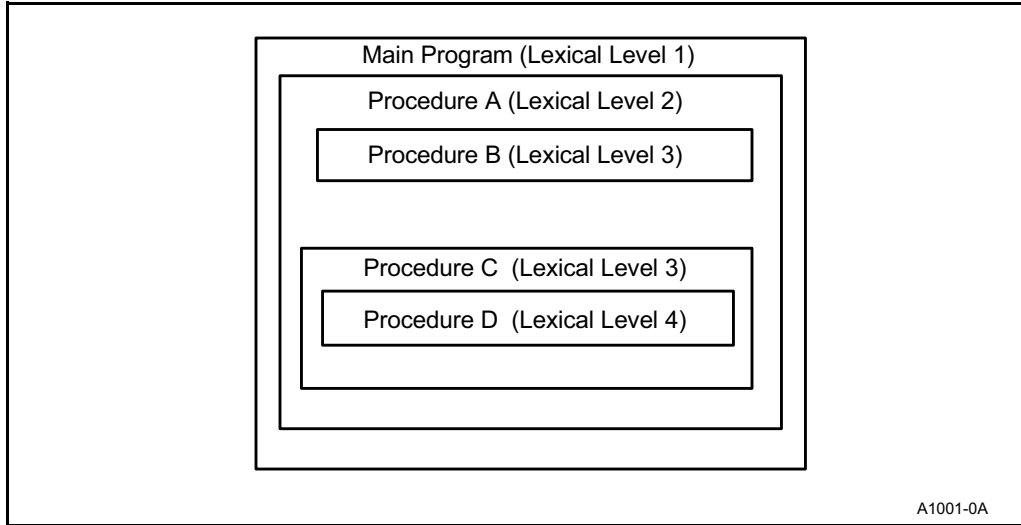
**Figure A-1. Formal Definition of ENTER**

ENTER treats a reentrant procedure as a procedure calling another procedure at the same lexical level. A reentrant procedure can address only its own variables and variables of higher-level calling procedures. ENTER ensures this by copying only stack frame pointers from higher-level procedures.

Block-structured high-level languages use lexical nesting levels to control access to variables of previously nested procedures. For example, assume for Figure A-2 that Procedure A calls Procedure B, which calls Procedure C, which calls Procedure D. Procedure C will have access to the variables of Main and Procedure A, but not to those of Procedure B because Procedures C and B operate at the same lexical nesting level.

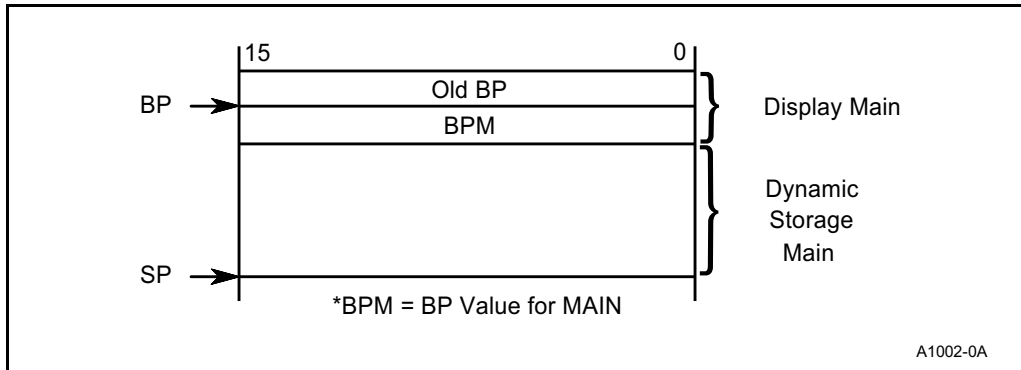
The following is a summary of the variable access for Figure A-2.

1. Main has variables at fixed locations.
2. Procedure A can access only the fixed variables of Main.
3. Procedure B can access only the variables of Procedure A and Main.  
Procedure B cannot access the variables of Procedure C or Procedure D.
4. Procedure C can access only the variables of Procedure A and Main.  
Procedure C cannot access the variables of Procedure B or Procedure D.
5. Procedure D can access the variables of Procedure C, Procedure A and Main.  
Procedure D cannot access the variables of Procedure B.



**Figure A-2. Variable Access in Nested Procedures**

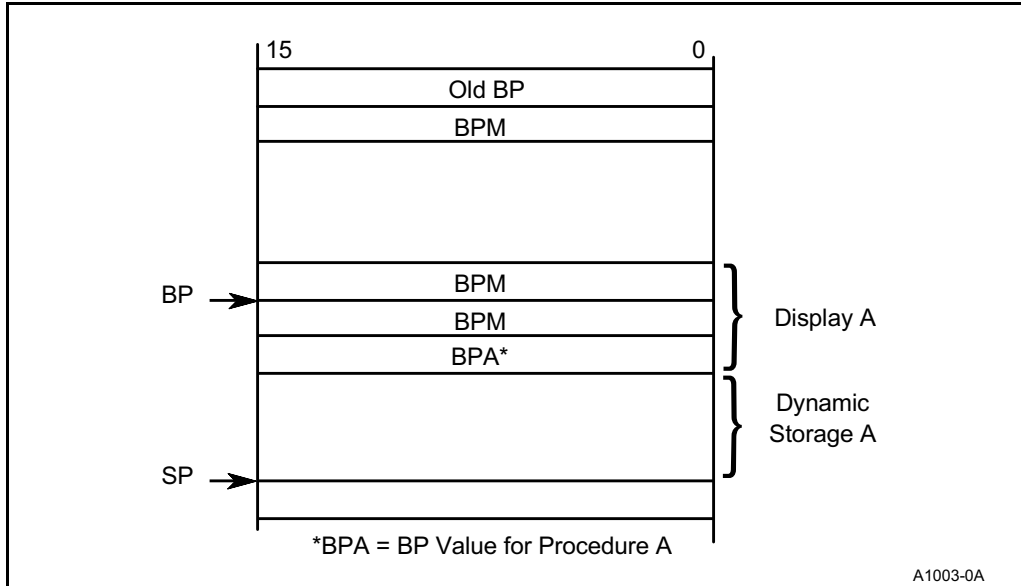
The first ENTER, executed in the Main Program, allocates dynamic storage space for Main, but no pointers are copied. The only word in the display points to itself because no previous value exists to return to after LEAVE is executed (see Figure A-3).



**Figure A-3. Stack Frame for Main at Level 1**

After Main calls Procedure A, ENTER creates a new display for Procedure A. The first word points to the previous value of BP (BPM). The second word points to the current value of BP (BPA). BPM contains the base for dynamic storage in Main. All dynamic variables for Main will be at a fixed offset from this value (see Figure A-4).





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

After Procedure A calls Procedure B, ENTER creates the display for Procedure B. The first word of the display points to the previous value of BP (BPA). The second word points to the value of BP for MAIN (BPM). The third word points to the BP for Procedure A (BPA). The last word points to the current BP (BPB). Procedure B can access variables in Procedure A or Main via the appropriate BP in the display (see Figure A-5).

After Procedure B calls Procedure C, ENTER creates the display for Procedure C. The first word of the display points to the previous value of BP (BPB). The second word points to the value of BP for MAIN (BPM). The third word points to the value of BP for Procedure A (BPA). The fourth word points to the current BP (BPC). Because Procedure B and Procedure C have the same lexical nesting level, Procedure C cannot access variables in Procedure B. The only pointer to Procedure B in the display of Procedure C exists to allow the LEAVE instruction to collapse the Procedure C stack frame (see Figure A-6).

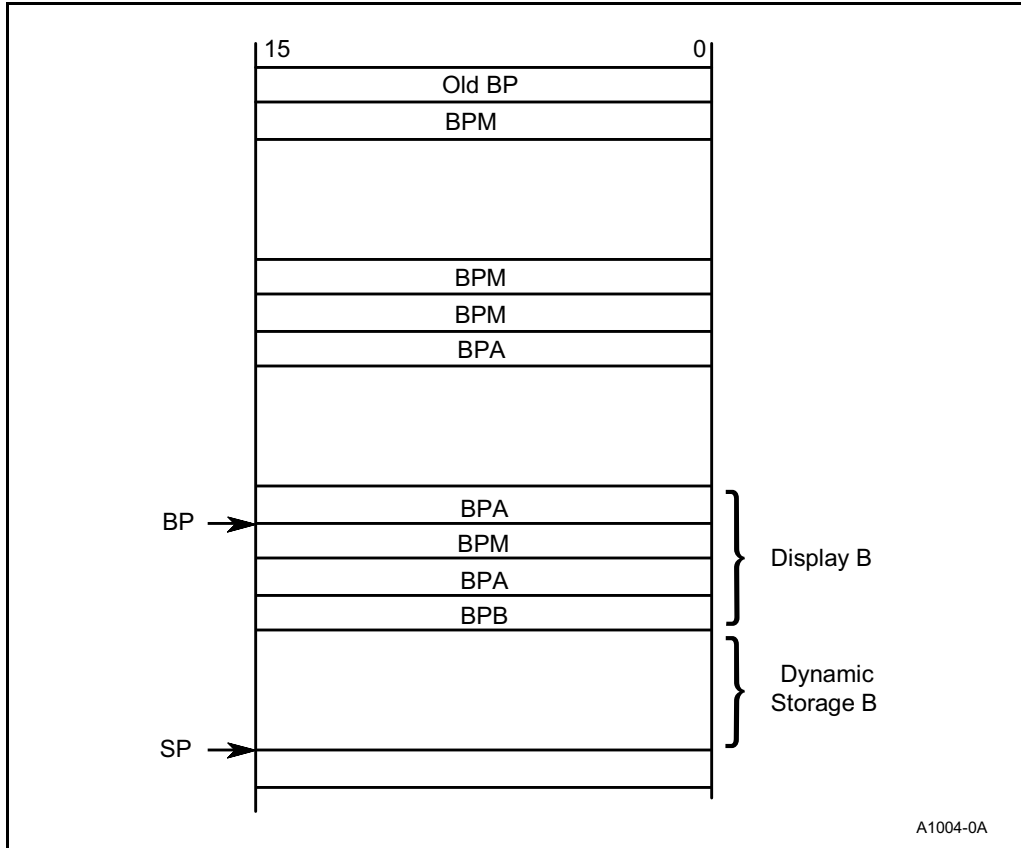


Figure A-5. Stack Frame for Procedure B at Level 3 Called from A



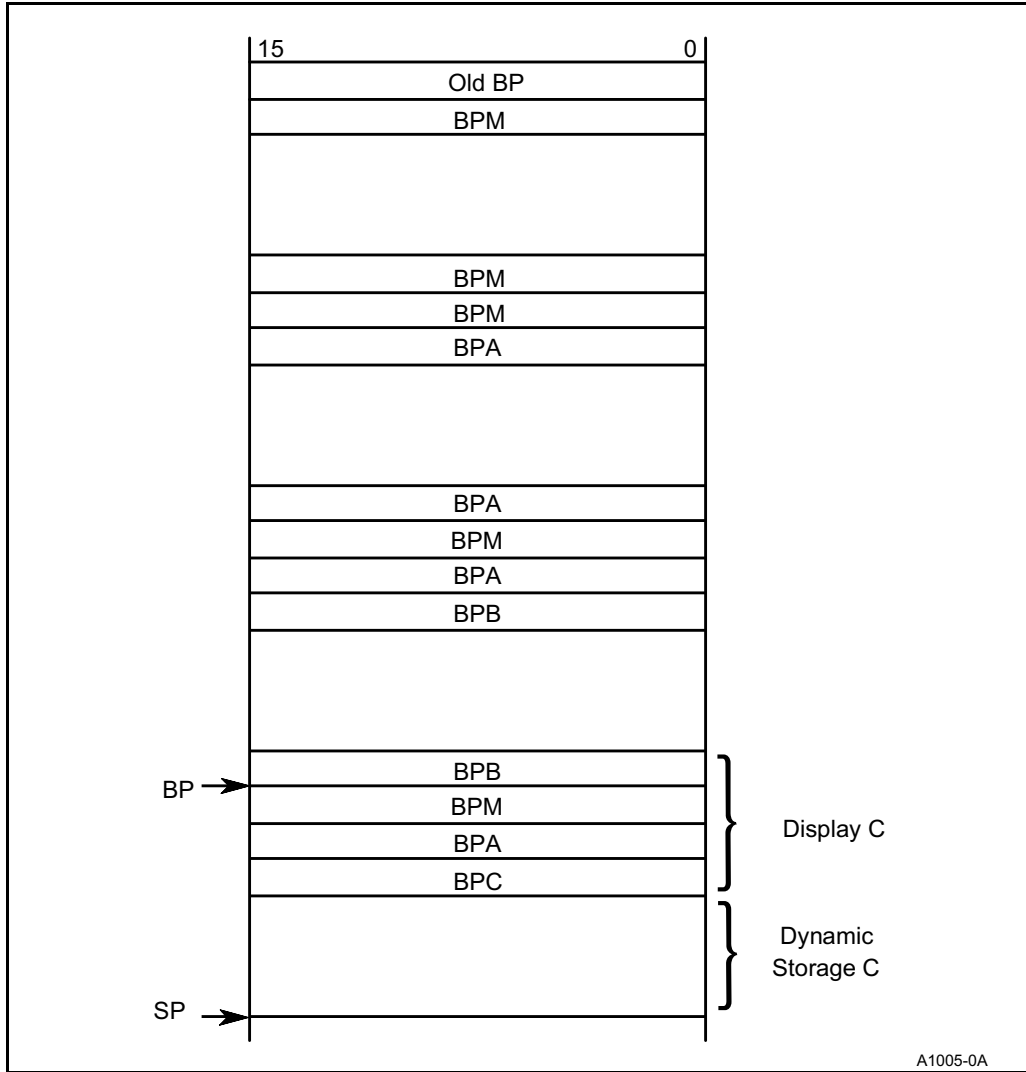


Figure A-6. Stack Frame for Procedure C at Level 3 Called from B

**LEAVE**

LEAVE reverses the action of the most recent ENTER instruction. It collapses the last stack frame created. First, LEAVE copies the current BP to the Stack Pointer, releasing the stack space allocated to the current procedure. Second, LEAVE pops the old value of BP from the stack, to return to the calling procedure's stack frame. A RET instruction will remove arguments stacked by the calling procedure for use by the called procedure.



**BOUND** *register, address*

BOUND verifies that the signed value in the specified register lies within specified limits. If the value does not lie within the bounds, an array bounds exception (type 5) occurs. BOUND is useful for checking array bounds before attempting to access an array element. This prevents the program from overwriting information outside the limits of the array.

BOUND has two operands. The first, *register*, specifies the register being tested. The second, *address*, contains the effective relative address of the two signed boundary values. The lower limit word is at this address and the upper limit word immediately follows. The limit values cannot be register operands (if they are, an invalid opcode exception occurs).

## A.2 80C186 INSTRUCTION SET ENHANCEMENTS

This section describes ten instructions that were available with the 8086/8088 but have been enhanced for the 80C186 Modular Core family.

- Data transfer instructions
  - PUSH
- Arithmetic instructions
  - IMUL
- Bit manipulation instructions (shifts and rotates)
  - SAL
  - SHL
  - SAR
  - SHR
  - ROL
  - ROR
  - RCL
  - RCR

### A.2.1 Data Transfer Instructions

**PUSH** *data*

PUSH (push immediate) allows an immediate argument, *data*, to be pushed onto the stack. The value can be either a byte or a word. Byte values are sign extended to word size before being pushed.







## A.2.2 Arithmetic Instructions

**IMUL** *destination, source, data*

IMUL (integer immediate multiply, signed) allows a value to be multiplied by an immediate operand. IMUL requires three operands. The first, *destination*, is the register where the result will be placed. The second, *source*, is the effective address of the multiplier. The source may be the same register as the destination, another register or a memory location. The third, *data*, is an immediate value used as the multiplicand. The *data* operand may be a byte or word. If *data* is a byte, it is sign extended to 16 bits. Only the lower 16 bits of the result are saved. The result must be placed in a general-purpose register.

## A.2.3 Bit Manipulation Instructions

This section describes the eight enhanced bit-manipulation instructions.

### A.2.3.1 Shift Instructions

**SAL** *destination, count*

SAL (immediate shift arithmetic left) shifts the destination operand left by an immediate value. SAL has two operands. The first, *destination*, is the effective address to be shifted. The second, *count*, is an immediate byte value representing the number of shifts to be made. The CPU will AND *count* with 1FH before shifting, to allow no more than 32 shifts. Zeros shift in on the right.

**SHL** *destination, count*

SHL (immediate shift logical left) is physically the same instruction as SAL (immediate shift arithmetic left).

**SAR** *destination, count*

SAR (immediate shift arithmetic right) shifts the destination operand right by an immediate value. SAR 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).





**B**

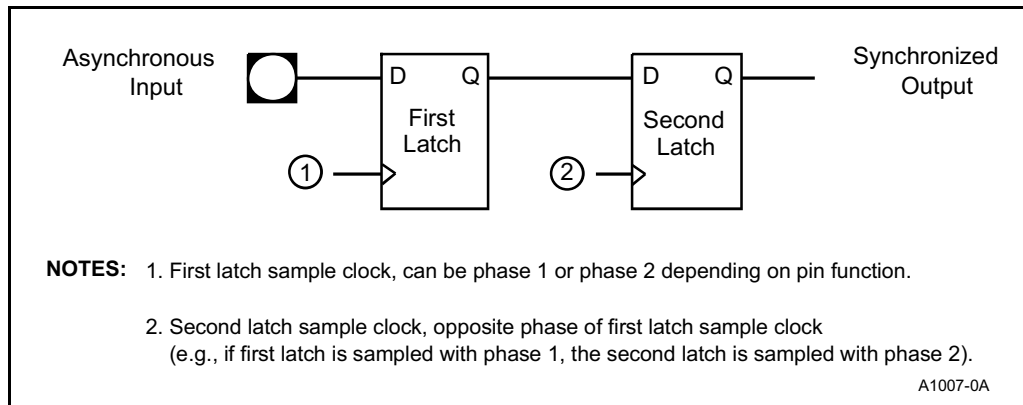
# **Input Synchronization**

**I**



## APPENDIX B INPUT SYNCHRONIZATION

Many input signals to an embedded processor are asynchronous. Asynchronous signals do not **require** a specified setup or hold time to ensure the device does not incur a failure. However, asynchronous setup and hold times are specified in the data sheet to ensure **recognition**. Associated with each of these inputs is a synchronizing circuit (see Figure B-1) that samples the asynchronous signal and synchronizes it to the internal operating clock. The output of the synchronizing circuit is then safely routed to the logic units.



**Figure B-1. Input Synchronization Circuit**

### B.1 WHY SYNCHRONIZERS ARE REQUIRED

Every data latch requires a specific setup and hold time to operate properly. The duration of the setup and hold time defines a *window* during which the device attempts to latch the data. If the input makes a transition within this window, the output may not attain a stable state. The data sheet specifies a setup and hold window larger than is actually required. However, variations in device operation (e.g., temperature, voltage) require that a larger window be specified to cover all conditions.

Should the input to the data latch make a transition during the sample and hold window, the output of the latch eventually attains a stable state. This stable state must be attained before the second stage of synchronization requires a valid input. To synchronize an asynchronous signal, the circuit in Figure B-1 samples the input into the first latch, allows the output to stabilize, then samples the stabilized value into a second latch. With the asynchronous signal resolved in this way, the input signal cannot cause an internal device failure.



## INPUT SYNCHRONIZATION

A synchronization failure can occur when the output of the first latch does not meet the setup and hold requirements of the input of the second latch. The rate of failure is determined by the actual size of the sampling window of the data latch and by the amount of time between the strobe signals of the two latches. As the sampling window gets smaller, the number of times an asynchronous transition occurs during the sampling window drops.

### B.2 ASYNCHRONOUS PINS

The 80C186EA/80C188EA inputs that use the two-stage synchronization circuit are T0IN, T1IN, NMI, TEST/BUSY, INT3:0, HOLD, DRQ0 and DRQ1.





**C**

# **Instruction Set Descriptions**

**I**







## APPENDIX C INSTRUCTION SET DESCRIPTIONS

This appendix provides reference information for the 80C186 Modular Core family instruction set. Tables C-1 through C-3 define the variables used in Table C-4, which lists the instructions with their descriptions and operations.

**Table C-1. Instruction Format Variables**

| Variable           | Description   |
|--------------------|---|
| dest               | A register or memory location that may contain data operated on by the instruction, and which receives (is replaced by) the result of the operation.  |
| src                | A register, memory location or immediate value that is used in the operation, but is not altered by the instruction   |
| target             | A label to which control is to be transferred directly, or a register or memory location whose content is the address of the location to which control is to be transferred indirectly.                   |
| disp8              | A label to which control is to be conditionally transferred; must lie within $-128$ to $+127$ bytes of the first byte of the next instruction.  |
| accum              | Register AX for word transfers, AL for bytes.   |
| port               | An I/O port number; specified as an immediate value of 0–255, or register DX (which contains port number in range 0–64K).   |
| src-string         | Name of a string in memory that is addressed by register SI; used only to identify string as byte or word and specify segment override, if any. This string is used in the operation, but is not altered. |
| dest-string        | Name of string in memory that is addressed by register DI; used only to identify string as byte or word. This string receives (is replaced by) the result of the operation.                               |
| count              | Specifies number of bits to shift or rotate; written as immediate value 1 or register CL (which contains the count in the range 0–255).   |
| interrupt-type     | Immediate value of 0–255 identifying interrupt pointer number.  |
| optional-pop-value | Number of bytes (0–64K, ordinarily an even number) to discard from the stack.   |
| external-opcode    | Immediate value (0–63) that is encoded in the instruction for use by an external processor.   |

Table C-2. Instruction Operands

| Operand     | Description  |
|-------------|--|
| reg         | An 8- or 16-bit general register.  |
| reg16       | An 16-bit general register.  |
| seg-reg     | A segment register.  |
| accum       | Register AX or AL  |
| immed       | A constant in the range 0–FFFFH.   |
| immed8      | A constant in the range 0–FFH.   |
| mem         | An 8- or 16-bit memory location.   |
| mem16       | A 16-bit memory location.  |
| mem32       | A 32-bit memory location.  |
| src-table   | Name of 256-byte translate table.  |
| src-string  | Name of string addressed by register SI.   |
| dest-string | Name of string addressed by register DI.   |
| short-label | A label within the –128 to +127 bytes of the end of the instruction.   |
| near-label  | A label in current code segment.   |
| far-label   | A label in another code segment.   |
| near-proc   | A procedure in current code segment.   |
| far-proc    | A procedure in another code segment.   |
| memptr16    | A word containing the offset of the location in the current code segment to which control is to be transferred.                                |
| memptr32    | A doubleword containing the offset and the segment base address of the location in another code segment to which control is to be transferred. |
| regptr16    | A 16-bit general register containing the offset of the location in the current code segment to which control is to be transferred.             |
| repeat      | A string instruction repeat prefix.  |



**Table C-3. Flag Bit Functions**

| Name | Function   |
|------|--|
| AF   | Auxiliary Flag:<br>Set on carry from or borrow to the low order four bits of AL; cleared otherwise.  |
| CF   | Carry Flag:<br>Set on high-order bit carry or borrow; cleared otherwise.   |
| DF   | Direction Flag:<br>Causes string instructions to auto decrement the appropriate index register when set. Clearing DF causes auto increment.    |
| IF   | Interrupt-enable Flag:<br>When set, maskable interrupts will cause the CPU to transfer control to an interrupt vector specified location.      |
| OF   | Overflow Flag:<br>Set if the signed result cannot be expressed within the number of bits in the destination operand; cleared otherwise.        |
| PF   | Parity Flag:<br>Set if low-order 8 bits of result contain an even number of 1 bits; cleared otherwise.   |
| SF   | Sign Flag:<br>Set equal to high-order bit of result (0 if positive, 1 if negative).  |
| TF   | Single Step Flag:<br>Once set, a single step interrupt occurs after the next instruction executes. TF is cleared by the single step interrupt. |
| ZF   | Zero Flag:<br>Set if result is zero; cleared otherwise.  |

