ISA System Architecture

Third Edition

MINDSHARE, INC.

TOM SHANLEY
AND
DON ANDERSON

EDITED AND REVISED BY
JOHN SWINDLE

Addison-Wesley Publishing Company

Reading, Massachusetts • Menlo Park, California • New York

Don Mills, Ontario • Wokingham, England • Amsterdam

Bonn • Sydney • Singapore • Tokyo • Madrid • San Juan

Paris • Seoul • Milan • Mexico City • Taipei

MindShare, Inc (r)
SINGLE-USER LICENSE AGREEMENT

Please read this document carefully before proceeding. This Agreement licenses this electronic book to you and contains warranty and liability disclaimers. By viewing this book, you are confirming your acceptance of the book and agreeing to become bound by the terms of this Agreement. If you do not wish to do so, immediately return the book to MindShare, Inc.

1. DEFINITIONS

(a) "book or electronic book" means the electronic book covered by this Agreement, and any related updates supplied by MindShare, Inc. The book consists of the encrypted PDF file supplied in electronic form.

2. LICENSE
This Agreement allows the **SINGLE END-USER** to:

(a) View the book on a computer or a stand-alone ebook viewer.
(b) You may make and distribute copies of the book and electronically transfer the book from one computer to another or over a network.
(c) Certain rights are not granted under this Agreement, but may be available under a separate agreement. If you would like to enter into a Site or Network License, please contact MindShare.

3. RESTRICTIONS
(a) You may not copy screen images of the book, or any portion thereof.
(b) You may not decompile, reverse engineer, disassemble, or otherwise reduce the book to a human-perceivable form.
(c) You may not modify, rent, resell for profit, distribute or create derivative works based upon the book or any part thereof.
(d) You will not export or reexport, directly or indirectly, the book into any country prohibited by the United States Export Administration Act and the regulations thereunder.
(e) The book may not be used in a group viewing environment.

4. OWNERSHIP

The foregoing license gives you limited rights to use the book. You do not become the owner of, and MindShare retains title to, the intellectual property contained within the book, and all copies thereof. All rights not specifically granted in this Agreement, including Federal and International Copyrights, are reserved by MindShare.

5. DISCLAIMER OF WARRANTIES AND OF TECHNICAL SUPPORT:
The book is provided to you on an "AS IS" basis, without any technical support or warranty of any kind from MindShare including, without limitation, a warranty of merchantability, fitness for a particular purpose and non-infringement. SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. YOU MAY ALSO HAVE OTHER LEGAL RIGHTS WHICH VARY FROM STATE TO

STATE. These limitations or exclusions of warranties and liability do not affect or prejudice the statutory rights of a consumer; i.e., a person acquiring goods otherwise than in the course of a business.

6. LIMITATION OF DAMAGES:
MINDSHARE SHALL NOT BE LIABLE FOR ANY INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES OR LOSS (INCLUDING DAMAGES FOR LOSS OF BUSINESS, LOSS OF PROFITS, OR THE LIKE), WHETHER BASED ON BREACH OF CONTRACT, TORT (INCLUDING NEGLIGENCE), PRODUCT LIABILITY OR OTHERWISE, EVEN IF MINDSHARE OR ITS REPRESENTATIVES HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. SOME STATES DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THIS LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU. The limited warranty, exclusive remedies and limited liability set forth above are fundamental elements of the basis of the bargain between Mindshare and you. You agree that Mindshare would not be able to provide the book on an economic basis without such limitations.

7. GOVERNMENT END USERS (USA only):
RESTRICTED RIGHTS LEGEND The book is "Restricted Computer Software." Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in this Agreement and as provided in DFARS 227.7202-1(a) and 227.7202-3(a) (1995), DFARS 252.227-7013 (OCT 1988), FAR 12.212(a)(1995), FAR 52.227-19, or FAR 52.227-14, as applicable." Manufacturer: Mindshare, Inc., 4285 Slash Pine Drive, Colorado Springs, CO 80908.

8. GENERAL:
This Agreement shall be governed by the internal laws of the State of Colorado. This Agreement contains the complete agreement between the parties with respect to the subject matter hereof, and supersedes all prior or contemporaneous agreements or understandings, whether oral or written. All questions concerning this Agreement shall be directed to: Mindshare, Inc., 4285 Slash Pine Drive, Colorado Springs, CO 80908, Attention: Chief Financial Officer.

Mindshare is registered trademark of Mindshare, Inc.

Single-User License Agreement 9/8/00.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Addison-Wesley was aware of a trademark claim, the designations have been printed in initial capital letters or all capital letters.

The authors and publishers have taken care in preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

Library of Congress Cataloging-in-Publication Data

ISBN: 0-201-40996-8
Copyright © 1995 by MindShare, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher.
Printed in the United States of America. Published simultaneously in Canada.

Sponsoring Editor: Keith Wollman
Project Manager: Eleanor McCarthy
Production Coordinator: Deborah McKenna
Cover design: Barbara T. Atkinson
Set in 10 point Palatino by MindShare, Inc.

1 2 3 4 5 6 7 8 9 -MA- 9998979695
First printing, February 1995

Addison-Wesley books are available for bulk purchases by corporations, institutions, and other organizations. For more information please contact the Corporate, Government, and Special Sales Department at (800) 238-9682.

Dedication

This book is dedicated to the numerous engineers and programmers who contributed to this work unknowingly by asking tough questions during MindShare seminars. These questions led us to a far greater understanding of ISA architecture and made this book more complete.

Contents

Foreword xxiii
Acknowledgments xxv

About This Book

The MindShare Architecture Series 1
Organization of This Book 2
Who This Book Is For 2
Prerequisite Knowledge 2
Documentation Conventions 3
Hex Notation 3
Binary Notation 3
Decimal Notation 3
Signal Name Representation 3
Identification of Bit Fields (logical groups of bits or signals) 4
We Want Your Feedback 4

Overview

System Kernel 6
Memory Subsystems 6
ISA Subsystem 7
Origins of ISA 7
The IBM PC 7
The IBM PC/AT 8
The ISA Concept 8

Part 1: The System Kernel

Chapter 1: Intro to Microprocessor Communications

Instruction Fetch and Execution 11
General 11
In-Line Code Fetching 13
Reading and Writing 15
Type of Information Read from Memory 16
Type of Information Written to Memory 16
The Buses 16
The Address Bus 17
Control Bus – Transaction Type and Synchronization 19
The Data Bus — Data Transfer Path 19

Chapter 2: Introduction to the Bus Cycle

Introduction 21
Automatic Dishwasher – Classic State Machine Example 21
The System Clock – a Metronome 22
Microprocessor's Bus Cycle State Machine 23
Address Time 24
Data Time 25
The Wait State 26

Chapter 3: Addressing I/O and Memory

Evolution of Memory and I/O Address Space 29
Intel 8080 Microprocessor Address Space 29
8086 and 8088 Microprocessor Address Space 33
286 and 386SX Address Space 34
386DX, 486 and Pentium Processor Address Space 35
Memory Mapped I/O 36
The I/O Device 36

Chapter 4: The Address Decode Logic

The Address Decoder Concept 39
Data Bus Contention (Address Conflicts) 41
How Address Decoders Work 41
Example 1– PC and PC/XT ROM Address Decoder 42
Background 42
The PC/XT ROM Address Decode Logic 44
Example 2 – System Board I/O Address Decoder 47

Chapter 5: The 80286 Microprocessor

The 80286 Functional Units 53
The Instruction Unit 55
The Execution Unit 55
General Registers 56
The Status and Control Registers 59
The Address Unit 63
The Segment Registers 63
Segment Register Usage in Real Mode 63
Code Segment (CS) & Instruction Pointer (IP) Registers 67
The Data Segment (DS) Register 69
The Extra Segment (ES) Register 70
Stack Segment (SS) & Stack Pointer (SP) Registers 71
Little-Endian Byte-Ordering Rule 74
Definition of Extended Memory 75
Accessing Extended Memory in Real Mode 76

The Bus Unit 81
Address Latches and Drivers 81
Instruction Prefetcher and 6-byte Prefetch Queue 81
Processor Extension Interface 82
Bus Control Logic 83
Data Transceivers 83
80286 Hardware Interface to External Devices 83
The Address Bus 83
How 80286 Addresses External Locations 85
The Data Bus 87
The Cardinal Rules 88
Cardinal Rule Number One 88
Cardinal Rule Number Two 89
Cardinal Rule Number Three 89
The Control Bus 89
Bus Cycle Definition Lines 90
Bus Mastering Lines 90
Protecting Access To Shared Resource 92
Ready Line 94
Interrupt Lines 95
Processor Extension Interface Lines 96
The Clock Line 97
The Reset Line 97
Protected Mode 98
Intro to Protected Mode and Multitasking Operating Systems 99
Segment Register Usage in Protected Mode 101

Chapter 6: The Reset Logic

The Power Supply Reset 107
Reset Button 108
Shutdown Detect 108
Hot Reset 109
Alternate (Fast) Hot Reset 111
Ctrl-Alt-Del Soft Reset 111

Chapter 7: The Power-Up Sequence

The Power Supply – Primary Reset Source 113
How RESET Affects the Microprocessor 115
Processor Reaction When Output Voltages Stabilize 115
The First Bus Cycle 116

Chapter 8: The 80286 System Kernel: the Engine

The Bus Control Logic 119

The Address Latch **121**
Address Pipelining **123**
The Data Bus Transceivers **124**
Data Bus Steering Logic **126**
Scenario One – Read Even-Addressed Location in 8-Bit Device 128
Scenario Two – 8-Bit Read from Odd-Addressed Location in 8-Bit Device 129
Scenario Three – 8-Bit Write to Odd-Addressed Location in 8-Bit Device 131
Scenario Four – 16-Bit Write to 8-Bit Device 132
Scenario Five – 16-Bit Read from 8-Bit Device 133
Scenario Six – 8-Bit Read from 16-Bit Device 134
Scenario Seven – 16-Bit Read from 16-Bit Device 134
Scenario Eight – 8-Bit Write to 16-Bit Device 134
Scenario Nine – 16-Bit Write to 16-Bit Device 134
The Ready Logic **134**
Access Time 134
Stretching the Transfer Time 135
The Default Ready Timer 135
Custom Ready Timers 136
Extending the Default Timing 137
Shortening the Default Timing 138

Chapter 9: Detailed View of the 80286 Bus Cycle

Address and Data Time Revisited **139**
The Read Bus Cycle **141**
Bus Cycle A 144
Bus Cycle B 145
The Write Bus Cycle **146**
The Halt or Shutdown Bus Cycle **150**
Halt 150
Shutdown 151

Chapter 10: The 80386 DX and SX Microprocessors

Introduction **153**
The 80386 Functional Units **154**
General 154
Code Prefetch Unit 155
Instruction Decode Unit 155
Execution Unit 156
General 156
The Registers 156
General Registers 156
Status, MSW and Instruction Registers 157
Debug Registers 159

Test Registers 160
Segmentation Unit 160
Paging Unit 161
Bus Unit 162
Protected Mode 162
Page Translation 164
Virtual Paging 164
Translation Lookaside Buffer 170
Virtual-8086 Mode 172
Automatic Self-Test 174
80386DX External Interface 175
The Address Bus 175
The Data Bus 183
The Control Bus 184
Bus Cycle Definition Outputs 185
Processor Extension Lines 185
Address Status Output 185
Pipelining Control Input 186
Dynamic Bus Sizing (BS16#) 186
80386SX External Interface 187
Interface Signal Differences 187
A0 or BLE# 188
Addressing Scheme, Data Bus Width Ramifications 188
Throughput and Compatibility Considerations 189

Chapter 11: The 80386 System Kernel

Introduction 191
The 80386SX System Kernel 192
The 80386DX System Kernel 194
80386DX System Kernel with Dynamic Bus Sizing 194
Introduction 194
Reading from an 8-Bit Device 196
One-Byte Read from an 8-Bit Device 196
Two-Byte Read from an 8-Bit Device 197
Four-Byte Read from an 8-Bit Device 198
Writing to an 8-Bit Device 200
One-Byte Write to an 8-Bit Device 200
Two-Byte Write to an 8-Bit Device 201
Four-Byte Write to an 8-Bit Device 203
Reading from a 16-Bit Device 205
Two-Byte Read from a 16-Bit Device 205
Four-Byte Read from a 16-Bit Device 206
Writing to a 16-Bit Device 207

Two-Byte Write to a 16-Bit Device 207
Four-Byte Write to a 16-Bit Device 208
Reading from a 32-Bit Device 210
Writing to a 32-Bit Device 211
80386DX System Kernel without Dynamic Bus Sizing 212
Introduction 212
Reading from a 16-Bit Device 214
Two-Byte Read from a 16-Bit Device 214
Four-Byte Read from a 16-Bit Device 215
Writing to a 16-Bit Device 216
Two-Byte Write to a 16-Bit Device 216
Four-Byte Write to a 16-Bit Device 217

Chapter 12: Detailed View of the 80386 Bus Cycles

The Bus Cycle Types 221
Address Pipelining Overview 222
Memory or I/O Read Bus Cycle 222
Memory or I/O Write Bus Cycle 224
Address Pipelining Example 226
Interrupt Acknowledge Bus Cycle 230
Halt or Shutdown Bus Cycle 230

Part 2: Memory Subsystems

Chapter 13: RAM Memory: Theory of Operation

Dynamic RAM (DRAM) Memory 235
DRAM Addressing Sequence 236
Row and Column Address Source 238
DRAM Addressing Logic 239
Detailed Description of DRAM Addressing Sequence 242
How Data is Stored in DRAM 244
DRAM Refresh 244
Refresh Logic and RAS-only Refresh 245
CAS-before-RAS Refresh 248
Hidden Refresh 248
Self-Refresh 248
Destructive Read: Pre-Charge Delay and Cycle Time 250
DRAM Bank 251
DRAM Bank Width 253
DRAM Error Detection and Correction 256
DRAM Parity 256
Error-Checking-and-Correcting Memory 259
Page-Mode DRAM and Its Variations 259

Page Mode DRAM 259
Enhanced Page Mode DRAM 264
Burst and Nibble Mode DRAM 265
Static Column RAM (SCRAM) 267
Synchronous DRAM 268
Interleaved Memory Architecture 269
Static RAM (SRAM) 271

Chapter 14: Cache Memory Concepts

The Problem 273
The Solution 274
Principles of Locality 277
Temporal Locality 277
Spatial Locality 277
Cache Performance 278
Overall System Performance 278
Cache Consistency 279
Components of a Cache Subsystem 279
Cache Memory 279
Cache Management Logic 281
Cache Memory Directory 281
Intro to Cache Architecture, Coherency, Write Policies and Organization 282
Cache Architectures 283
Look-Through Cache 283
Look-Aside Cache 287
First- and Second-Level Caches 289
Combined (Unified) and Split (Dedicated) Caches 290
Cache Consistency (Coherency) 291
Causes of Cache Consistency Problems 292
Write Policy 292
Write-Through Cache Designs 293
Buffered Write-Through Designs 293
Write-Back Cache Designs 294
Bus Master/Cache Interaction 295
Bus Snooping/Snarfing 295
Coherency via Cache Flushing 297
Software-Enforced Coherency 298
Cache Organization and Size 299
Fully-Associative Cache 299
Direct-Mapped Cache (One-Way Set-Associative) 301
Two-Way Set-Associative Cache 304
Four-Way Set-Associative Cache 307
Least-Recently Used (LRU) Algorithm 307

Cache Line Size 308
Cache Size 310
First-Level Cache Size 310
Second-Level Cache Size 310
Cache Addressing 311
I/O Information Not Cached 311
Non-Cacheable Memory 312
Testing Memory 313

Chapter 15: ROM Memory

ROM Memory — Theory of Operation 315
Introduction 315
Fusible-Link PROM 317
Masked ROM (MROM) 317
Eraseable Programmable Read-Only Memory (EPROM) 318
Electrically Eraseable Programmable Read-Only Memory (EEPROM) 319
Flash EEPROM 320
ROM's Interface to System 320
System Board ROM Memory 322
Testing 322
Shadow RAM 324
Shadow RAM and ROM Occupying Different Address Spaces 324
Shadow RAM and ROM Occupy Same Address Space 325
Double Mapping ROM and Shadow RAM Address Space 325
Recovering Unused ROM Address Space 326
32KB System Board ROM Configuration 327
64KB System Board ROM Configuration 329
ROMs on ISA Cards (Device ROMs) 330

Part 3: The Industry Standard Architecture

Chapter 16: ISA Bus Structure

Introduction 335
Address Bus 338
Data Bus 339
Bus Cycle Definition 339
Bus Cycle Timing 341
Device Size 343
Reset 344
DMA 345
Interrupts 347
Error Reporting Signal 349
Miscellaneous Signals 350

Chapter 17: Types of ISA Bus Cycles

Introduction **351**
Transfers with 8-bit Devices **352**
Transfers with 16-bit Devices **355**
Standard 16-bit Memory Device ISA Bus Cycle 355
Standard 16-bit I/O Device ISA Bus Cycle 359
0-Wait State Access to 16-bit Memory Device 362

Chapter 18: The Interrupt Subsystem

What Is an Interrupt? **365**
Microprocessor Response to Interrupt Request **366**
Interrupt Acknowledge Bus Cycles 369
Saving Pointer to Interrupted Program 373
Clearing the Interrupt Enable Flag 378
Jumping to the ISR 378
Resuming the Interrupted Program 379
When One 8259 Interrupt Controller Isn't Enough **383**
Servicing Requests to Slave Interrupt Controller **387**
Interrupt Table Entry Assignments **388**
IRQ2 Redirect **389**
Shareable Interrupts in ISA Machines **391**
Generating the Interrupt Request 391
Interrupt Table Initialization – Add-in Devices 392
Shared Interrupt Procedure 393
Phantom Interrupts **394**
Programming the 8259 ..... **396**
Introduction 396
Programming the Registers 396
Non-Maskable Interrupt Requests (NMI) **399**
Software Interrupts **402**
Software Exceptions 402
Software Interrupt Instruction 403
Protected Mode Interrupts **406**

Chapter 19: Direct Memory Access (DMA)

DMA Concept **409**
DMA Example **410**
DMA Controller (DMAC) **414**
DMA Transfer Types 414
DMA Transfer Modes 414
Single Transfer Mode 415
Block Transfer Mode 416

Demand Transfer Mode....416
Cascade Mode....417
DMAC Priority Logic....417
DMA Bus Cycle....420
Byte or Word Transfers....423
DMAC Addressing Capability....423
Addressing ISA Memory....427
Addressing Local Bus Memory....428
Address Translation....430
Data Bus Steering....430
DMA Transfer Rate....431
DMAC Initialization During POST....431

Chapter 20: ISA Bus Masters

ISA Bus Master Capability....433
ISA Bus Masters in 80386DX and Higher Systems....435
Address Translation....438
Data Bus Steering....438
Bus Masters and DRAM Refresh....439

Chapter 21: RTC and Configuration RAM

Introduction....441
Accessing the RAM Locations....442
Address Decoder and RTC's Hardware Interface....442
Real-Time Clock Function....443
Using BIOS to Control Real-Time Clock....445
Configuration RAM Usage....446

Chapter 22: Keyboard/Mouse Interface

Keyboard/Mouse Interface....449
Keyboard....450
Mouse....450
8042 Local I/O Ports....450
Hot Reset....451
A20 Gate....451
Local Port Definition....452
System Interface....453
Command/Status Port....454
Data Port....456
BIOS Routine....457

Chapter 23: Numeric Coprocessor

Introduction....459
Setup....460

With Numeric Coprocessor Installed **460**
Numeric Coprocessor Reset **463**
Without Numeric Coprocessor Installed (Emulation) **465**
Weitek Numeric Coprocessor **467**
Introduction 467
Systems Incorporating Just the Weitek 467
General 467
Weitek Handling of Intel Coprocessor Instructions 467
Systems Incorporating Both Coprocessor Types 468

Chapter 24: ISA Timers

ISA Oscillator, OSC **469**
System Timer (Timer 0) **469**
Refresh Timer (Timer 1) **471**
Speaker Timer (Timer 2) **471**
Watchdog Timer **472**
Slowdown Timer **474**

Appendices

I/O Address Map **479**
Glossary of Terms **491**

Index **507**

Figures

Figure 1. Simplified Block Diagram of Typical ISA System....5
Figure 1-1. Microprocessor's Relationship to External Devices....13
Figure 1-2. Sample Instruction Sequence....15
Figure 1-3. The 80286 Address Bus....18
Figure 1-4. The 80286 Data Bus....20
Figure 2-1. Crystal Oscillator Output....22
Figure 2-2. Relationship of CLOCK and PCLK....23
Figure 2-3. Each Cycle of PCLK Defines the Duration of a State....24
Figure 2-4. Bus Cycle Consisting of Address Time and Data Time....26
Figure 2-5. Bus Cycle with One Wait State Inserted....27
Figure 3-1. Single 64K Address Space....30
Figure 3-2. The 8085's Memory and I/O Address Spaces....32
Figure 3-3. 8086 and 8088 Memory and I/O Address Space....34
Figure 3-4. 286 and 386SX Memory and I/O Address Space....35
Figure 3-5. 386/486/Pentium Processor Memory and I/O Address Space....36
Figure 4-1. Relationship of Address Bus, Address Decoders and Devices....40
Figure 4-2. PC Memory Address Map....43
Figure 4-3. Example PC/XT ROM Address Decoder....45
Figure 4-4. System Board I/O Address Decoder....48
Figure 4-5. Microprocessor Placing I/O Address 0012h on the Address Bus....51
Figure 5-1. 80286 Microprocessor Block Diagram....54
Figure 5-2. The 80286 General Registers....57
Figure 5-3. The 80286 Status and Control Registers....59
Figure 5-4. The Flag Register's Status, Control and System Bits....60
Figure 5-5. The 80286 Machine Status Word (MSW) Register....62
Figure 5-6. Result If Segment Registers Were Restricted to 16 Bits....65
Figure 5-7. Segment Registers Point to Start Address of Memory Segments....66
Figure 5-8. CS and IP Registers....68
Figure 5-9. Illustration of Location within the Data Segment....70
Figure 5-10. The Stack....72
Figure 5-11. Extended Memory....76
Figure 5-12. A20 High....78
Figure 5-13. A20 Gate....80
Figure 5-14. The 80286 Address Bus....84
Figure 5-15. The 80286 Data Bus....88
Figure 5-16. The Lock Output....92
Figure 5-17. Relationship of the Microprocessor and the Numeric Coprocessor....96
Figure 5-18. The Reset Logic....98
Figure 5-19. The Segment Selector....102

Figure 5-20. The Segment Descriptor 104
Figure 7-1. RESET Is Derived from POWERGOOD....114
Figure 8-1. The Bus Control Logic 120
Figure 8-2. The Address Latch 122
Figure 8-3. The Data Bus Transceivers....125
Figure 8-4. The Data Bus Steering Logic....127
Figure 8-5. Access Time 135
Figure 8-6. Custom Ready Timer Logic 137
Figure 9-1. The Read Bus Cycle 142
Figure 9-2. The 80286 System Engine....143
Figure 9-3. The Write Bus Cycle 148
Figure 10-1. 80386 Microprocessor Block Diagram 154
Figure 10-2. The 80386 General Registers....157
Figure 10-3. The 80386 Status and Instruction Registers 158
Figure 10-4. The 80386 Debug Registers....159
Figure 10-5. The 80386 Test Registers 160
Figure 10-6. The 80386 Segment Registers....161
Figure 10-7. The 80286 Segment Descriptor Format 163
Figure 10-8. The 80386 General Segment Descriptor Format....164
Figure 10-9. Control Registers CR2 and CR3 166
Figure 10-10. Translation of Linear Memory Address to Physical Memory Address...167
Figure 10-11. Another View of the Page Directory, Page Tables and Memory Pages...170
Figure 10-12. The Translation Look-Aside Buffer 171
Figure 10-13. 80386DX Address/Byte Enable Relationship....179
Figure 10-14. The 80386DX Data Bus 184
Figure 11-1. The 80386SX-Based Kernel 193
Figure 11-2. The 80386DX-Based Kernel with Dynamic Bus Sizing Implemented....195
Figure 11-3. The 80386DX-Based Kernel without Dynamic Bus Sizing Implemented ..213
Figure 12-1. Non-Pipelined Read Bus Cycle 223
Figure 12-2. Non-Pipelined Write Bus Cycle 225
Figure 12-3. Transition from Non-Pipelined to Pipelined Operation 229
Figure 13-1. Block Diagram of a DRAM Memory Chip 237
Figure 13-2. The Address Output by the Current Bus Master....239
Figure 13-3. The Time Delay Device 240
Figure 13-4. Block Diagram of DRAM Addressing Logic 241
Figure 13-5. DRAM Addressing Timing Diagram 243
Figure 13-6. The Refresh Logic....246
Figure 13-7. Pre-Charge Delay and Cycle Time....251
Figure 13-8. The DRAM Bank 253
Figure 13-9. 16-Bit DRAM Bank....254
Figure 13-10. 32-Bit DRAM Bank....255
Figure 13-11. The System Board DRAM Parity Logic....258