Table C-4. Instruction Set

| Name | Description  | Operation  | Flags Affected  |
|------|--|--|---|
| AAA  | <p><b>ASCII Adjust for Addition:</b></p> <p>AAA</p> <p>Changes the contents of register AL to a valid unpacked decimal number; the high-order half-byte is zeroed.</p> <p><b>Instruction Operands:</b></p> <p>none</p>   | <p>if</p> <p><math>((AL) \text{ and } 0FH) &gt; 9 \text{ or } (AF) = 1</math></p> <p>then</p> <p><math>(AL) \leftarrow (AL) + 6</math></p> <p><math>(AH) \leftarrow (AH) + 1</math></p> <p><math>(AF) \leftarrow 1</math></p> <p><math>(CF) \leftarrow (AF)</math></p> <p><math>(AL) \leftarrow (AL) \text{ and } 0FH</math></p> | <p>AF ✓</p> <p>CF ✓</p> <p>DF –</p> <p>IF –</p> <p>OF ?</p> <p>PF ?</p> <p>SF ?</p> <p>TF –</p> <p>ZF ?</p> |
| AAD  | <p><b>ASCII Adjust for Division:</b></p> <p>AAD</p> <p>Modifies the numerator in AL before dividing two valid unpacked decimal operands so that the quotient produced by the division will be a valid unpacked decimal number. AH must be zero for the subsequent DIV to produce the correct result. The quotient is returned in AL, and the remainder is returned in AH; both high-order half-bytes are zeroed.</p> <p><b>Instruction Operands:</b></p> <p>none</p> | <p><math>(AL) \leftarrow (AH) \times 0AH + (AL)</math></p> <p><math>(AH) \leftarrow 0</math></p>   | <p>AF ?</p> <p>CF ?</p> <p>DF –</p> <p>IF –</p> <p>OF ?</p> <p>PF ✓</p> <p>SF ✓</p> <p>TF –</p> <p>ZF ✓</p> |
| AAM  | <p><b>ASCII Adjust for Multiply:</b></p> <p>AAM</p> <p>Corrects the result of a previous multiplication of two valid unpacked decimal operands. A valid 2-digit unpacked decimal number is derived from the content of AH and AL and is returned to AH and AL. The high-order half-bytes of the multiplied operands must have been 0H for AAM to produce a correct result.</p> <p><b>Instruction Operands:</b></p> <p>none</p>                                       | <p><math>(AH) \leftarrow (AL) / 0AH</math></p> <p><math>(AL) \leftarrow (AL) \% 0AH</math></p>   | <p>AF ?</p> <p>CF ?</p> <p>DF –</p> <p>IF –</p> <p>OF ?</p> <p>PF ✓</p> <p>SF ✓</p> <p>TF –</p> <p>ZF ✓</p> |

**NOTE:** The three symbols used in the Flags Affected column are defined as follows:

- the contents of the flag remain unchanged after the instruction is executed
- ? the contents of the flag is undefined after the instruction is executed
- ✓ the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

| Name | Description   | Operation  | Flags Affected  |
|------|---|--|---|
| AAS  | <p><b>ASCII Adjust for Subtraction:</b></p> <p>AAS</p> <p>Corrects the result of a previous subtraction of two valid unpacked decimal operands (the destination operand must have been specified as register AL). Changes the content of AL to a valid unpacked decimal number; the high-order half-byte is zeroed.</p> <p><b>Instruction Operands:</b></p> <p>none</p>   | <p>if</p> <p><math>((AL \text{ and } 0FH) &gt; 9 \text{ or } (AF) = 1)</math></p> <p>then</p> <p><math>(AL) \leftarrow (AL) - 6</math></p> <p><math>(AH) \leftarrow (AH) - 1</math></p> <p><math>(AF) \leftarrow 1</math></p> <p><math>(CF) \leftarrow (AF)</math></p> <p><math>(AL) \leftarrow (AL) \text{ and } 0FH</math></p> | <p>AF ✓</p> <p>CF ✓</p> <p>DF –</p> <p>IF –</p> <p>OF ?</p> <p>PF ?</p> <p>SF ?</p> <p>TF –</p> <p>ZF ?</p> |
| ADC  | <p><b>Add with Carry:</b></p> <p>ADC <i>dest, src</i></p> <p>Sums the operands, which may be bytes or words, adds one if CF is set and replaces the destination operand with the result. Both operands may be signed or unsigned binary numbers (see AAA and DAA). Since ADC incorporates a carry from a previous operation, it can be used to write routines to add numbers longer than 16 bits.</p> <p><b>Instruction Operands:</b></p> <p>ADC reg, reg</p> <p>ADC reg, mem</p> <p>ADC mem, reg</p> <p>ADC reg, imm</p> <p>ADC mem, imm</p> <p>ADC accum, imm</p> | <p>if</p> <p><math>(CF) = 1</math></p> <p>then</p> <p><math>(dest) \leftarrow (dest) + (src) + 1</math></p> <p>else</p> <p><math>(dest) \leftarrow (dest) + (src)</math></p>   | <p>AF ✓</p> <p>CF ✓</p> <p>DF –</p> <p>IF –</p> <p>OF ✓</p> <p>PF ✓</p> <p>SF ✓</p> <p>TF –</p> <p>ZF ✓</p> |

**NOTE:** The three symbols used in the Flags Affected column are defined as follows:

- the contents of the flag remain unchanged after the instruction is executed
- ? the contents of the flag is undefined after the instruction is executed
- ✓ the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

| Name | Description  | Operation   | Flags Affected   |
|------|--|---|--|
| ADD  | <p><b>Addition:</b><br/>           ADD <i>dest, src</i><br/>           Sums two operands, which may be bytes or words, replaces the destination operand. Both operands may be signed or unsigned binary numbers (see AAA and DAA).</p> <p><b>Instruction Operands:</b><br/>           ADD reg, reg<br/>           ADD reg, mem<br/>           ADD mem, reg<br/>           ADD reg, immed<br/>           ADD mem, immed<br/>           ADD accum, immed</p>   | $(dest) \leftarrow (dest) + (src)$  | AF ✓<br>CF ✓<br>DF –<br>IF –<br>OF ✓<br>PF ✓<br>SF ✓<br>TF –<br>ZF ✓ |
| AND  | <p><b>And Logical:</b><br/>           AND <i>dest, src</i><br/>           Performs the logical "and" of the two operands (byte or word) and returns the result to the destination operand. A bit in the result is set if both corresponding bits of the original operands are set; otherwise the bit is cleared.</p> <p><b>Instruction Operands:</b><br/>           AND reg, reg<br/>           AND reg, mem<br/>           AND mem, reg<br/>           AND reg, immed<br/>           AND mem, immed<br/>           AND accum, immed</p> | $(dest) \leftarrow (dest) \text{ and } (src)$<br>$(CF) \leftarrow 0$<br>$(OF) \leftarrow 0$ | AF ?<br>CF ✓<br>DF –<br>IF –<br>OF ✓<br>PF ✓<br>SF ✓<br>TF –<br>ZF ✓ |

**NOTE:** The three symbols used in the Flags Affected column are defined as follows:  
 – the contents of the flag remain unchanged after the instruction is executed  
 ? the contents of the flag is undefined after the instruction is executed  
 ✓ the flag is updated after the instruction is executed



Table C-4. Instruction Set (Continued)

| Name  | Description  | Operation  | Flags Affected  |
|-------|--|--|---|
| BOUND | <p><b>Detect Value Out of Range:</b><br/>           BOUND <i>dest, src</i></p> <p>Provides array bounds checking in hardware. The calculated array index is placed in one of the general purpose registers, and the upper and lower bounds of the array are placed in two consecutive memory locations. The contents of the register are compared with the memory location values, and if the register value is less than the first location or greater than the second memory location, a trap type 5 is generated.</p> <p><b>Instruction Operands:</b><br/>           BOUND reg, mem</p>                     | <p>if<br/>           ((dest) &lt; (src) or (dest) &gt; ((src) + 2)<br/>           then<br/>           (SP) ← (SP) – 2<br/>           ((SP) + 1 : (SP)) ← FLAGS<br/>           (IF) ← 0<br/>           (TF) ← 0<br/>           (SP) ← (SP) – 2<br/>           ((SP) + 1 : (SP)) ← (CS)<br/>           (CS) ← (1EH)<br/>           (SP) ← (SP) – 2<br/>           ((SP) + 1 : (SP)) ← (IP)<br/>           (IP) ← (1CH)</p> | <p>AF –<br/>           CF –<br/>           DF –<br/>           IF –<br/>           OF –<br/>           PF –<br/>           SF –<br/>           TF –<br/>           ZF –</p> |
| CALL  | <p><b>Call Procedure:</b><br/>           CALL <i>procedure-name</i></p> <p>Activates an out-of-line procedure, saving information on the stack to permit a RET (return) instruction in the procedure to transfer control back to the instruction following the CALL. The assembler generates a different type of CALL instruction depending on whether the programmer has defined the procedure name as NEAR or FAR.</p> <p><b>Instruction Operands:</b><br/>           CALL near-proc<br/>           CALL far-proc<br/>           CALL memptr16<br/>           CALL regptr16<br/>           CALL memptr32</p> | <p>if<br/>           Inter-segment<br/>           then<br/>           (SP) ← (SP) – 2<br/>           ((SP) + 1:(SP)) ← (CS)<br/>           (CS) ← SEG<br/>           (SP) ← (SP) – 2<br/>           ((SP) + 1:(SP)) ← (IP)<br/>           (IP) ← dest</p>  | <p>AF –<br/>           CF –<br/>           DF –<br/>           IF –<br/>           OF –<br/>           PF –<br/>           SF –<br/>           TF –<br/>           ZF –</p> |

**NOTE:** The three symbols used in the Flags Affected column are defined as follows:  
 – the contents of the flag remain unchanged after the instruction is executed  
 ? the contents of the flag is undefined after the instruction is executed  
 ✓ the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

| Name | Description  | Operation  | Flags Affected  |
|------|--|--|---|
| CBW  | <p><b>Convert Byte to Word:</b></p> <p>CBW</p> <p>Extends the sign of the byte in register AL throughout register AH. Use to produce a double-length (word) dividend from a byte prior to performing byte division.</p> <p><b>Instruction Operands:</b></p> <p>none</p>        | <p>if</p> <p>(AL) &lt; 80H</p> <p>then</p> <p>(AH) ← 0</p> <p>else</p> <p>(AH) ← FFH</p> | <p>AF –</p> <p>CF –</p> <p>DF –</p> <p>IF –</p> <p>OF –</p> <p>PF –</p> <p>SF –</p> <p>TF –</p> <p>ZF –</p> |
| CLC  | <p><b>Clear Carry flag:</b></p> <p>CLC</p> <p>Zeroes the carry flag (CF) and affects no other flags. Useful in conjunction with the rotate through carry left (RCL) and the rotate through carry right (RCR) instructions.</p> <p><b>Instruction Operands:</b></p> <p>none</p> | (CF) ← 0   | <p>AF –</p> <p>CF ✓</p> <p>DF –</p> <p>IF –</p> <p>OF –</p> <p>PF –</p> <p>SF –</p> <p>TF –</p> <p>ZF –</p> |
| CLD  | <p><b>Clear Direction flag:</b></p> <p>CLD</p> <p>Zeroes the direction flag (DF) causing the string instructions to auto-increment the source index (SI) and/or destination index (DI) registers.</p> <p><b>Instruction Operands:</b></p> <p>none</p>                          | (DF) ← 0   | <p>AF –</p> <p>CF –</p> <p>DF ✓</p> <p>IF –</p> <p>OF –</p> <p>PF –</p> <p>SF –</p> <p>TF –</p> <p>ZF –</p> |

**NOTE:** The three symbols used in the Flags Affected column are defined as follows:

- the contents of the flag remain unchanged after the instruction is executed
- ? the contents of the flag is undefined after the instruction is executed
- ✓ the flag is updated after the instruction is executed



**Table C-4. Instruction Set (Continued)**

| Name | Description   | Operation  | Flags Affected   |
|------|---|--|--|
| CLI  | <p><b>Clear Interrupt-enable Flag:</b><br/>CLI<br/>Zeroes the interrupt-enable flag (IF). When the interrupt-enable flag is cleared, the 8086 and 8088 do not recognize an external interrupt request that appears on the INTR line; in other words maskable interrupts are disabled. A non-maskable interrupt appearing on NMI line, however, is honored, as is a software interrupt.</p> <p><b>Instruction Operands:</b><br/>none</p> | (IF) ← 0   | AF –<br>CF –<br>DF –<br>IF ✓<br>OF –<br>PF –<br>SF –<br>TF –<br>ZF – |
| CMC  | <p><b>Complement Carry Flag:</b><br/>CMC<br/>Toggles complement carry flag (CF) to its opposite state and affects no other flags.</p> <p><b>Instruction Operands:</b><br/>none</p>  | if<br>(CF) = 0<br>then<br>(CF) ← 1<br>else<br>(CF) ← 0 | AF –<br>CF ✓<br>DF –<br>IF –<br>OF –<br>PF –<br>SF –<br>TF –<br>ZF – |

**NOTE:** The three symbols used in the Flags Affected column are defined as follows:  
 – the contents of the flag remain unchanged after the instruction is executed  
 ? the contents of the flag is undefined after the instruction is executed  
 ✓ the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

| Name | Description  | Operation  | Flags Affected   |
|------|--|--|--|
| CMP  | <p><b>Compare:</b><br/>           CMP <i>dest, src</i></p> <p>Subtracts the source from the destination, which may be bytes or words, but does not return the result. The operands are unchanged, but the flags are updated and can be tested by a subsequent conditional jump instruction. The comparison reflected in the flags is that of the destination to the source. If a CMP instruction is followed by a JG (jump if greater) instruction, for example, the jump is taken if the destination operand is greater than the source operand.</p> <p><b>Instruction Operands:</b></p> CMP reg, reg<br>CMP reg, mem<br>CMP mem, reg<br>CMP reg, immed<br>CMP mem, immed<br>CMP accum, immed | (dest) – (src)   | AF ✓<br>CF ✓<br>DF –<br>IF –<br>OF ✓<br>PF ✓<br>SF ✓<br>TF –<br>ZF ✓ |
| CMPS | <p><b>Compare String:</b><br/>           CMPS <i>dest-string, src-string</i></p> <p>Subtracts the destination byte or word from the source byte or word. The destination byte or word is addressed by the destination index (DI) register and the source byte or word is addresses by the source index (SI) register. CMPS updates the flags to reflect the relationship of the destination element to the source element but does not alter either operand and updates SI and DI to point to the next string element.</p> <p><b>Instruction Operands:</b></p> CMP dest-string, src-string<br>CMP (repeat) dest-string, src-string   | (dest-string) – (src-string)<br>if<br>(DF) = 0<br>then<br>(SI) ← (SI) + DELTA<br>(DI) ← (DI) + DELTA<br>else<br>(SI) ← (SI) – DELTA<br>(DI) ← (DI) – DELTA | AF ✓<br>CF ✓<br>DF –<br>IF –<br>OF ✓<br>PF ✓<br>SF ✓<br>TF –<br>ZF ✓ |

**NOTE:** The three symbols used in the Flags Affected column are defined as follows:  
 – the contents of the flag remain unchanged after the instruction is executed  
 ? the contents of the flag is undefined after the instruction is executed  
 ✓ the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

| Name | Description   | Operation  | Flags Affected  |
|------|---|--|---|
| CWD  | <p><b>Convert Word to Doubleword:</b><br/>CWD</p> <p>Extends the sign of the word in register AX throughout register DX. Use to produce a double-length (doubleword) dividend from a word prior to performing word division.</p> <p><b>Instruction Operands:</b><br/>none</p>   | <p>if<br/>(AX) &lt; 8000H<br/>then<br/>(DX) ← 0<br/>else<br/>(DX) ← FFFFH</p>  | <p>AF –<br/>CF –<br/>DF –<br/>IF –<br/>OF –<br/>PF –<br/>SF –<br/>TF –<br/>ZF –</p> |
| DAA  | <p><b>Decimal Adjust for Addition:</b><br/>DAA</p> <p>Corrects the result of previously adding two valid packed decimal operands (the destination operand must have been register AL). Changes the content of AL to a pair of valid packed decimal digits.</p> <p><b>Instruction Operands:</b><br/>none</p>                         | <p>if<br/>((AL) and 0FH) &gt; 9 or (AF) = 1<br/>then<br/>(AL) ← (AL) + 6<br/>(AF) ← 1<br/>if<br/>(AL) &gt; 9FH or (CF) = 1<br/>then<br/>(AL) ← (AL) + 60H<br/>(CF) ← 1</p> | <p>AF ✓<br/>CF ✓<br/>DF –<br/>IF –<br/>OF ?<br/>PF ✓<br/>SF ✓<br/>TF –<br/>ZF ✓</p> |
| DAS  | <p><b>Decimal Adjust for Subtraction:</b><br/>DAS</p> <p>Corrects the result of a previous subtraction of two valid packed decimal operands (the destination operand must have been specified as register AL). Changes the content of AL to a pair of valid packed decimal digits.</p> <p><b>Instruction Operands:</b><br/>none</p> | <p>if<br/>((AL) and 0FH) &gt; 9 or (AF) = 1<br/>then<br/>(AL) ← (AL) – 6<br/>(AF) ← 1<br/>if<br/>(AL) &gt; 9FH or (CF) = 1<br/>then<br/>(AL) ← (AL) – 60H<br/>(CF) ← 1</p> | <p>AF ✓<br/>CF ✓<br/>DF –<br/>IF –<br/>OF ?<br/>PF ✓<br/>SF ✓<br/>TF –<br/>ZF ✓</p> |

**NOTE:** The three symbols used in the Flags Affected column are defined as follows:  
 – the contents of the flag remain unchanged after the instruction is executed  
 ? the contents of the flag is undefined after the instruction is executed  
 ✓ the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

| Name | Description   | Operation                      | Flags Affected   |
|------|---|--------------------------------|--|
| DEC  | <p><b>Decrement:</b><br/>           DEC <i>dest</i></p> <p>Subtracts one from the destination operand. The operand may be a byte or a word and is treated as an unsigned binary number (see AAA and DAA).</p> <p><b>Instruction Operands:</b><br/>           DEC reg<br/>           DEC mem</p> | $(dest) \leftarrow (dest) - 1$ | AF ✓<br>CF –<br>DF –<br>IF –<br>OF ✓<br>PF ✓<br>SF ✓<br>TF –<br>ZF ✓ |

**NOTE:** The three symbols used in the Flags Affected column are defined as follows:  
 – the contents of the flag remain unchanged after the instruction is executed  
 ? the contents of the flag is undefined after the instruction is executed  
 ✓ the flag is updated after the instruction is executed



Table C-4. Instruction Set (Continued)

| Name | Description  | Operation  | Flags Affected  |
|------|--|--|---|
| DIV  | <p><b>Divide:</b></p> <p>DIV <i>src</i></p> <p>Performs an unsigned division of the accumulator (and its extension) by the source operand.</p> <p>If the source operand is a byte, it is divided into the two-byte dividend assumed to be in registers AL and AH. The byte quotient is returned in AL, and the byte remainder is returned in AH.</p> <p>If the source operand is a word, it is divided into the two-word dividend in registers AX and DX. The word quotient is returned in AX, and the word remainder is returned in DX.</p> <p>If the quotient exceeds the capacity of its destination register (FFH for byte source, FFFFH for word source), as when division by zero is attempted, a type 0 interrupt is generated, and the quotient and remainder are undefined. Nonintegral quotients are truncated to integers.</p> <p><b>Instruction Operands:</b></p> <p>DIV reg<br/>DIV mem</p> | <p><b>When Source Operand is a Byte:</b></p> <p>(temp) ← (byte-src)<br/>if<br/>    (temp) / (AX) &gt; FFH<br/>then (type 0 interrupt is generated)<br/>    (SP) ← (SP) – 2<br/>    ((SP) + 1:(SP)) ← FLAGS<br/>    (IF) ← 0<br/>    (TF) ← 0<br/>    (SP) ← (SP) – 2<br/>    ((SP) + 1:(SP)) ← (CS)<br/>    (CS) ← (2)<br/>    (SP) ← (SP) – 2<br/>    ((SP) + 1:(SP)) ← (IP)<br/>    (IP) ← (0)<br/>else<br/>    (AL) ← (temp) / (AX)<br/>    (AH) ← (temp) % (AX)</p> <p><b>When Source Operand is a Word:</b></p> <p>(temp) ← (word-src)<br/>if<br/>    (temp) / (DX:AX) &gt; FFFFH<br/>then (type 0 interrupt is generated)<br/>    (SP) ← (SP) – 2<br/>    ((SP) + 1:(SP)) ← FLAGS<br/>    (IF) ← 0<br/>    (TF) ← 0<br/>    (SP) ← (SP) – 2<br/>    ((SP) + 1:(SP)) ← (CS)<br/>    (CS) ← (2)<br/>    (SP) ← (SP) – 2<br/>    ((SP) + 1:(SP)) ← (IP)<br/>    (IP) ← (0)<br/>else<br/>    (AX) ← (temp) / (DX:AX)<br/>    (DX) ← (temp) % (DX:AX)</p> | <p>AF ?<br/>CF ?<br/>DF –<br/>IF –<br/>OF ?<br/>PF ?<br/>SF ?<br/>TF –<br/>ZF ?</p> |

**NOTE:** The three symbols used in the Flags Affected column are defined as follows:  
 – the contents of the flag remain unchanged after the instruction is executed  
 ? the contents of the flag is undefined after the instruction is executed  
 ✓ the flag is updated after the instruction is executed





Table C-4. Instruction Set (Continued)

| Name | Description   | Operation | Flags Affected   |
|------|---|-----------|--|
| HLT  | <b>Halt:</b><br>HLT<br>Causes the CPU to enter the halt state. The processor leaves the halt state upon activation of the RESET line, upon receipt of a non-maskable interrupt request on NMI, or upon receipt of a maskable interrupt request on INTR (if interrupts are enabled).<br><b>Instruction Operands:</b><br>none | None      | AF –<br>CF –<br>DF –<br>IF –<br>OF –<br>PF –<br>SF –<br>TF –<br>ZF – |

**NOTE:** The three symbols used in the Flags Affected column are defined as follows:  
– the contents of the flag remain unchanged after the instruction is executed  
? the contents of the flag is undefined after the instruction is executed  
✓ the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

| Name | Description   | Operation   | Flags Affected  |
|------|---|---|---|
| IDIV | <p><b>Integer Divide:</b></p> <p>IDIV <i>src</i></p> <p>Performs a signed division of the accumulator (and its extension) by the source operand. If the source operand is a byte, it is divided into the double-length dividend assumed to be in registers AL and AH; the single-length quotient is returned in AL, and the single-length remainder is returned in AH. For byte integer division, the maximum positive quotient is +127 (7FH) and the minimum negative quotient is -127 (81H).</p> <p>If the source operand is a word, it is divided into the double-length dividend in registers AX and DX; the single-length quotient is returned in AX, and the single-length remainder is returned in DX. For word integer division, the maximum positive quotient is +32,767 (7FFFH) and the minimum negative quotient is -32,767 (8001H).</p> <p>If the quotient is positive and exceeds the maximum, or is negative and is less than the minimum, the quotient and remainder are undefined, and a type 0 interrupt is generated. In particular, this occurs if division by 0 is attempted. Nonintegral quotients are truncated (toward 0) to integers, and the remainder has the same sign as the dividend.</p> <p><b>Instruction Operands:</b></p> <p>IDIV reg<br/>IDIV mem</p> | <p><b>When Source Operand is a Byte:</b></p> <p>(temp) ← (byte-src)</p> <p>if</p> <p>(temp) / (AX) &gt; 0 and<br/>(temp) / (AX) &gt; 7FH or<br/>(temp) / (AX) &lt; 0 and<br/>(temp) / (AX) &lt; 0 - 7FH - 1</p> <p>then (type 0 interrupt is generated)</p> <p>(SP) ← (SP) - 2<br/>((SP) + 1:(SP)) ← FLAGS<br/>(IF) ← 0<br/>(TF) ← 0<br/>(SP) ← (SP) - 2<br/>((SP) + 1:(SP)) ← (CS)<br/>(CS) ← (2)<br/>(SP) ← (SP) - 2<br/>((SP) + 1:(SP)) ← (IP)<br/>(IP) ← (0)</p> <p>else</p> <p>(AL) ← (temp) / (AX)<br/>(AH) ← (temp) % (AX)</p> <p><b>When Source Operand is a Word:</b></p> <p>(temp) ← (word-src)</p> <p>if</p> <p>(temp) / (DX:AX) &gt; 0 and<br/>(temp) / (DX:AX) &gt; 7FFFH or<br/>(temp) / (DX:AX) &lt; 0 and<br/>(temp) / (DX:AX) &lt; 0 - 7FFFH - 1</p> <p>then (type 0 interrupt is generated)</p> <p>(SP) ← (SP) - 2<br/>((SP) + 1:(SP)) ← FLAGS<br/>(IF) ← 0<br/>(TF) ← 0<br/>(SP) ← (SP) - 2<br/>((SP) + 1:(SP)) ← (CS)<br/>(CS) ← (2)<br/>(SP) ← (SP) - 2<br/>((SP) + 1:(SP)) ← (IP)<br/>(IP) ← (0)</p> <p>else</p> <p>(AX) ← (temp) / (DX:AX)<br/>(DX) ← (temp) % (DX:AX)</p> | <p>AF ?<br/>CF ?<br/>DF -<br/>IF -<br/>OF ?<br/>PF ?<br/>SF ?<br/>TF -<br/>ZF ?</p> |

**NOTE:** The three symbols used in the Flags Affected column are defined as follows:  
 - the contents of the flag remain unchanged after the instruction is executed  
 ? the contents of the flag is undefined after the instruction is executed  
 ✓ the flag is updated after the instruction is executed