Figure 13-12. Page Mode DRAM Support Logic 262
Figure 13-13. Page Mode DRAM RAS/CAS Timing 264
Figure 13-14. Burst Mode Access Timing 266
Figure 13-15. SCRAM Access Timing 268
Figure 13-16. Interleaved Memory Architecture 270
Figure 14-1. The Cache Memory Concept 275
Figure 14-2. Cache Subsystem Components 280
Figure 14-3. Relationship of Cache Directory to the Cache Memory 281
Figure 14-4. Look-Through Cache Architecture 284
Figure 14-5. Look-Through—Concurrent Bus Operation 285
Figure 14-6. Look-Aside Cache Architecture 288
Figure 14-7. First- and Second-Level Caches 289
Figure 14-8. Fully-Associative Cache Organization 300
Figure 14-9. Set-Associative Organization—Direct-Mapped 302
Figure 14-10. Direct-Mapped Cache Organization 303
Figure 14-11. Two-Way Set-Associative Cache Organization 306
Figure 15-1. The Bit Cell 316
Figure 15-2. The PROM Bit Cells 317
Figure 15-3. The EPROM Bit Cell 319
Figure 15-4. The ROM's Interface to the System 321
Figure 15-5. System Board ROM Configuration with 32KB ROMs 329
Figure 15-6. System Board ROM Configuration with 64KB ROMs 330
Figure 16-1. The 8-bit Portion of the ISA Bus 336
Figure 16-2. The 16-bit Portion of the ISA Bus 336
Figure 16-3. The ISA Connector 337
Figure 16-4. The ISA DMA Signals and Associated System Board Logic 346
Figure 16-5. The ISA Interrupt Signals and Associated System Board Logic 348
Figure 16-6. The Channel Check Logic 349
Figure 17-1. Standard Access to an 8-bit ISA Device 353
Figure 17-2. Standard Access to a 16-bit ISA Memory Device 357
Figure 17-3. Standard Access to 16-bit I/O device 360
Figure 17-4. Zero-Wait-State Access to a 16-bit ISA Memory Device 363
Figure 18-1. Flowchart of Interrupt Servicing 367
Figure 18-2. The Intel 8259 Programmable Interrupt Controller (PIC) 368
Figure 18-3. The Interrupt Table (in Real Mode) 370
Figure 18-4. The Stack 373
Figure 18-5. Stack After Flag Register Push Operation 375
Figure 18-6. Stack After CS Register Push Operation 376
Figure 18-7. Stack After IP Register Push Operation 377
Figure 18-8. Stack Before Pop Operation to Read IP Register 380
Figure 18-9. Stack Before Pop Operation to Read CS Register 381
Figure 18-10. Stack Before Pop Operation to Read Flag Register 382

Figure 18-11. Cascaded Interrupt Controllers....385
Figure 18-12. Causes of a NMI....401
Figure 19-1. Example Disk/DMA Hardware System....411
Figure 19-2. Cascaded DMA Controllers....419
Figure 19-3. Single Transfer Mode Timing....422
Figure 19-4. The Slave DMAC Memory Address Logic....425
Figure 19-5. The Master DMAC Memory Address Logic....426
Figure 19-6. Master DMAC Addressing 16-bit ISA Memory....427
Figure 19-7. DMA (16-bit channel) Addressing 32-bit System Memory....428
Figure 19-8. 16-bit Access to 32-bit Memory with DMA....429
Figure 20-1. DMA and Bus Mastering....434
Figure 20-2. ISA Bus Master Addressing 32-bit System Memory....436
Figure 20-3. Addressing 32-bit Memory with ISA Bus Masters....437
Figure 21-1. The RTC/Configuration RAM Chip....443
Figure 22-1. The Keyboard Controller....451
Figure 22-2. Sample 8042 Configuration....452
Figure 23-1. The 80287 Numeric Coprocessor....462
Figure 24-1. The System Timer, Timer 0....470
Figure 24-2. The Speaker Timer and Associated Logic....472
Figure 24-3. The Watchdog Timer and Associated Logic....474

Tables

Table 4-1. State of the Address Lines When Any Memory Address in Range F0000h to FFFFFh Is Addressed 43
Table 4-2. State of the Address Lines when any Memory Address in Range E0000h to EFFFFh Is Addressed 44
Table 4-3. System Board I/O Address Sub-Range Assignments....48
Table 4-4. 74138 Output Selection Criteria 50
Table 4-5. I/O Address Range Assigned to Each System Board Device 52
Table 5-1. The Flag Register's Status, Control and System Bits 61
Table 5-2. The 80286 MSW Register Bits....62
Table 5-3. Little-Endian Byte Ordering....74
Table 5-4. Big-Endian Byte Ordering (not used on x86) 74
Table 5-5. Memory Contents 75
Table 5-6. Binary Weighted Value of each Address Line 86
Table 5-7. Transfer Size Indicated by A0 and BHE# 87
Table 5-8. Examples of Even and Odd Addresses....89
Table 5-9. Bus Cycle Definition 90
Table 5-10. Attribute Byte Definition 105
Table 6-1. CMOS Reset Code Definition....112
Table 7-1. Values Preset Into 8086 and 8088 Microprocessor Registers by RESET 115
Table 7-2. State of 80386DX Outputs While RESET is Asserted 115
Table 7-3. State of 80286 Outputs While RESET Is Asserted 115
Table 8-1. Number of Wait States—Default Ready Timer 135
Table 9-1. 80286 Bus Cycle Definition....140
Table 9-2. 80286 and 80386 Bus Cycle State Names 140
Table 10-1. Format of Page Table Entry Attribute Bits 168
Table 10-2. Signals Common to the 80286 and 80386DX Microprocessors 175
Table 10-3. Example Addresses Output by an 80386DX 177
Table 10-4. Relationship of 80386DX Byte Enables, Data Paths and Locations in the Currently-Addressed Doubleword 178
Table 10-5. Relationship of Addresses and Byte Enables Output by an 80386DX....180
Table 10-6. 80386DX Addressing Examples....180
Table 10-7. 80386DX Byte Enable Combinations....181
Table 10-8. Example Instructions and Resultant Addresses 182
Table 10-9. 80386 Bus Cycle Definition....185
Table 10-10. Interface Signals That Differentiate the 80386SX from the 80386DX Microprocessor....188
Table 12-1. 80386 Bus Unit States 230
Table 14-1. Cache Size vs. Hit Rate....310
Table 15-1. The ROM's Interface Signals 321

Table 16-1. The ISA Address-Related Signals 338
Table 16-2. The ISA Data Bus Paths 339
Table 16-3. Bus Cycle Definition Line Decoding 339
Table 16-4. The ISA Command Lines 340
Table 16-5. Bus Cycle Timing Signals 341
Table 16-6. The Size 16 Signal Lines 344
Table 16-7. The ISA DMA Signal Lines and Associated System Board Logic 345
Table 16-8. ISA Bus Interrupt Request Line Assignment 347
Table 16-9. Interrupt Request Line Assignment, Lines Not on ISA Bus 347
Table 16-10. The ISA Error Reporting Signal 349
Table 16-11. Miscellaneous ISA Signals 350
Table 18-1. Standard ISA Interrupt Table entry number Assignments 389
Table 18-2. Initialization Sequence for the Master Interrupt Controller 397
Table 18-3. Initialization Sequence for the Slave Interrupt Controller 398
Table 18-4. System Control Port B at I/O Address 0061h (Read) 400
Table 18-5. System Control Port A at I/O Address 0092h 400
Table 18-6. Interrupt Table Entry Assignment 403
Table 19-1. DMA Clock Frequency 420
Table 19-2. DMAC States 421
Table 19-3. DMAC Memory Address and Page Registers—I/O Locations 424
Table 19-4. ISA DMA Transfer Rates 431
Table 19-5. Initialization of DMACs 432
Table 21-1. The Real-Time Clock Bytes 443
Table 21-2. RTC Status Register A 444
Table 21-3. RTC Status Register B 444
Table 21-4. RTC Status Register C 445
Table 21-5. RTC Status Register D 445
Table 21-6. Configuration RAM Usage 446
Table 21-7. The Reset Code Byte 447
Table 22-1. Sample Keyboard/Mouse Interface—Local Input Port 453
Table 22-2. Sample Keyboard/Mouse Interface—Local Output Port 453
Table 22-3. Keyboard/Mouse Interface—Test Port 453
Table 22-4. 8042 Sample Keyboard/Mouse Interface Command List 455
Table 23-1. The Numeric Coprocessor Control Bits in the MSW Register 460
Table 23-2. Intel Coprocessor/Processor Communication Signals 464
Table 23-3. Weitek Interpretation of the Memory Address 468

Foreword

Have you ever started using a new tool and shortly thereafter wondered how in the world you ever got along without it? Well, that is how I feel about this book, and many of my technical compatriots here at Dell Computer Corporation would agree. Let's face it, the fast moving computer world is treacherous and follows Darwin's Law of Survival of the Fittest, where the strong survive and the weak get eaten. Don Anderson and Tom Shanley have provided you with a mechanism to maximize your chances of survival in this exciting world of computers.

ISA System Architecture is what I call the missing link. There exist many complementary tools to this book such as various chip data books, programming manuals, "light" architecture books, "how to" books, logic analyzers, scopes, debuggers, digital logic textbooks and so forth, but none that describe the AT architecture from a system point of view in such an organized and understandable fashion. All of the fundamental and critical areas of a functional ISA system are explained in detail with diagrams and examples including bus cycles, addressing, I/O and memory, decode logic, reset logic, power-up, microprocessors, the system kernel, RAM, cache, ROM, interrupts, DMA, busmastering, RTC and configuration RAM, keyboard/mouse interface, coprocessor, timers, and more. Ergo, you've got a comprehensive reference on AT system functionality; you've got the missing link.

We were fortunate to have Don and Tom conduct several training classes based on this book here at Dell (and continue to do so). Members of my group consist of engineers and technicians who are first to test systems after they leave development; we perform a variety of tests including hardware, software, and environmental tests. The information obtained from this class and book was excellent, but what was pleasantly surprising was the reaction of the folks who attended these sessions. During the classes and right after, there were people throughout the lab with big grins saying something to the effect of "This is great. I've been wanting to tie all of this information together so I could not only understand it fully, but put it to good use." And they have put the information to good use, which has helped us improve our process and test smarter.

This book is quite valuable as a training tool. The first technical literature that a new engineer or technician in my group studies is this book. This allows him or her to learn not only the system but the fundamentals required to test and troubleshoot a system in the most efficient way. Many technical managers in other groups here such as design, manufacturing, sustaining, technical support, and repair use this book in a similar manner. As a matter of fact, training classes using this book are available on a fairly regular basis within our entire organization. Not only do our technical people have a sound foundation on the

building blocks of a system, but throughout the company, the terminology used is consistent. A BALE is a BALE is a BALE throughout the company.

Finally, I'd like to expand on the concept of testing smarter and improving your process. All of us in this business know that we must improve our processes to successfully compete. We need to decrease our time-to-market cycles, perform more complex processes in this shorter time period, and achieve this improvement on increasingly complex products. This is a real, not imaginary, burden to bear! However, if implemented successfully, the benefits I've mentioned can be attained in addition to improved methods and throughput, better troubleshooting techniques, reduction in field returns, and an increase in customer service quality. Quite simply, this text can be used as a part of this process.

In closing, I'd like to commend Don and Tom for a job well done. There is no doubt that their blood, sweat and tears have produced an immensely valuable technical work that they can well be proud of. The book is an invaluable tool and reference on ISA architecture and time reading it is time well spent. You might just end up wondering how you ever got along without it!

Dave Greenberg B.S.E.E, C.Q.E.
Manager, Test Operations
Dell Computer Corporation

Acknowledgments

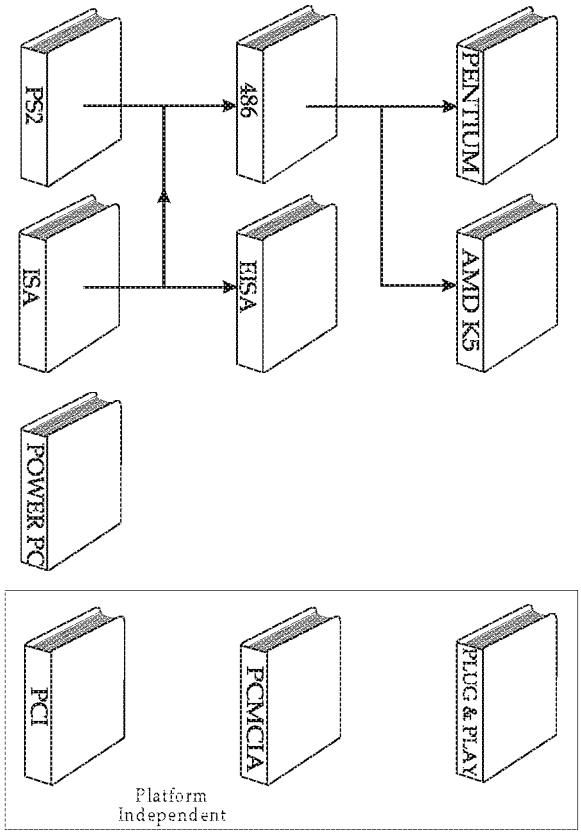
To Jerry Farnham for his help in editing the earlier versions of this book and for his many ideas that were incorporated.

To John Swindle for his tireless attention to detail during the editing and revision of the third edition.

The MindShare Architecture Series

The MindShare Architecture book series includes: *ISA System Architecture*, *EISA System Architecture, 80486 System Architecture, PCI System Architecture*, *Pentium System Architecture, PCMCIA System Architecture, PowerPC System Architecture, Plug-and-Play System Architecture*, and *AMD K5 System Architecture.*

Rather than duplicating common information in each book, the series uses the building-block approach. *ISA System Architecture* is the core book upon which the others build. The figure that follows illustrates the relationship of the books to each other.



Series Organization

Organization of This Book

ISA System Architecture is divided into three parts:

- **Part 1: The System Kernel**. This section provides a detailed tutorial on how Intel X86 microprocessors communicate with memory and I/O devices. Included is the support logic which allows the microprocessor to communicate with 8- and 16-bit devices and detailed descriptions of the signals and timing involved in all Bus Cycle types.
- **Part 2: Memory Subsystems**. The memory section provides a detailed theory of operation of RAM and ROM devices, along with implementations commonly used in ISA systems. The section also covers the concepts and terminology related to cache memory designs.
- **Part 3: The Industry Standard Architecture**. This section provides a detailed discussion of the ISA Bus Architecture including ISA bus cycles and timing, along with ISA implementations of Interrupts, DMA, Real-Time Clock and Configuration RAM, Numeric Coprocessors, Keyboard/Mouse Interface, and Timers.

Who This Book Is For

This book is intended for use by hardware and software design and support personnel. Due to the clear, concise explanatory methods used to describe each subject, personnel outside of the design field may also find the text useful.

Prerequisite Knowledge

The approach taken by this book assumes moderate technical knowledge on the part of the technical audience. It is assumed that the reader has had exposure to PCs in a technical capacity (hardware designer, programmer, technical support, repair, and so on). Familiarity with the binary and hexadecimal numbering systems and their application to electronic computing is essential. To ensure an optimum level of understanding, every attempt is made to explain terminology as it is encountered and to avoid mystifying terms.

Documentation Conventions

This section defines the typographical conventions used throughout this book.

Hex Notation

All hex numbers are followed by an "h." Examples:

```
9A4Eh
0100h
```

Binary Notation

All binary numbers are followed by a "b." Examples:

```
0001 0101b
01b
```

Decimal Notation

When required for clarity, all decimal numbers are followed by a "d." Examples:

```
256d
128d
```

Signal Name Representation

Each signal that assumes the logic low state when asserted is followed by a pound sign (#). As an example, the REFRESH# signal is asserted low when the refresh logic runs a refresh bus cycle.

Signals that are not followed by a pound sign are asserted when they assume the logic high state. As an example, DREQ3 is asserted high to indicate that a device using DMA Channel three is ready for data to be transferred.

Identification of Bit Fields (logical groups of bits or signals)

All bit fields are designated as follows:

[X:Y]

where "X" is the most-significant bit and "Y" is the least-significant bit of the field. As an example, the ISA data bus consists of SD[15:0], where SD15 is the most-significant and SD0 the least-significant bit of the field.

We Want Your Feedback

MindShare values your comments and suggestions. You can contact us via mail, phone, fax or internet email.

Phone (800) 633-1440
Fax (719) 487-1434
Email tom@mindshare.com

Web Site

Because we are constantly on the road teaching, we can be difficult to get hold of. To help alleviate problems associated with our migratory habits, we have a web site to supply the following services:

- Download course abstracts.
- Download tables of contents of each book in the series.
- Facility to inquire about public architecture seminars.
- Facility to view book errata and clarifications.

Web Site: www.mindshare.com

Mailing Address

MindShare, Inc.
4285 Slash Pine Dr.
Colorado Springs, CO 80908

This book details the operation of ISA systems. Figure 1 below illustrates most of the subsystems covered. The book is divided into three major parts:

- The System Kernel
- Memory Subsystems
- The Industry Standard Architecture

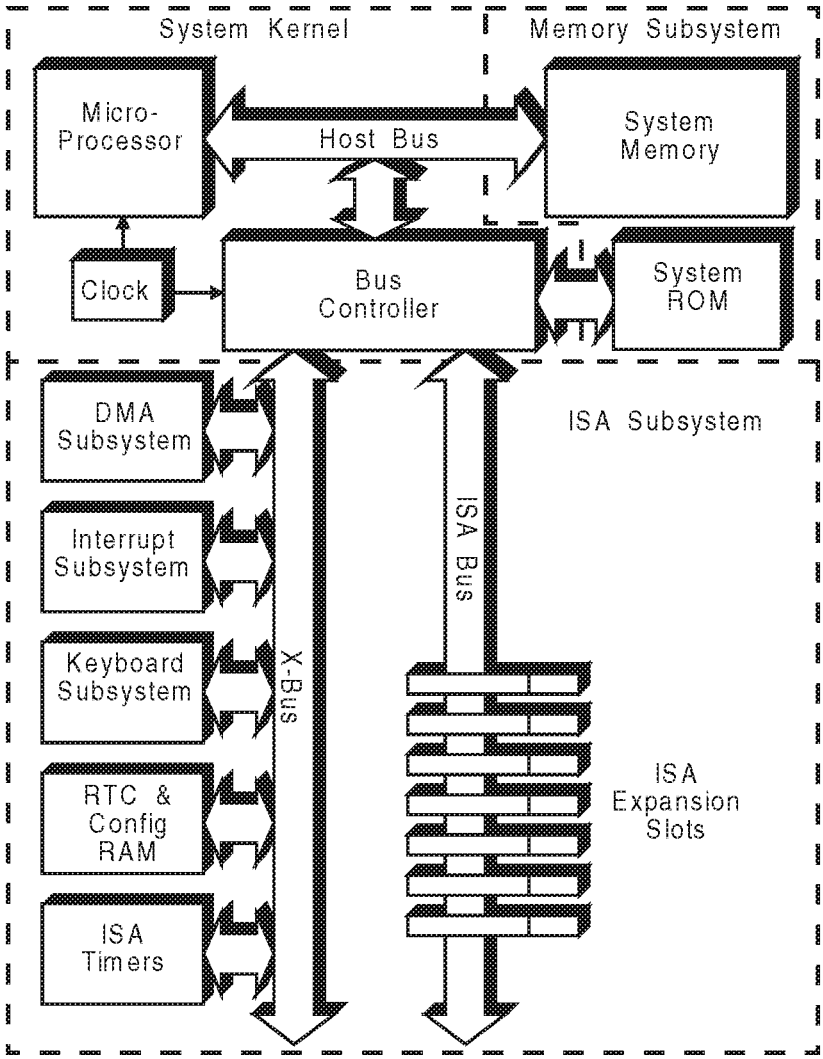


Figure 1. Simplified Block Diagram of Typical ISA System

In addition to detailed analysis of the hardware, many programming examples are covered with the purpose of tying the software and hardware together. This provides a very comprehensive understanding of system operation.

Refer to figure 1 during the following discussion of each major section.

System Kernel

The system kernel consists of the microprocessor and bus control logic, as well as other support circuitry not shown in figure 1. The kernel provides the basic signal, timing and protocols necessary for the microprocessor to communicate with the rest of the system.

This section provides a detailed tutorial on how Intel X86 microprocessors communicate with memory and I/O devices. Included is the support logic which allows the microprocessor to communicate with 8- and 16-bit devices and detailed descriptions of the signals and timing involved in all Bus Cycle types.

Several kernel implementations are possible with ISA systems depending on the exact processor used. Since ISA Architecture is based on the 8MHz IBM PC/AT, the initial kernel discussion is based on the Intel 80286 microprocessor running at 8MHz. 80286 microprocessors running at speeds faster than 8MHz require slight modification to the kernel. More drastic changes are required when using an 80386, 80486 or Pentium microprocessor in ISA systems. All major variations to kernel design and implementation are covered. Note that a thorough understanding of the 80286 kernel is essential before studying the 80386 kernel and that a thorough understanding of the 80386 kernel is essential before studying the 80486 and Pentium processors.

Memory Subsystems

Memory subsystems include main system memory, system ROM, and cache memory (not shown in figure 1). Design of the memory subsystem is key to obtaining maximum system performance at a reasonable price. A wide variety of memory devices are used in ISA systems to achieve high yet economical performance. Along with these devices come various memory architectures also designed to increase system performance.

The memory section provides a detailed theory of operation of RAM and ROM devices, along with implementations commonly used in ISA systems. The section also covers the concepts and terminology related to cache memory designs.

ISA Subsystem

The ISA subsystem includes the bus structures and standard devices used in all ISA systems. The ISA buses operate at speeds of 8MHz to 8.33MHz and include ability to communicate with both 8- and 16-bit devices. By having a range of allowable bus speeds, the system designer can easily divide the microprocessor's clock frequency down to the ISA clock frequency. Specific functions common to all ISA systems include Direct Memory Access (DMA), Interrupts, Keyboard, Real-Time Clock (RTC) and Configuration RAM, and the ISA Timers. Each of these functions must operate according to ISA standards.

This section provides a detailed discussion of the ISA Bus Architecture including ISA bus cycles and timing, along with ISA implementations of Interrupts, DMA, Real-Time Clock and Configuration RAM, Numeric Coprocessors, Keyboard/Mouse Interface, and Timers.

Some functions within ISA systems can be implemented in different ways depending on manufacturer preference, while others must conform to precise ISA standards. Items that can be implemented in different ways are described in general and followed by specific examples that have been chosen based on common or most recent implementations.

Origins of ISA

The IBM PC

The ISA bus structure is based partially on the first Intel-based PC: the IBM PC. The original PC used an Intel 8088 running at 4.77MHz, limiting the expansion bus for this system to eight data lines and twenty address lines. Subsequent designs of the PC included support for features such as fixed disk drives, but the bus structure remained the same to ensure continuing compatibility for a growing number of expansion devices.

The IBM PC/AT

The Intel 80286-based PC/AT maintained compatibility with the PC and offered additional capability. IBM designed the PC/AT so that 8-bit boards used in the original PC could work in the PC/AT. To accommodate the sixteen data lines and twenty-four address lines of the Intel 80286, IBM designed an additional connector to extend the 8-bit PC connector. These two connectors became the new industry standard.

System compatibility with ISA was originally based on the 8MHz IBM PC/AT; however, no written specification on the system architecture was available. The acid test for software and hardware compatibility was "if it ran on the 8MHz PC/AT it was compatible." That is, if software and hardware ran successfully on a PC/AT it should also run on a compatible machine. The lack of a standard however, created numerous problems for designers of both software and hardware.

The ISA Concept

The signals found on each ISA slot can be divided into three basic categories:

- The Address Bus group
- The Control Bus group
- The Data Bus group

These signal groups are present on the 8- and 16-bit expansion slots found in all ISA-compatible and IBM PC products. In an ISA system, each of these signal groups has been extended to provide additional capability not found in the IBM PC and PC/XT architecture.

The ISA standard encompasses more than just the structure of the expansion connectors. The ISA system must also provide support logic for interrupt handling, direct memory access, timers, error handling, the keyboard interface, and configuration RAM. Additionally, the IBM PC/AT and ISA compatibles, provide limited support for bus mastering. The following chapters provide detailed information regarding every aspect of the ISA architecture.

Part 1

The System Kernel

Part 1 covers the background information necessary to an understanding of the material covered in Parts 2 and 3. The focus is on the mechanism used by microprocessors in the Intel X86 family when communicating with memory and I/O devices.

Part 1 includes the following chapters:

- Chapter 1: Intro To Microprocessor Communications
- Chapter 2: Introduction To the Bus Cycle
- Chapter 3: Addressing I/O and Memory
- Chapter 4: The Address Decode Logic
- Chapter 5: The 80286 Microprocessor
- Chapter 6: The Reset Logic
- Chapter 7: The Power-Up Sequence
- Chapter 8: The 80286 System Kernel
- Chapter 9: Detailed View of the 80286 Bus cycle
- Chapter 10: The 80386DX and SX Microprocessors
- Chapter 11: The 80386 System Kernel

Chapter 1

This Chapter

This chapter defines the microprocessor's role in the system, the usage of memory, and defines the role of the address, control and data buses.