Table C-4. Instruction Set (Continued)

| Name | Description   | Operation  | Flags Affected  |
|------|---|--|---|
| IMUL | <p><b>Integer Multiply:</b></p> <p>IMUL <i>src</i></p> <p>Performs a signed multiplication of the source operand and the accumulator. If the source is a byte, then it is multiplied by register AL, and the double-length result is returned in AH and AL. If the source is a word, then it is multiplied by register AX, and the double-length result is returned in registers DX and AX. If the upper half of the result (AH for byte source, DX for word source) is not the sign extension of the lower half of the result, CF and OF are set; otherwise they are cleared. When CF and OF are set, they indicate that AH or DX contains significant digits of the result.</p> <p><b>Instruction Operands:</b></p> <p>IMUL reg<br/>IMUL mem<br/>IMUL immed</p> | <p><b>When Source Operand is a Byte:</b></p> <p>(AX) ← (byte-src) × (AL)</p> <p>if<br/>(AH) = sign-extension of (AL)<br/>then<br/>(CF) ← 0<br/>else<br/>(CF) ← 1<br/>(OF) ← (CF)</p> <p><b>When Source Operand is a Word:</b></p> <p>(DX:AX) ← (word-src) × (AX)</p> <p>if<br/>(DX) = sign-extension of (AX)<br/>then<br/>(CF) ← 0<br/>else<br/>(CF) ← 1<br/>(OF) ← (CF)</p> | <p>AF ?<br/>CF ✓<br/>DF –<br/>IF –<br/>OF ✓<br/>PF ?<br/>SF ?<br/>TF –<br/>ZF ?</p> |
| IN   | <p><b>Input Byte or Word:</b></p> <p>IN <i>accum, port</i></p> <p>Transfers a byte or a word from an input port to the AL register or the AX register, respectively. The port number may be specified either with an immediate byte constant, allowing access to ports numbered 0 through 255, or with a number previously placed in the DX register, allowing variable access (by changing the value in DX) to ports numbered from 0 through 65,535.</p> <p><b>Instruction Operands:</b></p> <p>IN AL, immed8<br/>IN AX, DX</p>  | <p><b>When Source Operand is a Byte:</b></p> <p>(AL) ← (port)</p> <p><b>When Source Operand is a Word:</b></p> <p>(AX) ← (port)</p>  | <p>AF –<br/>CF –<br/>DF –<br/>IF –<br/>OF –<br/>PF –<br/>SF –<br/>TF –<br/>ZF –</p> |

**NOTE:** The three symbols used in the Flags Affected column are defined as follows:  
 – the contents of the flag remain unchanged after the instruction is executed  
 ? the contents of the flag is undefined after the instruction is executed  
 ✓ the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

| Name | Description  | Operation                      | Flags Affected   |
|------|--|--------------------------------|--|
| INC  | <p><b>Increment:</b></p> <p>INC <i>dest</i></p> <p>Adds one to the destination operand. The operand may be byte or a word and is treated as an unsigned binary number (see AAA and DAA).</p> <p><b>Instruction Operands:</b></p> <p>INC reg<br/>INC mem</p>  | $(dest) \leftarrow (dest) + 1$ | AF ✓<br>CF –<br>DF –<br>IF –<br>OF ✓<br>PF ✓<br>SF ✓<br>TF –<br>ZF ✓ |
| INS  | <p><b>In String:</b></p> <p>INS <i>dest-string, port</i></p> <p>Performs block input from an I/O port to memory. The port address is placed in the DX register. The memory address is placed in the DI register. This instruction uses the ES register (which cannot be overridden). After the data transfer takes place, the DI register increments or decrements, depending on the value of the direction flag (DF). The DI register changes by 1 for byte transfers or 2 for word transfers.</p> <p><b>Instruction Operands:</b></p> <p>INS <i>dest-string, port</i><br/>INS (<i>repeat</i>) <i>dest-string, port</i></p> | $(dest) \leftarrow (src)$      | AF –<br>CF –<br>DF –<br>IF –<br>OF –<br>PF –<br>SF –<br>TF –<br>ZF – |

**NOTE:** The three symbols used in the Flags Affected column are defined as follows:  
 – the contents of the flag remain unchanged after the instruction is executed  
 ? the contents of the flag is undefined after the instruction is executed  
 ✓ the flag is updated after the instruction is executed



**Table C-4. Instruction Set (Continued)**

| Name | Description   | Operation   | Flags Affected  |
|------|---|---|---|
| INT  | <p><b>Interrupt:</b></p> <p>INT <i>interrupt-type</i></p> <p>Activates the interrupt procedure specified by the interrupt-type operand. Decrements the stack pointer by two, pushes the flags onto the stack, and clears the trap (TF) and interrupt-enable (IF) flags to disable single-step and maskable interrupts. The flags are stored in the format used by the PUSHF instruction. SP is decremented again by two, and the CS register is pushed onto the stack.</p> <p>The address of the interrupt pointer is calculated by multiplying interrupt-type by four; the second word of the interrupt pointer replaces CS. SP again is decremented by two, and IP is pushed onto the stack and is replaced by the first word of the interrupt pointer. If interrupt-type = 3, the assembler generates a short (1 byte) form of the instruction, known as the breakpoint interrupt.</p> <p><b>Instruction Operands:</b></p> <p>INT immed8</p> | $(SP) \leftarrow (SP) - 2$<br>$((SP) + 1:(SP)) \leftarrow \text{FLAGS}$<br>$(IF) \leftarrow 0$<br>$(TF) \leftarrow 0$<br>$(SP) \leftarrow (SP) - 2$<br>$((SP) + 1:(SP)) \leftarrow (CS)$<br>$(CS) \leftarrow (\text{interrupt-type} \times 4 + 2)$<br>$(SP) \leftarrow (SP) - 2$<br>$((SP) + 1:(SP)) \leftarrow (IP)$<br>$(IP) \leftarrow (\text{interrupt-type} \times 4)$ | <p>AF –</p> <p>CF –</p> <p>DF –</p> <p>IF ✓</p> <p>OF –</p> <p>PF –</p> <p>SF –</p> <p>TF ✓</p> <p>ZF –</p> |

**NOTE:** The three symbols used in the Flags Affected column are defined as follows:  
 – the contents of the flag remain unchanged after the instruction is executed  
 ? the contents of the flag is undefined after the instruction is executed  
 ✓ the flag is updated after the instruction is executed





Table C-4. Instruction Set (Continued)

| Name       | Description   | Operation   | Flags Affected  |
|------------|---|---|---|
| JAE<br>JNB | <p><b>Jump on Above or Equal:</b><br/> <b>Jump on Not Below:</b><br/>           JAE <i>disp8</i><br/>           JNB <i>disp8</i></p> <p>Transfers control to the target location if the tested condition (CF = 0) is true.</p> <p><b>Instruction Operands:</b><br/>           JAE short-label<br/>           JNB short-label</p>            | <p>if<br/>           (CF) = 0<br/>           then<br/>           (IP) ← (IP) + <i>disp8</i> (sign-ext to 16 bits)</p>                 | <p>AF –<br/>           CF –<br/>           DF –<br/>           IF –<br/>           OF –<br/>           PF –<br/>           SF –<br/>           TF –<br/>           ZF –</p> |
| JB<br>JNAE | <p><b>Jump on Below:</b><br/> <b>Jump on Not Above or Equal:</b><br/>           JB <i>disp8</i><br/>           JNAE <i>disp8</i></p> <p>Transfers control to the target location if the tested condition (CF = 1) is true.</p> <p><b>Instruction Operands:</b><br/>           JB short-label<br/>           JNAE short-label</p>            | <p>if<br/>           (CF) = 1<br/>           then<br/>           (IP) ← (IP) + <i>disp8</i> (sign-ext to 16 bits)</p>                 | <p>AF –<br/>           CF –<br/>           DF –<br/>           IF –<br/>           OF –<br/>           PF –<br/>           SF –<br/>           TF –<br/>           ZF –</p> |
| JBE<br>JNA | <p><b>Jump on Below or Equal:</b><br/> <b>Jump on Not Above:</b><br/>           JBE <i>disp8</i><br/>           JNA <i>disp8</i></p> <p>Transfers control to the target location if the tested condition ((C = 1) or (ZF=1)) is true.</p> <p><b>Instruction Operands:</b><br/>           JBE short-label<br/>           JNA short-label</p> | <p>if<br/>           ((CF) = 1) or ((ZF) = 1)<br/>           then<br/>           (IP) ← (IP) + <i>disp8</i> (sign-ext to 16 bits)</p> | <p>AF –<br/>           CF –<br/>           DF –<br/>           IF –<br/>           OF –<br/>           PF –<br/>           SF –<br/>           TF –<br/>           ZF –</p> |
| JC         | <p><b>Jump on Carry:</b><br/>           JC <i>disp8</i></p> <p>Transfers control to the target location if the tested condition (CF=1) is true.</p> <p><b>Instruction Operands:</b><br/>           JC short-label</p>   | <p>if<br/>           (CF) = 1<br/>           then<br/>           (IP) ← (IP) + <i>disp8</i> (sign-ext to 16 bits)</p>                 | <p>AF –<br/>           CF –<br/>           DF –<br/>           IF –<br/>           OF –<br/>           PF –<br/>           SF –<br/>           TF –<br/>           ZF –</p> |

**NOTE:** The three symbols used in the Flags Affected column are defined as follows:  
 – the contents of the flag remain unchanged after the instruction is executed  
 ? the contents of the flag is undefined after the instruction is executed  
 ✓ the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

| Name       | Description   | Operation   | Flags Affected   |
|------------|---|---|--|
| JCXZ       | <p><b>Jump if CX Zero:</b><br/> JCXZ <i>disp8</i></p> <p>Transfers control to the target location if CX is 0. Useful at the beginning of a loop to bypass the loop if CX has a zero value, i.e., to execute the loop zero times.</p> <p><b>Instruction Operands:</b><br/> JCXZ short-label</p>                  | <p>if<br/> (CX) = 0<br/> then<br/> (IP) ← (IP) + <i>disp8</i> (sign-ext to 16 bits)</p>                     | AF –<br>CF –<br>DF –<br>IF –<br>OF –<br>PF –<br>SF –<br>TF –<br>ZF – |
| JE<br>JZ   | <p><b>Jump on Equal:</b><br/> <b>Jump on Zero:</b><br/> JE <i>disp8</i><br/> JZ <i>disp8</i></p> <p>Transfers control to the target location if the condition tested (ZF = 1) is true.</p> <p><b>Instruction Operands:</b><br/> JE short-label<br/> JZ short-label</p>  | <p>if<br/> (ZF) = 1<br/> then<br/> (IP) ← (IP) + <i>disp8</i> (sign-ext to 16 bits)</p>                     | AF –<br>CF –<br>DF –<br>IF –<br>OF –<br>PF –<br>SF –<br>TF –<br>ZF – |
| JG<br>JNLE | <p><b>Jump on Greater Than:</b><br/> <b>Jump on Not Less Than or Equal:</b><br/> JG <i>disp8</i><br/> JNLE <i>disp8</i></p> <p>Transfers control to the target location if the condition tested (SF = OF) and (ZF=0) is true.</p> <p><b>Instruction Operands:</b><br/> JG short-label<br/> JNLE short-label</p> | <p>if<br/> ((SF) = (OF)) and ((ZF) = 0)<br/> then<br/> (IP) ← (IP) + <i>disp8</i> (sign-ext to 16 bits)</p> | AF –<br>CF –<br>DF –<br>IF –<br>OF –<br>PF –<br>SF –<br>TF –<br>ZF – |
| JGE<br>JNL | <p><b>Jump on Greater Than or Equal:</b><br/> <b>Jump on Not Less Than:</b><br/> JGE <i>disp8</i><br/> JNL <i>disp8</i></p> <p>Transfers control to the target location if the condition tested (SF=OF) is true.</p> <p><b>Instruction Operands:</b><br/> JGE short-label<br/> JNL short-label</p>              | <p>if<br/> (SF) = (OF)<br/> then<br/> (IP) ← (IP) + <i>disp8</i> (sign-ext to 16 bits)</p>                  | AF –<br>CF –<br>DF –<br>IF –<br>OF –<br>PF –<br>SF –<br>TF –<br>ZF – |

**NOTE:** The three symbols used in the Flags Affected column are defined as follows:  
– the contents of the flag remain unchanged after the instruction is executed  
? the contents of the flag is undefined after the instruction is executed  
✓ the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

| Name       | Description  | Operation  | Flags Affected  |
|------------|--|--|---|
| JL<br>JNGE | <p><b>Jump on Less Than:</b><br/> <b>Jump on Not Greater Than or Equal:</b><br/>           JL <i>disp8</i><br/>           JNGE <i>disp8</i></p> <p>Transfers control to the target location if the condition tested (SF≠OF) is true.</p> <p><b>Instruction Operands:</b><br/>           JL short-label<br/>           JNGE short-label</p>             | <p>if<br/>           (SF) ≠ (OF)<br/>           then<br/>           (IP) ← (IP) + <i>disp8</i> (sign-ext to 16 bits)</p>                 | <p>AF –<br/>           CF –<br/>           DF –<br/>           IF –<br/>           OF –<br/>           PF –<br/>           SF –<br/>           TF –<br/>           ZF –</p> |
| JLE<br>JNG | <p><b>Jump on Less Than or Equal:</b><br/> <b>Jump on Not Greater Than:</b><br/>           JGE <i>disp8</i><br/>           JNL <i>disp8</i></p> <p>Transfers control to the target location if the condition tested ((SF≠OF) or (ZF=0)) is true.</p> <p><b>Instruction Operands:</b><br/>           JGE short-label<br/>           JNL short-label</p> | <p>if<br/>           ((SF) ≠ (OF)) or ((ZF) = 1)<br/>           then<br/>           (IP) ← (IP) + <i>disp8</i> (sign-ext to 16 bits)</p> | <p>AF –<br/>           CF –<br/>           DF –<br/>           IF –<br/>           OF –<br/>           PF –<br/>           SF –<br/>           TF –<br/>           ZF –</p> |
| JMP        | <p><b>Jump Unconditionally:</b><br/>           JMP <i>target</i></p> <p>Transfers control to the target location.</p> <p><b>Instruction Operands:</b><br/>           JMP short-label<br/>           JMP near-label<br/>           JMP far-label<br/>           JMP memptr<br/>           JMP regptr</p>  | <p>if<br/>           Inter-segment<br/>           then<br/>           (CS) ← SEG<br/>           (IP) ← dest</p>                          | <p>AF –<br/>           CF –<br/>           DF –<br/>           IF –<br/>           OF –<br/>           PF –<br/>           SF –<br/>           TF –<br/>           ZF –</p> |
| JNC        | <p><b>Jump on Not Carry:</b><br/>           JNC <i>disp8</i></p> <p>Transfers control to the target location if the tested condition (CF=0) is true.</p> <p><b>Instruction Operands:</b><br/>           JNC short-label</p>  | <p>if<br/>           (CF) = 0<br/>           then<br/>           (IP) ← (IP) + <i>disp8</i> (sign-ext to 16 bits)</p>                    | <p>AF –<br/>           CF –<br/>           DF –<br/>           IF –<br/>           OF –<br/>           PF –<br/>           SF –<br/>           TF –<br/>           ZF –</p> |

**NOTE:** The three symbols used in the Flags Affected column are defined as follows:  
 – the contents of the flag remain unchanged after the instruction is executed  
 ? the contents of the flag is undefined after the instruction is executed  
 ✓ the flag is updated after the instruction is executed







## INSTRUCTION SET DESCRIPTIONS

Table C-4. Instruction Set (Continued)

| Name  | Description  | Operation   | Flags Affected   |
|-------|--|---|--|
| LDS   | <p><b>Load Pointer Using DS:</b></p> <p>LDS <i>dest, src</i></p> <p>Transfers a 32-bit pointer variable from the source operand, which must be a memory operand, to the destination operand and register DS. The offset word of the pointer is transferred to the destination operand, which may be any 16-bit general register. The segment word of the pointer is transferred to register DS.</p> <p><b>Instruction Operands:</b></p> <p>LDS reg16, mem32</p>  | $(dest) \leftarrow (EA)$<br>$(DS) \leftarrow (EA + 2)$                                    | AF –<br>CF –<br>DF –<br>IF –<br>OF –<br>PF –<br>SF –<br>TF –<br>ZF – |
| LEA   | <p><b>Load Effective Address:</b></p> <p>LEA <i>dest, src</i></p> <p>Transfers the offset of the source operand (rather than its value) to the destination operand.</p> <p><b>Instruction Operands:</b></p> <p>LEA reg16, mem16</p>  | $(dest) \leftarrow EA$  | AF –<br>CF –<br>DF –<br>IF –<br>OF –<br>PF –<br>SF –<br>TF –<br>ZF – |
| LEAVE | <p><b>Leave:</b></p> <p>LEAVE</p> <p>Reverses the action of the most recent ENTER instruction. Collapses the last stack frame created. First, LEAVE copies the current BP to the stack pointer releasing the stack space allocated to the current procedure. Second, LEAVE pops the old value of BP from the stack, to return to the calling procedure's stack frame. A return (RET) instruction will remove arguments stacked by the calling procedure for use by the called procedure.</p> <p><b>Instruction Operands:</b></p> <p>none</p> | $(SP) \leftarrow (BP)$<br>$(BP) \leftarrow ((SP) + 1:(SP))$<br>$(SP) \leftarrow (SP) + 2$ | AF –<br>CF –<br>DF –<br>IF –<br>OF –<br>PF –<br>SF –<br>TF –<br>ZF – |

**NOTE:** The three symbols used in the Flags Affected column are defined as follows:

- the contents of the flag remain unchanged after the instruction is executed
- ? the contents of the flag is undefined after the instruction is executed
- ✓ the flag is updated after the instruction is executed



## INSTRUCTION SET DESCRIPTIONS



**Table C-4. Instruction Set (Continued)**

| Name             | Description  | Operation  | Flags Affected   |
|------------------|--|--|--|
| LOOPNE<br>LOOPNZ | <p><b>Loop While Not Equal:</b><br/> <b>Loop While Not Zero:</b><br/>           LOOPNE <i>disp8</i><br/>           LOOPNZ <i>disp8</i></p> <p>Decrements CX by 1 and transfers control to the target location if CX is not 0 and if ZF is clear; otherwise the next sequential instruction is executed.</p> <p><b>Instruction Operands:</b><br/>           LOOPNE short-label<br/>           LOOPNZ short-label</p>  | $(CX) \leftarrow (CX) - 1$<br>if<br>$(ZF) = 0$ and $(CX) \neq 0$<br>then<br>$(IP) \leftarrow (IP) + disp8$ (sign-ext to 16 bits) | AF –<br>CF –<br>DF –<br>IF –<br>OF –<br>PF –<br>SF –<br>TF –<br>ZF – |
| MOV              | <p><b>Move (Byte or Word):</b><br/>           MOV <i>dest, src</i></p> <p>Transfers a byte or a word from the source operand to the destination operand.</p> <p><b>Instruction Operands:</b><br/>           MOV mem, accum<br/>           MOV accum, mem<br/>           MOV reg, reg<br/>           MOV reg, mem<br/>           MOV mem, reg<br/>           MOV reg, immed<br/>           MOV mem, immed<br/>           MOV seg-reg, reg16<br/>           MOV seg-reg, mem16<br/>           MOV reg16, seg-reg<br/>           MOV mem16, seg-reg</p> | $(dest) \leftarrow (src)$  | AF –<br>CF –<br>DF –<br>IF –<br>OF –<br>PF –<br>SF –<br>TF –<br>ZF – |

**NOTE:** The three symbols used in the Flags Affected column are defined as follows:  
 – the contents of the flag remain unchanged after the instruction is executed  
 ? the contents of the flag is undefined after the instruction is executed  
 ✓ the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

| Name | Description   | Operation  | Flags Affected   |
|------|---|--|--|
| MOVS | <p><b>Move String:</b></p> <p>MOVS <i>dest-string, src-string</i></p> <p>Transfers a byte or a word from the source string (addressed by SI) to the destination string (addressed by DI) and updates SI and DI to point to the next string element. When used in conjunction with REP, MOVS performs a memory-to-memory block transfer.</p> <p><b>Instruction Operands:</b></p> <p>MOVS <i>dest-string, src-string</i><br/> MOVS (repeat) <i>dest-string, src-string</i></p>  | ( <i>dest-string</i> ) ← ( <i>src-string</i> )   | AF –<br>CF –<br>DF –<br>IF –<br>OF –<br>PF –<br>SF –<br>TF –<br>ZF – |
| MUL  | <p><b>Multiply:</b></p> <p>MUL <i>src</i></p> <p>Performs an unsigned multiplication of the source operand and the accumulator. If the source is a byte, then it is multiplied by register AL, and the double-length result is returned in AH and AL. If the source operand is a word, then it is multiplied by register AX, and the double-length result is returned in registers DX and AX. The operands are treated as unsigned binary numbers (see AAM). If the upper half of the result (AH for byte source, DX for word source) is non-zero, CF and OF are set; otherwise they are cleared.</p> <p><b>Instruction Operands:</b></p> <p>MUL <i>reg</i><br/> MUL <i>mem</i></p> | <p><b>When Source Operand is a Byte:</b></p> <p>(AX) ← (AL) × (<i>src</i>)</p> <p>if<br/> (AH) = 0<br/> then<br/> (CF) ← 0<br/> else<br/> (CF) ← 1<br/> (OF) ← (CF)</p> <p><b>When Source Operand is a Word:</b></p> <p>(DX:AX) ← (AX) × (<i>src</i>)</p> <p>if<br/> (DX) = 0<br/> then<br/> (CF) ← 0<br/> else<br/> (CF) ← 1<br/> (OF) ← (CF)</p> | AF ?<br>CF ✓<br>DF –<br>IF –<br>OF ✓<br>PF ?<br>SF ?<br>TF –<br>ZF ? |

**NOTE:** The three symbols used in the Flags Affected column are defined as follows:

- the contents of the flag remain unchanged after the instruction is executed
- ? the contents of the flag is undefined after the instruction is executed
- ✓ the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

| Name | Description   | Operation   | Flags Affected   |
|------|---|---|--|
| NEG  | <p><b>Negate:</b><br/>           NEG <i>dest</i></p> <p>Subtracts the destination operand, which may be a byte or a word, from 0 and returns the result to the destination. This forms the two's complement of the number, effectively reversing the sign of an integer. If the operand is zero, its sign is not changed. Attempting to negate a byte containing -128 or a word containing -32,768 causes no change to the operand and sets OF.</p> <p><b>Instruction Operands:</b><br/>           NEG reg<br/>           NEG mem</p> | <p><b>When Source Operand is a Byte:</b><br/> <math>(dest) \leftarrow FFH - (dest)</math><br/> <math>(dest) \leftarrow (dest) + 1</math> (affecting flags)</p> <p><b>When Source Operand is a Word:</b><br/> <math>(dest) \leftarrow FFFFH - (dest)</math><br/> <math>(dest) \leftarrow (dest) + 1</math> (affecting flags)</p> | AF ✓<br>CF ✓<br>DF -<br>IF -<br>OF ✓<br>PF ✓<br>SF ✓<br>TF -<br>ZF ✓ |
| NOP  | <p><b>No Operation:</b><br/>           NOP</p> <p>Causes the CPU to do nothing.</p> <p><b>Instruction Operands:</b><br/>           none</p>   | None  | AF -<br>CF -<br>DF -<br>IF -<br>OF -<br>PF -<br>SF -<br>TF -<br>ZF - |
| NOT  | <p><b>Logical Not:</b><br/>           NOT <i>dest</i></p> <p>Inverts the bits (forms the one's complement) of the byte or word operand.</p> <p><b>Instruction Operands:</b><br/>           NOT reg<br/>           NOT mem</p>   | <p><b>When Source Operand is a Byte:</b><br/> <math>(dest) \leftarrow FFH - (dest)</math></p> <p><b>When Source Operand is a Word:</b><br/> <math>(dest) \leftarrow FFFFH - (dest)</math></p>   | AF -<br>CF -<br>DF -<br>IF -<br>OF -<br>PF -<br>SF -<br>TF -<br>ZF - |

**NOTE:** The three symbols used in the Flags Affected column are defined as follows:  
 - the contents of the flag remain unchanged after the instruction is executed  
 ? the contents of the flag is undefined after the instruction is executed  
 ✓ the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

| Name | Description  | Operation  | Flags Affected   |
|------|--|--|--|
| OR   | <p><b>Logical OR:</b><br/>OR <i>dest,src</i></p> <p>Performs the logical "inclusive or" of the two operands (bytes or words) and returns the result to the destination operand. A bit in the result is set if either or both corresponding bits in the original operands are set; otherwise the result bit is cleared.</p> <p><b>Instruction Operands:</b></p> <p>OR reg, reg<br/>OR reg, mem<br/>OR mem, reg<br/>OR accum, immed<br/>OR reg, immed<br/>OR mem, immed</p>  | $(dest) \leftarrow (dest) \text{ or } (src)$<br>$(CF) \leftarrow 0$<br>$(OF) \leftarrow 0$ | AF ?<br>CF ✓<br>DF –<br>IF –<br>OF ✓<br>PF ✓<br>SF ✓<br>TF –<br>ZF ✓ |
| OUT  | <p><b>Output:</b><br/>OUT <i>port, accumulator</i></p> <p>Transfers a byte or a word from the AL register or the AX register, respectively, to an output port. The port number may be specified either with an immediate byte constant, allowing access to ports numbered 0 through 255, or with a number previously placed in register DX, allowing variable access (by changing the value in DX) to ports numbered from 0 through 65,535.</p> <p><b>Instruction Operands:</b></p> <p>OUT immed8, AL<br/>OUT DX, AX</p> | $(dest) \leftarrow (src)$  | AF –<br>CF –<br>DF –<br>IF –<br>OF –<br>PF –<br>SF –<br>TF –<br>ZF – |