The Next Chapter

The next chapter, "Introduction to the Bus Cycle," introduces the concept of the bus cycle and defines the basic sequence of events when the microprocessor uses a bus cycle to communicate with memory or an I/O device.

Instruction Fetch and Execution

General

The microprocessor is an engine with only one task in life — to continually read instructions from memory and execute them (perform the operations specified by the instructions).

An instruction tells the microprocessor to perform one of three basic types of operations:

- Read data from an external device.
- Write data to an external device.
- Perform an internal operation that doesn't involve reading from or writing to the outside world (such as math functions).

Note — As used in this book, the term external device refers to a device external to the microprocessor chip itself. Refer to figure 1-1.

The instructions fetched (read) from memory tell the microprocessor what to do. When a microprocessor-based system is initially powered up, the microprocessor knows what address in memory to fetch (read) its first instruction from. This location is known as the power-on restart address. A group of in-

structions that cause the microprocessor to perform a particular task is referred to as a program. After fetching the first instruction from the restart address, the microprocessor is totally dependent on the program to tell it what to do.

In a properly functioning system, the microprocessor is never idle. At any given moment in time, the microprocessor is reading data from an external device, writing data to an external device, or executing an instruction that doesn't require that a read or write take place (for example, an instruction to add the contents of two of the microprocessor's internal registers together).

The microprocessor communicates with all external devices by reading data from them or writing data to them. The terms read and write are extremely important in any discussion of microprocessors. Always think of reading and writing from the point of view of the microprocessor rather than from that of the device the microprocessor is communicating with. The microprocessor does the reading and writing. Devices are read from and written to by the microprocessor. If you think of the terms read and write from the device's point of view, much of the information in this book will not make sense.

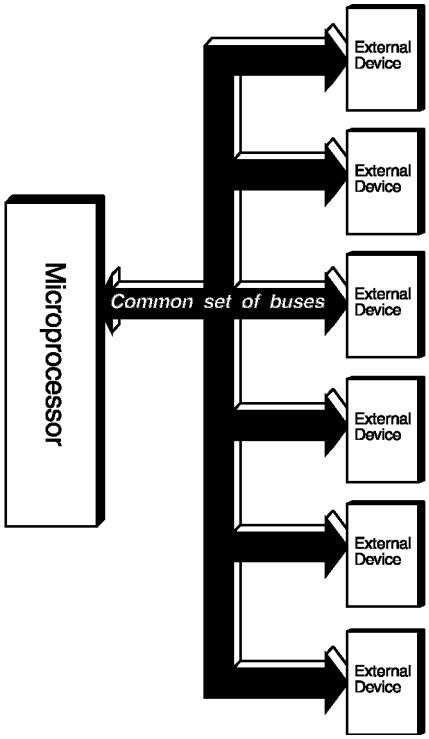


Figure 1-1. Microprocessor's Relationship to External Devices

In-Line Code Fetching

When a microprocessor fetches an instruction from a memory location and executes it, the address of the next instruction may or may not be specified as part of the currently executing instruction.

Case 1: The currently executing instruction doesn't specify the memory address of the next instruction. The microprocessor automatically assumes the next instruction is to be fetched from the next sequential memory location.

or

Case 2: The currently executing instruction specifies the memory address of the next instruction. Execution of the instruction, called a jump instruction, causes the microprocessor to alter its program flow (which is usually sequential). Rather than fetching its next instruction from the

next sequential memory location, the microprocessor fetches it from the memory location specified by the instruction.

After executing a JUMP instruction, the microprocessor resumes fetching instructions from sequential memory locations until another JUMP instruction is executed. Most well-written programs do not contain an excessive number of JUMPs. Statistically then, the microprocessor is executing non-jump instructions the majority of the time. This means that the microprocessor is performing what is commonly referred to as in-line code fetches most of the time.

As an example, refer to figure 1-2. Assume that the microprocessor has just fetched the ADD instruction from location 00000h in memory. The flow of instructions fetched and executed proceeds as follows:

1. The microprocessor executes the ADD instruction fetched from location 00000h. Since it's not a JUMP instruction, the microprocessor fetches its next instruction from memory location 00001h.
2. The microprocessor executes the SUBTRACT instruction fetched from location 1. Since it's not a JUMP, the microprocessor fetches its next instruction from memory location 00002h.
3. The microprocessor executes the MOVE instruction fetched from location 2. Since it's not a JUMP, the microprocessor fetches its next instruction from memory location 00003h.
4. The microprocessor executes the JUMP 10000 instruction fetched from location 3. The JUMP instruction alters program flow. Rather than fetch its next instruction from the next sequential memory location (00004h), the microprocessor fetches it from memory location 10000h.
5. The microprocessor executes the MOVE instruction fetched from location 10000h. Since it's not a JUMP, the microprocessor fetches its next instruction from memory location 10001h.
6. And so on.

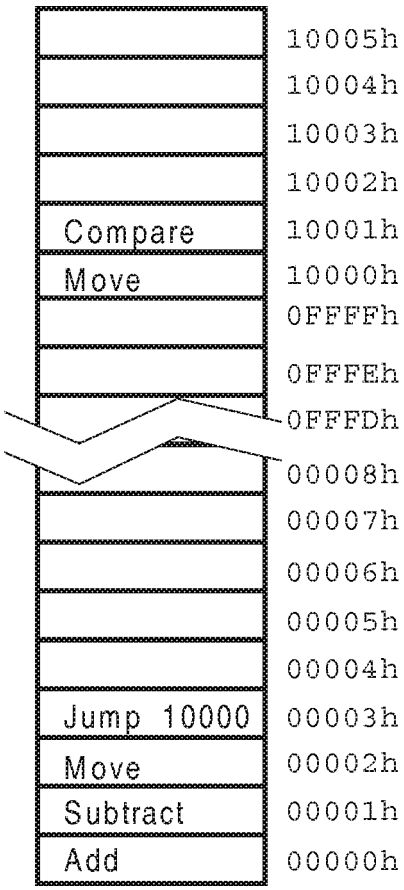


Figure 1-2. Sample Instruction Sequence

Reading and Writing

The microprocessor communicates with external devices under the following circumstances:

- To fetch (read) the next instruction from memory.
- When the currently executing instruction directs the microprocessor to read data from an external device.
- When the currently executing instruction directs the microprocessor to write data to an external device.

The microprocessor communicates with external devices under some other special circumstances as well, but reading or writing occurs for one of the three reasons cited above most of the time.

Type of Information Read from Memory

The microprocessor reads the following types of information from memory:

- Instructions
- Data

The microprocessor reads instructions from memory on an on-going basis. Data, written into a memory location by a previously executed instruction, is read from memory when the microprocessor executes an instruction that tells it to read data from the memory location.

Type of Information Written to Memory

The Intel x86 microprocessors can only write data, not instructions, into memory locations. Instructions are never written into memory by the microprocessor. Instructions (the program) are read from the floppy disk and written into memory by the DMA Controller, not by the microprocessor. Similarly, instructions may be read from the hard disk and written into memory by a bus mastering disk controller, instead of by the microprocessor. DMA and bus mastering are covered in the chapters entitled "Direct Memory Access (DMA)" and "ISA Bus Masters."

Back in the bad old days, programmers would sometimes write programs that modified themselves on-the-fly by writing new instructions into memory and then executing them. This is known as self-modifying code and, generally speaking, programmers who write this type of program should be cast into the pit. Programs of this nature are a nightmare to understand and maintain, even for the person who wrote them.

The Buses

When the microprocessor must read data from or write data to an external device, it uses three sets of signal lines to perform the read or write operation. They are:

- The address bus
- The data bus
- The control bus

The following sections describe the role played by each of these three buses during a read or write operation.

The Address Bus

The microprocessor uses the address bus to identify the external device (and location within the device) that it wishes to communicate with.

The address bus consists of a number of signal lines known as address lines. As an example, the 80286 microprocessor has 24 address lines, referred to as A[23:0]. Refer to figure 1-3.

When the microprocessor must read data from or write data to an external device, it will place the address on the address bus signal lines as a pattern of ones and zeros. This address pattern identifies the external device and the location within the device. In Figure 1-3, the microprocessor has placed address 000105h on the address bus. For a review of the hexadecimal numbering system, refer to the appendix.

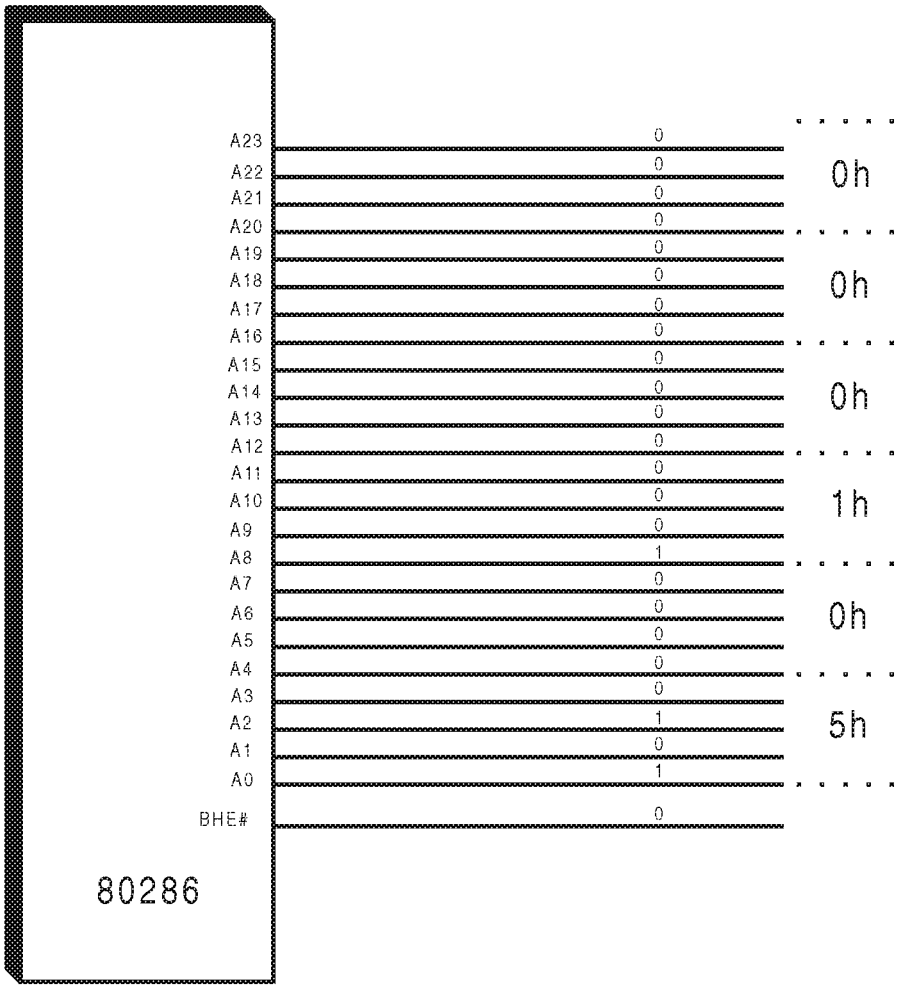


Figure 1-3. The 80286 Address Bus

Each device has an address decoder that detects if the address currently on the address bus is one assigned to its respective device. If it is, the address decoder then informs its associated device that the microprocessor is communicating with it. The addressed device then examines the address further to identify the exact location the microprocessor is communicating with within the device. This subject is covered in the chapter entitled "The Address Decode Logic."

Control Bus – Transaction Type and Synchronization

The control bus basically consists of all the microprocessor's signal lines other than the address bus and data bus lines. The fundamental purposes of the control bus are to:

- Identify the type of transaction (for example, as a read or write transaction).
- Synchronize the fast processor to the slow external devices it is reading from or writing to.

The Data Bus — Data Transfer Path

The microprocessor's only purpose in performing a read or write transaction is to transfer data between itself and the currently addressed external device. The sole purpose of the data bus is to provide a data path for the transmission of the data between the two devices. As an example, the 80286 microprocessor has 16 data bus signal lines, designated D[15:0]. This means that the 80286 microprocessor can read or write 16 bits of information between itself and an external device during a single transaction.

The data bus is often referred to as a bi-directional bus. This doesn't mean that data can be transferred in both directions simultaneously; rather, it means that data can be transferred from the microprocessor to an external device during a write operation, or from an external device to the microprocessor during a read operation. If data is ever sent in both directions at once, something is broken! Refer to figure 1-4.

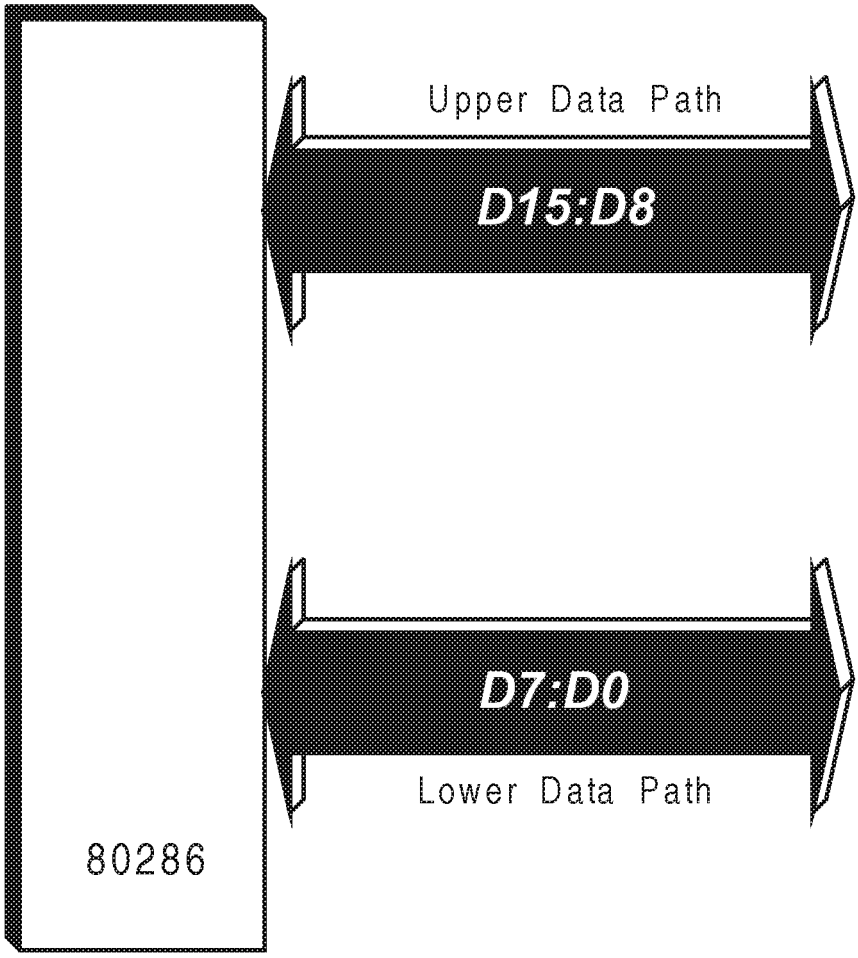


Figure 1-4. The 80286 Data Bus

Note that the 80286 data bus has been illustrated as two 8-bit data paths, rather than as one 16-bit data path. This was done for a very good reason that is covered in the chapter entitled "The 80286 Microprocessor."

Chapter 2

The Previous Chapter

In "Introduction to Microprocessor Communications," the microprocessor's role in the system, usage of memory, and the role of the address, control and data buses were defined.

This Chapter

When the microprocessor must communicate with an external device, it uses its buses. The sequence of events necessary when using the buses to perform a read or write transaction is referred to as a bus cycle. This chapter introduces the microprocessor's bus unit, the concept of a state machine, and defines address time, data time and the wait state.

The Next Chapter

The next chapter, "Addressing I/O and Memory," provides a definition of an I/O device. It defines the method used by the x86 processors to distinguish memory and I/O addresses and the range of memory and I/O addresses available to the 8088, 8086, 80286, 80386, 80486 and Pentium microprocessors.

Introduction

The microprocessor's internal bus unit is a state machine that performs the required bus cycle when the microprocessor must communicate with another device. The concept of the state machine is introduced, as well as the concept of the clock, or timebase, used by the state machine to define the duration of each state.

Automatic Dishwasher – Classic State Machine Example

Any task is easier to perform when divided into logical steps. A state machine is a device (either mechanical, electronic, or software-based) designed to per-

form a task that can be divided into steps. Prior to starting the task, the state machine is said to be idle. When commanded to perform the task, the state machine leaves the idle state and moves through a series of steps, or states. Predefined portions of the overall task are performed during each state. The duration of each state is defined by a clock, or timebase.

In the case of the automatic dishwasher, the dishwasher is idle until you start it. A timer then begins to run, defining the duration of each state the dishwasher must pass through in order to accomplish the overall task of washing dishes. These states are:

- During the first state, the state machine wets down the dishes.
- During the second state, the dishes are soaped.
- During the third state, the dishes are rinsed.
- During the fourth and final state, the dishes are dried.
- The state machine then returns to the idle state.

Most dishwashers also have one or more switches the user can manipulate to alter the sequence of states or possibly to cause the machine to execute a particular state more than once. A perfect example would be the "Pot Scrubber" button. When pressed, this might cause the state machine to execute the "rinse" state two times instead of one.

The System Clock – a Metronome

Every x86 microprocessor has an input called CLOCK (or a similar name). This signal is produced when a voltage is applied to a crystal oscillator. It then begins to generate an electrical signal of a specific frequency. In essence, the oscillator acts as a highly accurate electronic tuning fork. The signal looks like Figure 2-1.

Figure 2-1. Crystal Oscillator Output

All microprocessors perform their operations in a highly organized, predefined fashion. Portions of each task are always performed during predefined time slots. The duration of each time slot is defined by the output of the crystal oscillator.

Indirectly, the 80286 and 80386 microprocessors use the CLOCK input to define the length of a time slot. They divide the frequency of the CLOCK input by two to yield an internal timebase referred to as the Processor Clock (PCLK). The CLOCK input is referred to as a double-frequency clock because its frequency is double that of the required PCLK. The 486DX microprocessor, on the other hand, uses the CLOCK input as the PCLK without dividing it. Clock-doubling or clock-multiplying processors, such as the DX4, use phase-locked loops or similar technology to multiply (instead of divide) the CLOCK signal to produce a PCLK that is faster than CLOCK.

PCLK is the real metronome that defines the duration of the time slots during which the microprocessor performs a task or a pre-defined portion of a task. Refer to Figure 2-2.

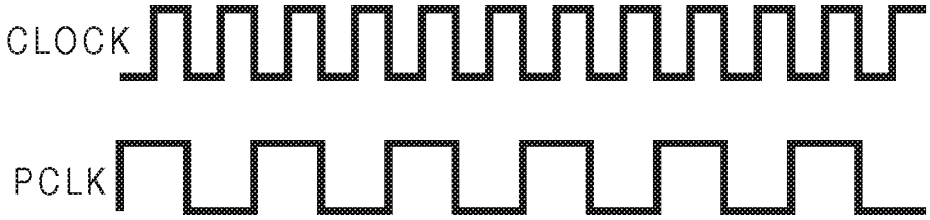


Figure 2-2. Relationship of CLOCK and PCLK

As illustrated in figure 2-2, PCLK is half the frequency of CLOCK. When a PC manufacturer refers to the operating speed of their computer as 8MHz (megahertz, or million cycles-per-second), 10MHz, etc., they are referring to the microprocessor's PCLK frequency, not that of CLOCK.

Microprocessor's Bus Cycle State Machine

When the microprocessor performs a read or a write operation, it initiates a sequence of events called a bus cycle.

During a bus cycle, the microprocessor places the address on the address bus, sets the control bus lines to indicate the type of transaction (such as a memory read or I/O write bus cycle), and transfers the data between the target location and itself. This happens in a very orderly fashion, with each step occurring at the proper point during the appropriate time slot.

x86 microprocessor chips include a subsystem called a bus unit, tasked with the job of running bus cycles when required. The bus unit is a state machine

that is stepped through its various states by the PCLK signal. The duration of each state is one cycle of PCLK.

Refer to figure 2-3. As an example, if the CLOCK input frequency is 40MHz (40 million cycles per second), the PCLK frequency is half that frequency, or 20MHz. In order to determine the duration of one cycle of PCLK, just divide 20 million cycles-per-second into one second. In this example, a PCLK cycle is 50ns in duration (50 nanoseconds; 0.000000050 seconds or 50 billionths of a second). Since the bus unit state machine's states are each one PCLK in duration, this means that each bus unit state is 50ns in duration.

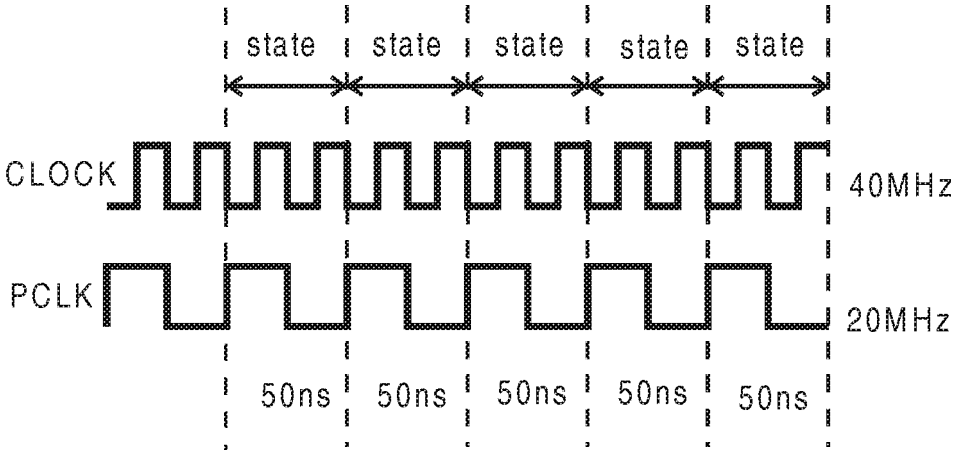


Figure 2-3. Each Cycle of PCLK Defines the Duration of a State

If not currently engaged in a bus cycle (a read or a write operation), the bus unit state machine is said to be in the idle state. It remains in the idle state until the microprocessor must perform a bus cycle.

Address Time

When the microprocessor must perform a read or a write, its bus unit initiates a bus cycle. The bus unit leaves the idle state and enters a state that will be referred to in this text as address time. Although Intel uses a different name for this state, address time is the name that will be used in this text for clarity's sake. During address time, one PCLK cycle in duration, the microprocessor places the address on the address bus and the bus cycle definition (type of bus cycle) on the control bus lines.

Data Time

After performing all actions required during address time, the bus unit state machine immediately enters the state we will refer to as data time. As with address time, Intel uses a different name for this state, but, for clarity's sake, data time is the name that will be used in this text. During data time, one PCLK cycle in duration, the microprocessor expects the data to be transferred between itself and the currently addressed device. At the end, or trailing edge, of data time (refer to reference point 1 in figure 2-4), the microprocessor samples (tests the state of) its READY# input to see if the currently addressed device is ready to complete the bus cycle. If the READY# input is asserted (low, as indicated by the pound sign), the bus unit state machine terminates the bus cycle.

If a read were in progress, the bus unit interprets an asserted level on READY# to mean that the currently addressed device has placed the requested data onto the data bus for the microprocessor. The microprocessor reads the data from the data bus and terminates the bus cycle.

If a write were in progress, the bus unit interprets an asserted level on READY# to mean that the currently addressed device has accepted the data being written to it. The microprocessor terminates the bus cycle. The chapter entitled "The 80286 System Kernel: the Engine" provides a detailed operational description of the Ready logic. Figure 2-4 illustrates a bus cycle consisting of address time and data time.

The 0-wait-state bus cycle is the fastest type of bus cycle the 80286, 80386, 80486 and Pentium microprocessors are capable of performing. In other words, the fastest x86 bus cycle takes two "ticks" (cycles) of PCLK. If this were an 80386 running at 20MHz (PCLK speed), a 0-wait-state bus cycle would take 100ns (50ns each for address time and data time).