**NOTE:** The three symbols used in the Flags Affected column are defined as follows:  
– the contents of the flag remain unchanged after the instruction is executed  
? the contents of the flag is undefined after the instruction is executed  
✓ the flag is updated after the instruction is executed





**Table C-4. Instruction Set (Continued)**

| Name | Description   | Operation   | Flags Affected   |
|------|---|---|--|
| OUTS | <p><b>Out String:</b><br/>           OUTS <i>port, src_string</i></p> <p>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.</p> <p><b>Instruction Operands:</b><br/>           OUTS <i>port, src_string</i><br/>           OUTS (repeat) <i>port, src_string</i></p> | $(dst) \leftarrow (src)$  | AF –<br>CF –<br>DF –<br>IF –<br>OF –<br>PF –<br>SF –<br>TF –<br>ZF – |
| POP  | <p><b>Pop:</b><br/>           POP <i>dest</i></p> <p>Transfers the word at the current top of stack (pointed to by SP) to the destination operand and then increments SP by two to point to the new top of stack.</p> <p><b>Instruction Operands:</b><br/>           POP <i>reg</i><br/>           POP <i>seg-reg</i> (CS illegal)<br/>           POP <i>mem</i></p>  | $(dest) \leftarrow ((SP) + 1:(SP))$<br>$(SP) \leftarrow (SP) + 2$ | AF –<br>CF –<br>DF –<br>IF –<br>OF –<br>PF –<br>SF –<br>TF –<br>ZF – |

**NOTE:** The three symbols used in the Flags Affected column are defined as follows:  
 – the contents of the flag remain unchanged after the instruction is executed  
 ? the contents of the flag is undefined after the instruction is executed  
 ✓ the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

| Name | Description   | Operation   | Flags Affected   |
|------|---|---|--|
| POPA | <p><b>Pop All:</b><br/>POPA</p> <p>Pops all data, pointer, and index registers off of the stack. The SP value popped is discarded.</p> <p><b>Instruction Operands:</b><br/>none</p>   | $(DI) \leftarrow ((SP) + 1:(SP))$<br>$(SP) \leftarrow (SP) + 2$<br>$(SI) \leftarrow ((SP) + 1:(SP))$<br>$(SP) \leftarrow (SP) + 2$<br>$(BP) \leftarrow ((SP) + 1:(SP))$<br>$(SP) \leftarrow (SP) + 2$<br>$(BX) \leftarrow ((SP) + 1:(SP))$<br>$(SP) \leftarrow (SP) + 2$<br>$(DX) \leftarrow ((SP) + 1:(SP))$<br>$(SP) \leftarrow (SP) + 2$<br>$(CX) \leftarrow ((SP) + 1:(SP))$<br>$(SP) \leftarrow (SP) + 2$<br>$(AX) \leftarrow ((SP) + 1:(SP))$<br>$(SP) \leftarrow (SP) + 2$ | AF –<br>CF –<br>DF –<br>IF –<br>OF –<br>PF –<br>SF –<br>TF –<br>ZF – |
| POPF | <p><b>Pop Flags:</b><br/>POPF</p> <p>Transfers specific bits from the word at the current top of stack (pointed to by register SP) into the 8086/8088 flags, replacing whatever values the flags previously contained. SP is then incremented by two to point to the new top of stack.</p> <p><b>Instruction Operands:</b><br/>none</p> | $Flags \leftarrow ((SP) + 1:(SP))$<br>$(SP) \leftarrow (SP) + 2$  | AF ✓<br>CF ✓<br>DF ✓<br>IF ✓<br>OF ✓<br>PF ✓<br>SF ✓<br>TF ✓<br>ZF ✓ |
| PUSH | <p><b>Push:</b><br/>PUSH <i>src</i></p> <p>Decrements SP by two and then transfers a word from the source operand to the top of stack now pointed to by SP.</p> <p><b>Instruction Operands:</b><br/>PUSH reg<br/>PUSH seg-reg (CS legal)<br/>PUSH mem</p>   | $(SP) \leftarrow (SP) - 2$<br>$((SP) + 1:(SP)) \leftarrow (src)$  | AF –<br>CF –<br>DF –<br>IF –<br>OF –<br>PF –<br>SF –<br>TF –<br>ZF – |

**NOTE:** The three symbols used in the Flags Affected column are defined as follows:  
 – the contents of the flag remain unchanged after the instruction is executed  
 ? the contents of the flag is undefined after the instruction is executed  
 ✓ the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

| Name  | Description   | Operation  | Flags Affected  |
|-------|---|--|---|
| PUSHA | <p><b>Push All:</b><br/>PUSHA<br/>Pushes all data, pointer, and index registers onto the stack. The order in which the registers are saved is: AX, CX, DX, BX, SP, BP, SI, and DI. The SP value pushed is the SP value before the first register (AX) is pushed.</p> <p><b>Instruction Operands:</b><br/>none</p> | $temp \leftarrow (SP)$<br>$(SP) \leftarrow (SP) - 2$<br>$((SP) + 1:(SP)) \leftarrow (AX)$<br>$(SP) \leftarrow (SP) - 2$<br>$((SP) + 1:(SP)) \leftarrow (CX)$<br>$(SP) \leftarrow (SP) - 2$<br>$((SP) + 1:(SP)) \leftarrow (DX)$<br>$(SP) \leftarrow (SP) - 2$<br>$((SP) + 1:(SP)) \leftarrow (BX)$<br>$(SP) \leftarrow (SP) - 2$<br>$((SP) + 1:(SP)) \leftarrow (temp)$<br>$(SP) \leftarrow (SP) - 2$<br>$((SP) + 1:(SP)) \leftarrow (BP)$<br>$(SP) \leftarrow (SP) - 2$<br>$((SP) + 1:(SP)) \leftarrow (SI)$<br>$(SP) \leftarrow (SP) - 2$<br>$((SP) + 1:(SP)) \leftarrow (DI)$ | <p>AF –<br/>CF –<br/>DF –<br/>IF –<br/>OF –<br/>PF –<br/>SF –<br/>TF –<br/>ZF –</p> |
| PUSHF | <p><b>Push Flags:</b><br/>PUSHF<br/>Decrements SP by two and then transfers all flags to the word at the top of stack pointed to by SP.</p> <p><b>Instruction Operands:</b><br/>none</p>  | $(SP) \leftarrow (SP) - 2$<br>$((SP) + 1:(SP)) \leftarrow \text{Flags}$  | <p>AF –<br/>CF –<br/>DF –<br/>IF –<br/>OF –<br/>PF –<br/>SF –<br/>TF –<br/>ZF –</p> |

**NOTE:** The three symbols used in the Flags Affected column are defined as follows:  
 – the contents of the flag remain unchanged after the instruction is executed  
 ? the contents of the flag is undefined after the instruction is executed  
 ✓ the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

| Name | Description  | Operation   | Flags Affected  |
|------|--|---|---|
| RCL  | <p><b>Rotate Through Carry Left:</b></p> <p>RCL <i>dest, count</i></p> <p>Rotates the bits in the byte or word destination operand to the left by the number of bits specified in the count operand. The carry flag (CF) is treated as "part of" the destination operand; that is, its value is rotated into the low-order bit of the destination, and itself is replaced by the high-order bit of the destination.</p> <p><b>Instruction Operands:</b></p> <p>RCL reg, n<br/>RCL mem, n<br/>RCL reg, CL<br/>RCL mem, CL</p> | <pre>(temp) ← count do while (temp) ≠ 0   (tmpcf) ← (CF)   (CF) ← high-order bit of (dest)   (dest) ← (dest) × 2 + (tmpcf)   (temp) ← (temp) – 1 if   count = 1 then   if     high-order bit of (dest) ≠ (CF)   then     (OF) ← 1   else     (OF) ← 0 else   (OF) undefined</pre>   | <p>AF –</p> <p>CF ✓</p> <p>DF –</p> <p>IF –</p> <p>OF ✓</p> <p>PF –</p> <p>SF –</p> <p>TF –</p> <p>ZF –</p> |
| RCR  | <p><b>Rotate Through Carry Right:</b></p> <p>RCR <i>dest, count</i></p> <p>Operates exactly like RCL except that the bits are rotated right instead of left.</p> <p><b>Instruction Operands:</b></p> <p>RCR reg, n<br/>RCR mem, n<br/>RCR reg, CL<br/>RCR mem, CL</p>  | <pre>(temp) ← count do while (temp) ≠ 0   (tmpcf) ← (CF)   (CF) ← low-order bit of (dest)   (dest) ← (dest) / 2   high-order bit of (dest) ← (tmpcf)   (temp) ← (temp) – 1 if   count = 1 then   if     high-order bit of (dest) ≠     next-to-high-order bit of (dest)   then     (OF) ← 1   else     (OF) ← 0 else   (OF) undefined</pre> | <p>AF –</p> <p>CF ✓</p> <p>DF –</p> <p>IF –</p> <p>OF ✓</p> <p>PF –</p> <p>SF –</p> <p>TF –</p> <p>ZF –</p> |

**NOTE:** The three symbols used in the Flags Affected column are defined as follows:  
 – the contents of the flag remain unchanged after the instruction is executed  
 ? the contents of the flag is undefined after the instruction is executed  
 ✓ the flag is updated after the instruction is executed

**Table C-4. Instruction Set (Continued)**

| Name                                  | Description  | Operation   | Flags Affected  |
|---------------------------------------|--|---|---|
| REP<br>REPE<br>REPZ<br>REPNE<br>REPNZ | <p><b>Repeat:</b></p> <p><b>Repeat While Equal:</b></p> <p><b>Repeat While Zero:</b></p> <p><b>Repeat While Not Equal:</b></p> <p><b>Repeat While Not Zero:</b></p> <p>Controls subsequent string instruction repetition. The different mnemonics are provided to improve program clarity.</p> <p>REP is used in conjunction with the MOVS (Move String) and STOS (Store String) instructions and is interpreted as "repeat while not end-of-string" (CX not 0).</p> <p>REPE and REPZ operate identically and are physically the same prefix byte as REP. These instructions are used with the CMPS (Compare String) and SCAS (Scan String) instructions and require ZF (posted by these instructions) to be set before initiating the next repetition.</p> <p>REPNE and REPNZ are mnemonics for the same prefix byte. These instructions function the same as REPE and REPZ except that the zero flag must be cleared or the repetition is terminated. ZF does not need to be initialized before executing the repeated string instruction.</p> <p><b>Instruction Operands:</b></p> <p>none</p> | <p>do while (CX) ≠ 0</p> <p>service pending interrupts (if any)</p> <p>execute primitive string</p> <p>Operation in succeeding byte</p> <p>(CX) ← (CX) – 1</p> <p>if</p> <p>primitive operation is CMPB, CMPW, SCAB, or SCAW and (ZF) ≠ 0</p> <p>then</p> <p>exit from while loop</p> | <p>AF –</p> <p>CF –</p> <p>DF –</p> <p>IF –</p> <p>OF –</p> <p>PF –</p> <p>SF –</p> <p>TF –</p> <p>ZF –</p> |

**NOTE:** The three symbols used in the Flags Affected column are defined as follows:

- the contents of the flag remain unchanged after the instruction is executed
- ? the contents of the flag is undefined after the instruction is executed
- ✓ the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

| Name | Description   | Operation   | Flags Affected   |
|------|---|---|--|
| RET  | <p><b>Return:</b></p> <p>RET <i>optional-pop-value</i></p> <p>Transfers control from a procedure back to the instruction following the CALL that activated the procedure. The assembler generates an intra-segment RET if the programmer has defined the procedure near, or an intersegment RET if the procedure has been defined as far. RET pops the word at the top of the stack (pointed to by register SP) into the instruction pointer and increments SP by two. If RET is intersegment, the word at the new top of stack is popped into the CS register, and SP is again incremented by two. If an optional pop value has been specified, RET adds that value to SP.</p> <p><b>Instruction Operands:</b></p> <p>RET immed8</p> | $(IP) \leftarrow ((SP) = 1:(SP))$<br>$(SP) \leftarrow (SP) + 2$<br>if<br>inter-segment<br>then<br>$(CS) \leftarrow ((SP) + 1:(SP))$<br>$(SP) \leftarrow (SP) + 2$<br>if<br>add immed8 to SP<br>then<br>$(SP) \leftarrow (SP) + \text{data}$   | AF –<br>CF –<br>DF –<br>IF –<br>OF –<br>PF –<br>SF –<br>TF –<br>ZF – |
| ROL  | <p><b>Rotate Left:</b></p> <p>ROL <i>dest, count</i></p> <p>Rotates the destination byte or word left by the number of bits specified in the count operand.</p> <p><b>Instruction Operands:</b></p> <p>ROL reg, n<br/>           ROL mem, n<br/>           ROL reg, CL<br/>           ROL mem CL</p>  | $(\text{temp}) \leftarrow \text{count}$<br>do while $(\text{temp}) \neq 0$<br>$(CF) \leftarrow \text{high-order bit of } (\text{dest})$<br>$(\text{dest}) \leftarrow (\text{dest}) \times 2 + (CF)$<br>$(\text{temp}) \leftarrow (\text{temp}) - 1$<br>if<br>count = 1<br>then<br>if<br>high-order bit of $(\text{dest}) \neq (CF)$<br>then<br>$(OF) \leftarrow 1$<br>else<br>$(OF) \leftarrow 0$<br>else<br>$(OF)$ undefined | AF –<br>CF ✓<br>DF –<br>IF –<br>OF ✓<br>PF –<br>SF –<br>TF –<br>ZF – |

**NOTE:** The three symbols used in the Flags Affected column are defined as follows:  
 – the contents of the flag remain unchanged after the instruction is executed  
 ? the contents of the flag is undefined after the instruction is executed  
 ✓ the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

| Name | Description  | Operation   | Flags Affected   |
|------|--|---|--|
| ROR  | <p><b>Rotate Right:</b><br/> ROR <i>dest, count</i></p> <p>Operates similar to ROL except that the bits in the destination byte or word are rotated right instead of left.</p> <p><b>Instruction Operands:</b><br/> ROR reg, n<br/> ROR mem, n<br/> ROR reg, CL<br/> ROR mem, CL</p> | <pre>(temp) ← count do while (temp) ≠ 0   (CF) ← low-order bit of (dest)   (dest) ← (dest) / 2   high-order bit of (dest) ← (CF)   (temp) ← (temp) - 1 if   count = 1 then   if     high-order bit of (dest) ≠     next-to-high-order bit of (dest)   then     (OF) ← 1   else     (OF) ← 0 else   (OF) undefined</pre> | AF –<br>CF ✓<br>DF –<br>IF –<br>OF ✓<br>PF –<br>SF –<br>TF –<br>ZF – |
| SAHF | <p><b>Store Register AH Into Flags:</b><br/> SAHF</p> <p>Transfers bits 7, 6, 4, 2, and 0 from register AH into SF, ZF, AF, PF, and CF, respectively, replacing whatever values these flags previously had.</p> <p><b>Instruction Operands:</b><br/> none</p>                        | <pre>(SF):(ZF):X:(AF):X:(PF):X:(CF) ← (AH)</pre>  | AF ✓<br>CF ✓<br>DF –<br>IF –<br>OF –<br>PF ✓<br>SF ✓<br>TF –<br>ZF ✓ |

**NOTE:** The three symbols used in the Flags Affected column are defined as follows:  
– the contents of the flag remain unchanged after the instruction is executed  
? the contents of the flag is undefined after the instruction is executed  
✓ the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

| Name       | Description  | Operation   | Flags Affected  |
|------------|--|---|---|
| SHL<br>SAL | <p><b>Shift Logical Left:</b></p> <p><b>Shift Arithmetic Left:</b></p> <p>SHL <i>dest, count</i><br/>SAL <i>dest, count</i></p> <p>Shifts the destination byte or word left by the number of bits specified in the count operand. Zeros are shifted in on the right. If the sign bit retains its original value, then OF is cleared.</p> <p><b>Instruction Operands:</b></p> <p>SHL reg, n      SAL reg, n<br/>SHL mem, n     SAL mem, n<br/>SHL reg, CL    SAL reg, CL<br/>SHL mem, CL   SAL mem, CL</p>  | <pre>(temp) ← count do while (temp) ≠ 0   (CF) ← high-order bit of (dest)   (dest) ← (dest) × 2   (temp) ← (temp) – 1 if   count = 1 then   if     high-order bit of (dest) ≠ (CE)   then     (OF) ← 1   else     (OF) ← 0 else   (OF) undefined</pre>                          | <p>AF ?</p> <p>CF ✓</p> <p>DF –</p> <p>IF –</p> <p>OF ✓</p> <p>PF ✓</p> <p>SF ✓</p> <p>TF –</p> <p>ZF ✓</p> |
| SAR        | <p><b>Shift Arithmetic Right:</b></p> <p>SAR <i>dest, count</i></p> <p>Shifts the bits in the destination operand (byte or word) to the right by the number of bits specified in the count operand. Bits equal to the original high-order (sign) bit are shifted in on the left, preserving the sign of the original value. Note that SAR does not produce the same result as the dividend of an "equivalent" IDIV instruction if the destination operand is negative and 1 bits are shifted out. For example, shifting –5 right by one bit yields –3, while integer division –5 by 2 yields –2. The difference in the instructions is that IDIV truncates all numbers toward zero, while SAR truncates positive numbers toward zero and negative numbers toward negative infinity.</p> <p><b>Instruction Operands:</b></p> <p>SAR reg, n<br/>SAR mem, n<br/>SAR reg, CL<br/>SAR mem, CL</p> | <pre>(temp) ← count do while (temp) ≠ 0   (CF) ← low-order bit of (dest)   (dest) ← (dest) / 2   (temp) ← (temp) – 1 if   count = 1 then   if     high-order bit of (dest) ≠     next-to-high-order bit of (dest)   then     (OF) ← 1   else     (OF) ← 0 else   (OF) ← 0</pre> | <p>AF ?</p> <p>CF ✓</p> <p>DF –</p> <p>IF –</p> <p>OF ✓</p> <p>PF ✓</p> <p>SF ✓</p> <p>TF –</p> <p>ZF ✓</p> |

**NOTE:** The three symbols used in the Flags Affected column are defined as follows:

- the contents of the flag remain unchanged after the instruction is executed
- ? the contents of the flag is undefined after the instruction is executed
- ✓ the flag is updated after the instruction is executed



**Table C-4. Instruction Set (Continued)**

| Name | Description  | Operation   | Flags Affected  |
|------|--|---|---|
| SBB  | <p><b>Subtract With Borrow:</b></p> <p>SBB <i>dest, src</i></p> <p>Subtracts the source from the destination, subtracts one if CF is set, and returns the result to the destination operand. Both operands may be bytes or words. Both operands may be signed or unsigned binary numbers (see AAS and DAS)</p> <p><b>Instruction Operands:</b></p> <p>SBB reg, reg<br/>           SBB reg, mem<br/>           SBB mem, reg<br/>           SBB accum, immed<br/>           SBB reg, immed<br/>           SBB mem, immed</p> | <p>if<br/>           (CF) = 1<br/>           then<br/>           (dest) = (dest) – (src) – 1<br/>           else<br/>           (dest) ← (dest) – (src)</p> | <p>AF ✓<br/>           CF ✓<br/>           DF –<br/>           IF –<br/>           OF ✓<br/>           PF ✓<br/>           SF ✓<br/>           TF –<br/>           ZF ✓</p> |

**NOTE:** The three symbols used in the Flags Affected column are defined as follows:  
 – the contents of the flag remain unchanged after the instruction is executed  
 ? the contents of the flag is undefined after the instruction is executed  
 ✓ the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

| Name | Description  | Operation   | Flags Affected  |
|------|--|---|---|
| SCAS | <p><b>Scan String:</b></p> <p>SCAS <i>dest-string</i></p> <p>Subtracts the destination string element (byte or word) addressed by DI from the content of AL (byte string) or AX (word string) and updates the flags, but does not alter the destination string or the accumulator. SCAS also updates DI to point to the next string element and AF, CF, OF, PF, SF and ZF to reflect the relationship of the scan value in AL/AX to the string element. If SCAS is prefixed with REPE or REPZ, the operation is interpreted as "scan while not end-of-string (CX not 0) and string-element = scan-value (ZF = 1)." This form may be used to scan for departure from a given value. If SCAS is prefixed with REPNE or REPNZ, the operation is interpreted as "scan while not end-of-string (CX not 0) and string-element is not equal to scan-value (ZF = 0)."</p> <p><b>Instruction Operands:</b></p> <p>SCAS <i>dest-string</i><br/> SCAS (repeat) <i>dest-string</i></p> | <p><b>When Source Operand is a Byte:</b></p> <p>(AL) ← (byte-string)</p> <p>if<br/> (DF) = 0<br/> then<br/> (DI) ← (DI) + DELTA<br/> else<br/> (DI) ← (DI) – DELTA</p> <p><b>When Source Operand is a Word:</b></p> <p>(AX) ← (word-string)</p> <p>if<br/> (DF) = 0<br/> then<br/> (DI) ← (DI) + DELTA<br/> else<br/> (DI) ← (DI) – DELTA</p> | <p>AF ✓<br/> CF ✓<br/> DF –<br/> IF –<br/> OF ✓<br/> PF ✓<br/> SF ✓<br/> TF –<br/> ZF ✓</p> |

**NOTE:** The three symbols used in the Flags Affected column are defined as follows:  
– the contents of the flag remain unchanged after the instruction is executed  
? the contents of the flag is undefined after the instruction is executed  
✓ the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

| Name | Description   | Operation   | Flags Affected  |
|------|---|---|---|
| SHR  | <p><b>Shift Logical Right:</b><br/>SHR <i>dest, src</i></p> <p>Shifts the bits in the destination operand (byte or word) to the right by the number of bits specified in the count operand. Zeros are shifted in on the left. If the sign bit retains its original value, then OF is cleared.</p> <p><b>Instruction Operands:</b><br/>SHR reg, n<br/>SHR mem, n<br/>SHR reg, CL<br/>SHR mem, CL</p> | <pre>(temp) ← count do while (temp) ≠ 0   (CF) ← low-order bit of (dest)   (dest) ← (dest) / 2   (temp) ← (temp) - 1 if   count = 1 then   if     high-order bit of (dest) ≠     next-to-high-order bit of (dest)   then     (OF) ← 1   else     (OF) ← 0 else   (OF) undefined</pre> | <p>AF ?<br/>CF ✓<br/>DF –<br/>IF –<br/>OF ✓<br/>PF ✓<br/>SF ✓<br/>TF –<br/>ZF ✓</p> |
| STC  | <p><b>Set Carry Flag:</b><br/>STC</p> <p>Sets CF to 1.</p> <p><b>Instruction Operands:</b><br/>none</p>   | <pre>(CF) ← 1</pre>   | <p>AF –<br/>CF ✓<br/>DF –<br/>IF –<br/>OF –<br/>PF –<br/>SF –<br/>TF –<br/>ZF –</p> |
| STD  | <p><b>Set Direction Flag:</b><br/>STD</p> <p>Sets DF to 1 causing the string instructions to auto-decrement the SI and/or DI index registers.</p> <p><b>Instruction Operands:</b><br/>none</p>  | <pre>(DF) ← 1</pre>   | <p>AF –<br/>CF –<br/>DF ✓<br/>IF –<br/>OF –<br/>PF –<br/>SF –<br/>TF –<br/>ZF –</p> |

**NOTE:** The three symbols used in the Flags Affected column are defined as follows:  
 – the contents of the flag remain unchanged after the instruction is executed  
 ? the contents of the flag is undefined after the instruction is executed  
 ✓ the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

| Name | Description  | Operation   | Flags Affected  |
|------|--|---|---|
| STI  | <p><b>Set Interrupt-enable Flag:</b></p> <p>STI</p> <p>Sets IF to 1, enabling processor recognition of maskable interrupt requests appearing on the INTR line. Note however, that a pending interrupt will not actually be recognized until the instruction following STI has executed.</p> <p><b>Instruction Operands:</b></p> <p>none</p>                            | $(IF) \leftarrow 1$   | <p>AF –</p> <p>CF –</p> <p>DF –</p> <p>IF ✓</p> <p>OF –</p> <p>PF –</p> <p>SF –</p> <p>TF –</p> <p>ZF –</p> |
| STOS | <p><b>Store (Byte or Word) String:</b></p> <p>STOS <i>dest-string</i></p> <p>Transfers a byte or word from register AL or AX to the string element addressed by DI and updates DI to point to the next location in the string. As a repeated operation.</p> <p><b>Instruction Operands:</b></p> <p>STOS <i>dest-string</i></p> <p>STOS (repeat) <i>dest-string</i></p> | <p><b>When Source Operand is a Byte:</b></p> <p><math>(DEST) \leftarrow (AL)</math></p> <p>if</p> <p><math>(DF) = 0</math></p> <p>then</p> <p><math>(DI) \leftarrow (DI) + DELTA</math></p> <p>else</p> <p><math>(DI) \leftarrow (DI) - DELTA</math></p> <p><b>When Source Operand is a Word:</b></p> <p><math>(DEST) \leftarrow (AX)</math></p> <p>if</p> <p><math>(DF) = 0</math></p> <p>then</p> <p><math>(DI) \leftarrow (DI) + DELTA</math></p> <p>else</p> <p><math>(DI) \leftarrow (DI) - DELTA</math></p> | <p>AF –</p> <p>CF –</p> <p>DF –</p> <p>IF –</p> <p>OF –</p> <p>PF –</p> <p>SF –</p> <p>TF –</p> <p>ZF –</p> |

**NOTE:** The three symbols used in the Flags Affected column are defined as follows:

- the contents of the flag remain unchanged after the instruction is executed
- ? the contents of the flag is undefined after the instruction is executed
- ✓ the flag is updated after the instruction is executed

**Table C-4. Instruction Set (Continued)**

| Name | Description  | Operation   | Flags Affected   |
|------|--|---|--|
| SUB  | <p><b>Subtract:</b><br/>           SUB <i>dest, src</i></p> <p>The source operand is subtracted from the destination operand, and the result replaces the destination operand. The operands may be bytes or words. Both operands may be signed or unsigned binary numbers (see AAS and DAS).</p> <p><b>Instruction Operands:</b><br/>           SUB reg, reg<br/>           SUB reg, mem<br/>           SUB mem, reg<br/>           SUB accum, immed<br/>           SUB reg, immed<br/>           SUB mem, immed</p>   | $(dest) \leftarrow (dest) - (src)$  | AF ✓<br>CF ✓<br>DF –<br>IF –<br>OF ✓<br>PF ✓<br>SF ✓<br>TF –<br>ZF ✓ |
| TEST | <p><b>Test:</b><br/>           TEST <i>dest, src</i></p> <p>Performs the logical "and" of the two operands (bytes or words), updates the flags, but does not return the result, i.e., neither operand is changed. If a TEST instruction is followed by a JNZ (jump if not zero) instruction, the jump will be taken if there are any corresponding one bits in both operands.</p> <p><b>Instruction Operands:</b><br/>           TEST reg, reg<br/>           TEST reg, mem<br/>           TEST accum, immed<br/>           TEST reg, immed<br/>           TEST mem, immed</p> | $(dest) \text{ and } (src)$<br>$(CF) \leftarrow 0$<br>$(OF) \leftarrow 0$ | AF ?<br>CF ✓<br>DF –<br>IF –<br>OF ✓<br>PF ✓<br>SF ✓<br>TF –<br>ZF ✓ |

**NOTE:** The three symbols used in the Flags Affected column are defined as follows:  
 – the contents of the flag remain unchanged after the instruction is executed  
 ? the contents of the flag is undefined after the instruction is executed  
 ✓ the flag is updated after the instruction is executed

Table C-4. Instruction Set (Continued)

| Name | Description   | Operation  | Flags Affected   |
|------|---|--|--|
| WAIT | <p><b>Wait:</b><br/>WAIT<br/>Causes the CPU to enter the wait state while its test line is not active.</p> <p><b>Instruction Operands:</b><br/>none</p>   | None   | AF –<br>CF –<br>DF –<br>IF –<br>OF –<br>PF –<br>SF –<br>TF –<br>ZF – |
| XCHG | <p><b>Exchange:</b><br/>XCHG <i>dest, src</i><br/>Switches the contents of the source and destination operands (bytes or words). When used in conjunction with the LOCK prefix, XCHG can test and set a semaphore that controls access to a resource shared by multiple processors.</p> <p><b>Instruction Operands:</b><br/>XCHG accum, reg<br/>XCHG mem, reg<br/>XCHG reg, reg</p> | $(temp) \leftarrow (dest)$<br>$(dest) \leftarrow (src)$<br>$(src) \leftarrow (temp)$ | AF –<br>CF –<br>DF –<br>IF –<br>OF –<br>PF –<br>SF –<br>TF –<br>ZF – |

**NOTE:** The three symbols used in the Flags Affected column are defined as follows:  
 – the contents of the flag remain unchanged after the instruction is executed  
 ? the contents of the flag is undefined after the instruction is executed  
 ✓ the flag is updated after the instruction is executed



**Table C-4. Instruction Set (Continued)**

| Name | Description   | Operation   | Flags Affected   |
|------|---|---|--|
| XLAT | <p><b>Translate:</b><br/> <i>XLAT translate-table</i></p> <p>Replaces a byte in the AL register with a byte from a 256-byte, user-coded translation table. Register BX is assumed to point to the beginning of the table. The byte in AL is used as an index into the table and is replaced by the byte at the offset in the table corresponding to AL's binary value. The first byte in the table has an offset of 0. For example, if AL contains 5H, and the sixth element of the translation table contains 33H, then AL will contain 33H following the instruction. XLAT is useful for translating characters from one code to another, the classic example being ASCII to EBCDIC or the reverse.</p> <p><b>Instruction Operands:</b><br/>           XLAT src-table</p> | $AL \leftarrow ((BX) + (AL))$   | AF –<br>CF –<br>DF –<br>IF –<br>OF –<br>PF –<br>SF –<br>TF –<br>ZF – |
| XOR  | <p><b>Exclusive Or:</b><br/> <i>XOR dest, src</i></p> <p>Performs the logical "exclusive or" of the two operands and returns the result to the destination operand. A bit in the result is set if the corresponding bits of the original operands contain opposite values (one is set, the other is cleared); otherwise the result bit is cleared.</p> <p><b>Instruction Operands:</b><br/>           XOR reg, reg<br/>           XOR reg, mem<br/>           XOR mem, reg<br/>           XOR accum, immed<br/>           XOR reg, immed<br/>           XOR mem, immed</p>  | $(dest) \leftarrow (dest) \text{ xor } (src)$<br>$(CF) \leftarrow 0$<br>$(OF) \leftarrow 0$ | AF ?<br>CF ✓<br>DF –<br>IF –<br>OF ✓<br>PF ✓<br>SF ✓<br>TF –<br>ZF ✓ |

**NOTE:** The three symbols used in the Flags Affected column are defined as follows:  
 – the contents of the flag remain unchanged after the instruction is executed  
 ? the contents of the flag is undefined after the instruction is executed  
 ✓ the flag is updated after the instruction is executed







**D**

**Instruction Set  
Opcodes and Clock  
Cycles**

**I**



intel<sup>®</sup>

I



Table D-2. Instruction Set Summary

| Function                            | Format          |               |             |             | Clocks | Notes |
|-------------------------------------|-----------------|---------------|-------------|-------------|--------|-------|
| <b>DATA TRANSFER INSTRUCTIONS</b>   |                 |               |             |             |        |       |
| <b>MOV = Move</b>                   |                 |               |             |             |        |       |
| register to register/memory         | 1 0 0 0 1 0 0 w | mod reg r/m   |             |             | 2/12   |       |
| register/memory to register         | 1 0 0 0 1 0 1 w | mod reg r/m   |             |             | 2/9    |       |
| immediate to register/memory        | 1 1 0 0 0 1 1 w | mod 000 r/m   | data        | data if w=1 | 12/13  | (1)   |
| immediate to register               | 1 0 1 1 w reg   | data          | data if w=1 |             | 3/4    | (1)   |
| memory to accumulator               | 1 0 1 0 0 0 0 w | addr-low      | addr-high   |             | 9      |       |
| accumulator to memory               | 1 0 1 0 0 0 1 w | addr-low      | addr-high   |             | 8      |       |
| register/memory to segment register | 1 0 0 0 1 1 1 0 | mod 0 reg r/m |             |             | 2/9    |       |
| segment register to register/memory | 1 0 0 0 1 1 0 0 | mod 0 reg r/m |             |             | 2/11   |       |
| <b>PUSH = Push</b>                  |                 |               |             |             |        |       |
| memory                              | 1 1 1 1 1 1 1 1 | mod 110 r/m   |             |             | 16     |       |
| register                            | 0 1 0 1 0 reg   |               |             |             | 10     |       |
| segment register                    | 0 0 0 reg 1 1 0 |               |             |             | 9      |       |
| immediate                           | 0 1 1 0 1 0 s 0 | data          | data if s=0 |             | 10     |       |
| <b>POP = Pop</b>                    |                 |               |             |             |        |       |
| memory                              | 1 0 0 0 1 1 1 1 | mod 000 r/m   |             |             | 20     |       |
| register                            | 0 1 0 1 1 reg   |               |             |             | 10     |       |
| segment register                    | 0 0 0 reg 1 1 1 | (reg ?01)     |             |             | 8      |       |
| <b>PUSHA = Push all</b>             | 0 1 1 0 0 0 0 0 |               |             |             | 36     |       |
| <b>POPA = Pop all</b>               | 0 1 1 0 0 0 0 1 |               |             |             | 51     |       |
| <b>XCHG = Exchange</b>              |                 |               |             |             |        |       |
| 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      |       |
| <b>XLAT = Translate byte to AL</b>  | 1 1 0 1 0 1 1 1 |               |             |             | 11     |       |
| <b>IN = Input from</b>              |                 |               |             |             |        |       |
| fixed port                          | 1 1 1 0 0 1 0 w | port          |             |             | 10     |       |
| variable port                       | 1 1 1 0 1 1 0 w |               |             |             | 8      |       |
| <b>OUT = Output from</b>            |                 |               |             |             |        |       |
| fixed port                          | 1 1 1 0 0 1 0 w | port          |             |             | 9      |       |
| variable port                       | 1 1 1 0 1 1 0 w |               |             |             | 7      |       |

**NOTES:**

1. Clock cycles are given for 8-bit/16-bit operations.
2. Clock cycles are given for jump not taken/jump taken.
3. Clock cycles are given for interrupt taken/interrupt not taken.
4. If TEST = 0

Shading indicates additions and enhancements to the 8086/8088 instruction set. See Appendix A, "80C186 Instruction Set Additions and Extensions," for details.





INSTRUCTION SET OPCODES AND CLOCK CYCLES

Table D-2. Instruction Set Summary (Continued)

| Function                                      | Format  | Clocks     | Notes |
|---|---|------------|-------|
| <b>DATA TRANSFER INSTRUCTIONS</b> (Continued) |   |            |       |
| LEA = Load EA to register                     | 1 0 0 0 1 1 0 1    mod reg r/m                        | 6          |       |
| LDS = Load pointer to DS                      | 1 1 0 0 0 1 0 1    mod reg r/m    (mod ?11)           | 18         |       |
| LES = Load pointer to ES                      | 1 1 0 0 0 1 0 0    mod reg r/m    (mod ?11)           | 18         |       |
| ENTER = Build stack frame                     | 1 1 0 0 1 0 0 0    data-low    data-high    L         |            |       |
| L = 0   |   | 15         |       |
| L = 1   |   | 25         |       |
| L > 1   |   | 22+16(n-1) |       |
| LEAVE = Tear down stack frame                 | 1 1 0 0 1 0 0 1                                       | 8          |       |
| LAHF = Load AH with flags                     | 1 0 0 1 1 1 1 1                                       | 2          |       |
| SAHF = Store AH into flags                    | 1 0 0 1 1 1 1 0                                       | 3          |       |
| PUSHF = Push flags                            | 1 0 0 1 1 1 0 0                                       | 9          |       |
| POPF = Pop flags                              | 1 0 0 1 1 1 0 1                                       | 8          |       |
| <b>ARITHMETIC INSTRUCTIONS</b>                |   |            |       |
| <b>ADD = Add</b>                              |   |            |       |
| reg/memory with register to either            | 0 0 0 0 0 d w    mod reg r/m                          | 3/10       |       |
| immediate to register/memory                  | 1 0 0 0 0 s w    mod 000 r/m    data    data if sw=01 | 4/16       |       |
| immediate to accumulator                      | 0 0 0 0 1 0 w    data    data if w=1                  | 3/4        | (1)   |
| <b>ADC = Add with carry</b>                   |   |            |       |
| 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 s w    mod 010 r/m    data    data if sw=01 | 4/16       |       |
| immediate to accumulator                      | 0 0 0 1 0 1 0 w    data    data if w=1                | 3/4        | (1)   |
| <b>INC = Increment</b>                        |   |            |       |
| register/memory                               | 1 1 1 1 1 1 w    mod 000 r/m                          | 3/15       |       |
| register                                      | 0 1 0 0 0 reg   | 3          |       |
| AAA = ASCII adjust for addition               | 0 0 1 1 0 1 1 1                                       | 8          |       |
| DAA = Decimal adjust for addition             | 0 0 1 0 0 1 1 1                                       | 4          |       |

**NOTES:**

1. Clock cycles are given for 8-bit/16-bit operations.
2. Clock cycles are given for jump not taken/jump taken.
3. Clock cycles are given for interrupt taken/interrupt not taken.
4. If TEST = 0

Shading indicates additions and enhancements to the 8086/8088 instruction set. See Appendix A, "80C186 Instruction Set Additions and Extensions," for details.



Table D-2. Instruction Set Summary (Continued)

| Function                                    | Format   | Clocks          | Notes |
|---|--|-----------------|-------|
| <b>ARITHMETIC INSTRUCTIONS (Continued)</b>  |  |                 |       |
| <b>SUB</b> = Subtract                       |  |                 |       |
| reg/memory with register to either          | 0 0 1 0 1 0 d w   mod reg r/m                      | 3/10            |       |
| immediate from register/memory              | 1 0 0 0 0 s w   mod 101 r/m   data   data if sw=01 | 4/16            |       |
| immediate from accumulator                  | 0 0 0 1 1 1 0 w   data   data if w=1               | 3/4             | (1)   |
| <b>SBB</b> = Subtract with borrow           |  |                 |       |
| reg/memory with register to either          | 0 0 0 1 1 0 d w   mod reg r/m                      | 3/10            |       |
| immediate from register/memory              | 1 0 0 0 0 s w   mod 011 r/m   data   data if sw=01 | 4/16            |       |
| immediate from accumulator                  | 0 0 0 1 1 1 0 w   data   data if w=1               | 3/4             | (1)   |
| <b>DEC</b> = Decrement                      |  |                 |       |
| register/memory                             | 1 1 1 1 1 1 w   mod 001 r/m                        | 3/15            |       |
| register                                    | 0 1 0 0 1 reg                                      | 3               |       |
| <b>NEG</b> = Change sign                    | 1 1 1 1 0 1 1 w   mod reg r/m                      | 3               |       |
| <b>CMP</b> = 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 s w   mod 111 r/m   data   data if sw=01 | 3/10            |       |
| immediate with accumulator                  | 0 0 1 1 1 1 0 w   data   data if w=1               | 3/4             | (1)   |
| <b>AAS</b> = ASCII adjust for subtraction   | 0 0 1 1 1 1 1 1                                    | 7               |       |
| <b>DAS</b> = Decimal adjust for subtraction | 0 0 1 0 1 1 1 1                                    | 4               |       |
| <b>MUL</b> = multiply (unsigned)            | 1 1 1 1 0 1 1 w   mod 100 r/m                      |                 |       |
| register-byte                               |  | 26-28           |       |
| register-word                               |  | 35-37           |       |
| memory-byte                                 |  | 32-34           |       |
| memory-word                                 |  | 41-43           |       |
| <b>IMUL</b> = Integer multiply (signed)     | 1 1 1 1 0 1 1 w   mod 101 r/m                      |                 |       |
| register-byte                               |  | 25-28           |       |
| register-word                               |  | 34-37           |       |
| memory-byte                                 |  | 31-34           |       |
| memory-word                                 |  | 40-43           |       |
| integer <b>immediate</b> multiply (signed)  | 0 1 1 0 1 0 s 1   mod reg r/m   data   data if s=0 | 22-25/<br>29-32 |       |

**NOTES:**

1. Clock cycles are given for 8-bit/16-bit operations.
2. Clock cycles are given for jump not taken/jump taken.
3. Clock cycles are given for interrupt taken/interrupt not taken.
4. If  $\overline{\text{TEST}} = 0$

Shading indicates additions and enhancements to the 8086/8088 instruction set. See Appendix A, "80C186 Instruction Set Additions and Extensions," for details.



## INSTRUCTION SET OPCODES AND CLOCK CYCLES

**Table D-2. Instruction Set Summary (Continued)**

| Function                                   | Format  | Clocks | Notes |
|--|---|--------|-------|
| <b>ARITHMETIC INSTRUCTIONS (Continued)</b> |   |        |       |
| <b>AAM</b> = ASCII adjust for multiply     | 1 1 0 1 0 1 0 0    0 0 0 0 1 0 1 0                    | 19     |       |
| <b>DIV</b> = Divide (unsigned)             | 1 1 1 1 0 1 1 w    mod 110 r/m                        |        |       |
| register-byte                              |   | 29     |       |
| register-word                              |   | 38     |       |
| memory-byte                                |   | 35     |       |
| memory-word                                |   | 44     |       |
| <b>IDIV</b> = Integer divide (signed)      | 1 1 1 1 0 1 1 w    mod 111 r/m                        |        |       |
| register-byte                              |   | 29     |       |
| register-word                              |   | 38     |       |
| memory-byte                                |   | 35     |       |
| memory-word                                |   | 44     |       |
| <b>AAD</b> = ASCII adjust for divide       | 1 1 0 1 0 1 0 1    0 0 0 0 1 0 1 0                    | 15     |       |
| <b>CBW</b> = Convert byte to word          | 1 0 0 1 1 0 0 0                                       | 2      |       |
| <b>CWD</b> = Convert word to double-word   | 1 0 0 1 1 0 0 1                                       | 4      |       |
| <b>BIT MANIPULATION INSTRUCTIONS</b>       |   |        |       |
| <b>NOT</b> = Invert register/memory        | 1 1 1 1 0 1 1 w    mod 010 r/m                        | 3      |       |
| <b>AND</b> = And                           |   |        |       |
| reg/memory and register to either          | 0 0 1 0 0 0 d w    mod reg r/m                        | 3/10   |       |
| immediate to register/memory               | 1 0 0 0 0 0 0 w    mod 100 r/m    data    data if w=1 | 4/16   |       |
| immediate to accumulator                   | 0 0 1 0 0 1 0 w    data    data if w=1                | 3/4    | (1)   |
| <b>OR</b> = Or                             |   |        |       |
| reg/memory and register to either          | 0 0 0 0 1 0 d w    mod reg r/m                        | 3/10   |       |
| immediate to register/memory               | 1 0 0 0 0 0 0 w    mod 001 r/m    data    data if w=1 | 4/10   |       |
| immediate to accumulator                   | 0 0 0 0 1 1 0 w    data    data if w=1                | 3/4    | (1)   |
| <b>XOR</b> = 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 110 r/m    data    data if w=1 | 4/10   |       |
| immediate to accumulator                   | 0 0 1 1 0 1 0 w    data    data if w=1                | 3/4    | (1)   |

**NOTES:**

1. Clock cycles are given for 8-bit/16-bit operations.
2. Clock cycles are given for jump not taken/jump taken.
3. Clock cycles are given for interrupt taken/interrupt not taken.
4. If TEST = 0

Shading indicates additions and enhancements to the 8086/8088 instruction set. See Appendix A, "80C186 Instruction Set Additions and Extensions," for details.

Table D-2. Instruction Set Summary (Continued)

| Function   | Format          |                 |             |             | Clocks   | Notes |
|--|-----------------|-----------------|-------------|-------------|----------|-------|
| <b>BIT MANIPULATION INSTRUCTIONS</b> (Continued) |                 |                 |             |             |          |       |
| <b>TEST</b> = 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 000 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      | (1)   |
| <b>Shifts/Rotates</b>                            |                 |                 |             |             |          |       |
| 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 |       |
| <b>STRING MANIPULATION INSTRUCTIONS</b>          |                 |                 |             |             |          |       |
| <b>MOVS</b> = Move byte/word                     | 1 0 1 0 0 1 0 w |                 |             |             | 14       |       |
| <b>INS</b> = Input byte/word from DX port        | 0 1 1 0 1 1 0 w |                 |             |             | 14       |       |
| <b>OUTS</b> = Output byte/word to DX port        | 0 1 1 0 1 1 1 w |                 |             |             | 14       |       |
| <b>CMPS</b> = Compare byte/word                  | 1 0 1 0 0 1 1 w |                 |             |             | 22       |       |
| <b>SCAS</b> = Scan byte/word                     | 1 0 1 0 1 1 1 w |                 |             |             | 15       |       |
| <b>LODS</b> = Load byte/word to AL/AX            | 1 0 1 0 1 1 0 w |                 |             |             | 12       |       |
| <b>STOS</b> = Store byte/word from AL/AX         | 1 0 1 0 1 0 1 w |                 |             |             | 10       |       |
| <b>Repeated by count in CX:</b>                  |                 |                 |             |             |          |       |
| <b>MOVS</b> = Move byte/word                     | 1 1 1 1 0 0 1 0 | 1 0 1 0 0 1 0 w |             |             | 8+8n     |       |
| <b>INS</b> = Input byte/word from DX port        | 1 1 1 1 0 0 1 0 | 0 1 1 0 1 1 0 w |             |             | 8-8n     |       |
| <b>OUTS</b> = Output byte/word to DX port        | 1 1 1 1 0 0 1 0 | 0 1 1 0 1 1 1 w |             |             | 8+8n     |       |
| <b>CMPS</b> = Compare byte/word                  | 1 1 1 1 0 0 1 z | 1 0 1 0 0 1 1 w |             |             | 5+22n    |       |
| <b>SCAS</b> = Scan byte/word                     | 1 1 1 1 0 0 1 z | 1 0 1 0 1 1 1 w |             |             | 5+15n    |       |
| <b>LODS</b> = Load byte/word to AL/AX            | 1 1 1 1 0 0 1 0 | 0 1 0 1 0 0 1 w |             |             | 6+11n    |       |
| <b>STOS</b> = Store byte/word from AL/AX         | 1 1 1 1 0 1 0 0 | 0 1 0 1 0 0 1 w |             |             | 6+9n     |       |

**NOTES:**

1. Clock cycles are given for 8-bit/16-bit operations.
2. Clock cycles are given for jump not taken/jump taken.
3. Clock cycles are given for interrupt taken/interrupt not taken.
4. If **TEST** = 0

Shading indicates additions and enhancements to the 8086/8088 instruction set. See Appendix A, "80C186 Instruction Set Additions and Extensions," for details.





INSTRUCTION SET OPCODES AND CLOCK CYCLES

Table D-2. Instruction Set Summary (Continued)

| Function                                   | Format                                      | Clocks | Notes |
|--|---|--------|-------|
| <b>PROGRAM TRANSFER INSTRUCTIONS</b>       |   |        |       |
| <b>Conditional Transfers — jump if:</b>    |   |        |       |
| <b>JE/JZ</b> = equal/zero                  | 0 1 1 1 0 1 0 0    disp                     | 4/13   | (2)   |
| <b>JL/JNGE</b> = less/not greater or equal | 0 1 1 1 1 1 0 0    disp                     | 4/13   | (2)   |
| <b>JLE/JNG</b> = less or equal/not greater | 0 1 1 1 1 1 1 0    disp                     | 4/13   | (2)   |
| <b>JB/JNAE</b> = below/not above or equal  | 0 1 1 1 0 0 1 0    disp                     | 4/13   | (2)   |
| <b>JC</b> = carry                          | 0 1 1 1 0 0 1 0    disp                     | 4/13   | (2)   |
| <b>JBE/JNA</b> = below or equal/not above  | 0 1 1 1 0 1 1 0    disp                     | 4/13   | (2)   |
| <b>JP/JPE</b> = parity/parity even         | 0 1 1 1 1 0 1 0    disp                     | 4/13   | (2)   |
| <b>JO</b> = overflow                       | 0 1 1 1 0 0 0 0    disp                     | 4/13   | (2)   |
| <b>JS</b> = sign                           | 0 1 1 1 1 0 0 0    disp                     | 4/13   | (2)   |
| <b>JNE/JNZ</b> = not equal/not zero        | 0 1 1 1 0 1 0 1    disp                     | 4/13   | (2)   |
| <b>JNL/JGE</b> = not less/greater or equal | 0 1 1 1 1 1 0 1    disp                     | 4/13   | (2)   |
| <b>JNLE/JG</b> = not less or equal/greater | 0 1 1 1 1 1 1 1    disp                     | 4/13   | (2)   |
| <b>JNB/JAE</b> = not below/above or equal  | 0 1 1 1 0 0 1 1    disp                     | 4/13   | (2)   |
| <b>JNC</b> = not carry                     | 0 1 1 1 0 0 1 1    disp                     | 4/13   | (2)   |
| <b>JNBE/JA</b> = not below or equal/above  | 0 1 1 1 0 1 1 1    disp                     | 4/13   | (2)   |
| <b>JNP/JPO</b> = not parity/parity odd     | 0 1 1 1 1 0 1 1    disp                     | 4/13   | (2)   |
| <b>JNO</b> = not overflow                  | 0 1 1 1 0 0 0 1    disp                     | 4/13   | (2)   |
| <b>JNS</b> = not sign                      | 0 1 1 1 1 0 0 1    disp                     | 5/15   | (2)   |
| <b>Unconditional Transfers</b>             |   |        |       |
| <b>CALL</b> = Call procedure               |   |        |       |
| direct within segment                      | 1 1 1 0 1 0 0 0    disp-low    disp-high    | 15     |       |
| reg/memory indirect within segment         | 1 1 1 1 1 1 1 1    mod 010 r/m              | 13/19  |       |
| indirect intersegment                      | 1 1 1 1 1 1 1 1    mod 011 r/m    (mod ?11) | 38     |       |
| direct intersegment                        | 1 0 0 1 1 0 1 0    segment offset           | 23     |       |
|  | selector                                    |        |       |

**NOTES:**

1. Clock cycles are given for 8-bit/16-bit operations.
2. Clock cycles are given for jump not taken/jump taken.
3. Clock cycles are given for interrupt taken/interrupt not taken.
4. If  $\overline{TEST} = 0$

Shading indicates additions and enhancements to the 8086/8088 instruction set. See Appendix A, "80C186 Instruction Set Additions and Extensions," for details.