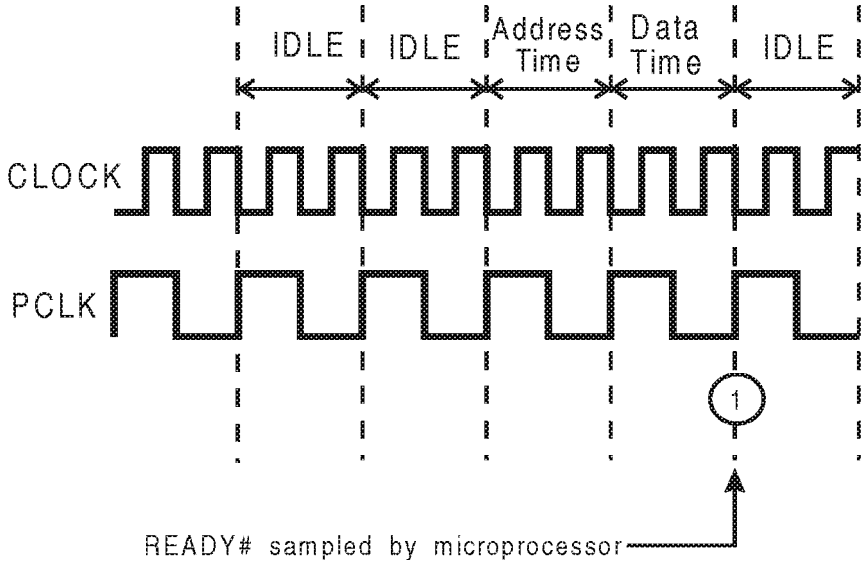


Figure 2-4. Bus Cycle Consisting of Address Time and Data Time

The Wait State

All bus cycles consist of at least two states: address time and data time. At the end of address time and data time, however, not every device will be ready to end a bus cycle. Some devices are slower than others in responding to the microprocessor's data transfer requests, and are referred to as slow access devices.

Refer to figure 2-5. When the microprocessor samples READY# at the end of the first data time (see reference point 1 in figure 2-5), READY# may not be asserted. This is the currently addressed device's way of saying "Whoa! Wait up thar big fella!" When the READY# line is sampled deasserted, it causes the bus unit state machine to re-enter the data time state again.

During this additional data time, the microprocessor keeps all of its outputs the same. In other words, nothing changes; the microprocessor just waits one tick of PCLK. This is referred to as a wait state. At the end of this additional data time, the microprocessor samples the READY# input again (see reference point 2 in figure 2-5) to see if the currently addressed device is ready to complete the transaction.

If READY# is sampled deasserted again, the microprocessor inserts another data time or Wait State into the bus cycle. When READY# is sampled asserted at the end of data time, the microprocessor terminates the bus cycle.

By holding off the READY# line, the currently addressed device can force the microprocessor to stretch out the bus cycle as long as necessary until the device is ready to complete the data transfer. The currently addressed device deasserts READY# until it's ready to end the transaction.

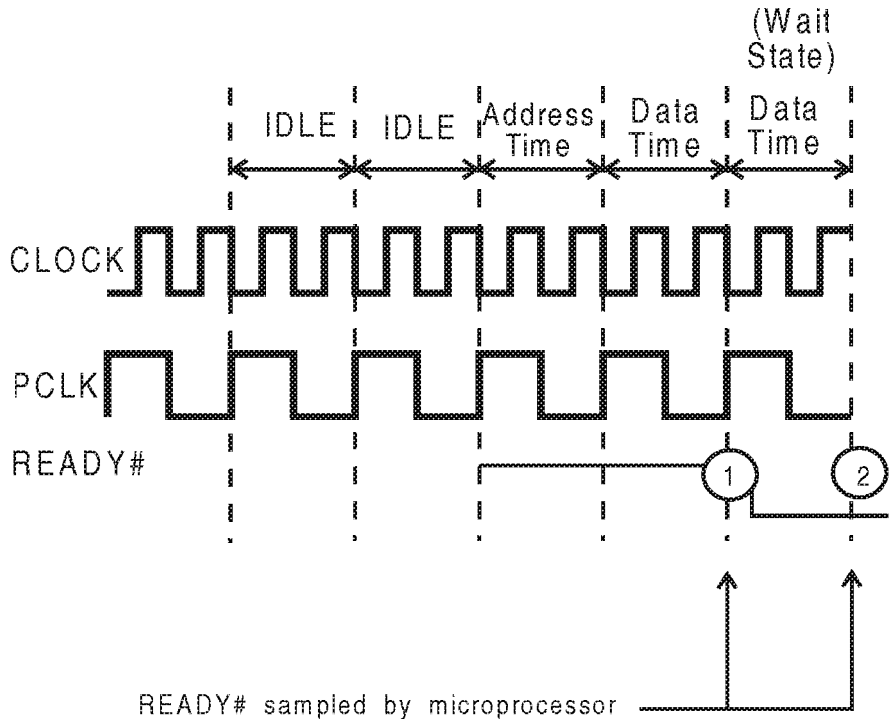


Figure 2-5. Bus Cycle with One Wait State Inserted

Chapter 3

The Previous Chapter

The previous chapter, "Introduction To the Bus Cycle," provided a basic description of the microprocessor's communications scheme. Using the three buses, the microprocessor performs bus cycles to read information from or write information to memory or I/O devices.

This Chapter

This chapter explores the addressing scheme used by the x86 processors. It defines the addressing range provided by the 8086, 8088, 286, 386, 486 and Pentium microprocessors. A basic definition of the term I/O device is also provided.

The Next Chapter

Having covered the addressing scheme used by the x86 processors in this chapter, the next chapter, "The Address Decode Logic," introduces the concept of the address decoder. Several examples of address decoders are examined in detail.

Evolution of Memory and I/O Address Space

The Intel 8080 microprocessor is the ancestor of the entire x86 microprocessor family. Many of the family's characteristics stem from this common ancestor.

Intel 8080 Microprocessor Address Space

The 8080 microprocessor's address bus consists of 16 address lines, A[15:0]. With 16 address lines, the microprocessor can place any address pattern on the address bus from all zeros (0000h) to all ones (FFFFh). In other words, the 8080 can place any one of 65536 addresses on the address bus and can therefore address any one of 65536 locations located in external devices. In the computer

industry, 65536 is referred to as 64K (K stands for Kilo, Greek for 1,000, but in the computer industry, kilo is 2^{10} or 1,024). Refer to figure 3-1. This means that the designer of an 8080-based system can include memory devices with a total of no more than 64K locations in the system.

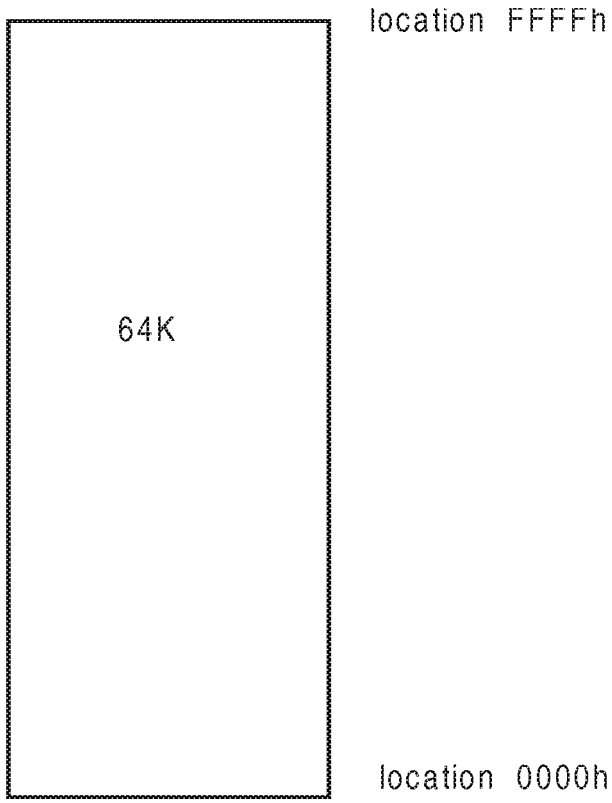


Figure 3-1. Single 64K Address Space

In addition to memory devices used for program and data storage, the designer must also use a number of locations for input/output ports (I/O ports) used to communicate with I/O devices. In simple terms, any device that the microprocessor can read data from or write data to that isn't a memory device (RAM or ROM) is an I/O device. Implementing these ports certainly is necessary, but depletes the number of addressable locations available to be assigned to memory devices.

Rather than use up already scarce memory locations to implement I/O devices, Intel provided two new instructions to the 8080 microprocessor: IN and OUT, and added a way for the 8080 to indicate to the system that the 8080 wishes to communicate with an I/O device instead of a memory device. The system typically included an Intel 8228 controller chip which detected the 8080's intention to communicate with an I/O device.

Some of the functionality of the 8228 was incorporated in the 8085 microprocessor, a successor of the 8080. The 8085 has 16 address lines, as does the 8080, so the 8085 can address 64K locations.

The 8085 has a new output pin called IO/M#. In Intel signal names, the "/" character should always be read as the English word "or." IO/M# therefore stands for "IO or Memory." Since there is no pound sign after the "IO", a high on this signal line means an I/O address is being output by the microprocessor. The pound sign after the "M" portion of the signal name indicates that a memory address is present on the address bus when IO/M# is low.

The addition of the IO/M# pin actually created two separate address spaces (or maps), one for Memory and the other for I/O. In essence, the IO/M# pin "points" to the appropriate address space when the microprocessor places an address on the address bus.

Refer to figure 3-2. As previously mentioned, the 8085 provides 64K of memory address space. However, due to the limited number of I/O devices typically found in systems, Intel only implemented 256 I/O locations.

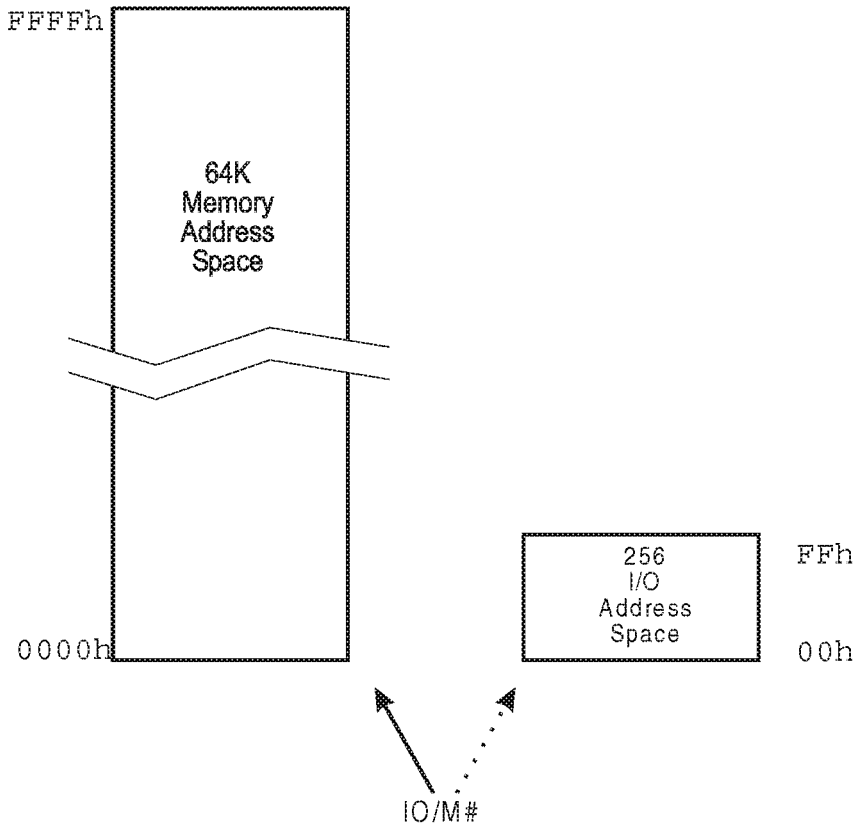


Figure 3-2. The 8085's Memory and I/O Address Spaces

As an example, if the microprocessor places address 0010h on the address bus and places a high on IO/M#, it is addressing I/O location 0010h, not memory location 0010h. On the other hand, if the microprocessor places address 0010h on the address bus and places a low on IO/M#, it is addressing memory location 0010h, not I/O location 0010h.

In addition to the IO/M# pin, Intel also added two I/O instructions, IN and OUT, to the 8085's instruction set. When executed, the IN instruction causes the microprocessor to perform a read from the I/O address specified by the programmer. This address is placed on the address bus and the IO/M# pin is set high, thereby indicating the presence of an I/O address on the address bus to external logic. In the same way, the OUT instruction also causes the microprocessor to place an I/O address on the address bus with IO/M# set high to indicate the I/O address it wishes to write data to.

A separate instruction, MOV, is used by the programmer to specify a memory address to move data to or from. When executed, the MOV instruction causes the specified memory address to be placed on the address bus and the IO/M# pin to be set low, indicating the presence of a memory address.

These changes allowed the designer to implement I/O ports without encroaching on memory space.

8086 and 8088 Microprocessor Address Space

When Intel designed the 8086 and 8088 microprocessors, they increased the number of address lines from 16 to 20. This means that the 8086 and 8088 microprocessors can place any address from 00000h to FFFFFh on the address bus, giving them an address range of 1,048,576 or 1M locations. "M" stands for the Greek prefix Mega, meaning large or great. In physics, M is 1,000,000, but in the computer industry, M is 2^{20} or 1,048,576. Some computer companies use M to mean 1,000,000 when referring to disk storage while still using M to mean 2^{20} when referring to memory size, so beware!

As with the 8085 microprocessor, the 8086 and 8088 microprocessors have a pin to indicate memory or I/O addresses. Note that the pin name and function is slightly different (or completely different, depending on your point of view). It's named M/IO#, meaning that the 8086 and 8088 set the M/IO# pin low when executing an IN or OUT instruction. The 8086 and 8088 set the M/IO# pin high when executing a MOV instruction to access memory. This is just the opposite of the way the 8085 did it.

For the 8086 and 8088 microprocessors, Intel also increased the size of the I/O address space to 64K. The programmer may specify any memory address within the 1M memory address range, but is restricted to I/O addresses in the first 64K locations of address space (0000h to FFFFh). There were two reasons for this decision:

1. The 8086 and 8088 microprocessors remain backward-compatible with the I/O instructions in the 8080 instruction set, although these instructions will not take full advantage of the processors address capability.
2. Virtually no designer requires more than 64K I/O ports (locations) to implement a full complement of I/O devices, no matter how complex they might be.

If you take the example of an IBM PC/AT fully loaded with every conceivable type of add-in I/O expansion boards, no more than 768 I/O ports are used. This is less than 1/64th of the total I/O space available. As previously noted, 64K I/O locations are more than enough for anyone's needs. Figure 3-3 illustrates the 8086 and 8088 microprocessor memory and I/O address space.

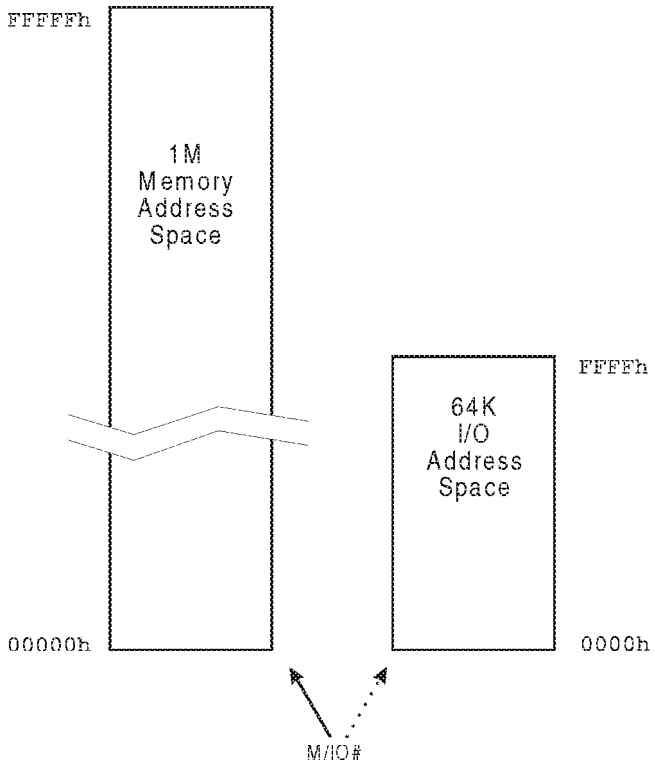


Figure 3-3. 8086 and 8088 Memory and I/O Address Space

286 and 386SX Address Space

With the introduction of the 286 microprocessor, the number of address lines was increased from 20 to 24. Refer to figure 3-4. Although this increases the size of the memory address space available to the designer from 1M to 16M (2^{24}; 16,777,216 locations), the I/O space is still restricted to 64K locations. The 386SX microprocessor also has 24 address lines, allowing access to 16M of memory and 64K I/O ports.

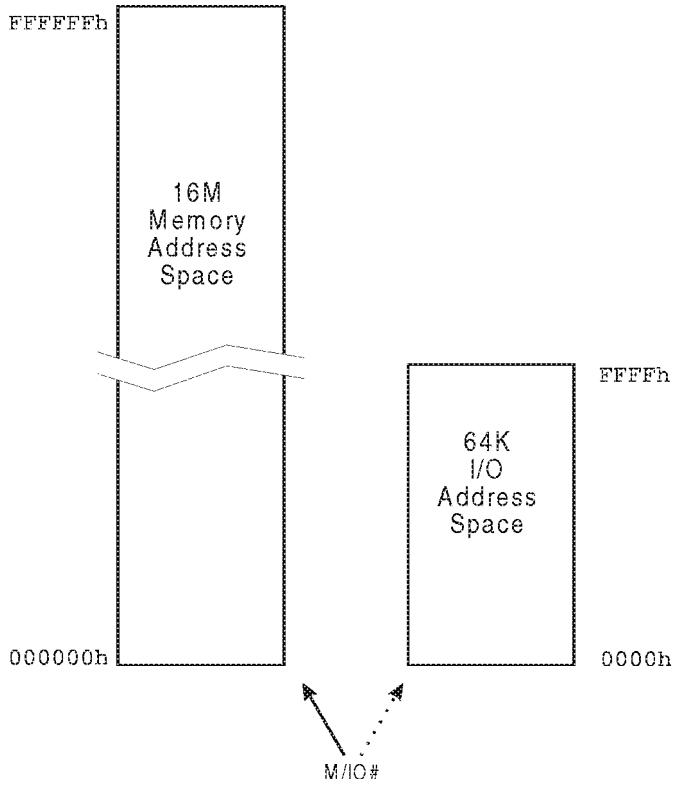


Figure 3-4. 286 and 386SX Memory and I/O Address Space

386DX, 486 and Pentium Processor Address Space

The 386DX, 486 and Pentium microprocessors each have 32 address lines, providing 4G possible memory locations ("G" is borrowed from the Greek word Giga, meaning giant; used as a prefix in physics, it means 1,000,000,000, but in the computer industry, it means 2^{30} or 1,073,741,824; thirty-two address lines yield 2^{32} or 4,294,967,296 locations). Refer to figure 3-5. The I/O space is still restricted to 64K locations.

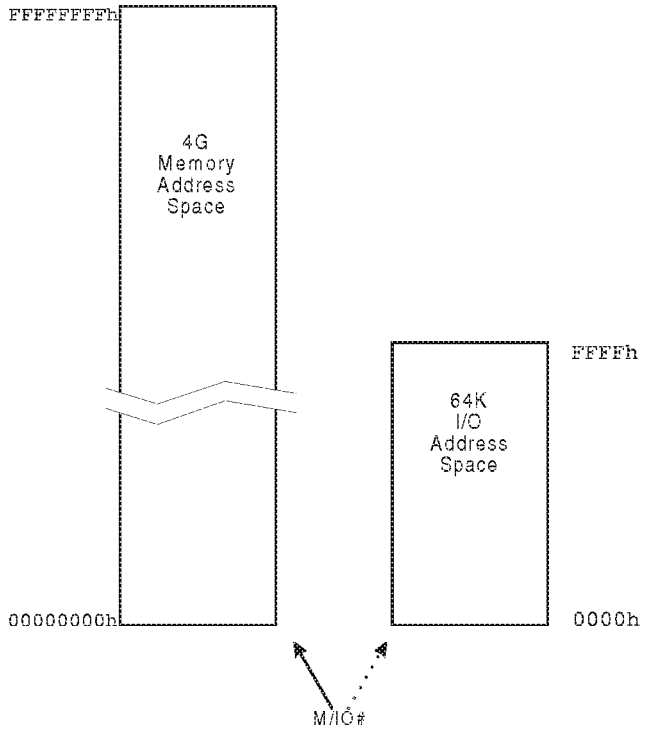


Figure 3-5. 386/486/Pentium Processor Memory and I/O Address Space

Memory Mapped I/O

Designers can also map I/O devices into memory space if they so choose. Care must be exercised when mapping I/O devices into memory so that the operating system and application software do not conflict with the address space chosen for the I/O device. For this reason, many designers map I/O devices in the high ranges of memory address space so that it is well beyond the maximum memory range supported by most PCs.

The I/O Device

I/O Ports fall into these following basic categories:

- **Command Port**. The data written to this port by the programmer acts as a command to the associated I/O device. The individual bits of data written

to the command port control different aspects of the device's operation. As an example, one bit in the data written to the keyboard command port could be a command to enable or disable the Keyboard.

- **Status Port**. Data read from this port represents the current status of the associated I/O device. The individual bits of data read from the status port reflect different aspects of the device's current status. As an example, the state of a bit in the data read from the printer status port could indicate that the printer is out of paper.
- **Data Port**. When an I/O device has data to be transferred to the microprocessor, the programmer reads the data from the device's data port. Similarly, when an I/O device is ready for the next character (for example, a character for the printer), the programmer writes the character to the device's data port.

The number of I/O locations needed to control, sense the status of, read data from, or write data to a particular type of I/O device is device-dependent. Simple I/O devices may only require a few I/O locations. As an example, the parallel port requires only one location each for its command port, status port, and data port.

More complex I/O devices such as the VGA display interface, can require many more locations.

Chapter 4

The Previous Chapter

The previous chapter, "Addressing I/O and Memory," explored the addressing scheme used by the x86 processors. It defined the addressing range provided by the 8086, 8088, 80286, 80386, 80486 and Pentium microprocessors, and provided a basic definition of the term "I/O device."

This Chapter

This chapter describes the function performed by address decoders, discusses the address decoder design process, defines data bus contention, and provides a detailed analysis of two example address decoders found in ISA systems.

The Next Chapter

The next chapter, "The 80286 Microprocessor," provides a detailed description of the internal hardware architecture of the 80286 microprocessor, describes its register set, and supplies definitions of real and protected modes.

The Address Decoder Concept

Refer to figure 4-1. When the microprocessor must read data from or write data to an external device, it places the address on the address bus. Every device that can be read from or written to has an associated piece of logic called an address decoder. The address decoder selects its associated device when an address within its defined range is detected.

In most cases, the selected device contains multiple internal locations. When selected, the addressed device examines the address to identify the exact internal location with which the microprocessor wishes to perform a data transfer.

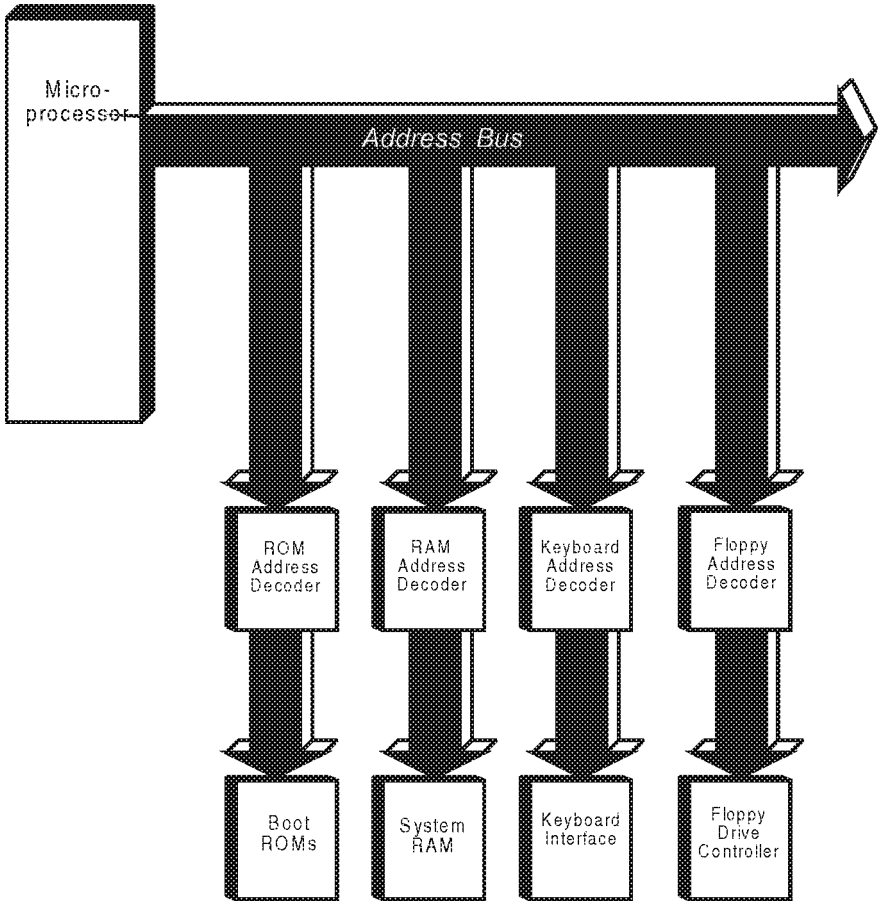


Figure 4-1. Relationship of Address Bus, Address Decoders and Devices

A memory or I/O chip does not respond to any pre-assigned address range. Rather, the device's address decode logic informs it that the address currently on the address bus is assigned to one of its internal locations by asserting the device's chip-select input pin.