Table D-2. Instruction Set Summary (Continued)

| Function   | Format          | Clocks | Notes                      |
|--|-----------------|--------|----------------------------|
| <b>PROGRAM TRANSFER INSTRUCTIONS (Continued)</b>     |                 |        |                            |
| <b>RET = Return from procedure</b>                   |                 |        |                            |
| within segment                                       | 1 1 0 0 0 1 1   | 16     |                            |
| within segment adding immed to SP                    | 1 1 0 0 0 1 0   | 18     | data-low data-high         |
| intersegment   | 1 1 0 0 1 0 1 1 | 22     |                            |
| intersegment adding immed to SP                      | 1 1 0 0 1 0 1 0 | 25     | data-low data-high         |
| <b>JMP = Unconditional jump</b>                      |                 |        |                            |
| short/long   | 1 1 1 0 1 0 1 1 | 14     | disp-low                   |
| direct within segment                                | 1 1 1 0 1 0 0 1 | 14     | disp-low disp-high         |
| reg/memory indirect within segment                   | 1 1 1 1 1 1 1 1 | 26     | mod 100 r/m                |
| indirect intersegment                                | 1 1 1 1 1 1 1 1 | 11/17  | mod 101 r/m (mod ?11)      |
| direct intersegment                                  | 1 1 1 0 1 0 1 0 | 14     | segment offset<br>selector |
| <b>Iteration Control</b>                             |                 |        |                            |
| <b>LOOP = Loop CX times</b>                          | 1 1 1 0 0 0 1 0 | 6/16   | (2)                        |
| <b>LOOPZ/LOOPE = Loop while zero/equal</b>           | 1 1 1 0 0 0 0 1 | 5/16   | (2)                        |
| <b>LOOPNZ/LOOPNE = Loop while not zero/not equal</b> | 1 1 1 0 0 0 0 0 | 5/16   | (2)                        |
| <b>JCXZ = Jump if CX = zero</b>                      | 1 1 1 0 0 0 1 1 | 6/16   | (2)                        |
| <b>Interrupts</b>                                    |                 |        |                            |
| <b>INT = Interrupt</b>                               |                 |        |                            |
| Type specified                                       | 1 1 0 0 1 1 0 1 | 47     | type                       |
| Type 3   | 1 1 0 0 1 1 0 0 | 45     |                            |
| <b>INTO = Interrupt on overflow</b>                  | 1 1 0 0 1 1 1 0 | 48/4   | (3)                        |
| <b>BOUND = Detect value out of range</b>             | 0 1 1 0 0 0 1 0 | 33-35  | mod reg r/m                |
| <b>IRET = Interrupt return</b>                       | 1 1 0 0 1 1 1 1 | 28     |                            |

**NOTES:**

1. Clock cycles are given for 8-bit/16-bit operations.
2. Clock cycles are given for jump not taken/jump taken.
3. Clock cycles are given for interrupt taken/interrupt not taken.
4. If **TEST** = 0

Shading indicates additions and enhancements to the 8086/8088 instruction set. See Appendix A, "80C186 Instruction Set Additions and Extensions," for details.





## INSTRUCTION SET OPCODES AND CLOCK CYCLES

**Table D-2. Instruction Set Summary (Continued)**

| Function                              | Format          | Clocks | Notes |             |
|---------------------------------------|-----------------|--------|-------|-------------|
| <b>PROCESSOR CONTROL INSTRUCTIONS</b> |                 |        |       |             |
| CLC = Clear carry                     | 1 1 1 1 1 0 0 0 | 2      | (4)   |             |
| 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      |       |             |
| LOCK = Bus lock prefix                | 1 1 1 1 0 0 0 0 | 2      |       |             |
| ESC = Math coprocessor escape         | 1 1 0 1 1 M M M | 6      |       | mod PPP r/m |
| NOP = No operation                    | 1 0 0 1 0 0 0 0 | 3      |       |             |
| <b>SEGMENT OVERRIDE PREFIX</b>        |                 |        |       |             |
| CS                                    | 0 0 1 0 1 1 1 0 | 2      |       |             |
| SS                                    | 0 0 1 1 0 1 1 0 | 2      |       |             |
| DS                                    | 0 0 1 1 1 1 1 0 | 2      |       |             |
| ES                                    | 0 0 1 0 0 1 1 0 | 2      |       |             |

**NOTES:**

1. Clock cycles are given for 8-bit/16-bit operations.
2. Clock cycles are given for jump not taken/jump taken.
3. Clock cycles are given for interrupt taken/interrupt not taken.
4. If TEST = 0

Shading indicates additions and enhancements to the 8086/8088 instruction set. See Appendix A, "80C186 Instruction Set Additions and Extensions," for details.

**Table D-3. Machine Instruction Decoding Guide**

| Byte 1 |           | Byte 2      | Bytes 3–6           | ASM-86 Instruction Format |                   |
|--------|-----------|-------------|---------------------|---------------------------|-------------------|
| Hex    | Binary    |             |                     |                           |                   |
| 00     | 0000 0000 | mod reg r/m | (disp-lo),(disp-hi) | add                       | reg8/mem8, reg8   |
| 01     | 0000 0001 | mod reg r/m | (disp-lo),(disp-hi) | add                       | reg16/mem16,reg16 |
| 02     | 0000 0010 | mod reg r/m | (disp-lo),(disp-hi) | add                       | reg8,reg8/mem8    |
| 03     | 0000 0011 | mod reg r/m | (disp-lo),(disp-hi) | add                       | reg16,reg16/mem16 |
| 04     | 0000 0100 | data-8      |                     | add                       | AL,immed8         |
| 05     | 0000 0101 | data-lo     | data-hi             | add                       | AX,immed16        |
| 06     | 0000 0110 |             |                     | push                      | ES                |
| 07     | 0000 0111 |             |                     | pop                       | ES                |
| 08     | 0000 0100 | mod reg r/m | (disp-lo),(disp-hi) | or                        | reg8/mem8,reg8    |

Table D-3. Machine Instruction Decoding Guide (Continued)

| Byte 1 |           | Byte 2      | Bytes 3–6           | ASM-86 Instruction Format |                           |
|--------|-----------|-------------|---------------------|---------------------------|---------------------------|
| Hex    | Binary    |             |                     |                           |                           |
| 09     | 0000 1001 | mod reg r/m | (disp-lo),(disp-hi) | or                        | reg16/mem16,reg16         |
| 0A     | 0000 1010 | mod reg r/m | (disp-lo),(disp-hi) | or                        | reg8,reg8/mem8            |
| 0B     | 0000 1011 | mod reg r/m | (disp-lo),(disp-hi) | or                        | reg16,reg16/mem16         |
| 0C     | 0000 1100 | data-8      |                     | or                        | AL, immed8                |
| 0D     | 0000 1101 | data-lo     | data-hi             | or                        | AX,immed16                |
| 0E     | 0000 1110 |             |                     | push                      | CS                        |
| 0F     | 0000 1111 |             |                     | —                         |                           |
| 10     | 0001 0000 | mod reg r/m | (disp-lo),(disp-hi) | adc                       | reg8/mem8,reg8            |
| 11     | 0001 0001 | mod reg r/m | (disp-lo),(disp-hi) | adc                       | reg16/mem16,reg16         |
| 12     | 0001 0010 | mod reg r/m | (disp-lo),(disp-hi) | adc                       | reg8,reg8/mem8            |
| 13     | 0001 0011 | mod reg r/m | (disp-lo),(disp-hi) | adc                       | reg16,reg16/mem16         |
| 14     | 0001 0100 | data-8      |                     | adc                       | AL,immed8                 |
| 15     | 0001 0101 | data-lo     | data-hi             | adc                       | AX,immed16                |
| 16     | 0001 0110 |             |                     | push                      | SS                        |
| 17     | 0001 0111 |             |                     | pop                       | SS                        |
| 18     | 0001 1000 | mod reg r/m | (disp-lo),(disp-hi) | sbb                       | reg8/mem8,reg8            |
| 19     | 0001 1001 | mod reg r/m | (disp-lo),(disp-hi) | sbb                       | reg16/mem16,reg16         |
| 1A     | 0001 1010 | mod reg r/m | (disp-lo),(disp-hi) | sbb                       | reg8,reg8/mem8            |
| 1B     | 0001 1011 | mod reg r/m | (disp-lo),(disp-hi) | sbb                       | reg16,reg16/mem16         |
| 1C     | 0001 1100 | data-8      |                     | sbb                       | AL,immed8                 |
| 1D     | 0001 1101 | data-lo     | data-hi             | sbb                       | AX,immed16                |
| 1E     | 0001 1110 |             |                     | push                      | DS                        |
| 1F     | 0001 1111 |             |                     | pop                       | DS                        |
| 20     | 0010 0000 | mod reg r/m | (disp-lo),(disp-hi) | and                       | reg8/mem8,reg8            |
| 21     | 0010 0001 | mod reg r/m | (disp-lo),(disp-hi) | and                       | reg16/mem16,reg16         |
| 22     | 0010 0010 | mod reg r/m | (disp-lo),(disp-hi) | and                       | reg8,reg8/mem8            |
| 23     | 0010 0011 | mod reg r/m | (disp-lo),(disp-hi) | and                       | reg16,reg16/mem16         |
| 24     | 0010 0100 | data-8      |                     | and                       | AL,immed8                 |
| 25     | 0010 0101 | data-lo     | data-hi             | and                       | AX,immed16                |
| 26     | 0010 0110 |             |                     | ES:                       | (segment override prefix) |
| 27     | 0010 0111 |             |                     | daa                       |                           |
| 28     | 0010 1000 | mod reg r/m | (disp-lo),(disp-hi) | sub                       | reg8/mem8,reg8            |
| 29     | 0010 1001 | mod reg r/m | (disp-lo),(disp-hi) | sub                       | reg16/mem16,reg16         |
| 2A     | 0010 1010 | mod reg r/m | (disp-lo),(disp-hi) | sub                       | reg8,reg8/mem8            |
| 2B     | 0010 1011 | mod reg r/m | (disp-lo),(disp-hi) | sub                       | reg16,reg16/mem16         |
| 2C     | 0010 1100 | data-8      |                     | sub                       | AL,immed8                 |
| 2D     | 0010 1101 | data-lo     | data-hi             | sub                       | AX,immed16                |



## INSTRUCTION SET OPCODES AND CLOCK CYCLES

**Table D-3. Machine Instruction Decoding Guide (Continued)**

| Byte 1 |           | Byte 2      | Bytes 3–6           | ASM-86 Instruction Format |                           |
|--------|-----------|-------------|---------------------|---------------------------|---------------------------|
| Hex    | Binary    |             |                     |                           |                           |
| 2E     | 0010 1110 |             |                     | DS:                       | (segment override prefix) |
| 2F     | 0010 1111 |             |                     | das                       |                           |
| 30     | 0011 0000 | mod reg r/m | (disp-lo),(disp-hi) | xor                       | reg8/mem8,reg8            |
| 31     | 0011 0001 | mod reg r/m | (disp-lo),(disp-hi) | xor                       | reg16/mem16,reg16         |
| 32     | 0011 0010 | mod reg r/m | (disp-lo),(disp-hi) | xor                       | reg8,reg8/mem8            |
| 33     | 0011 0011 | mod reg r/m | (disp-lo),(disp-hi) | xor                       | reg16,reg16/mem16         |
| 34     | 0011 0100 | data-8      |                     | xor                       | AL,immed8                 |
| 35     | 0011 0101 | data-lo     | data-hi             | xor                       | AX,immed16                |
| 36     | 0011 0110 |             |                     | SS:                       | (segment override prefix) |
| 37     | 0011 0111 |             |                     | aaa                       |                           |
| 38     | 0011 1000 | mod reg r/m | (disp-lo),(disp-hi) | xor                       | reg8/mem8,reg8            |
| 39     | 0011 1001 | mod reg r/m | (disp-lo),(disp-hi) | xor                       | reg16/mem16,reg16         |
| 3A     | 0011 1010 | mod reg r/m | (disp-lo),(disp-hi) | xor                       | reg8,reg8/mem8            |
| 3B     | 0011 1011 | mod reg r/m | (disp-lo),(disp-hi) | xor                       | reg16,reg16/mem16         |
| 3C     | 0011 1100 | data-8      |                     | xor                       | AL,immed8                 |
| 3D     | 0011 1101 | data-lo     | data-hi             | xor                       | AX,immed16                |
| 3E     | 0011 1110 |             |                     | DS:                       | (segment override prefix) |
| 3F     | 0011 1111 |             |                     | aas                       |                           |
| 40     | 0100 0000 |             |                     | inc                       | AX                        |
| 41     | 0100 0001 |             |                     | inc                       | CX                        |
| 42     | 0100 0010 |             |                     | inc                       | DX                        |
| 43     | 0100 0011 |             |                     | inc                       | BX                        |
| 44     | 0100 0100 |             |                     | inc                       | SP                        |
| 45     | 0100 0101 |             |                     | inc                       | BP                        |
| 46     | 0100 0110 |             |                     | inc                       | SI                        |
| 47     | 0100 0111 |             |                     | inc                       | DI                        |
| 48     | 0100 1000 |             |                     | dec                       | AX                        |
| 49     | 0100 1001 |             |                     | dec                       | CX                        |
| 4A     | 0100 1010 |             |                     | dec                       | DX                        |
| 4B     | 0100 1011 |             |                     | dec                       | BX                        |
| 4C     | 0100 1100 |             |                     | dec                       | SP                        |
| 4D     | 0100 1101 |             |                     | dec                       | BP                        |
| 4E     | 0100 1110 |             |                     | dec                       | SI                        |
| 4F     | 0100 1111 |             |                     | dec                       | DI                        |
| 50     | 0101 0000 |             |                     | push                      | AX                        |
| 51     | 0101 0001 |             |                     | push                      | CX                        |
| 52     | 0101 0010 |             |                     | push                      | DX                        |

Table D-3. Machine Instruction Decoding Guide (Continued)

| Byte 1 |           | Byte 2      | Bytes 3–6        | ASM-86 Instruction Format |             |
|--------|-----------|-------------|------------------|---------------------------|-------------|
| Hex    | Binary    |             |                  |                           |             |
| 53     | 0101 0011 |             |                  | push                      | BX          |
| 54     | 0101 0100 |             |                  | push                      | SP          |
| 55     | 0101 0101 |             |                  | push                      | BP          |
| 56     | 0101 0110 |             |                  | push                      | SI          |
| 57     | 0101 0111 |             |                  | push                      | DI          |
| 58     | 0101 1000 |             |                  | pop                       | AX          |
| 59     | 0101 1001 |             |                  | pop                       | CX          |
| 5A     | 0101 1010 |             |                  | pop                       | DX          |
| 5B     | 0101 1011 |             |                  | pop                       | BX          |
| 5C     | 0101 1100 |             |                  | pop                       | SP          |
| 5D     | 0101 1101 |             |                  | pop                       | BP          |
| 5E     | 0101 1110 |             |                  | pop                       | SI          |
| 5F     | 0101 1111 |             |                  | pop                       | DI          |
| 60     | 0110 0000 |             |                  | pusha                     |             |
| 61     | 0110 0001 |             |                  | popa                      |             |
| 62     | 0110 0010 | mod reg r/m |                  | bound                     | reg16,mem16 |
| 63     | 0110 0011 |             |                  | —                         |             |
| 64     | 0110 0100 |             |                  | —                         |             |
| 65     | 0110 0101 |             |                  | —                         |             |
| 66     | 0110 0110 |             |                  | —                         |             |
| 67     | 0110 0111 |             |                  | —                         |             |
| 68     | 0110 1000 | data-lo     | data-hi          | push                      | immed16     |
| 69     | 0110 1001 | mod reg r/m | data-lo, data-hi | imul                      | immed16     |
| 70     | 0111 0000 | IP-inc-8    |                  | jo                        | short-label |
| 71     | 0111 0001 | IP-inc-8    |                  | jno                       | short-label |
| 72     | 0111 0010 | IP-inc-8    |                  | jb/jnae/jc                | short-label |
| 73     | 0111 0011 | IP-inc-8    |                  | jnb/jae/jnc               | short-label |
| 74     | 0111 0100 | IP-inc-8    |                  | je/jz                     | short-label |
| 75     | 0111 0101 | IP-inc-8    |                  | jne/jnz                   | short-label |
| 76     | 0111 0110 | IP-inc-8    |                  | jbe/jna                   | short-label |
| 77     | 0111 0111 | IP-inc-8    |                  | jnb/ja                    | short-label |
| 78     | 0111 1000 | IP-inc-8    |                  | js                        | short-label |
| 79     | 0111 1001 | IP-inc-8    |                  | jns                       | short-label |
| 7A     | 0111 1010 | IP-inc-8    |                  | jp/jpe                    | short-label |
| 7B     | 0111 1011 | IP-inc-8    |                  | jnp/jpo                   | short-label |
| 7C     | 0111 1100 | IP-inc-8    |                  | jl/jnge                   | short-label |
| 7D     | 0111 1101 | IP-inc-8    |                  | jnl/jge                   | short-label |



INSTRUCTION SET OPCODES AND CLOCK CYCLES

Table D-3. Machine Instruction Decoding Guide (Continued)

| Byte 1 |           | Byte 2      | Bytes 3–6                            | ASM-86 Instruction Format |                     |
|--------|-----------|-------------|--------------------------------------|---------------------------|---------------------|
| Hex    | Binary    |             |                                      |                           |                     |
| 7E     | 0111 1110 | IP-inc-8    |                                      | jle/jng                   | short-label         |
| 7F     | 0111 1111 | IP-inc-8    |                                      | jnl/jg                    | short-label         |
| 80     | 1000 0000 | mod 000 r/m | (disp-lo),(disp-hi), data-8          | add                       | reg8/mem8,immed8    |
|        |           | mod 001 r/m | (disp-lo),(disp-hi), data-8          | or                        | reg8/mem8,immed8    |
|        |           | mod 010 r/m | (disp-lo),(disp-hi), data-8          | adc                       | reg8/mem8,immed8    |
|        |           | mod 011 r/m | (disp-lo),(disp-hi), data-8          | sbb                       | reg8/mem8,immed8    |
|        |           | mod 100 r/m | (disp-lo),(disp-hi), data-8          | and                       | reg8/mem8,immed8    |
|        |           | mod 101 r/m | (disp-lo),(disp-hi), data-8          | sub                       | reg8/mem8,immed8    |
|        |           | mod 110 r/m | (disp-lo),(disp-hi), data-8          | xor                       | reg8/mem8,immed8    |
|        |           | mod 111 r/m | (disp-lo),(disp-hi), data-8          | cmp                       | reg8/mem8,immed8    |
| 81     | 1000 0001 | mod 000 r/m | (disp-lo),(disp-hi), data-lo,data-hi | add                       | reg16/mem16,immed16 |
|        |           | mod 001 r/m | (disp-lo),(disp-hi), data-lo,data-hi | or                        | reg16/mem16,immed16 |
|        |           | mod 010 r/m | (disp-lo),(disp-hi), data-lo,data-hi | adc                       | reg16/mem16,immed16 |
|        |           | mod 011 r/m | (disp-lo),(disp-hi), data-lo,data-hi | sbb                       | reg16/mem16,immed16 |
|        |           | mod 100 r/m | (disp-lo),(disp-hi), data-lo,data-hi | and                       | reg16/mem16,immed16 |
| 81     | 1000 0001 | mod 101 r/m | (disp-lo),(disp-hi), data-lo,data-hi | sub                       | reg16/mem16,immed16 |
|        |           | mod 110 r/m | (disp-lo),(disp-hi), data-lo,data-hi | xor                       | reg16/mem16,immed16 |
|        |           | mod 111 r/m | (disp-lo),(disp-hi), data-lo,data-hi | cmp                       | reg16/mem16,immed16 |
| 82     | 1000 0010 | mod 000 r/m | (disp-lo),(disp-hi), data-8          | add                       | reg8/mem8,immed8    |
|        |           | mod 001 r/m |                                      | —                         |                     |
|        |           | mod 010 r/m | (disp-lo),(disp-hi), data-8          | adc                       | reg8/mem8,immed8    |
|        |           | mod 011 r/m | (disp-lo),(disp-hi), data-8          | sbb                       | reg8/mem8,immed8    |
|        |           | mod 100 r/m |                                      | —                         |                     |
|        |           | mod 101 r/m | (disp-lo),(disp-hi), data-8          | sub                       | reg8/mem8,immed8    |
|        |           | mod 110 r/m |                                      | —                         |                     |
|        |           | mod 111 r/m | (disp-lo),(disp-hi), data-8          | cmp                       | reg8/mem8,immed8    |
| 83     | 1000 0011 | mod 000 r/m | (disp-lo),(disp-hi), data-SX         | add                       | reg16/mem16,immed8  |
|        |           | mod 001 r/m |                                      | —                         |                     |
|        |           | mod 010 r/m | (disp-lo),(disp-hi), data-SX         | adc                       | reg16/mem16,immed8  |
|        |           | mod 011 r/m | (disp-lo),(disp-hi), data-SX         | sbb                       | reg16/mem16,immed8  |
|        |           | mod 100 r/m |                                      | —                         |                     |
|        |           | mod 101 r/m | (disp-lo),(disp-hi), data-SX         | sub                       | reg16/mem16,immed8  |
|        |           | mod 110 r/m |                                      | —                         |                     |
|        |           | mod 111 r/m | (disp-lo),(disp-hi), data-SX         | cmp                       | reg16/mem16,immed8  |
| 84     | 1000 0100 | mod reg r/m | (disp-lo),(disp-hi)                  | test                      | reg8/mem8,reg8      |
| 85     | 1000 0101 | mod reg r/m | (disp-lo),(disp-hi)                  | test                      | reg16/mem16,reg16   |
| 86     | 1000 0110 | mod reg r/m | (disp-lo),(disp-hi)                  | xchg                      | reg8,reg8/mem8      |

Table D-3. Machine Instruction Decoding Guide (Continued)

| Byte 1 |           | Byte 2      | Bytes 3–6             | ASM-86 Instruction Format |                      |
|--------|-----------|-------------|-----------------------|---------------------------|----------------------|
| Hex    | Binary    |             |                       |                           |                      |
| 87     | 1000 0111 | mod reg r/m | (disp-lo),(disp-hi)   | xchg                      | reg16,reg16/mem16    |
| 88     | 1000 0100 | mod reg r/m | (disp-lo),(disp-hi)   | mov                       | reg8/mem8,reg8       |
| 89     | 1000 1001 | mod reg r/m | (disp-lo),(disp-hi)   | mov                       | reg16/mem16,reg16    |
| 8A     | 1000 1010 | mod reg r/m | (disp-lo),(disp-hi)   | mov                       | reg8,reg8/mem8       |
| 8B     | 1000 1011 | mod reg r/m | (disp-lo),(disp-hi)   | mov                       | reg16,reg16/mem16    |
| 8C     | 1000 1100 | mod OSR r/m | (disp-lo),(disp-hi)   | mov                       | reg16/mem16,SEGREG   |
|        |           | mod 1 - r/m |                       | —                         |                      |
| 8D     | 1000 1101 | mod reg r/m | (disp-lo),(disp-hi)   | lea                       | reg16,mem16          |
| 8E     | 1000 1110 | mod OSR r/m | (disp-lo),(disp-hi)   | mov                       | SEGREG,reg16/mem16   |
|        |           | mod 1 - r/m |                       | —                         |                      |
| 8F     | 1000 1111 |             |                       | pop                       | mem16                |
| 90     | 1001 0000 |             |                       | nop                       | (xchg AX,AX)         |
| 91     | 1001 0001 |             |                       | xchg                      | AX,CX                |
| 92     | 1001 0010 |             |                       | xchg                      | AX,DX                |
| 93     | 1001 0011 |             |                       | xchg                      | AX,BX                |
| 94     | 1001 0100 |             |                       | xchg                      | AX,SP                |
| 95     | 1001 0101 |             |                       | xchg                      | AX,BP                |
| 96     | 1001 0110 |             |                       | xchg                      | AX,SI                |
| 97     | 1001 0111 |             |                       | xchg                      | AX,DI                |
| 98     | 1001 1000 |             |                       | cbw                       |                      |
| 99     | 1001 1001 |             |                       | cwd                       |                      |
| 9A     | 1001 1010 | disp-lo     | disp-hi,seg-lo,seg-hi | call                      | far-proc             |
| 9B     | 1001 1011 |             |                       | wait                      |                      |
| 9C     | 1001 1100 |             |                       | pushf                     |                      |
| 9D     | 1001 1101 |             |                       | popf                      |                      |
| 9E     | 1001 1110 |             |                       | sahf                      |                      |
| 9F     | 1001 1111 |             |                       | lahf                      |                      |
| A0     | 1010 0000 | addr-lo     | addr-hi               | mov                       | AL,mem8              |
| A1     | 1010 0001 | addr-lo     | addr-hi               | mov                       | AX,mem16             |
| A2     | 1010 0010 | addr-lo     | addr-hi               | mov                       | mem8,AL              |
| A3     | 1010 0011 | addr-lo     | addr-hi               | mov                       | mem16,AL             |
| A4     | 1010 0100 |             |                       | movs                      | dest-str8,src-str8   |
| A5     | 1010 0101 |             |                       | movs                      | dest-str16,src-str16 |
| A6     | 1010 0110 |             |                       | cmps                      | dest-str8,src-str8   |
| A7     | 1010 0111 |             |                       | cmps                      | dest-str16,src-str16 |
| A8     | 1010 1000 | data-8      |                       | test                      | AL,immed8            |
| A9     | 1010 1001 | data-lo     | data-hi               | test                      | AX,immed16           |