The overall range of memory and I/O locations addressable by a particular microprocessor are commonly referred to as address maps. The design of its associated address decoder, not that of the memory or I/O device itself, defines where a device lives within the overall memory or I/O address map of a microprocessor.

Data Bus Contention (Address Conflicts)

Under no circumstances should the address decoders for two devices installed in a system be designed in such a fashion that they both detect the same address ranges. This would cause data bus contention.

Every expansion card installed in an ISA system has its own address decode logic that defines the address range the card responds to. Address conflicts are a common problem in the PC world. Consider the problem that arises if the address decoders on two expansion cards are both designed to detect the same address range.

As an example, assume that two cards respond to the I/O address range 03F0h to 03F7h. If the microprocessor executes an instruction that causes a read from I/O location 03F1h, both devices are chip-selected by their respective address decoders. Each then places the contents of its respective location 03F1h onto the common data bus simultaneously. One card may be driving a particular data bus signal line high while the other card is driving the same data bus signal line low. In this situation, we have two separate current sources trying to drive five volts and zero volts onto the same piece of wire.

At best, this situation will cause garbled data and, at worst, hardware damage. The proper term for this condition is data bus contention, a problem quite common in the PC world. A user populates a PC with many cards purchased from various manufacturers. The prospect of an address conflict is very real.

In order to permit resolution of address conflicts, a card designer should allow the user to easily alter the address range the board responds to. In ISA systems, this is usually accomplished with configuration switches (or jumpers) on the board. By changing the switch setting, the user causes the card's address decoder to detect a different address range, thereby resolving the conflict.

How Address Decoders Work

The address decoder function is similar to that performed by the post office. Because it more or less identifies a neighborhood, the zip code is analogous to the high-order part of the address (upper address bits). The low-order portion of the address identifies the exact location within the neighborhood.

In much the same way, the address decoder associated with a device examines the high-order part of the address being output by the microprocessor. It de-

termines if the address is within the range assigned to its device. If it is, the address decoder chip-selects the device. The chip-selected device then examines the lower part of the address to determine the exact location within it the microprocessor is addressing.

It should be noted that an address decoder doesn't have to look at the entire address to determine that the current address is within the address range assigned to its respective device. If you were traveling in a car and were told to keep an eye out for the two thousand block, you would only need to look at the thousands digit of the address to determine if it was within the desired range. The lower digits would be insignificant to you.

The following two sections describe example address decoders and how they function. The first example is that of the ROM address decoder in an IBM PC/XT. The second example is that of the address decoder that detects addresses assigned to the I/O devices found on a typical ISA system board.

Example 1– PC and PC/XT ROM Address Decoder

Background

The original IBM PC was designed around the 8088 microprocessor. With an address bus consisting of 20 lines, designated A[19:0], the 8088 could address any one of 1,048,576 memory locations. The designers chose to populate this memory map as illustrated in Figure 4-2.

The memory address range from F0000h to FFFFFh was assigned to the system board ROM. This ROM is frequently referred to as the boot or POST/BIOS ROM. The 64K addresses from F0000h to FFFFFh are set aside for the boot ROM. It should therefore be selected whenever the uppermost digit on the address bus is Fh and the M/IO# pin is high (indicating the presence of a memory address on the address bus). Table 4-1 illustrates the state of the address lines when any memory address in this range is accessed.

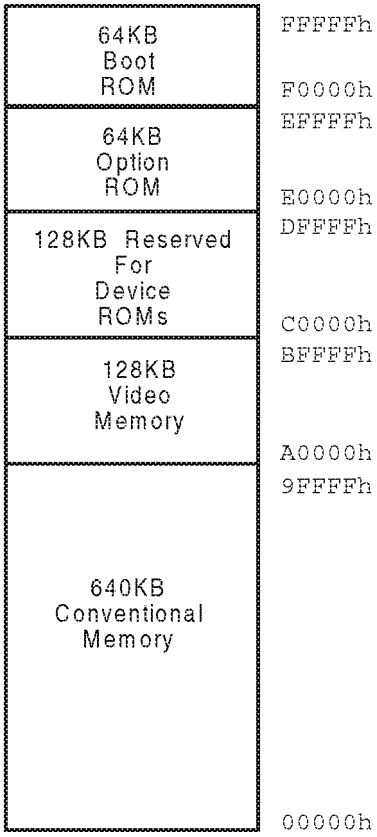


Figure 4-2. PC Memory Address Map

Table 4-1. State of the Address Lines When Any Memory Address in Range F0000h to FFFFFh Is Addressed

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
F				X				X				X				X			

In addition, memory address range E0000h to EFFFFh was assigned to the optional ROM that could be installed in the empty ROM socket provided on the system board. This capability was provided to allow the installation of a ROM containing the BIOS routines and interrupt service routines for optional devices that could be added to the machine. This add-in ROM is commonly referred to as the option ROM. The 64K addresses from E0000h to EFFFFh are set aside for the option ROM. It should therefore be selected whenever the uppermost digit on the address bus is Eh and the M/IO# pin is high (indicating the presence of

a memory address on the address bus). Table 4-2 illustrates the state of the address lines when any memory address in this range is accessed.

Table 4-2. State of the Address Lines when any Memory Address in Range E0000h to EFFFFh Is Addressed

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
E				X				X				X				X			

An address decoder that detects addresses for either ROM must look for any memory address that has address lines A[19:17] all set to a one. That means its high-order digit is either Eh or Fh. To discriminate between addresses for the boot and option ROMs, the decoder would then look at address line A16. If it is low, the high-order digit of the address is Eh and the option ROM should be chip-selected. If address line A16 is high, however, the high-order digit of the address is Fh and the Boot ROM should be chip-selected.

The PC/XT ROM Address Decode Logic

The PC/XT ROM address decoder pictured in figure 4-3 consists of gates U1, U2, U3 and U4.

The ROM address decoder's inputs consist of address bits A[19:16] from the microprocessor. It has two chip-select outputs, CS EXXXX# and CS FXXXX#. CS EXXXX# is attached to the chip-enable (CE) input of the option ROM, while CS FXXXX# is attached to the chip-enable (CE) input of the boot ROM.

To chip-select the option ROM, the signal CS EXXXX# (chip-select E0000h to EFFFFh range) would have to be asserted (low, as indicated by the pound sign). Pins 1 and 2 of gate U4 must both be high in order to set U4-3, CS EXXXX#, low. U4-2 will be high if pins 1, 2 and 4 of U2 are high. In addition, address bus lines A[19:17] must all be ones.

A zero on A16 is inverted (flipped to its opposite state) by U1, applying a one to U4-1. U3-1 will be low, keeping U3-3 high and preventing the boot ROM from being chip-selected.

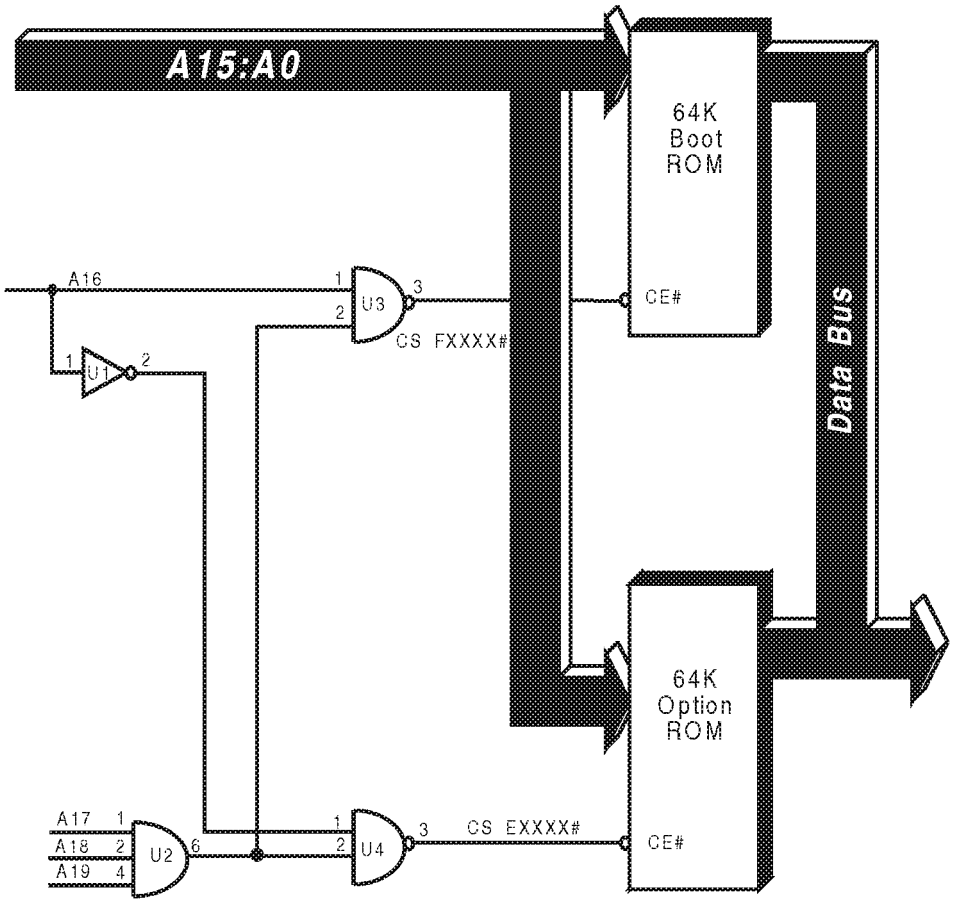


Figure 4-3. Example PC/XT ROM Address Decoder

Since the highest I/O address is FFFFh, there is no conflict between the ROM address decoder and the I/O address decoders. The ROM address decoder is looking for addresses that are well above the highest I/O address. The designer of the address decoder can therefore ignore the M/IO# signal from the microprocessor because the 8086, 8088 and 286 microprocessors set the upper address lines (A16 and higher) to 0 when M/IO# is low.

Address bus lines A[15:0] aren't connected to the ROM address decoder and are considered to be "don't care" bits (from the ROM address decoder's view point).

Notice, however, that the "don't care" bits are connected to the boot and option ROMs. Neither of the ROMs will look at these address bus lines unless its respective chip-select signal is asserted by the ROM address decoder.

Table 4-2 illustrates the state the address bus inputs (as defined above) must assume in order to chip-select the option ROM.

Address bus lines A[15:0] are shown as "x", representing a "don't care" condition (can be either a 1 or a 0 because the ROM address decoder doesn't look at them). If the high-order digit of the memory address on the address bus is an Eh, the address is considered to be for the option ROM and CS EXXXX# is asserted (low).

This enables the option ROM, which then examines the low-order portion of the memory address, A[15:0], to identify the exact location in the ROM that the microprocessor wishes to read data from. The option ROM then places the contents of the respective location on the data bus to be sent back to the microprocessor. The microprocessor reads the data from the data bus, completing the memory read operation.

To chip-select the boot ROM, the signal CS FXXXX# (chip-select F0000h to FFFFFh range) must be asserted (low, as indicated by the pound sign).

Pins 1 and 2 of gate U3 must both be high in order to set U3-3, CS FXXXX#, low. U3-2 will be high if pins 1, 2 and 4 of U2 are high. In addition, address bus lines A[19:17] must all be ones.

A one on address bus line A16 places a high on U3-1. A16 is inverted by U1, applying a low to U4-1, preventing the option ROM from being chip-selected by the ROM address decoder.

Address bus bits A[15:0] aren't connected to the ROM address decoder and are considered to be "don't care" bits (from the ROM address decoder's view point).

Table 4-1 illustrates the state the address bus lines (as defined in the steps above) must assume in order to chip-select the boot ROM.

Address bus bits A[15:0] are shown as "x", representing a "don't care" condition (can be either a 1 or a 0 because the ROM address decoder doesn't look at them). If the high-order digit of the memory address on the address bus is an Fh, the address is for the boot ROM and CS FXXXX# is asserted (low).

This will enable the boot ROM, which then examines the low-order portion of the memory address, A[15:0], to identify the exact location in the ROM the microprocessor wishes to read data from. The boot ROM then places the contents of the respective location on the data bus to be sent back to the microprocessor. The microprocessor reads the data from the data bus, completing the memory read operation.

Example 2 – System Board I/O Address Decoder

When the microprocessor addresses an I/O location, only address bus lines A[15:0] are used. This is because the microprocessor is restricted to I/O addresses in the range 0000h to FFFFh. The 8086, 8088 and 286 microprocessors always set address bus lines above A15 to all zeros when placing I/O addresses on the address bus. The 80386 microprocessor is slightly different. Intel reserved some of the 80386's I/O addresses above FFFFh for use with the 80387 numeric coprocessor. This is covered in the chapter entitled "The Numeric Coprocessor."

This example describes an I/O address decoder that decodes the chip-selects for many of the I/O devices that are integrated (included) on ISA system boards.

The address decoder pictured in figure 4-4 detects most of the I/O addresses assigned to the I/O devices integrated onto the system board. The overall I/O address range detected is from 0000h to 00FFh. The address decoder further breaks this address range down into sub-ranges of 32 locations each. When an I/O address within one of these sub-ranges is detected, the respective chip-select line is asserted (set low) to select the addressed I/O device. The sub-ranges and the I/O devices associated with them are listed in Table 4-3. The I/O devices referred to are described in subsequent chapters.

Table 4-3. System Board I/O Address Sub-Range Assignments

I/O Sub-Range	Device(s) Assigned to Sub-Range
0000h to 001Fh	DMA controller (DMAC) number 1
0020h to 003Fh	Programmable interrupt controller (PIC) number 1
0040h to 005Fh	Programmable interval timers (PIT) 1 and 2
0060h to 007Fh	System control port B, keyboard interface, NMI control and battery-backed RAM ports
0080h to 009Fh	DMA page registers, PS/2 compatibility port
00A0h to 00BFh	Programmable interrupt controller (PIC) number 2
00C0h to 00DFh	DMA controller (DMAC) number 2
00E0h to 00FFh	Intel 80X87 numeric coprocessor

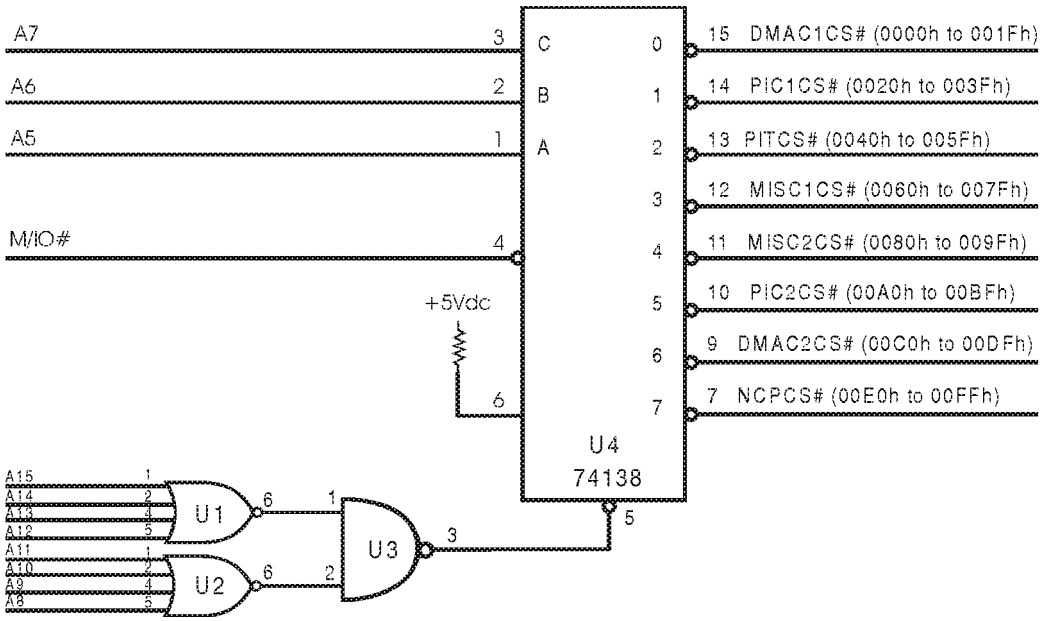


Figure 4-4. System Board I/O Address Decoder

The system board I/O address decoder consists of three gates and a 74138 three-to-eight line decoder. The operation of the 74138 is described in the following paragraphs.

When the microprocessor begins an I/O read or write operation, it places the I/O address on the address bus and sets the M/IO# line low to indicate the

presence of an I/O address. The system board I/O address decoder responds to the address as described in the following paragraphs.

The enabling of the 74138 is fundamental to the operation of the address decoder. This chip has 3 enable inputs, pins 4, 5 and 6. All three must be asserted to enable the chip:

- Pin 6 does not have a state indicator (a bubble), meaning a high is needed on this pin to enable the 74138. The pin 6 enable is tied to a high (+5Vdc), so this enable is always present.
- Pin 4 has a state indicator, indicating it requires a low input to provide an enable to the chip. The pin 4 enable input is tied to the M/IO# signal line and is therefore only asserted when the microprocessor places an I/O address on the address bus.
- Pin 5 has a state indicator, indicating it requires a low input to provide an enable to the chip.

Gates U1 and U2 are used to detect all zeros on address lines A[15:8] (an I/O address in the range 0000h to 00FFh). When all of these address lines are low, both gates output highs to U3. This causes U3 to output a low, providing the final enable to the 74138. Only I/O addresses in the range 0000h to 00FFh cause the 74138 to receive all three enables, enabling it to decode the low-order address lines to assert the appropriate chip-select line.

Assuming that the I/O address currently being output by the microprocessor is within the range 0000h to 00FFh, the 74138 is enabled and can perform its job.

The 74138 is called a three-to-eight line decoder because, when enabled, it uses its A, B and C inputs to select one of its eight outputs to assert (they are asserted when low, as indicated by the state indicator on each output). The 74138 uses the binary-weighted value of the three lines on its inputs to select an output to assert. The A input has a value of 1, the B input a value of 2, and the C input a value of 4. Table 4-4 indicates the output selected for each possible set of inputs.

Table 4-4. 74138 Output Selection Criteria

C	B	A	Output Selected	Output Pin Number
0	0	0	0	15
0	0	1	1	14
0	1	0	2	13
0	1	1	3	12
1	0	0	4	11
1	0	1	5	10
1	1	0	6	9
1	1	1	7	7

Assume that the microprocessor is currently performing a read or write with I/O address 0012h. Address 0012h is placed on the address bus and the microprocessor's M/IO# output is set low to indicate it's an I/O address. Figure 4-5 illustrates how this address appears on the address bus.

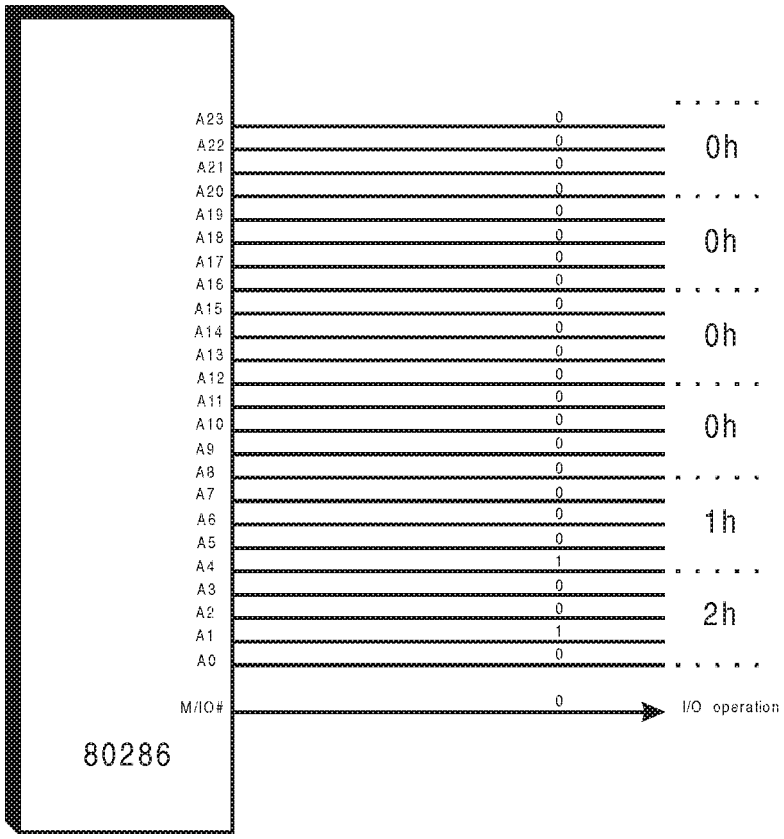


Figure 4-5. Microprocessor Placing I/O Address 0012h on the Address Bus

Since address bus lines A[15:8] are all zeros and an I/O operation is in progress (as indicated by a low on M/IO#), the 74138 is enabled.

Note that address bus lines A[7:5] are all zeros in this address. Since these three address lines are connected to the 74138's inputs and have a binary-weighted value of 0, the 74138 will assert (set low) its "0" output, pin 15. This asserts the chip-select line for DMA controller number one (DMAC1CS# goes low), enabling DMA controller number one. The DMA controller then uses address lines A[4:0] to identify the exact location within it the microprocessor wishes to perform a data transfer with.

To summarize, address lines A[15:8] set to all zeros during an I/O operation enables the system board I/O address decoder. The address decoder then uses address lines A[7:5] to select the chip-select line to assert.

Address lines A[4:0] aren't connected to the system board I/O address decoder, so they are "don't care" bits to the address decoder.

This address decoder is designed to detect 32-location I/O address blocks within the overall address range from 0000h to 00FFh. Any address within a given block selects the associated device's chip-select line. The designer could implement up to 32 I/O locations for each device to control it, sense its status and read data from or write data to it. A read or write to any location in the 32-location block assigned to the device by the address decoder causes the device to be chip-selected. The device itself can then use address lines A[4:0] to select the exact location within itself that the microprocessor wishes to talk to.

Table 4-5 illustrates the I/O address ranges assigned to each of the device chip-selects.

Table 4-5. I/O Address Range Assigned to Each System Board Device

I/O Sub-Range	Device(s) Assigned to Sub-Range
0000h to 001Fh	DMA controller (DMAC) number 1
0020h to 003Fh	Programmable interrupt controller (PIC) number 1
0040h to 005Fh	Programmable interval timers (PIT) 1 and 2
0060h to 007Fh	System control port B, keyboard interface, NMI control and battery-backed RAM ports
0080h to 009Fh	DMA page registers, PS/2 compatibility port
00A0h to 00BFh	Programmable interrupt controller (PIC) number 2
00C0h to 00DFh	DMA controller (DMAC) number 2
00E0h to 00FFh	Intel 80X87 numeric coprocessor

ISA I/O Decoders

Many ISA I/O cards perform an inadequate I/O address decode. Appendix C provides a description of this problem.

Chapter 5

The Previous Chapter

The basics of microprocessor communication have been introduced. This includes concept of the bus cycle and the types of devices that the microprocessor communicates with (memory and I/O), along with how these devices are addressed.

This Chapter

This chapter provides a description of the 80286 microprocessor. Emphasis is placed on the hardware interface between the 80286 microprocessor and the remainder of the system. Memory address formation in real mode is covered in detail and an introduction to protected mode is provided.

The Next Chapter

The next chapter provides a description of the six possible sources for various reset signals found in ISA systems and the effect each of them has on the system. It also explains the actions taken when Ctrl/Alt/Delete are depressed.

The 80286 Functional Units

Any microprocessor is really a system consisting of a number of functional units. Each unit has a specific job to do and all of the units working together comprise the microprocessor. Figure 5-1 illustrates the units which make up the 80286 microprocessor.

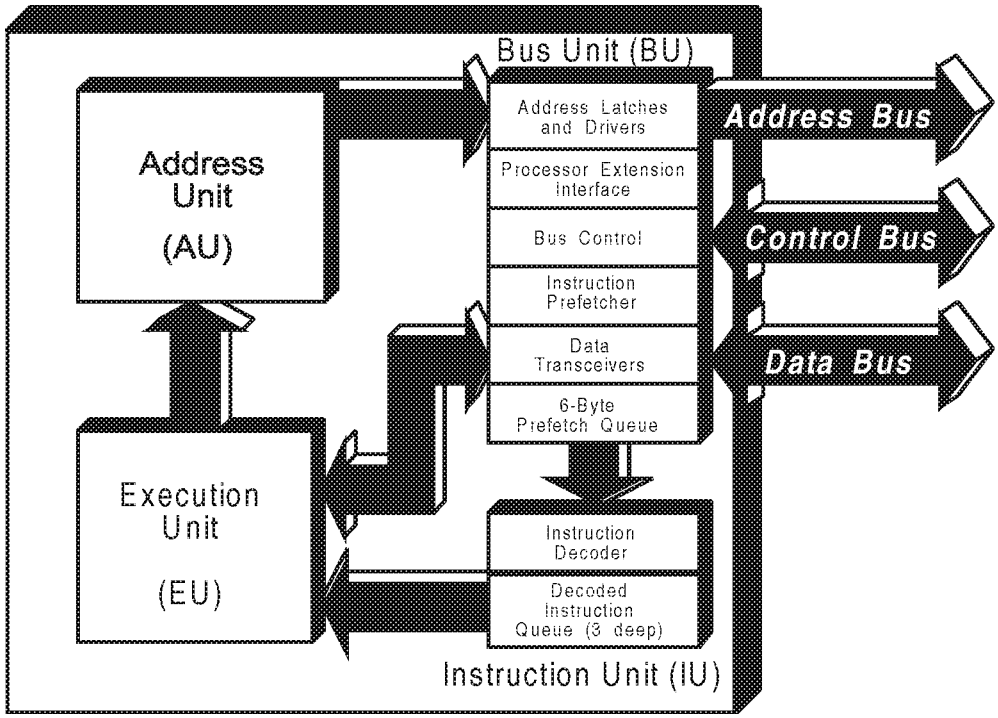


Figure 5-1. 80286 Microprocessor Block Diagram

Refer to 5-1. When the microprocessor starts operation, it will automatically begin fetching (reading) instructions from memory. It is the bus unit's task to perform the bus cycles required to fetch all instructions from memory. Instructions are read from memory over the data bus and placed into the instruction Prefetch Queue. From there they go to the instruction unit where they are decoded and sent to the execution unit one at a time for processing.

Execution of some instructions requires that data be read from or written to memory or I/O devices. If a memory device is specified, the execution unit directs the address unit to form the memory address of the location specified by the instruction. I/O addresses specified by an instruction do not require special address formation as do memory address. Instead they are sent directly from the execution unit to the bus unit to run the bus cycle specified by the instruction.

In summary, two types of information may be read into the microprocessor; instructions (code) and data. Instructions are read from memory over the data bus, go through the data transceivers, and are placed into the prefetch queue. Data to be processed is read from either memory or I/O devices over the data bus and is transferred via the data transceivers to the execution unit.

An instruction may also specify that data be written to a memory or I/O location, in which case data is sent from the execution unit to the data transceivers for transfer over the data bus.

In short, the 80286 microprocessor consists of four functional units:

1. **Instruction unit**. Decodes the instruction prior to passing it to the execution unit for execution.
2. **Execution unit**. Handles the actual execution of the instruction.
3. **Address unit**. When the microprocessor must address a memory location, the address unit forms the memory address that is driven out onto the address bus by the bus unit during the bus cycle.
4. **Bus unit**. Handles communication with the world outside the microprocessor chip (by performing bus cycles).

The following sections provide a more detailed description of each of these units.

The Instruction Unit

One at a time, the 80286 instruction unit pops (reads) instructions from the prefetch queue, decodes them into their component parts and places them into the decoded instruction queue to be forwarded to the execution unit.

The Execution Unit

Instructions are executed by the 80286 execution unit. In general, instructions cause either internal processing of data already stored within the execution unit registers, or reading data from or writing data to devices external to the microprocessor.

When the currently executing instruction requires writing or reading a memory or I/O location, the execution unit issues a bus cycle request to the bus unit. The execution unit then stalls until the bus unit completes the requested data transfer.

Two types of information may be read into the microprocessor: instructions (code) or data. Instructions are always read from memory and are placed into the instruction prefetch queue. From there, they are routed to the instruction unit where they are decoded and sent to the execution unit for processing. Data to be processed goes directly to registers inside the execution unit. These registers have several functions as described in the following paragraphs.

A register can be thought of as a storage location within the microprocessor. Some registers can only be read from, and some can only be written to, while others are both readable and writable. Registers within the execution unit are used by the programmer or the microprocessor itself for the following purposes:

- to temporarily hold values to be used in calculations
- to receive data being read from an external memory or I/O location
- to provide the data to be written to an external memory or I/O location
- to keep track of the microprocessor's current status
- to control certain aspects of the microprocessor's operation

Although they are storage locations, the programmer refers to the microprocessor's registers by names (rather then by the hex addresses used to address external memory and I/O locations). Most of the 80286 registers are 100% backward-compatible with the 8086 and 8088 microprocessor's registers. The registers inside the execution unit can be logically divided into two categories:

- General Registers
- Status and Control Registers

The following sections describe each of the registers in the General and the Status and Control register sets.

General Registers

This section offers a description of each of the general registers and examples of their usage. Figure 5-2 details the registers in the general register set.

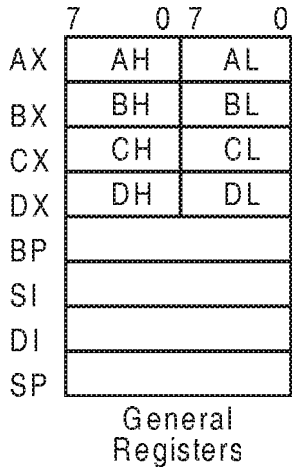


Figure 5-2. The 80286 General Registers

The AX, BX, CX and DX Registers—Each of these four registers can hold 16 bits (two bytes) of information. As an example, the following instruction will move the 16-bit value 1234h into the AX register:

```
MOV AX,1234     ;move the value 1234h into AX
```

In addition, the programmer can individually refer to each of the 8-bit registers that make up the upper and lower half of each of these 16-bit registers. As an example, the AX register actually consists of two 8-bit registers: the AL (Lower half of AX register) and AH (upper half of the AX register) registers. The same is true of the BX, CX and DX registers.

If the programmer wants to move only one byte of information, he or she would specify one of the 8-bit registers rather than a 16-bit register. In the following example, the programmer wants to write the value 02h into I/O location 60h:

```
MOV AL,02 ;move value 02h into AL
OUT 60,AL ;write AL contents to I/O port 60h
```

In the example above, the programmer first moves the value 02h into the 8-bit AL register and then performs an I/O write instruction to write the contents of the AL register to I/O location 60h. This will cause an I/O write bus cycle with address 000060h on the address bus and 02h on the lower part of the data bus

(see cardinal rule 3 later in this chapter — the 80286 always uses lower data path for data transfers with even addresses).
In general, the AX, BX, CX and DX registers are used for the following purposes:

- to temporarily hold values to be used in calculations
- to receive data being read from an external memory or I/O location
- to provide the data to be written to an external memory or I/O location
- to hold values to be used in calculating a memory address

The BP Register—Although available for general use by the programmer, the Base Pointer (BP) register is frequently used in forming a memory address. The programmer places the start address of a table of data items in memory (often referred to as a data structure) into the BP register. A move instruction is then executed that adds an index value to this base address to form the exact address of a data item in the table in memory. This is known as indexing into a table. The BX register can also be used in the same manner. The BP register is often used to access the stack without affecting the contents of the SP register.

The index registers—The two index registers are:

- the Source index (SI) register
- the Destination index (DI) register

These two registers are frequently used to implement string operations. This is an operation that is performed on a string of locations, rather than just one. A classic example would be the move string operation where the programmer wants to perform a series of memory reads and writes to transfer a block of data from one area of memory to another. To accomplish this, the programmer would code (write) the following instructions:

```
MOV  SI,XXXX   ;SI=start address of source
MOV  DI,YYYY   ;DI=destination start address
MOV  CX,ZZZZ   ;CX=number bytes to move
REP  MOVSB     ;move string of bytes
               ;repeat until done
```

In this example, the programmer places the start memory address of the area of memory containing the source data into the SI register and the start memory address of the destination area in memory into the DI register. The number of bytes to move is placed into CX, which acts as the count register in this context.

The programmer then executes the Repeat Move String of Bytes (REP MOVSB) instruction. When executed, this instruction performs a memory read from the memory address specified in the SI register. The byte received from this location is then written to the memory location specified in the DI register. The CX register is then decremented and the SI and DI registers are incremented. When the count in the CX register is exhausted, all of the bytes have been moved and the REP MOVSB instruction has completed execution. The next instruction is then executed. If the count is not exhausted, however, the REP MOVSB is repeated as many times as necessary until the move is completed.

The Status and Control Registers

This section describes the Flag and Machine Status Word registers. Three registers comprise the Status and Control Register group as shown in 5-3.

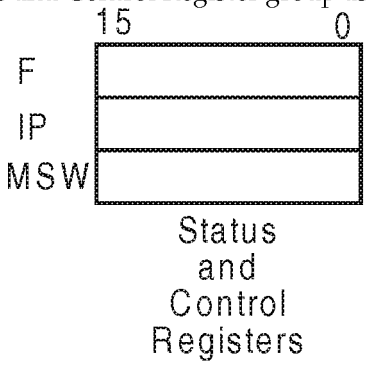


Figure 5-3. The 80286 Status and Control Registers

The Flag register—Refer to figure 5-4. The 16 bits in the Flag register may be divided into three fields, or groups:

1. Status Flag bits. Generally speaking, these bits reflect the result of the previously executed instruction.
2. Control Flag bits. By setting or clearing these bits, the programmer can modify certain of the microprocessor's operations.
3. System Flag fields. These three bits are related to protected mode operation and are outside the scope of this book.

Refer to table 5-1 for the definition of the status, control and system flag bits.

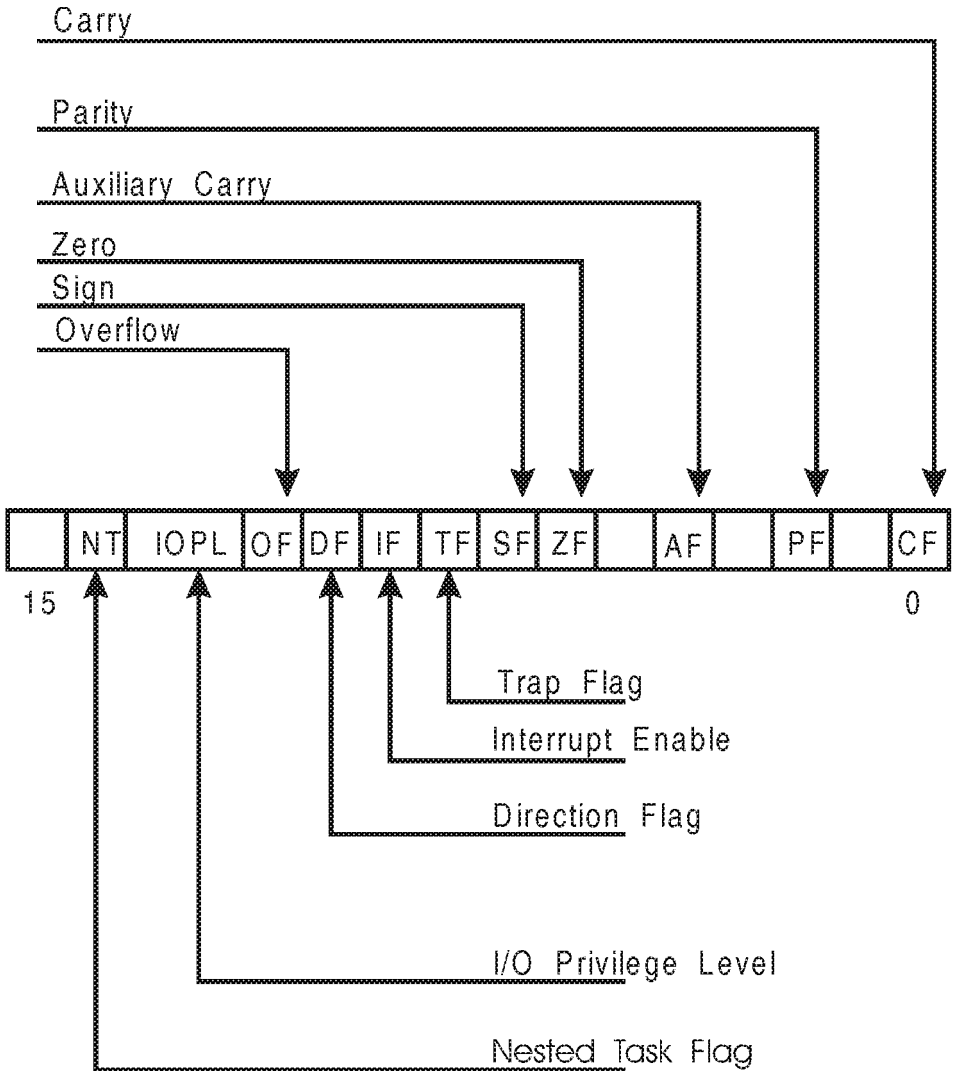


Figure 5-4. The Flag Register's Status, Control and System Bits

Table 5-1. The Flag Register's Status, Control and System Bits

Bit	Type	Description
CF	Status	Carry Flag. Indicates a carry or a borrow from the high-order bit in the result. Caused by an arithmetic operation.
PF	Status	Parity Flag. Set to 1 if low-order 8 bits of result contains an even number of 1 bits. Cleared to 0 otherwise.
AF	Status	Auxiliary Carry Flag. Indicates a carry or a borrow on the low-order 4 bits of the AL. Caused by an arithmetic operation.
ZF	Status	Zero Flag. Set to 1 if result is 0. Cleared to 0 otherwise.
SF	Status	Sign Flag. Set equal to the high-order bit of the result. 0 = positive result, 1 = negative result.
OF	Status	Overflow Flag. Set to 1 if the positive numeric result is too large (or the negative result is too small) to fit in the destination register.
TF	Control	Trap Flag. Also called the Single-Step Flag. When set to 1, causes a Single-Step Exception Interrupt after the execution of each instruction. Only used by Debug programs to aid in program debugging.
IF	Control	Interrupt Flag. When set to 1, the microprocessor will recognize interrupts on the INTR line. Programmer can set by executing STI instruction and clear via the CLI instruction.
DF	Control	Direction Flag. When cleared to 0, causes string instructions to increment SI and DI. They are decremented when DF is set to 1.
IOPL	System	I/O Privilege Level. 2-bit field. Causes the processor to generate an exception before accesses to specified I/O devices if the currently running task's privilege level is less privileged than the IOPL during protected mode operation.
NT	System	Nested Task Flag. When set to 1 during protected mode operation, indicates that the currently executing task (the nested task) was started by a CALL or interrupt while another task was running.

The Machine Status Word (MSW) Register—Refer to figure 5-5. The 80286 MSW register is a 16-bit register, but only the lower four bits are used. They are defined in table 5-2.

When RESET is asserted, the value FFF0h is forced into the MSW register. This means that the microprocessor operates in real mode, no Numeric Coprocessor is installed and the microprocessor will not emulate floating-point instructions. Real mode is described later in this chapter.

Table 5-2. The 80286 MSW Register Bits

Bit	Description
PE	Protect Enable. When cleared to 0, the 80286 operates in Real Mode (also known as Real-address Mode). When set to 1, microprocessor operates in Protected Mode.
MP	Monitor Numeric Coprocessor. When set to a 1, the microprocessor will check the 80287 Numeric Coprocessor's BUSY# signal before forwarding all floating-point instructions to the coprocessor for execution. Described in detail in the Chapter entitled, "The Numeric Coprocessor."
EM	Emulate Numeric Coprocessor. When set to a 1, recognition of a floating-point instruction causes an Exception 7 Interrupt. The Interrupt Service Routine, which handles this exception interrupt, emulates the particular floating-point instruction that would have been executed by the Numeric Coprocessor if it were present. See the Chapter entitled, "The Numeric Coprocessor" for details.
TS	Task Switch. This bit is related to protected mode operation and is outside the scope of this book.

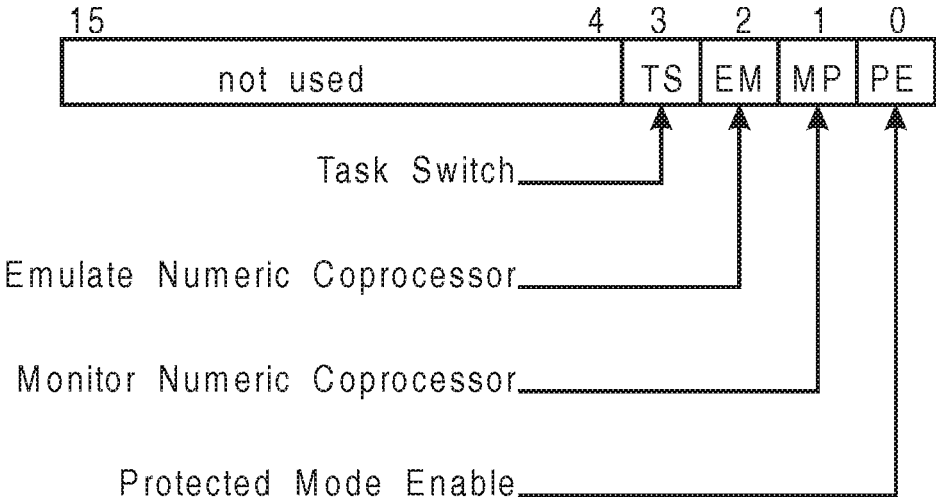


Figure 5-5. The 80286 Machine Status Word (MSW) Register

The Address Unit

When the microprocessor needs to read or write a memory location, the address unit forms the address to be placed on the address bus during the bus cycle.

The Segment Registers

Four segment registers exist in the 80286. The purpose of the segment register is to point to the start address of an area of memory containing a specific type of information (for example, a program or the data used by a program).

When Intel designed the 8086 and 8088 microprocessors (circa 1978), a programming discipline commonly referred to as structured programming was becoming very popular. Programmers were paying more attention to building programs in a logical fashion.

To aid the programmer in organizing complex programs, Intel created the segment registers. The idea was to identify a particular area (segment) of memory to hold the program, another to hold the data upon which the program acts, a "scratchpad" area of memory known as the stack and a fourth area known as the extra data segment.

In order to implement this feature, Intel included four 16-bit registers known as the segment registers:

- Code Segment (CS) register
- Data Segment (DS) register
- Stack Segment (SS) register
- Extra Segment (ES) register

Segment Register Usage in Real Mode

The following description provides a detailed view of how the microprocessor operates when in real mode. A description of how the segment registers are used when the microprocessor is in protected mode can be found in this chapter in the section entitled "Protected Mode."

When an 80286-, 80386-, 80486- or Pentium processor-based system is powered up, RESET is asserted until the power has stabilized. The asserted level on RESET causes certain default values to be forced into the microprocessor's internal

registers. The fact that the microprocessor always starts with the same values in its registers causes it to begin execution in exactly the same way every time the system is powered up. The contents of the Machine Status Word (MSW) register is forced to FFF0h by RESET. Bit 0 in this register is the PE (Protect Enable) bit. The fact that this bit always starts off cleared to 0 ensures that the microprocessor will always come up in real mode.

When in real mode, ISA machines based on the 80286, 80386, 80486 and Pentium microprocessors operate similarly as an 8086 or 8088 microprocessor. In other words, although they physically have more than the twenty address lines possessed by an 8086 or 8088 microprocessor, they aren't able to use all of these additional lines when running programs written for the 8086 or 8088. The 80286 has a limited ability to produce addresses above 1MB in real mode, but logic on ISA system boards defeats this ability so that the 80286 acts as an 8086 or 8088. This is done to ensure software compatibility. (See the section entitled, "Accessing Extended Memory in Real Mode.") This restricts the microprocessor to generating memory addresses from 000000h to 0FFFFFh. The upper address bits above A19 are forced to 0. The microprocessor is therefore limited to the lower 1MB of memory address space.

The 80386 and higher microprocessors can address all of their physical memory address space while in real mode, so long as they have at least once been switched to protected mode and back to real mode since the last reset. The 80286 lacks this capability since it cannot be switched from Protected to real mode without resetting the microprocessor and thus clearing the contents of the segment registers.

The address placed in each of the segment registers points to the actual start address of the segments in memory. Since each of the segment registers is only a 16-bit register, however, you can only place an address between 0000h and FFFFh into a segment register. On the surface, this would mean that the start address of each segment must be somewhere in the first 64KB of memory address space (locations 0000h to FFFFh). Refer to figure 5-6. Although the 8086 and 8088 microprocessors can actually address 1MB of memory address space, this would mean that the program, data, stack and extra segments would be limited to the first 64KB of this space and the rest of the memory would be inaccessible.

In actuality, the address the programmer places in a segment register is the four high-order hex digits of the address. The least-significant digit is imaginary and always considered to be zero. Placing 1000h in a segment register causes it to point to the memory segment (area) starting at address 10000h

(1000h + imaginary 0h). This means that a 16-bit segment register can be used to point to the start address of an area in memory anywhere from location 00000h to FFFF0h. This encompasses all but the last 16 locations of the 1MB memory address range of the 8086 and 8088 microprocessors.

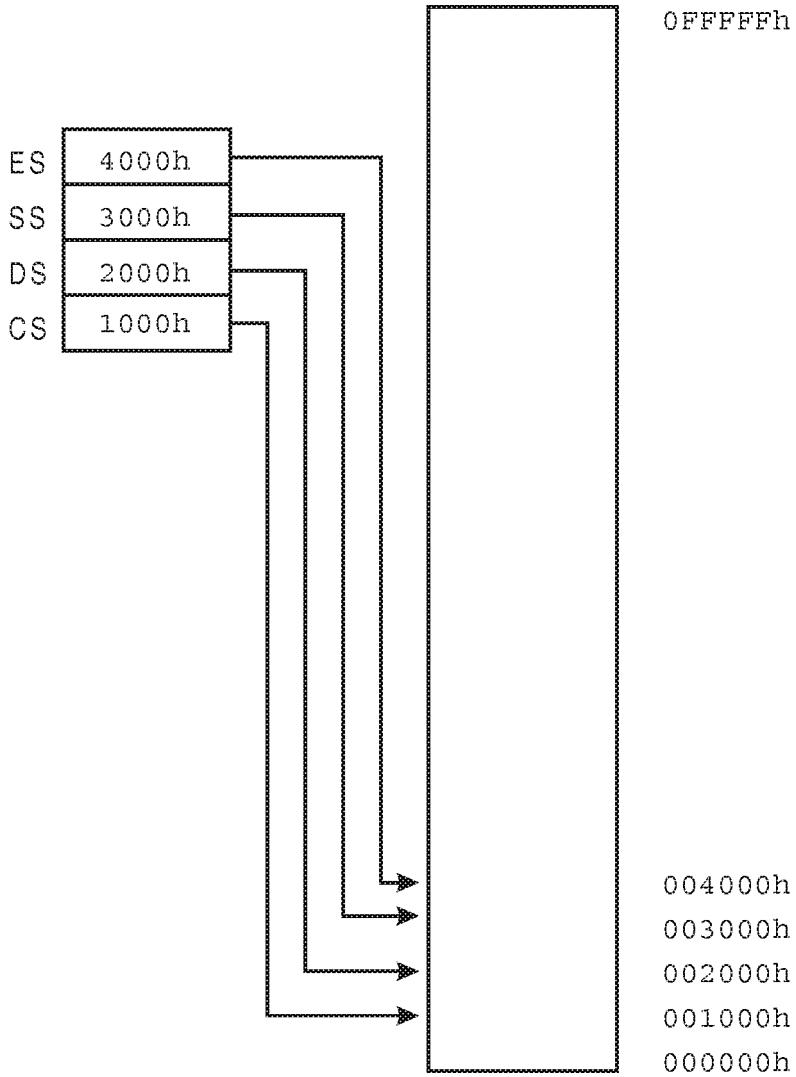


Figure 5-6. Result If Segment Registers Were Restricted to 16 Bits

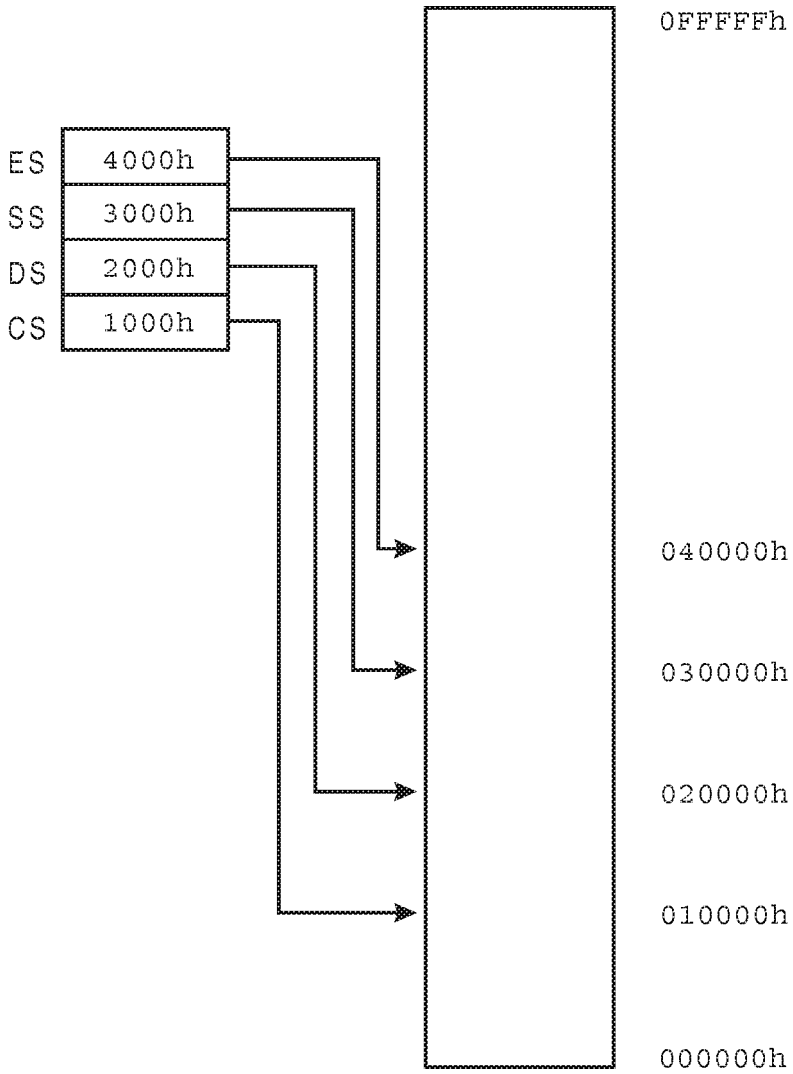


Figure 5-7. Segment Registers Point to Start Address of Memory Segments

In summary, then, the segment registers tell the microprocessor the start address of the Code, Data, Stack and Extra segments in memory. Figure 5-7 illustrates how the four segment registers are used to identify the four segments

available to a programmer. The usage of each of the segment registers is defined in the following sections.