## INSTRUCTION SET OPCODES AND CLOCK CYCLES

**Table D-3. Machine Instruction Decoding Guide (Continued)**

| Byte 1 |           | Byte 2      | Bytes 3–6 | ASM-86 Instruction Format |                     |
|--------|-----------|-------------|-----------|---------------------------|---------------------|
| Hex    | Binary    |             |           |                           |                     |
| AA     | 1010 1010 |             |           | stos                      | dest-str8           |
| AB     | 1010 1011 |             |           | stos                      | dest-str16          |
| AC     | 1010 1100 |             |           | lods                      | src-str8            |
| AD     | 1010 1101 |             |           | lods                      | src-str16           |
| AE     | 1010 1110 |             |           | scas                      | dest-str8           |
| AF     | 1010 1111 |             |           | scas                      | dest-str16          |
| B0     | 1011 0000 | data-8      |           | mov                       | AL,immed8           |
| B1     | 1011 0001 | data-8      |           | mov                       | CL,immed8           |
| B2     | 1011 0010 | data-8      |           | mov                       | DL,immed8           |
| B3     | 1011 0011 | data-8      |           | mov                       | BL,immed8           |
| B4     | 1011 0100 | data-8      |           | mov                       | AH,immed8           |
| B5     | 1011 0101 | data-8      |           | mov                       | CH,immed8           |
| B6     | 1011 0110 | data-8      |           | mov                       | DH,immed8           |
| B7     | 1011 0111 | data-8      |           | mov                       | BH,immed8           |
| B8     | 1011 1000 | data-lo     | data-hi   | mov                       | AX,immed16          |
| B9     | 1011 1001 | data-lo     | data-hi   | mov                       | CX,immed16          |
| BA     | 1011 1010 | data-lo     | data-hi   | mov                       | DX,immed16          |
| BB     | 1011 1011 | data-lo     | data-hi   | mov                       | BX,immed16          |
| BC     | 1011 1100 | data-lo     | data-hi   | mov                       | SP,immed16          |
| BD     | 1011 1101 | data-lo     | data-hi   | mov                       | BP,immed16          |
| BE     | 1011 1110 | data-lo     | data-hi   | mov                       | SI,immed16          |
| BF     | 1011 1111 | data-lo     | data-hi   | mov                       | DI,immed16          |
| C0     | 1100 0000 | mod 000 r/m | data-8    | rol                       | reg8/mem8, immed8   |
|        |           | mod 001 r/m | data-8    | ror                       | reg8/mem8, immed8   |
|        |           | mod 010 r/m | data-8    | rcl                       | reg8/mem8, immed8   |
|        |           | mod 011 r/m | data-8    | rcr                       | reg8/mem8, immed8   |
|        |           | mod 100 r/m | data-8    | shl/sal                   | reg8/mem8, immed8   |
|        |           | mod 101 r/m | data-8    | shr                       | reg8/mem8, immed8   |
|        |           | mod 110 r/m |           | —                         |                     |
|        |           | mod 111 r/m | data-8    | sar                       | reg8/mem8, immed8   |
| C1     | 1100 0001 | mod 000 r/m | data-8    | rol                       | reg16/mem16, immed8 |
|        |           | mod 001 r/m | data-8    | ror                       | reg16/mem16, immed8 |
|        |           | mod 010 r/m | data-8    | rcl                       | reg16/mem16, immed8 |
|        |           | mod 011 r/m | data-8    | rcr                       | reg16/mem16, immed8 |
|        |           | mod 100 r/m | data-8    | shl/sal                   | reg16/mem16, immed8 |
|        |           | mod 101 r/m | data-8    | shr                       | reg16/mem16, immed8 |
|        |           | mod 110 r/m |           | —                         |                     |

Table D-3. Machine Instruction Decoding Guide (Continued)

| Byte 1 |           | Byte 2             | Bytes 3–6                           | ASM-86 Instruction Format |                         |
|--------|-----------|--------------------|-------------------------------------|---------------------------|-------------------------|
| Hex    | Binary    |                    |                                     |                           |                         |
|        |           | mod 111 <i>r/m</i> | data-8                              | sar                       | reg16/mem16, immed8     |
| C2     | 1100 0010 | data-lo            | data-hi                             | ret                       | immed16 (intra-segment) |
| C3     | 1100 0011 |                    |                                     | ret                       | (intra-segment)         |
| C4     | 1100 0100 | mod reg <i>r/m</i> | (disp-lo),(disp-hi)                 | les                       | reg16,mem16             |
| C5     | 1100 0101 | mod reg <i>r/m</i> | (disp-lo),(disp-hi)                 | lds                       | reg16,mem16             |
| C6     | 1100 0110 | mod 000 <i>r/m</i> | (disp-lo),(disp-hi),data-8          | mov                       | mem8,immed8             |
|        |           | mod 001 <i>r/m</i> |                                     | —                         |                         |
|        |           | mod 010 <i>r/m</i> |                                     | —                         |                         |
|        |           | mod 011 <i>r/m</i> |                                     | —                         |                         |
|        |           | mod 100 <i>r/m</i> |                                     | —                         |                         |
|        |           | mod 101 <i>r/m</i> |                                     | —                         |                         |
|        |           | mod 110 <i>r/m</i> |                                     | —                         |                         |
| C6     | 1100 0110 | mod 111 <i>r/m</i> |                                     | —                         |                         |
| C7     | 1100 0111 | mod 000 <i>r/m</i> | (disp-lo),(disp-hi),data-lo,data-hi | mov                       | mem16,immed16           |
|        |           | mod 001 <i>r/m</i> |                                     | —                         |                         |
|        |           | mod 010 <i>r/m</i> |                                     | —                         |                         |
|        |           | mod 011 <i>r/m</i> |                                     | —                         |                         |
|        |           | mod 100 <i>r/m</i> |                                     | —                         |                         |
|        |           | mod 101 <i>r/m</i> |                                     | —                         |                         |
|        |           | mod 110 <i>r/m</i> |                                     | —                         |                         |
|        |           | mod 111 <i>r/m</i> |                                     | —                         |                         |
| C8     | 1100 1000 | data-lo            | data-hi, level                      | enter                     | immed16, immed8         |
| C9     | 1100 1001 |                    |                                     | leave                     |                         |
| CA     | 1100 1010 | data-lo            | data-hi                             | ret                       | immed16 (inter-segment) |
| CB     | 1100 1011 |                    |                                     | ret                       | (inter-segment)         |
| 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 <i>r/m</i> | (disp-lo),(disp-hi)                 | rol                       | reg8/mem8,1             |
|        |           | mod 001 <i>r/m</i> | (disp-lo),(disp-hi)                 | ror                       | reg8/mem8,1             |
|        |           | mod 010 <i>r/m</i> | (disp-lo),(disp-hi)                 | rcl                       | reg8/mem8,1             |
|        |           | mod 011 <i>r/m</i> | (disp-lo),(disp-hi)                 | rcr                       | reg8/mem8,1             |
|        |           | mod 100 <i>r/m</i> | (disp-lo),(disp-hi)                 | sal/shl                   | reg8/mem8,1             |
|        |           | mod 101 <i>r/m</i> | (disp-lo),(disp-hi)                 | shr                       | reg8/mem8,1             |
|        |           | mod 110 <i>r/m</i> |                                     | —                         |                         |
|        |           | mod 111 <i>r/m</i> | (disp-lo),(disp-hi)                 | sar                       | reg8/mem8,1             |



INSTRUCTION SET OPCODES AND CLOCK CYCLES

Table D-3. Machine Instruction Decoding Guide (Continued)

| Byte 1 |           | Byte 2      | Bytes 3–6           | ASM-86 Instruction Format |                |
|--------|-----------|-------------|---------------------|---------------------------|----------------|
| Hex    | Binary    |             |                     |                           |                |
| D1     | 1101 0001 | mod 000 r/m | (disp-lo),(disp-hi) | rol                       | reg16/mem16,1  |
|        |           | mod 001 r/m | (disp-lo),(disp-hi) | ror                       | reg16/mem16,1  |
| D1     | 1101 0001 | mod 010 r/m | (disp-lo),(disp-hi) | rcl                       | reg16/mem16,1  |
|        |           | mod 011 r/m | (disp-lo),(disp-hi) | rcr                       | reg16/mem16,1  |
|        |           | mod 100 r/m | (disp-lo),(disp-hi) | sal/shl                   | reg16/mem16,1  |
|        |           | mod 101 r/m | (disp-lo),(disp-hi) | shr                       | reg16/mem16,1  |
|        |           | mod 110 r/m |                     | —                         |                |
|        |           | mod 111 r/m | (disp-lo),(disp-hi) | sar                       | reg16/mem16,1  |
| D2     | 1101 0010 | mod 000 r/m | (disp-lo),(disp-hi) | rol                       | reg8/mem8,CL   |
|        |           | mod 001 r/m | (disp-lo),(disp-hi) | ror                       | reg8/mem8,CL   |
|        |           | mod 010 r/m | (disp-lo),(disp-hi) | rcl                       | reg8/mem8,CL   |
|        |           | mod 011 r/m | (disp-lo),(disp-hi) | rcr                       | reg8/mem8,CL   |
|        |           | mod 100 r/m | (disp-lo),(disp-hi) | sal/shl                   | reg8/mem8,CL   |
|        |           | mod 101 r/m | (disp-lo),(disp-hi) | shr                       | reg8/mem8,CL   |
|        |           | mod 110 r/m |                     | —                         |                |
|        |           | mod 111 r/m | (disp-lo),(disp-hi) | sar                       | reg8/mem8,CL   |
| D3     | 1101 0011 | mod 000 r/m | (disp-lo),(disp-hi) | rol                       | reg16/mem16,CL |
|        |           | mod 001 r/m | (disp-lo),(disp-hi) | ror                       | reg16/mem16,CL |
|        |           | mod 010 r/m | (disp-lo),(disp-hi) | rcl                       | reg16/mem16,CL |
|        |           | mod 011 r/m | (disp-lo),(disp-hi) | rcr                       | reg16/mem16,CL |
|        |           | mod 100 r/m | (disp-lo),(disp-hi) | sal/shl                   | reg16/mem16,CL |
|        |           | mod 101 r/m | (disp-lo),(disp-hi) | shr                       | reg16/mem16,CL |
|        |           | mod 110 r/m |                     | —                         |                |
|        |           | mod 111 r/m | (disp-lo),(disp-hi) | sar                       | reg16/mem16,CL |
| D4     | 1101 0100 | 0000 1010   |                     | aam                       |                |
| D5     | 1101 0101 | 0000 1010   |                     | aad                       |                |
| D6     | 1101 0110 |             |                     | —                         |                |
| D7     | 1101 0111 |             |                     | xlat                      | source-table   |
| D8     | 1101 1000 | mod 000 r/m | (disp-lo),(disp-hi) | esc                       | opcode,source  |
| D9     | 1101 1001 | mod 001 r/m | (disp-lo),(disp-hi) | esc                       | opcode,source  |
| DA     | 1101 1010 | mod 010 r/m | (disp-lo),(disp-hi) | esc                       | opcode,source  |
| DB     | 1101 1011 | mod 011 r/m | (disp-lo),(disp-hi) | esc                       | opcode,source  |
| DC     | 1101 1100 | mod 100 r/m | (disp-lo),(disp-hi) | esc                       | opcode,source  |
| DD     | 1101 1101 | mod 101 r/m | (disp-lo),(disp-hi) | esc                       | opcode,source  |
| DE     | 1101 1110 | mod 110 r/m | (disp-lo),(disp-hi) | esc                       | opcode,source  |
| DF     | 1101 1111 | mod 111 r/m | (disp-lo),(disp-hi) | esc                       | opcode,source  |
| E0     | 1110 0000 | IP-inc-8    |                     | loopne/loopnz             | short-label    |

Table D-3. Machine Instruction Decoding Guide (Continued)

| Byte 1 |           | Byte 2      | Bytes 3–6                           | ASM-86 Instruction Format |                     |
|--------|-----------|-------------|-------------------------------------|---------------------------|---------------------|
| Hex    | Binary    |             |                                     |                           |                     |
| E1     | 1110 0001 | IP-inc-8    |                                     | loope/loopz               | short-label         |
| E2     | 1110 0010 | IP-inc-8    |                                     | loop                      | short-label         |
| E3     | 1110 0011 | IP-inc-8    |                                     | jcxz                      | short-label         |
| E4     | 1110 0100 | data-8      |                                     | in                        | AL,immed8           |
| E5     | 1110 0101 | data-8      |                                     | in                        | AX,immed8           |
| E6     | 1110 0110 | data-8      |                                     | out                       | AL,immed8           |
| E7     | 1110 0111 | data-8      |                                     | out                       | AX,immed8           |
| E8     | 1110 1000 | IP-inc-lo   | IP-inc-hi                           | call                      | near-proc           |
| E9     | 1110 1001 | IP-inc-lo   | IP-inc-hi                           | jmp                       | near-label          |
| EA     | 1110 1010 | IP-lo       | IP-hi,CS-lo,CS-hi                   | jmp                       | far-label           |
| EB     | 1110 1011 | IP-inc-8    |                                     | jmp                       | short-label         |
| EC     | 1110 1100 |             |                                     | in                        | AL,DX               |
| ED     | 1110 1101 |             |                                     | in                        | AX,DX               |
| EE     | 1110 1110 |             |                                     | out                       | AL,DX               |
| EF     | 1110 1111 |             |                                     | out                       | AX,DX               |
| F0     | 1111 0000 |             |                                     | lock                      | (prefix)            |
| F1     | 1111 0001 |             |                                     | —                         |                     |
| F2     | 1111 0010 |             |                                     | repne/repnz               |                     |
| F3     | 1111 0011 |             |                                     | rep/repe/repz             |                     |
| F4     | 1111 0100 |             |                                     | hlt                       |                     |
| F5     | 1111 0101 |             |                                     | cmc                       |                     |
| F6     | 1111 0110 | mod 000 r/m | (disp-lo),(disp-hi),data-8          | test                      | reg8/mem8,immed8    |
|        |           | mod 001 r/m |                                     | —                         |                     |
|        |           | mod 010 r/m | (disp-lo),(disp-hi)                 | not                       | reg8/mem8           |
|        |           | mod 011 r/m | (disp-lo),(disp-hi)                 | neg                       | reg8/mem8           |
|        |           | mod 100 r/m | (disp-lo),(disp-hi)                 | mul                       | reg8/mem8           |
|        |           | mod 101 r/m | (disp-lo),(disp-hi)                 | imul                      | reg8/mem8           |
|        |           | mod 110 r/m | (disp-lo),(disp-hi)                 | div                       | reg8/mem8           |
|        |           | mod 111 r/m | (disp-lo),(disp-hi)                 | idiv                      | reg8/mem8           |
| F7     | 1111 0111 | mod 000 r/m | (disp-lo),(disp-hi),data-lo,data-hi | test                      | reg16/mem16,immed16 |
|        |           | mod 001 r/m |                                     | —                         |                     |
|        |           | mod 010 r/m | (disp-lo),(disp-hi)                 | not                       | reg16/mem16         |
|        |           | mod 011 r/m | (disp-lo),(disp-hi)                 | neg                       | reg16/mem16         |
|        |           | mod 100 r/m | (disp-lo),(disp-hi)                 | mul                       | reg16/mem16         |
|        |           | mod 101 r/m | (disp-lo),(disp-hi)                 | imul                      | reg16/mem16         |
|        |           | mod 110 r/m | (disp-lo),(disp-hi)                 | div                       | reg16/mem16         |
|        |           | mod 111 r/m | (disp-lo),(disp-hi)                 | idiv                      | reg16/mem16         |



## INSTRUCTION SET OPCODES AND CLOCK CYCLES

**Table D-3. Machine Instruction Decoding Guide (Continued)**

| Byte 1 |           | Byte 2      | Bytes 3–6           | ASM-86 Instruction Format |                             |
|--------|-----------|-------------|---------------------|---------------------------|-----------------------------|
| Hex    | Binary    |             |                     |                           |                             |
| F8     | 1111 1000 |             |                     | clc                       |                             |
| F9     | 1111 1001 |             |                     | stc                       |                             |
| FA     | 1111 1010 |             |                     | cli                       |                             |
| FB     | 1111 1011 |             |                     | sti                       |                             |
| FC     | 1111 1100 |             |                     | cld                       |                             |
| FD     | 1111 1101 |             |                     | std                       |                             |
| FE     | 1111 1110 | mod 000 r/m | (disp-lo),(disp-hi) | inc                       | mem16                       |
|        |           | mod 001 r/m | (disp-lo),(disp-hi) | dec                       | mem16                       |
|        |           | mod 010 r/m |                     | —                         |                             |
| FE     | 1111 1110 | mod 011 r/m |                     | —                         |                             |
|        |           | mod 100 r/m |                     | —                         |                             |
|        |           | mod 101 r/m |                     | —                         |                             |
|        |           | mod 110 r/m |                     | —                         |                             |
|        |           | mod 111 r/m |                     | —                         |                             |
| FF     | 1111 1111 | mod 000 r/m | (disp-lo),(disp-hi) | inc                       | mem16                       |
|        |           | mod 001 r/m | (disp-lo),(disp-hi) | dec                       | mem16                       |
|        |           | mod 010 r/m | (disp-lo),(disp-hi) | call                      | reg16/mem16 (intra-segment) |
|        |           | mod 011 r/m | (disp-lo),(disp-hi) | call                      | mem16 (inter-segment)       |
|        |           | mod 100 r/m | (disp-lo),(disp-hi) | jmp                       | reg16/mem16 (intra-segment) |
|        |           | mod 101 r/m | (disp-lo),(disp-hi) | jmp                       | mem16 (inter-segment)       |
|        |           | mod 110 r/m | (disp-lo),(disp-hi) | push                      | mem16                       |
|        |           | mod 111 r/m |                     | —                         |                             |





**Table D-4. Mnemonic Encoding Matrix (Left Half)**

|           | <b>x0</b>      | <b>x1</b>      | <b>x2</b>      | <b>x3</b>      | <b>x4</b>   | <b>x5</b>   | <b>x6</b>  | <b>x7</b>  |
|-----------|----------------|----------------|----------------|----------------|-------------|-------------|------------|------------|
| <b>0x</b> | ADD            | ADD            | ADD            | ADD            | ADD         | ADD         | PUSH       | POP        |
|           | b,f,r/m<br>ADC | w,f,r/m<br>ADC | b,t,r/m<br>ADC | w,t,r/m<br>ADC | b,ia<br>ADC | w,ia<br>ADC | ES<br>PUSH | ES<br>POP  |
| <b>1x</b> | b,f,r/m<br>AND | w,f,r/m<br>AND | b,t,r/m<br>AND | w,t,r/m<br>AND | b,i<br>AND  | w,i<br>AND  | SS<br>SEG  | SS<br>DAA  |
|           | b,f,r/m<br>XOR | w,f,r/m<br>XOR | b,t,r/m<br>XOR | w,t,r/m<br>XOR | b,i<br>XOR  | w,i<br>XOR  | =ES<br>SEG | AAA        |
| <b>3x</b> | b,f,r/m<br>INC | w,f,r/m<br>INC | b,t,r/m<br>INC | w,t,r/m<br>INC | b,i<br>INC  | w,i<br>INC  | =SS<br>INC | INC        |
|           | AX<br>PUSH     | CX<br>PUSH     | DX<br>PUSH     | BX<br>PUSH     | SP<br>PUSH  | BP<br>PUSH  | SI<br>PUSH | DI<br>PUSH |
| <b>5x</b> | AX<br>PUSHA    | CX<br>POPA     | DX<br>BOUND    | BX             | SP          | BP          | SI         | DI         |
|           | <b>6x</b>      |                | w,f,r/m        |                |             |             |            |            |





## INSTRUCTION SET OPCODES AND CLOCK CYCLES

**Table D-4. Mnemonic Encoding Matrix (Right Half)**

| x8             | x9             | xA                | xB                 | xC                | xD               | xE               | xF               |           |
|----------------|----------------|-------------------|--------------------|-------------------|------------------|------------------|------------------|-----------|
| OR             | OR             | OR                | OR                 | OR                | OR               | PUSH             |                  | <b>0x</b> |
| b,f,r/m<br>SBB | w,f,r/m<br>SBB | b,t,r/m<br>SBB    | w,t,r/m<br>SBB     | b,i<br>SBB        | w,i<br>SBB       | CS<br>PUSH       | POP              | <b>1x</b> |
| b,f,r/m<br>SUB | w,f,r/m<br>SUB | b,t,r/m<br>SUB    | w,t,r/m<br>SUB     | b,i<br>SUB        | w,i<br>SUB       | DS<br>SEG        | DS<br>DAS        | <b>2x</b> |
| b,f,r/m<br>CMP | w,f,r/m<br>CMP | b,t,r/m<br>CMP    | w,t,r/m<br>CMP     | b,i<br>CMP        | w,i<br>CMP       | =CS<br>SEG       | AAS              | <b>3x</b> |
| b,f,r/m<br>DEC | w,f,r/m<br>DEC | b,t,r/m<br>DEC    | w,t,r/m<br>DEC     | b,i<br>DEC        | w,i<br>DEC       | =DS<br>DEC       | DEC              | <b>4x</b> |
| AX<br>POP      | CX<br>POP      | DX<br>POP         | BX<br>POP          | SP<br>POP         | BP<br>POP        | SI<br>POP        | DI<br>POP        | <b>5x</b> |
| AX<br>PUSH     | CX<br>IMUL     | DX<br>PUSH        | BX<br>IMUL         | SP<br>INS         | BP<br>INS        | SI<br>OUTS       | DI<br>OUTS       | <b>6x</b> |
| w,i<br>JS      | w,i<br>JNS     | b,i<br>JP/<br>JPE | w,i<br>JNP/<br>JPO | b<br>JL/<br>JNGE  | w<br>JNL/<br>JGE | b<br>JLE/<br>JNG | w<br>JNLE/<br>JG | <b>7x</b> |
| MOV            | MOV            | MOV               | MOV                | MOV               | LEA              | MOV              | POP              | <b>8x</b> |
| b,f,r/m<br>CBW | w,f,r/m<br>CWD | b,t,r/m<br>CALL   | w,t,r/m<br>WAIT    | sr,f,r/m<br>PUSHF | POPF             | sr,t,r/m<br>SAHF | r/m<br>LAHF      | <b>9x</b> |
| TEST           | TEST           | L,D<br>STOS       | STOS               | LODS              | LODS             | SCAS             | SCAS             | <b>Ax</b> |
| b,ia<br>MOV    | w,ia<br>MOV    | MOV               | MOV                | MOV               | MOV              | MOV              | MOV              | <b>Bx</b> |
| i→AX<br>ENTER  | i→CX<br>LEAVE  | i→DX<br>RET       | i→BX<br>RET        | i→SP<br>INT       | i→BP<br>INT      | i→SI<br>INTO     | i→DI<br>IRET     | <b>Cx</b> |
| ESC            | ESC            | I(i+SP)<br>ESC    | I<br>ESC           | type 3<br>ESC     | (any)<br>ESC     | ESC              | ESC              | <b>Dx</b> |
| 0<br>CALL      | 1<br>JMP       | 2<br>JMP          | 3<br>JMP           | 4<br>IN           | 5<br>IN          | 6<br>OUT         | 7<br>OUT         | <b>Ex</b> |
| CLC            | STC            | CLI               | STI                | CLS               | STD              | Grp2<br>b,r/m    | Grp2<br>w,r/m    | <b>Fx</b> |

**NOTE:** Table D-5 defines abbreviations used in this matrix. Shading indicates reserved opcodes.

Table D-5. Abbreviations for Mnemonic Encoding Matrix

| Abbr | Definition        | Abbr | Definition                    | Abbr | Definition         | Abbr | Definition      |
|------|-------------------|------|-------------------------------|------|--------------------|------|-----------------|
| b    | byte operation    | ia   | immediate to accumulator      | m    | memory             | t    | to CPU register |
| d    | direct            | id   | indirect                      | r/m  | EA is second byte  | v    | variable        |
| f    | from CPU register | is   | immediate byte, sign extended | si   | short intrasegment | w    | word operation  |
| i    | immediate         | l    | long (intersegment)           | sr   | segment register   | z    | zero            |

| Byte 2      | Immed | Shift   | Grp1 | Grp2       |
|-------------|-------|---------|------|------------|
| mod 000 r/m | ADD   | ROL     | TEST | INC        |
| mod 001 r/m | OR    | ROR     | —    | DEC        |
| mod 010 r/m | ADC   | RCL     | NOT  | CALL id    |
| mod 011 r/m | SBB   | RCR     | NEG  | CALL I, id |
| mod 100 r/m | AND   | SHL/SAL | MUL  | JMP id     |
| mod 101 r/m | SUB   | SHR     | IMUL | JMP i, id  |
| mod 110 r/m | XOR   | —       | DIV  | PUSH       |
| mod 111 r/m | CMP   | SAR     | IDIV | —          |

*mod* and *r/m* determine the Effective Address (EA) calculation. See Table D-1 for definitions.