Code Segment (CS) & Instruction Pointer (IP) Registers

Instructions are frequently referred to as code. The Code Segment (CS) register is always used in conjunction with another register called the instruction Pointer (IP) register. Together, their only task is to point to the memory location the microprocessor should fetch its next instruction from.

Refer to figure 5-8. The CS register points to the start address of the code segment in memory, while the 16-bit IP register points to the exact location within the code segment where the next instruction should be read from.

The value in the CS register supplies the SEGMENT portion of the address, while the IP register supplies the OFFSET, or exact location, within the segment. Intel almost always refers to memory addresses using the SEGMENT:OFFSET format, using the colon as a separator. For instance, if the current contents of CS and IP are 3000:0010, this means that the next instruction will be fetched (read) from memory location 0010h in the code segment which starts at memory address 30000h. The 10th (hex) location in the segment starting at 30000h is 30010h. Location 30010h is the actual memory address the microprocessor will place on the address bus to read the next instruction from. The address placed on the address bus is called the physical address.

A severe limitation of the segmentation scheme used by the 8086 and 8088 is tied to the width of the IP register. This register is only 16 bits wide and is used to point to the exact location (in the code segment) from which the next instruction will be fetched. The fact that it's only 16 bits wide means that it's physically impossible to point to any location within the segment greater than 64KB from the segment's start address (the segment register value). In other words, FFFFh is the largest value that could be placed in the IP register. The result— segments are limited to 64KB in length.

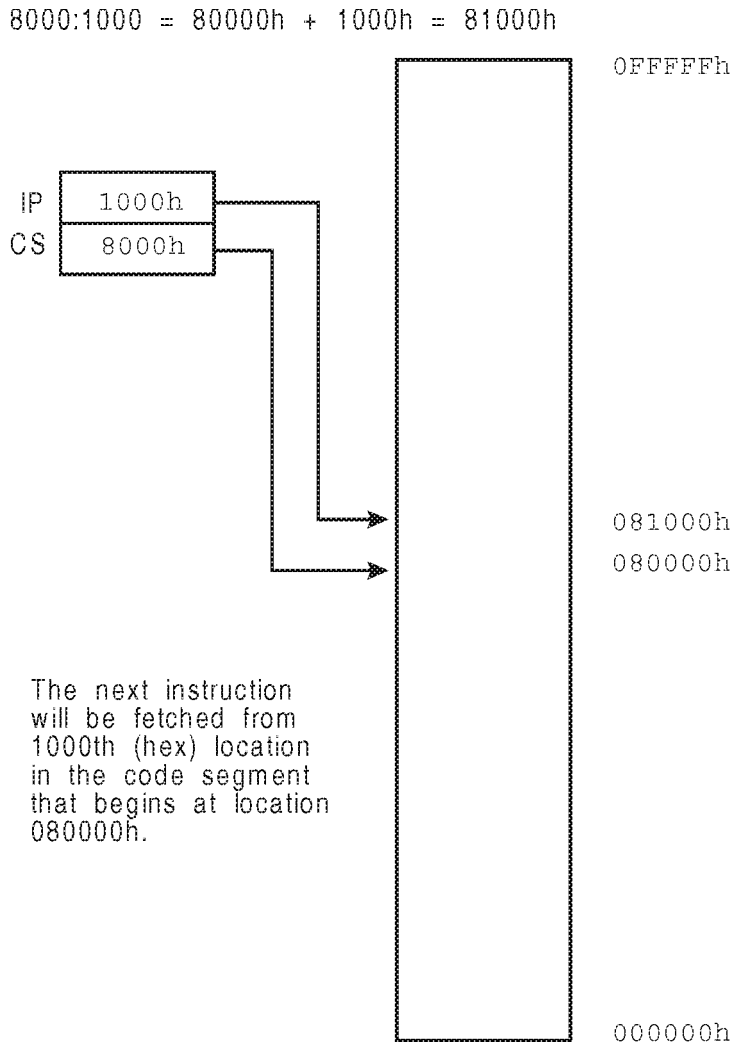


Figure 5-8. CS and IP Registers

This means that an entire program would have to fit in 64KB of memory (unless it resides in more than one segment). In order to have a program bigger than 64KB, the program must occupy several 64KB blocks of memory and the programmer must constantly be aware of which segment of memory the next instruction to execute resides in. The start address of the memory segment

must then be loaded into the CS register. The offset of the actual instruction within the new segment is loaded into the IP register.

New values are loaded into the CS and IP registers by executing JUMP instructions. A JUMP to a location within the same code segment loads a new value (the offset of the next instruction) into the IP register. Because the JUMP is within the same segment, this is called a NEAR JUMP.

A JUMP to a location within a different code segment loads new values into both the CS (loaded with start address of the new code segment) and IP (loaded with the offset of the instruction to jump to in the new code segment) registers. Because you're jumping to a different code segment, this is called a FAR JUMP.

The Data Segment (DS) Register

Refer to figure 5-9. The DS register points to the area of memory that holds data associated with the program pointed to by the CS register. As an example, assume that the microprocessor executes the following instructions:

```
MOV  AX,2000    ;value 2000h into AX
MOV  DS,AX      ;contents of AX into DS
MOV  AL,[0100]  ;contents of 2000:0100 to AL
```

This series of instructions will first place the value 2000h into the DS register. It's illegal to move a value directly into the DS register, so it must be moved first into AX and then into DS from AX. The data segment then starts at location 20000h in memory.

The MOV AL,[0100] instruction is then interpreted as "move the contents of memory location 0100h in the data segment into the AL register." As a result, the microprocessor executes a memory read bus cycle from physical memory location 20100h (20000h + 0100h) and the byte read will be placed into the microprocessor's AL register. The brackets are read as "the contents of memory location."

When a MOVE instruction doesn't specify what segment, the microprocessor always assumes the data segment. The value placed within the brackets is the OFFSET within the segment. Just as the OFFSET portion of the address is restricted to 16 bits in the code segment (by the size of the IP register), it is in the other segments as well. It is illegal to specify an OFFSET address larger than FFFFh within the brackets.

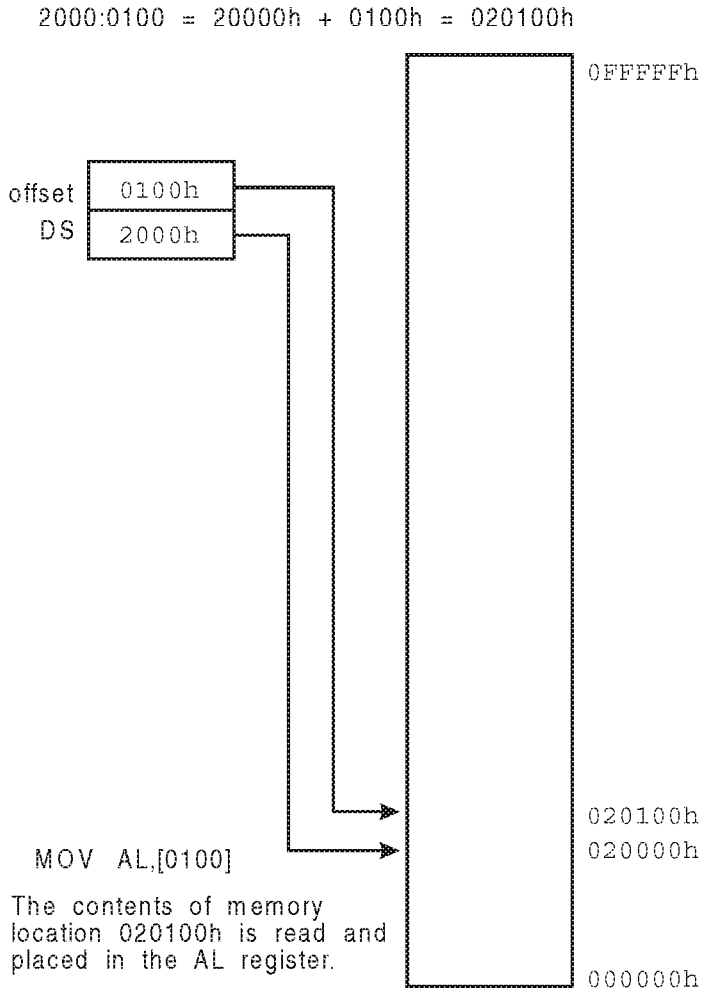


Figure 5-9. Illustration of Location within the Data Segment

The Extra Segment (ES) Register

The extra segment register is used in the same manner as the data segment register. The following series of instructions is the same as that used in the data segment example above, with one small change:

```
MOV  AX,3000        ;value 3000h into AX
MOV  ES,AX          ;contents of AX into ES
MOV  AL,ES:[0100]   ;contents of 3000:0100
                    ;to AL
```

The "ES:" added just before the brackets is called a Segment Override. In this way, the programmer tells the microprocessor to use the ES register to supply the segment portion of the address, rather than the default DS register. The end result of executing this series of instructions would be exactly the same as the series illustrated for the data segment above.

Stack Segment (SS) & Stack Pointer (SP) Registers

The area of memory designated as the stack is used as "scratch-pad memory" by the programmer and the microprocessor. Whenever the programmer needs to save a value for a little while and get it back later, the stack is frequently used for this purpose. The programmer doesn't need to specify a memory address when writing to or reading from stack memory. This makes it a very easy method for temporarily storing information.

The Stack Segment (SS) register points to the start address of the area of memory to be used as the stack. The Stack Pointer (SP) register provides the OFFSET portion of the address and points to the exact location in the stack segment where the last item was stored. At the beginning of a program, the programmer places the start address of the program's stack segment in the SS register. As data items are stored in the program's stack, the stack grows downward towards the stack segment start address specified in the SS register. The value placed in the SP register determines the size of the program's stack. As an example, if the programmer wishes this program's stack to be 512 bytes in size, the SP register should be initialized to 0200h to point to the five-hundred-twelfth location in the stack.

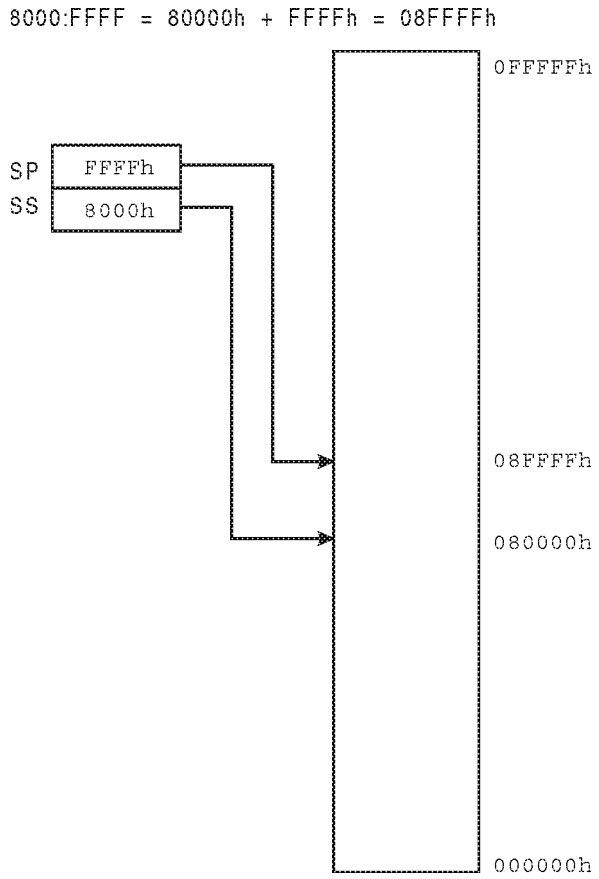


Figure 5-10. The Stack

Refer to figure 5-10. When the programmer wants to store a value on the stack, a PUSH instruction is executed. As an example, PUSH AX will cause the contents of the AX register to be written into stack memory where SS:SP is currently pointing. Assume that the SS register contains 8000h and the stack is empty. The SP register contains FFFFh. Also assume that the AX register currently contains 1234h and BX contains AA55h. Now consider the following:

1. When the PUSH AX is executed, the microprocessor first decrements the SP by 2. It then writes the two bytes from AX, 12h and 34h, into memory starting at 8FFFDh (80000h + FFFDh). AL is stored in memory location 8FFFDh and AH is stored in 8FFFEh because of the Little-Endian Byte-Ordering rule which is described later in this chapter.

2. If BX were now pushed onto the stack, the SP is first decremented by 2 and the two bytes from BX, AAh and 55h, are then stored in memory starting at location 8FFFBh.
3. Each time the microprocessor executes a subsequent push operation, it will first decrement the SP by 2 and then store the data into stack memory.

As you can see, the stack grows downward in memory from the highest memory location in the stack towards the segment start address.

To read data back from the stack, the programmer uses the POP instruction. When the programmer executes a POP instruction, such as POP BX, the microprocessor reads two bytes off the stack using the current value in SS:SP to form the memory address.

1. Continuing the example used above, a POP BX would cause the microprocessor to read the two bytes (55h and AAh) from locations 8FFFBh and 8FFFCh and place them into the BX register.
2. The microprocessor then increments SP by two, so SS:SP are now pointing at 8000:FFFD.
3. A POP AX would then cause the microprocessor to read the two bytes (34h and 12h) from locations 8FFFDh and 8FFFEh and place them into the AX register.
4. The microprocessor then increments SP by two, so SS:SP are now pointing at 8000:FFFF, the top of the stack.

As implemented by the Intel x86 microprocessors, the stack is a LIFO (Last In, First Out) buffer— the last object in is the first out.

If the programmer attempts to pop more data off the stack than was pushed onto it, the microprocessor will generate a special type of interrupt called a Stack Underflow Exception interrupt to indicate that the stack is empty. Conversely, if the programmer pushes data onto the stack until the entire 64KB stack segment (maximum possible size) is full and then attempts to push one more word onto the stack, the microprocessor will generate a Stack Overflow Exception interrupt. Interrupts are covered in the chapter entitled "Interrupt Subsystem."

Little-Endian Byte-Ordering Rule

When the microprocessor writes two bytes from a 16-bit register to memory, then there are two possibilities for how the data could appear in memory. As-

sume that the data segment starts at 000000h (DS register contains 0000h) and consider the following code:

```
MOV AX,1234     ;move 1234h into AX
MOV [0200],AX   ;move the contents of AX to
                ;memory locations 0200h and
                ;0201h
```

AH contains 12h and AL contains 34h after the first instruction is executed. When the microprocessor executes the second instruction, it will fill in memory with the following values:

Table 5-3. Little-Endian Byte Ordering

Memory Address	Data Byte
000200h	34h
000201h	12h

Notice that the other possibility is that the microprocessor theoretically could have put data byte 12h at memory address 000200h and byte 34h at address 000201h. That's only theoretical, because the x86 microprocessors use the Little-Endian Byte-Ordering Rule to determine which byte goes where, so only one way is actually used. Little-Endian Byte Ordering requires that the least-significant byte must be associated with the lowest address. In this case, the lowest address is 000200h and the least-significant byte is 34h.

The other byte-ordering method is Big-Endian Byte Ordering, which is used on most non-x86-based computers. The Big-Endian Byte-Ordering Rule requires that the most significant byte must be associated with the lowest address. Table 5-4 shows how the data would appear in memory.

Table 5-4. Big-Endian Byte Ordering (not used on x86)

Memory Address	Data Byte
000200h	12h
000201h	34h

When the x86 microprocessor reads data from memory, it once again uses the Little-Endian Byte-Ordering Rule to route the data bytes to the proper ends of the registers. For example, consider the following code and memory contents:

Table 5-5. Memory Contents

Memory Address	Data Byte
001003h	56h
001004h	78h

```
MOV AX,[1003]   ;move the contents of memory
                ;locations 1003h and 1004h
                ;to AX
```

What's in AX when the instruction is finished? The Little-Endian Rule says that the least-significant end of AX (AL) must be associated with the lowest address (1003h), so AX contains 7856h.

An easy way to remember the Little-Endian Byte-Ordering Rule is "Little End First." (The Big-Endian Byte-Ordering Rule is "Big End First.") Note that the Little-Endian Byte-Ordering Rule applies to I/O accesses as well as to memory accesses.

Definition of Extended Memory

Refer to figure 5-11. Simply put, extended memory is any memory that resides at address 1MB or higher (above address 0FFFFFh). Memory in the lower 1MB (000000h to 0FFFFFh) is frequently referred to as conventional memory. Since the 8086 and 8088 microprocessors only have twenty address lines, they are only capable of addressing memory in conventional memory space. This means that extended memory doesn't exist in machines based on these microprocessors.

Unless the microprocessor has been set to protected mode at least once since the last reset, then the 80286/80386/80486/Pentium microprocessors cannot access Extended Memory, with one exception. The next section describes the exception.

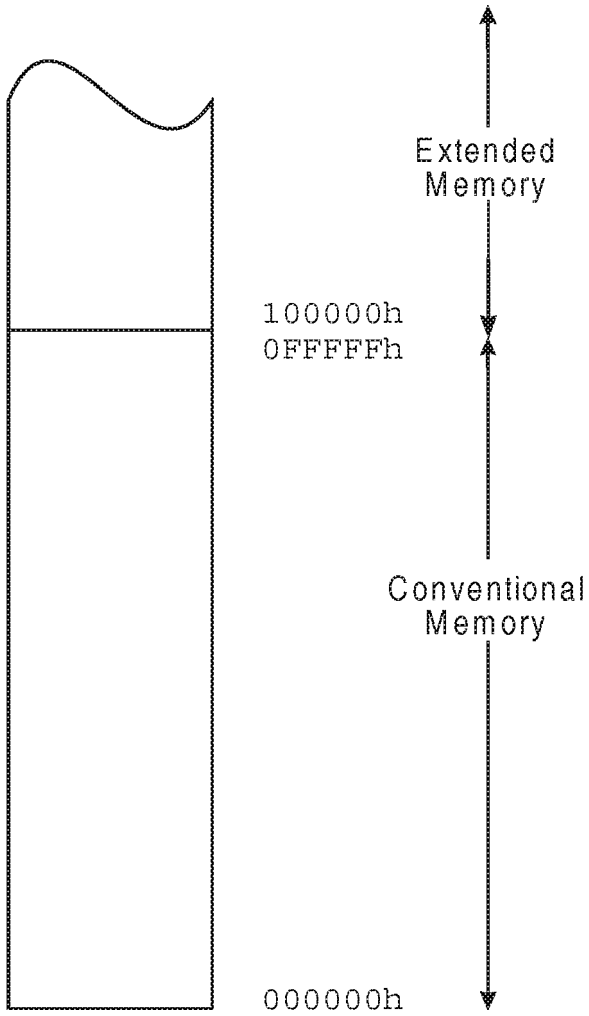


Figure 5-11. Extended Memory

Accessing Extended Memory in Real Mode

It is possible to access a small amount of extended memory (memory above 1MB) while in real mode. Consider the following example:

```
MOV  AX,FFFF   ;FFFFh to AX
MOV  DS,AX     ;transfer FFFFh to DS
MOV  AL,[0010] ;transfer contents of memory
               ;location FFFF:0010 to AL
```

In order to form the physical memory address to place on the address bus when executing the third instruction, the microprocessor would place a 0 on the end of the data segment value (FFFFh) to point to the start address of the data segment (FFFF0h). It would then add the offset (0010h) to the data segment start address to create the physical memory address:

```
                DS + 0h   =     FFFF0h
                OFFSET    =      0010h
Physical memory address   =    100000h
```

Refer to figure 5-12. The microprocessor then performs a memory read bus cycle, driving the resultant physical memory address, 100000h, onto the address bus. Notice that the 21st address bit, A20, is high. The processor is addressing the first memory location of the 2nd megabyte of memory address space. This is extended memory and the processor is addressing it in real mode!

Now consider this example:

```
MOV  AX,FFFF   ;FFFFh to AX
MOV  DS,AX     ;transfer FFFFh to DS
MOV  AL,[FFFF] ;read byte from memory address
               ;FFFF:FFFF to AL
```

Just as before, in order to form the physical memory address to place on the address bus when executing the third instruction, the microprocessor would place a 0h on the end of the data segment value (FFFFh) to point to the start address of the data segment (FFFF0h). It would then add the offset (FFFFh) to the data segment start address to create the physical memory address:

```
                DS + 0h   =     FFFF0h
                OFFSET    =      FFFFh
Physical memory address   =    10FFEFh
```

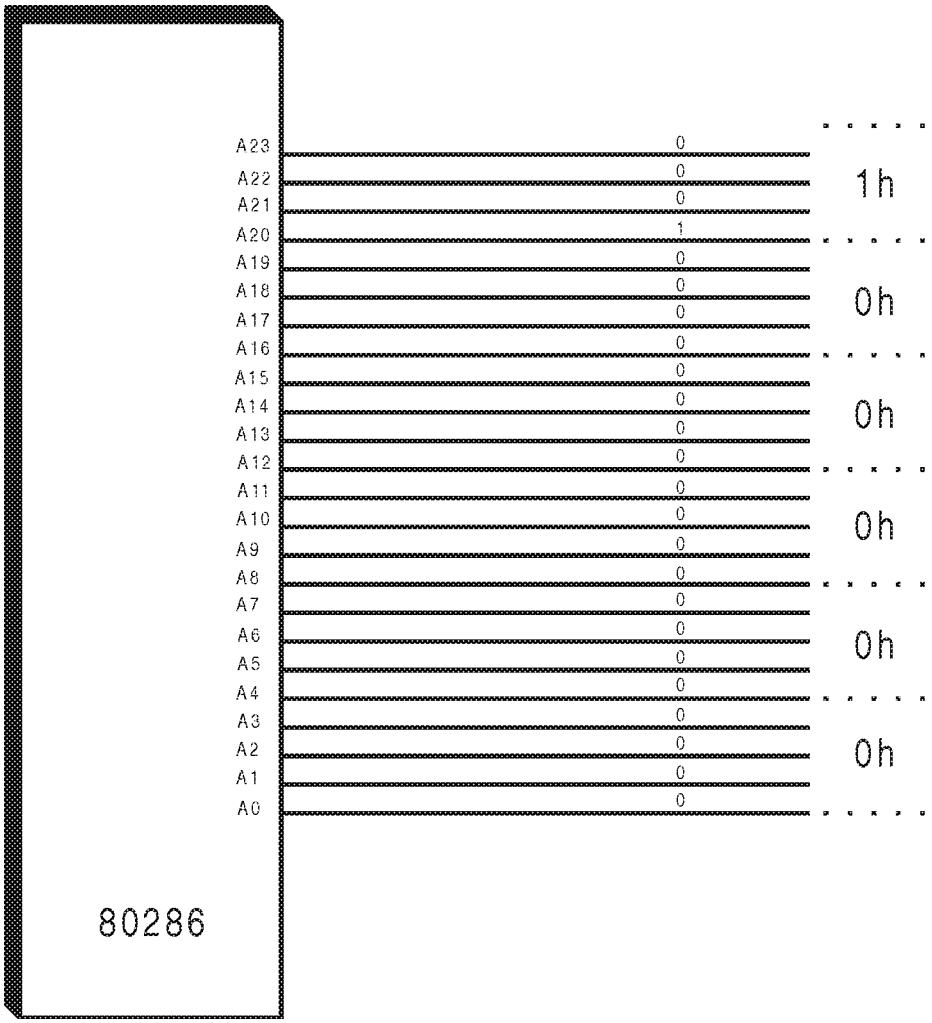


Figure 5-12. A20 High

The microprocessor then performs a memory read bus cycle, driving the resultant physical memory address, 10FFEFh, onto the address bus. With the value FFFFh in the segment register and by supplying any offset in the range 0010h to FFFFh, we are able to access any extended memory location from 100000h to 10FFEFh. This is a total of 65520 extended memory locations we can access while still in real mode.

This method is used by many MS-DOS memory management programs to access extended memory while remaining in real mode. This memory area is usually referred to as the High Memory Area (HMA).

The 80386 and higher microprocessors can address all of their physical memory address space while in real mode, so long as they have at least once been switched to protected mode and back to real mode since the last reset. The 80286 lacks this capability since it cannot be switched from Protected to real mode without resetting the microprocessor and thus clearing the contents of the segment registers.