# Index

I





## INDEX

80C187 Math Coprocessor, 11-2–11-8  
  accessing, 11-10–11-11  
  arithmetic instructions, 11-3–11-4  
  bus cycles, 11-11  
  clocking, 11-10  
  code examples, 11-13–11-16  
  comparison instructions, 11-5  
  constant instructions, 11-6  
  data transfer instructions, 11-3  
  data types, 11-7–11-8  
  design considerations, 11-10–11-11  
  example floating point routine, 11-16  
  exceptions, 11-13  
  I/O port assignments, 11-10  
  initialization example, 11-13–11-16  
  instruction set, 11-2  
  interface, 11-7–11-13  
    and chip-selects, 6-17, 11-11  
    and PCB location, 4-7  
    exception trapping, 11-13  
    generating READY, 11-11  
  processor control instructions, 11-6  
  testing for presence, 11-10  
  transcendental instructions, 11-5

8259A Programmable Interrupt Controllers, 8-1  
  and special fully nested mode, 8-8  
  cascading, 8-7, 8-8  
  interrupt type, 8-9  
  priority structure, 8-8

82C59A Programmable Interrupt Controller  
  interfacing with, 3-25–3-27

### A

Address and data bus, 3-1–3-6  
  16-bit, 3-1–3-5  
    considerations, 3-7  
  8-bit, 3-5–3-6  
    considerations, 3-7  
  *See also Bus cycles, Data transfers*

Address bus, *See Address and data bus*

Address space, *See Memory space, I/O space*

Addressing modes, 2-27–2-36  
  and string instructions, 2-34  
  based, 2-30, 2-31, 2-32

  based index, 2-34, 2-35  
  direct, 2-29  
  immediate operands, 2-28  
  indexed, 2-32, 2-33  
  indirect, 2-36  
  memory operands, 2-28  
  register indirect, 2-30, 2-31  
  register operands, 2-27

AH register, 2-5

AL register, 2-5, 2-18, 2-23

ApBUILDER software, downloading, 1-5

Application BBS, 1-5

Architecture  
  CPU block diagram, 2-2  
  device feature comparisons, 1-2  
  family introduction, 1-1  
  overview, 1-1, 2-1

ARDY, *See READY*

Arithmetic  
  instructions, 2-19–2-20  
  interpretation of 8-bit numbers, 2-20

Arithmetic Logic Unit (ALU), 2-1

Array bounds trap (Type 5 exception), 2-44

ASCII, defined, 2-37

Auxiliary Flag (AF), 2-7, 2-9

AX register, 2-1, 2-5, 2-18, 2-23, 3-6

### B

Base Pointer (BP), *See BP register*

BBS, 1-5

BCD, defined, 2-37

Bit manipulation instructions, 2-21–2-22

BOUND instruction, 2-44, A-8

BP register, 2-1, 2-13, 2-30, 2-34

Breakpoint interrupt (Type 3 exception), 2-44

Bulletin board system (BBS), 1-5

Bus cycles, 3-20–3-47  
  address/status phase, 3-10–3-12  
  and 80C187, 11-11  
  and CSU, 6-17  
  and Idle mode, 5-13  
  and PCB accesses, 4-4  
  and Powerdown mode, 5-16  
  and T-states, 3-9

- data phase, 3-13
  - HALT cycle, 3-28–3-35
    - and chip-selects, 6-5
  - HALT state, exiting, 3-31–3-35
  - idle states, 3-18
  - instruction prefetch, 3-20
  - interrupt acknowledge (INTA) cycles, 3-6, 3-25–3-26, 8-9
    - and chip-selects, 6-5
  - interrupt acknowledge cycles, 8-29
  - operation, 3-7–3-20
  - priorities, 3-46–3-47, 7-2
  - read cycles, 3-20–3-21
  - refresh cycles, 3-22, 7-4, 7-5
    - control signals, 7-5, 7-6
    - during HOLD, 3-43–3-45, 7-12–7-13
  - wait states, 3-13–3-18
  - write cycles, 3-22–3-25
  - See also Data transfers*
  - Bus hold protocol, 3-41–3-46
    - and CLKOUT, 5-6
    - and CSU, 6-18
    - and Idle mode, 5-14
    - and refresh cycles, 3-43–3-45, 7-12–7-13
    - and reset, 5-9
    - latency, 3-42–3-43
  - Bus Interface Unit (BIU), 2-1, 2-3, 2-11, 3-1–3-47
    - and DMA, 10-8
    - and DRAM refresh requests, 7-4
    - and TCU, 9-1
    - buffering the data bus, 3-36–3-38
    - modifying interface, 3-35–3-38, 3-38
    - relationship to RCU, 7-1
    - synchronizing software and hardware events, 3-38–3-39
    - using a locked bus, 3-39–3-40
    - using multiple bus masters, 3-41–3-46
    - using the queue status signals, 3-40–3-41
  - BX register, 2-1, 2-5, 2-30
- C**
- Carry Flag (CF), 2-7, 2-9
  - Chip-Select Unit (CSU), 6-1
    - and DMA, 10-8
    - and DMA acknowledge signal, 10-22
    - and HALT bus cycles, 3-28
    - and READY, 6-15–6-16
    - and wait states, 6-15–6-16
  - block diagram, 6-3
  - bus cycle decoding, 6-17
  - examples, 6-18–6-22
  - features and benefits, 6-1
  - functional overview, 6-2–6-5
  - programming, 6-6–6-17
  - registers, 6-6–6-12
  - system diagram, 6-19
  - See also Chip selects*
  - Chip-selects
    - activating, 6-5
    - and 80C187 interface, 6-17, 11-11
    - and bus hold protocol, 6-18
    - and DMA acknowledge signal, 10-22
    - and DRAM controllers, 7-1
    - and reserved I/O locations, 6-17
    - initializing, 6-6–6-18
    - methods for generating, 6-1
    - overlapping, 6-16–6-17
    - programming considerations, 6-17
    - start address, 6-17
    - timing, 6-4
  - CL register, 2-5, 2-21, 2-22
  - CLKOUT
    - and bus hold, 5-6
    - and power management modes, 5-6
    - and reset, 5-6
  - Clock divider, 5-19
    - control register, 5-20
  - Clock generator, 5-6–5-10
    - and system reset, 5-6–5-7
    - output, 5-6
    - synchronizing CLKOUT and RESOUT, 5-6–5-7
  - Clock sources, TCU, 9-12
  - Code (programs), *See Software*
  - Code segment, 2-5
  - CompuServe forums, 1-6
  - Counters, *See Timer Counter Unit (TCU)*
  - CPU, block diagram, 2-2
  - Crystal, *See Oscillator*
  - CS register, 2-1, 2-5, 2-6, 2-13, 2-23, 2-39, 2-41
  - Customer service, 1-4
  - CX register, 2-1, 2-5, 2-23, 2-25, 2-26

**D**

- Data, 3-6
- Data bus, *See Address and data bus*
- Data segment, 2-5
- Data sheets, online, 1-6
- Data transfers, 3-1–3-6
  - instructions, 2-18
  - PCB considerations, 4-5
  - PSW flag storage formats, 2-19
  - See also Bus cycles*
- Data types, 2-37–2-38
- DI register, 2-1, 2-5, 2-13, 2-22, 2-23, 2-30, 2-32, 2-34
- Digital one-shot, code example, 9-17–9-23
- Direct Memory Access (DMA) Unit, 10-1–10-27
  - and BIU, 10-8
  - and CSU, 10-8
  - and PCB, 10-3
  - arming channel, 10-18
  - DMA acknowledge signal, 10-2, 10-22
  - DRQ timing, 10-20
  - examples, 10-22–10-27
  - HALT bit, 8-22, 8-23, 10-20
  - hardware considerations, 10-20–10-22
  - initialization code, 10-22–10-27
  - initializing, 10-20
  - interrupts, 10-8
    - generating on terminal count, 10-19
  - introduction, 10-1
  - latency, 10-21
  - modules, 10-8–10-9
  - overview, 10-1–10-10
  - pointers, programming, 10-10–10-14
  - priority
    - channel, 10-8–10-9, 10-19
    - fixed, 10-8–10-10
    - rotating, 10-10
  - programming, 10-17–10-20
    - arming channel, 10-18
    - channel priority, 10-19
    - initializing, 10-20
    - interrupts, 10-19
    - suspending transfers, 10-20
    - synchronization, 10-18
    - transfer count, 10-18–10-19
  - programming, pointers, 10-10–10-14
  - requests, 10-3
    - external, 10-4
    - internal, 10-6
      - software, 10-6
      - Timer 2, 10-6
    - selecting source, 10-17
  - synchronization
    - destination-synchronized, 10-5
    - selecting, 10-18
    - source-synchronized, 10-5
    - unsynchronized, 10-6
  - timed DMA transfer example, 10-22–10-27
  - transfers, 10-1–10-27
    - count, 10-7
      - programming, 10-18–10-19
    - direction, 10-3
    - rates, 10-21
    - size, 10-3
      - selecting, 10-14
    - suspending, 10-7, 10-20
    - terminating, 10-7
- Direction Flag (DF), 2-7, 2-9, 2-23
- Display, defined, A-2
- Divide Error trap (Type 0 exception), 2-43
- DMA Control Register (DxCON), 10-15
- DMA Destination Pointer Register, 10-13, 10-14
- DMA Source Pointer Register, 10-11, 10-12
- Documents, related, 1-3
- DRAM controllers
  - and wait state control, 7-5
  - clocked, 7-5
  - design guidelines, 7-5
  - unclocked, 7-5
  - See also Refresh Control Unit*
- DS register, 2-1, 2-5, 2-6, 2-13, 2-30, 2-34, 2-43
- DX register, 2-1, 2-5, 2-36, 3-6

**E**

- Effective Address (EA), 2-13
  - calculation, 2-28
- Emulation mode, 12-1
- End-of-Interrupt (EOI)
  - command, 8-21
  - register, 8-21, 8-22, 8-27, 8-28
- ENTER instruction, A-2
- ES register, 2-1, 2-5, 2-6, 2-13, 2-30, 2-34
- Escape opcode fault (Type 7 exception), 2-44, 11-2

## INDEX

- Exceptions, 2-43–2-44
  - priority, 2-46–2-49
- Execution Unit (EU), 2-1, 2-2
- Extra segment, 2-5
- F**
- Fault exceptions, 2-43
- FaxBack service, 1-4
- F-Bus
  - and PCB, 4-5
  - operation, 4-5
- Flags, *See Processor Status Word (PSW)*
- Floating Point, defined, 2-37
- H**
- HALT bus cycle, *See Bus cycles*
- HOLD/HLDA protocol, *See Bus hold protocol*
- Hypertext manuals and datasheets,
  - downloading, 1-6
- I**
- I/O devices
  - interfacing with, 3-6–3-7
  - memory-mapped, 3-6
- I/O ports
  - addressing, 2-36
- I/O space, 3-1–3-7
  - accessing, 3-6
  - reserved locations, 2-15, 6-17
- Idle mode, 5-11–5-16, 5-16
  - bus operation, 5-13
  - control register, 5-12
  - entering, 5-11, 5-13
  - exiting, 5-14–5-15
  - exiting HALT bus cycle, 3-35
  - initialization code, 5-15–5-16
- Idle states
  - and bus cycles, 3-18
- Immediate operands, 2-28
- IMUL instruction, A-9
- Inputs, asynchronous, synchronizing, B-1
- INS instruction, A-2
- In-Service register, 8-5, 8-7, 8-18, 8-19, 8-28
- Instruction Pointer (IP), 2-1, 2-6, 2-13, 2-23, 2-39, 2-41
  - reset status, 2-6
- Instruction prefetch bus cycle, *See Bus cycles*
- Instruction set, 2-17, A-1, D-1
  - additions, A-1
  - arithmetic instructions, 2-19–2-20, A-9
  - bit manipulation instructions, 2-21–2-22, A-9
  - data transfer instructions, 2-18–2-20, A-1, A-8
  - data types, 2-37–2-38
  - enhancements, A-8
  - high-level instructions, A-2
  - nesting, A-2
  - processor control instructions, 2-27
  - program transfer instructions, 2-23–2-24
  - reentrant procedures, A-2
  - rotate instructions, A-10
  - shift instructions, A-9
  - string instructions, 2-22–2-23, A-2
- INT instruction, single-byte, *See Breakpoint interrupt*
- INT0 instruction, 2-44
- INTA bus cycle, *See Bus cycles*
- Integer, defined, 2-37, 11-7
- Interrupt Control register, 8-28
- Interrupt Control registers, 8-12
  - for external pins, 8-14, 8-15
  - for internal sources, 8-13
- Interrupt Control Unit (ICU), 8-1–8-31
  - block diagram, 8-2, 8-24
  - cascade mode, 8-7
  - initializing, 8-30, 8-31
  - interfacing with an 82C59A Programmable Interrupt Controller, 3-25–3-27
  - master mode, 8-1, 8-2–8-23
    - initializing, 8-30
  - operation in slave mode, 8-29
  - operation with nesting, 8-4
  - programming, 8-11, 8-25
  - registers, 8-11, 8-25
    - master mode, 8-11
  - slave mode, 8-1, 8-23–8-30
  - special fully nested mode, 8-8
    - with cascade mode, 8-8
    - without cascade mode, 8-8
  - typical interrupt sequence, 8-5
- Interrupt Enable Flag (IF), 2-7, 2-9, 2-41
- Interrupt Mask register, 8-16, 8-17, 8-28
- Interrupt Request register, 8-16, 8-28
- Interrupt Status register, 8-7, 8-22, 8-23, 8-28
- Interrupt Vector register, 8-26, 8-27

Interrupt Vector Table, 2-39, 2-40  
 Interrupt-on-overflow trap (Type 4 exception), 2-44  
 Interrupts, 2-39–2-43  
   and CSU initialization, 6-6  
   controlling priority, 8-12  
   edge- and level-sensitive, 8-10  
     and external 8259As, 8-10  
   enabling cascade mode, 8-12  
   enabling special fully nested mode, 8-12  
   latency, 2-45  
     reducing, 3-28  
   latency and response times, 8-10, 8-11, 8-30  
   maskable, 2-43  
   masking, 8-3, 8-12, 8-16  
     priority-based, 8-17  
   multiplexed, 8-7  
   nesting, 8-4  
   NMI, 2-42  
   nonmaskable, 2-45  
   overview, 8-1, 8-2  
   priority, 2-46–2-49, 8-3  
     default, 8-3  
     resolution, 8-5, 8-6  
   processing, 2-39–2-42  
   reserved, 2-39  
   response time, 2-46  
   selecting edge- or level-triggering, 8-12  
   slave mode sources, 8-25  
   software, 2-45  
   timer interrupts, 9-16  
   types, 8-9, 8-26, 8-27  
   *See also Exceptions, Interrupt Control Unit*  
 INTh instruction, 2-45  
 Invalid opcode trap (Type 6 exception), 2-44  
 IRET instruction, 2-41

**L**

Latency, *See Bus hold protocol, Direct Memory Access (DMA) Unit, Interrupts*  
 LEAVE instruction, A-7  
 Literature, ordering, 1-6  
 Local bus, 3-1, 3-41, 11-11  
 Long integer, defined, 11-7  
 Long real, defined, 11-7

**M**

Manuals, online, 1-5  
 Math coprocessing, 11-1  
   hardware support, 11-1  
   overview, 11-1  
 Memory  
   addressing, 2-28–2-36  
   operands, 2-28  
   reserved locations, 2-15  
 Memory devices, interfacing with, 3-6–3-7  
 Memory segments, 2-8  
   accessing, 2-5, 2-10, 2-11, 2-13  
   address  
     base value, 2-10, 2-11, 2-12  
     Effective Address (EA), 2-13  
     logical, 2-10, 2-12  
     offset value, 2-10, 2-13  
     overriding, 2-11, 2-13  
     physical, 2-3, 2-10, 2-12  
     and dynamic code relocation, 2-13  
 Memory space, 3-1–3-6

**N**

Normally not-ready signal, *See Ready*  
 Normally ready signal, *See Ready*  
 Numerics coprocessor fault (Type 16 exception), 2-44, 11-13

**O**

ONCE mode, 12-1  
 One-shot, code example, 9-17–9-23  
 Ordinal, defined, 2-37  
 Oscillator  
   external  
     and powerdown, 5-19  
     selecting crystal, 5-5  
     using canned, 5-6  
   internal crystal, 5-1–5-10  
     controlling gating to internal clocks, 5-18  
     operation, 5-2–5-3  
     selecting C<sub>1</sub> and L<sub>1</sub> components, 5-3–5-6  
 OUTS instruction, A-2  
 Overflow Flag (OF), 2-7, 2-9, 2-44

**P**

Packed BCD, defined, 2-37  
 Packed decimal, defined, 11-7  
 Parity Flag (PF), 2-7, 2-9  
 PCB Relocation Register, 4-1, 4-3, 4-6  
   and math coprocessing, 11-2  
 PDTMR pin, 5-18  
 Peripheral Control Block (PCB), 4-1  
   accessing, 4-4  
   and DMA Unit, 10-3  
   and F-Bus operation, 4-5  
   base address, 4-6–4-7  
   bus cycles, 4-4  
   READY signals, 4-4  
   reserved locations, 4-6  
   wait states, 4-4  
 Peripheral control registers, 4-1, 4-6  
 Pointer, defined, 2-37  
 Poll register, 8-9, 8-19, 8-20  
 Poll Status register, 8-9, 8-19, 8-20, 8-21  
 Polling, 8-1, 8-9  
 POPA instruction, A-1  
 Power consumption, reducing, 3-28, 5-23  
 Power Control Register, 5-12  
 Power management, 5-10–5-23  
 Power management modes  
   and HALT bus cycles, 3-28, 3-31  
   compared, 5-23  
 Powerdown mode, 5-16–5-19, 7-2  
   and bus cycles, 5-16  
   control register, 5-12  
   entering, 5-17  
   exiting, 5-18–5-19  
   exiting HALT bus cycle, 3-34  
   initialization code, 5-15–5-18  
 Power-Save mode, 5-19–5-22, 7-2  
   and DRAM refresh rate, 5-21  
   and refresh interval, 7-7  
   control register, 5-20  
   entering, 5-19  
   exiting, 5-21  
   initialization code, 5-21–5-22  
 Power-Save Register, 5-20  
 Priority Mask register, 8-17, 8-18, 8-28  
 Processor control instructions, 2-27  
 Processor Status Word (PSW), 2-1, 2-7, 2-41  
   bits defined, 2-7, 2-9

  flag storage formats, 2-19  
   reset status, 2-7

Program transfer instructions, 2-23–2-24  
   conditional transfers, 2-24, 2-26  
   interrupts, 2-26  
   iteration control, 2-25  
   unconditional transfers, 2-24

PUSH instruction, A-8  
 PUSHA instruction, A-1

**Q**

Queue status signals, 3-40

**R**

RCL instruction, A-10  
 RCR instruction, A-10  
 Read bus cycles, *See Bus cycles*  
 READY  
   and chip-selects, 6-15  
   and normally not-ready signal, 3-17–3-18  
   and normally ready signal, 3-16–3-17  
   and PCB accesses, 4-4  
   and wait states, 3-13–3-18  
   implementation approaches, 3-13  
   timing concerns, 3-17  
 Real, defined, 11-7  
 Real-time clock, code example, 9-17–9-20  
 Refresh address, 7-4  
 Refresh Base Address Register (RFBASE), 7-8  
 Refresh bus cycle, *See Bus cycles*  
 Refresh Clock Interval Register (RFTIME), 7-7,  
   7-8  
 Refresh Control Register (RFCON), 7-9, 7-10  
 Refresh Control Unit (RCU), 7-1–7-13  
   and bus hold protocol, 7-12–7-13  
   and Powerdown mode, 7-2  
   and Power-Save mode, 5-19, 7-2, 7-7  
   block diagram, 7-1  
   bus latency, 7-7  
   calculating refresh interval, 7-7  
   control registers, 7-7–7-10  
   initialization code, 7-10  
   operation, 7-2  
   overview, 7-2–7-4  
   programming, 7-7–7-11  
   relationship to BIU, 7-1  
 Register operands, 2-27



Registers, 2-1  
 control, 2-1  
 data, 2-4, 2-5  
 general, 2-1, 2-4, 2-5  
 H & L group, 2-4  
 index, 2-5, 2-13, 2-34  
 P & I group, 2-4  
 pointer, 2-1, 2-5, 2-13  
 pointer and index, 2-4  
 segment, 2-1, 2-5, 2-11, 2-12  
 status, 2-1

Relocation Register, *See PCB Relocation Register*

Reset  
 and bus hold protocol, 5-6  
 and clock synchronization, 5-6–5-10  
 cold, 5-7, 5-8  
 RC circuit for reset input, 5-7  
 warm, 5-7, 5-9

ROL instruction, A-10  
 ROR instruction, A-10

## S

SAL instruction, A-9  
 SAR instruction, A-9  
 SHL instruction, A-9  
 Short integer, defined, 11-7  
 Short real, defined, 11-7  
 SHR instruction, A-9  
 SI register, 2-1, 2-5, 2-13, 2-22, 2-23, 2-30, 2-32, 2-34  
 Sign Flag (SF), 2-7, 2-9  
 Single-step trap (Type 1 exception), 2-43

Software  
 code example  
 80C187 floating-point routine, 11-16  
 80C187 initialization, 11-13–11-15  
 digital one-shot, 9-17–9-23  
 DMA initialization, 10-22–10-27  
 ICU initialization, 8-31  
 real-time clock, 9-17–9-19  
 square-wave generator, 9-17–9-22  
 TCU configurations, 9-17–9-23  
 timed DMA transfers, 10-22–10-27  
 data types, 2-37, 2-38  
 dynamic code relocation, 2-13, 2-14  
 interrupts, 2-45  
 overview, 2-17

*See also Addressing modes, Instruction set*

Square-wave generator, code example, 9-17–9-22  
 SRDY, *See READY*

SS register, 2-1, 2-5, 2-6, 2-13, 2-15, 2-30, 2-45  
 Stack frame pointers, A-2  
 Stack Pointer, 2-1, 2-5, 2-13, 2-15, 2-45  
 Stack segment, 2-5  
 Stacks, 2-15  
 START registers, CSU, 6-6, 6-7  
 STOP registers, CSU, 6-6, 6-8  
 String instructions, 2-22–2-23  
 and addressing modes, 2-34  
 and memory-mapped I/O ports, 2-36  
 operand locations, 2-13  
 operands, 2-36

Strings  
 accessing, 2-13, 2-34  
 defined, 2-37

Synchronizing asynchronous inputs, B-1

## T

Technical support, 1-6  
 Temporary real, defined, 11-7  
 Terminology  
 "above" vs "greater", 2-26  
 "below" vs "less", 2-26  
 device names, 1-2

Timer Control Registers (TxCON), 9-7, 9-8  
 Timer Count Registers (TxCNT), 9-10  
 Timer Counter Unit (TCU), 9-1–9-23  
 and Power-Save mode, 5-19  
 application examples, 9-17–9-23  
 block diagram, 9-2  
 clock sources, 9-12  
 configuring a digital one-shot, 9-17–9-23  
 configuring a real-time clock, 9-17–9-19  
 configuring a square-wave generator, 9-17–9-22  
 counting sequence, 9-12–9-13  
 dual maxcount mode, 9-13–9-14  
 enabling and disabling counters, 9-15–9-16  
 frequency, maximum, 9-17  
 initializing, 9-11  
 input synchronization, 9-17  
 interrupts, 9-16  
 overview, 9-1–9-6



## INDEX

- programming, 9-6–9-16
  - considerations, 9-16
- pulsed output, 9-14–9-15
- retriggering, 9-13–9-14
- setup and hold times, 9-16
- single maxcount mode, 9-13, 9-14–9-16
- timer delay, 9-1
- timing, 9-1
  - and BIU, 9-1
  - considerations, 9-16
  - TxOUT signal, 9-15
  - variable duty cycle output, 9-14–9-15
- Timer Maxcount Compare Registers (TxCMPA, TxCMPB), 9-11
- Timers, *See Timer/Counter Unit (TCU)*
- Trap exceptions, 2-43
- Trap Flag (TF), 2-7, 2-9, 2-43, 2-48
- T-state
  - and bus cycles, 3-9
  - and CLKOUT, 3-8
  - defined, 3-7
  
- W**
- Wait states
  - and bus cycles, 3-13
  - and bus ready inputs, 3-13
  - and chip-selects, 6-15–6-17
  - and DRAM controllers, 7-1
  - and PCB accesses, 4-4
  - and READY input, 3-13
- Word integer, defined, 11-7
- World Wide Web, 1-6
- Write bus cycle, 3-22
  
- Z**
- Zero Flag (ZF), 2-7, 2-9, 2-23