Notice that the code fragments shown above are valid to execute on an 8086 or 8088 microprocessor. The results are different, however. If an 8086 or 8088 executes that code, address bit A20 will not be high since the 8086 and 8088 do not have such a bit. Therefore, the 8086 or 8088 produces an address that is between 00000h and 0FFEFh, for the examples shown above. This is called address wraparound or segment wraparound. The address space of the segment wraps around from the highest physical addresses to the lowest physical addresses.

Some programs written for the 8086 and 8088 may be depending on the segment wraparound to occur so that they can access data that is in the low addresses, such as the Interrupt Table or the BIOS Data Area. If these programs are executed on an 80286 or higher microprocessor, they will fail to operate correctly because they will not be accessing the same memory locations that they were expecting to access. In order for machines based on the 80286 and higher microprocessors to be compatible with such old code, the microprocessor's address must be truncated (cut off) to simulate the 20-bit address of the 8086 and 8088.

Notice that the highest address bit that was set when the largest segment value and largest offset were used is address bit 20, A20. Therefore, only A20 needs to be truncated. A23, A22 and A21 will not be high when executing the old code. Refer to figure 5-13. On the system board, outside the microprocessor, A20 is connected to one input of an AND gate. The other input of the AND gate is A20 GATE (also named A20 MASK#, FORCE A20, etc.). The output of the AND gate becomes the new A20 address bit that is broadcast with the rest of the microprocessor's address bits to the system. When A20 GATE is deasserted (low), the AND gate's output is low, so the system's A20 bit is low, simulating an address coming from an 8086 or 8088. The A20 GATE signal comes from the keyboard controller. See the chapter entitled "The Keyboard/Mouse Interface" for more information.

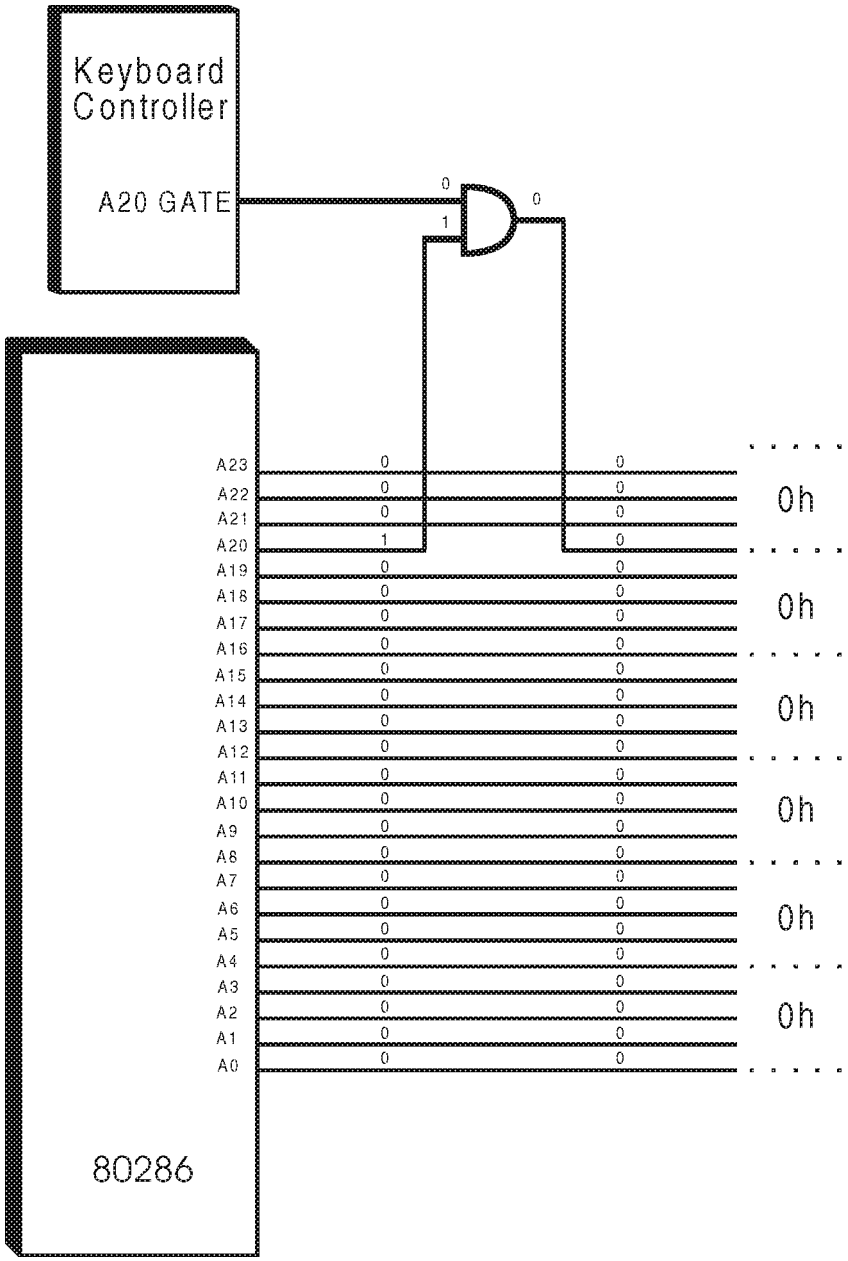


Figure 5-13. A20 Gate

The Bus Unit

Whenever the 80286 microprocessor must communicate with an external device, the bus unit performs the required bus cycle using the address, data and control buses. The bus unit consists of the following sub-units:

- Address Latches and Drivers
- Instruction prefetcher
- Processor Extension Interface
- Bus Control Logic
- Data Transceivers
- 6-byte prefetch queue

The following sections describe the purpose of each of the sub-units that comprise the bus unit.

Address Latches and Drivers

When the microprocessor must place an address on the address bus, the address is latched into (held by) the bus unit's internal address latch. This should not to be confused with the external address latch that is discussed in a subsequent chapter. The address drivers then drive the latched address out onto the address bus.

Instruction Prefetcher and 6-byte Prefetch Queue

As described in chapter one, the microprocessor is performing in-line (sequential) code fetches (instruction fetching) on an on-going basis. Only jump instructions alter program flow and they are statistically rare. Intel designed the instruction prefetcher to take advantage of this characteristic. Rather than wait for the execution unit to request the next instruction fetch, the prefetcher gambles that the next instruction fetch requested will be from the next sequential memory location and, on its own, performs a memory read bus cycle to get the next instruction. The 80286 instruction prefetcher always fetches two bytes at a time when fetching instructions from memory. It then places these two instruction bytes into the 6-byte prefetch queue.

Since the queue isn't full yet, the prefetcher initiates another memory read bus cycle to get the next two instruction bytes. These two bytes are also placed into the prefetch queue. The prefetcher then initiates a third memory read bus cycle

to get the next two bytes and places them into the prefetch queue. The queue is now full.

The instructions are decoded by the instruction unit and placed into the decoded instruction queue one at a time. This takes place in parallel with instruction execution by the execution unit and prefetching.

Once the prefetch queue is full, the instruction prefetcher ceases prefetching instructions. If the currently executing instruction isn't a JUMP instruction, upon execution completion the execution unit gets the next decoded instruction from the instruction unit's decoded instruction queue. The instruction unit then, in turn, gets the next instruction from the 6-byte prefetch queue, decodes it and places it into the decoded instruction queue. When two or more bytes open up in the prefetch queue, the instruction prefetcher resumes operation to fill the queue back up again.

Most of the time, the execution unit's request for the next instruction can be filled from the on-chip queues rather than forcing the microprocessor to perform a memory read bus cycle to get the next instruction.

When the execution unit executes a JUMP instruction, however, the prefetch queue and decoded instruction queue are "flushed." In other words, the prefetcher gambled and lost. All of the entries in the two queues are marked as invalid. The instruction prefetcher must then initiate a memory read bus cycle to get the requested instruction from memory. This stalls the execution unit until the next instruction becomes available.

Once the instruction has been fetched and placed in the prefetch queue, the prefetcher begins to prefetch once again to get ahead of the game.

Since JUMP instructions really don't occur very frequently in most programs, the execution unit may feed off the decoded instruction queue most of the time without having to wait for memory read bus cycles to be performed. In the overall scheme of things, the time lost when the queue must be flushed is minimal when compared with the advantage gained by prefetching.

Processor Extension Interface

This bus unit logic allows the microprocessor to communicate with the numeric coprocessor (if it is installed). This subject is discussed in the chapter entitled "The Numeric Coprocessor."

Bus Control Logic

This is the bus unit's state machine that performs bus cycles upon request. Bus cycles are performed under the following circumstances:

- Memory read bus cycle request from the instruction prefetcher.
- Request from the execution unit to perform a memory read, memory write, I/O read or I/O write bus cycle.
- In response to an interrupt request (refer to the chapter entitled "The Interrupt Subsystem").

Data Transceivers

A transceiver is a device that passes data through itself from left to right or visa versa. During read bus cycles, the bus unit's data transceivers pass the data coming from the currently addressed device into the microprocessor. During write bus cycles, the transceivers pass data from the microprocessor onto the data bus to be written to the currently addressed device.

80286 Hardware Interface to External Devices

This section provides a detailed description of the signal lines the 80286 microprocessor uses to communicate with the rest of the system. They have been grouped into the following categories:

- Address Bus
- Data Bus
- Control Bus

The Address Bus

The 80286 microprocessor's address bus consists of twenty-five signal lines:

- twenty-four address lines, A[23:0]
- BHE# (Bus High Enable)

Figure 5-14 illustrates the 80286 address bus.

During a bus cycle, the microprocessor places the full 24-bit address of the target location on the address lines, A[23:0]. A 24-bit address bus gives the 80286

the ability to address any of 16,777,216 individual addresses from location 000000h to FFFFFFh.

In addition, the microprocessor asserts BHE# (low) when performing a data transfer between itself and an odd address.

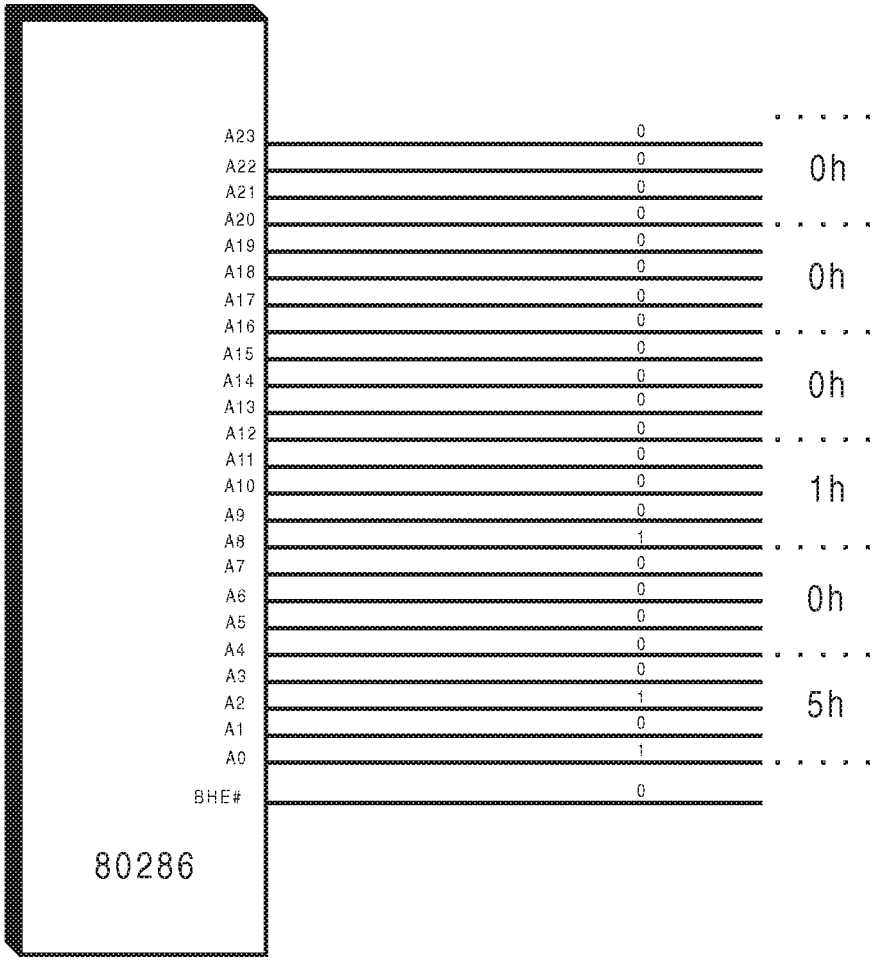


Figure 5-14. The 80286 Address Bus

How 80286 Addresses External Locations

During a bus cycle, the 80286 microprocessor can address just an even-addressed location, just an odd-addressed location, or an even location and the next sequential odd-addressed location simultaneously.

Refer to table 5-6. Address bit 0 (A0), has a weighted value of 1. When the microprocessor outputs an address, a 0 on A0 indicates that the address is an even address and a 1 on A0 indicates that it is an odd address. For example, address 000100h is an even address (A0 = 0), while address 000509h is an odd address (A0 = 1).

Since the microprocessor can only place one 24-bit address on the address bus at a time, it is incapable of telling external logic that it wants to address an even address and an odd address during the same bus cycle. A0 would have to be low and high at the same time; an impossibility.

To indicate that it wishes to address an even and an odd address simultaneously, the 80286 microprocessor places the even address out on the address bus and asserts its BHE# output (Bus High Enable). External logic interprets this to mean that the microprocessor wishes to perform a data transfer with the currently addressed even address and the next sequential odd address during the same bus cycle.

The data to be transferred between the microprocessor and the even address will be routed over the lower data path, D[7:0], while the data to be transferred with the odd address will be transferred over the upper data path, D[15:8]. This is stated as Rule 3 in the next section entitled, "The Cardinal Rules of the Intel World."

When the microprocessor outputs an address at the start of a bus cycle, external logic interprets A0 and BHE# as indicated in table 5-7. Remember that BHE# is asserted when low.

Table 5-6. Binary Weighted Value of each Address Line

Address Line	Binary Weighted Value
A0	1
A1	2
A2	4
A3	8
A4	16
A5	32
A6	64
A7	128
A8	256
A9	512
A10	1024
A11	2048
A12	4096
A13	8192
A14	16384
A15	32768
A16	65536
A17	131072
A18	262144
A19	524288
A20	1048576
A21	2097152
A22	4194304
A23	8388608

Table 5-7. Transfer Size Indicated by A0 and BHE#

A0	BHE#	Type of Transfer
0	0	The microprocessor is addressing an even address (A0=0) and will therefore use the lower data path to transfer the data byte between itself and the addressed location. Since the microprocessor only uses the upper data path to transfer data between itself and odd addresses, the asserted level on BHE# indicates that it wants to perform a data transfer with the odd location immediately following the even address it is outputting. This is a 16-bit data transfer using both data paths.
0	1	The microprocessor is addressing an even address (A0 = 0) and will therefore use the lower data path to transfer the data byte between itself and the addressed location. The deasserted level on BHE# indicates that the microprocessor will not be using the upper data path to talk to an odd address. This is therefore an 8-bit data transfer using the lower data path.
1	0	The microprocessor is addressing an odd address (A0 = 1) and will therefore use the upper data path to transfer the data byte between itself and the addressed location. The asserted level on BHE# indicates that the microprocessor will use the upper data path to talk to the odd address. This is an 8-bit data transfer using the upper data path.
1	1	Illegal (will not occur).

The Data Bus

Although it is technically correct to say that the 80286 microprocessor has a 16-bit data bus, it is more correct to say that it has two 8-bit data paths. The lower data path consists of data lines D[7:0], while the upper path consists of D[15:8]. Rule three describes the usage of these two data paths. Figure 5-15 illustrates them.

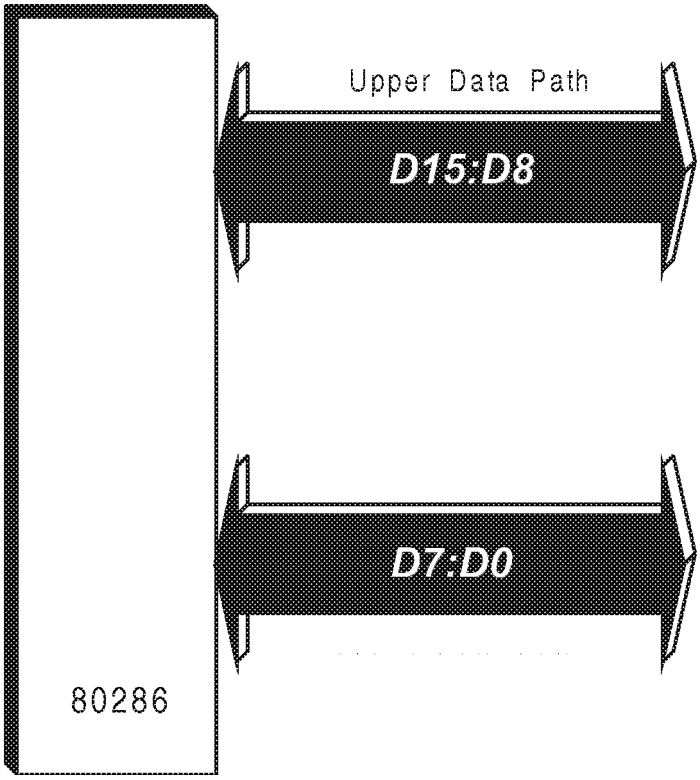


Figure 5-15. The 80286 Data Bus

The Cardinal Rules

The three rules stated in this section apply directly to the operation of the 80286 and the 80386SX microprocessors. The 80386DX, 80486 and Pentium microprocessors use a slight variation on rule number three. The chapter on the 80386 microprocessor explains the differences.

Cardinal Rule Number One

In any system based on the X86 microprocessors, every memory and I/O storage location contains one byte (8 bits) of information, no more, no less.

The temptation is great to say that storage locations in a system based on the 80286 microprocessor each contain one word (two bytes) of information. Many people think this is the case because the 80286 has a 16-bit data bus and can transfer a word at a time. Its 16-bit data bus gives the 80286 the ability to access

two locations simultaneously. This doesn't alter the fact that each location contains one byte of information.

Cardinal Rule Number Two

Every memory and I/O address is considered to be either an even address or an odd address.

When the 80286 microprocessor must read data from or write data to an external device, it begins a bus cycle. The address is placed on the address bus signal lines. Logic external to the microprocessor looks at the least-significant address line, A0, to determine if the address is even or odd. Consider the sample addresses in table 5-8.

Table 5-8. Examples of Even and Odd Addresses

Address	State of A0	Address Type
000000h	0	Even
000001h	1	Odd
0AFC12h	0	Even
F12345h	1	Odd
101003h	1	Odd

In each case where A0 is 0, the address is even. The address is odd if A0 is a 1.

In itself, this rule may not seem important, but in light of Rule 3, it assumes a great deal of importance.

Cardinal Rule Number Three

When the 80286 microprocessor reads from or writes to an even address, the data (one byte) is transferred over the lower data path, D[7:0]. When reading from or writing to an odd address, the data (one byte) is transferred over the upper data path, D[15:8].

The Control Bus

The control bus consists of all the 80286 signal lines other than the address bus and the data bus. The control bus signal lines can be separated into the following subcategories:

- Bus Cycle Definition lines
- Bus Mastering lines
- Ready line
- Interrupt lines
- Processor Extension lines
- The Clock line
- The Reset line

The following sections describe each of these signal groups.

Bus Cycle Definition Lines

When the 80286 bus unit must read data from or write data to an external location, it must run a bus cycle to do so. During the bus cycle, the microprocessor indicates the type of bus cycle by placing the proper pattern on the bus cycle definition signal lines. Table 5-9 indicates each type of bus cycle and the respective pattern which is output onto the bus cycle definition lines.

Table 5-9. Bus Cycle Definition

M/IO#	S1#	S0#	Bus Cycle Type
0	0	0	Interrupt Acknowledge
0	0	1	I/O Read
0	1	0	I/O Write
1	0	0	Halt or Shutdown
1	0	1	Memory Read
1	1	0	Memory Write

Technically speaking, there is a fourth bus cycle definition line, COD/INTA#. It is the Code or Interrupt Acknowledge line and has something in common with your appendix— if you lost it you wouldn't miss it. A system only needs to use this line if it must differentiate between a memory read to get data versus a memory read to get an instruction. In most systems, this pin isn't even hooked up.

Bus Mastering Lines

Most of the time, the microprocessor on the system board “owns” the three buses and uses them to communicate with external devices. The microprocessor is the Bus Master. However, there are situations where it is necessary to surrender the buses to other bus masters.

When a device other than the microprocessor wishes to use the buses to communicate with another device, it asserts the microprocessor's HOLD (Hold Request) input. When HOLD is asserted, the manner in which the microprocessor responds depends on whether or not a microprocessor bus cycle (data transfer) is currently in progress.

If a bus cycle is in progress when HOLD is asserted by the prospective bus master, the microprocessor completes the current bus cycle prior to responding to the HOLD. If a bus cycle isn't in progress, the microprocessor responds to the hold request immediately.

In response to HOLD, the microprocessor electrically disconnects itself from the address, data and control buses. It's just as if the microprocessor opens dozens of switches, disconnecting itself from the external buses. This disconnect process is commonly referred to as "floating" the buses.

The microprocessor then asserts its Hold Acknowledge (HLDA) output to inform the requesting bus master that it now owns the buses and is therefore bus master. The new bus master may now use the buses to run one or more bus cycles. It may keep the buses as long as necessary by keeping HOLD asserted. When it no longer requires the use of the buses, the bus master releases HOLD. In response, the microprocessor's bus unit re-attaches itself to the buses and initiates a bus cycle (if it has one pending).

In addition to HOLD and HLDA, there is a third microprocessor line involved in bus mastering. In a system designed to take advantage of it, the 80286's LOCK# output can be used by the programmer to prevent other bus masters from stealing the buses away from the microprocessor. Refer to figure 5-16.

The microprocessor automatically asserts the LOCK# signal during certain operations that require multiple bus cycles to complete such as:

- Interrupt Acknowledge Bus Cycles
- Execution of Exchange instructions
- Segment Descriptor Reads

There are times when the programmer must force the microprocessor to assert the LOCK# signal. These situations are described in the next section.

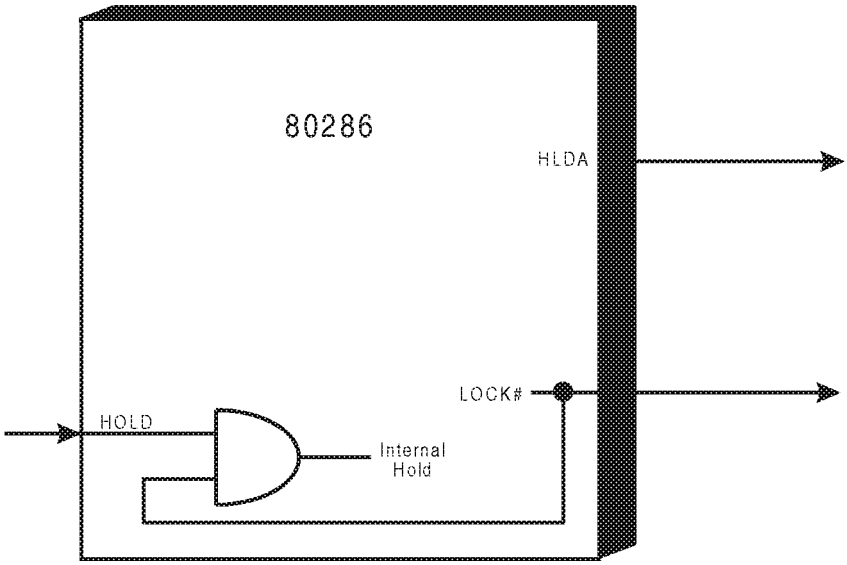


Figure 5-16. The Lock Output

Protecting Access To Shared Resource

Two processes (programs in a multi-tasking system) or two processors (in a multi-processing system) may need to access a shared resource at the same time. Shared resources are things that each of the processes or processors can access, such as the printer and data structures in memory. Assume that two microprocessors, CPU A and CPU B, need to print a file. In order for CPU A and CPU B to share the printer, they must share the bus that connects to the printer. An arbiter prevents bus contention between CPU A and CPU B by controlling the HOLD inputs to both microprocessors. Only one microprocessor owns the bus at a time.

If CPU A were granted the buses by the arbiter, it could start writing to the printer. It may take CPU A several bus cycles to write the entire file to the printer. CPU B might request the buses from the arbiter before CPU A finished writing the entire file to the printer. If the arbiter granted the buses to CPU B, then CPU A would have to give up the buses. When CPU B started writing to the printer, the result would be a mish-mash of printout from CPU A and CPU B, all mixed up.

CPU B needs to be told to wait until CPU A has finished the print job. This is usually handled by "semaphores," meaning "sign bearers." (Did you realize you'd learn Greek while reading this?) Somewhere in memory, a semaphore indicates whether the printer is available for use. Don't confuse this with the printer status byte which indicates whether the printer is busy or out of paper. The semaphore is like a reservationist. Both microprocessors would have to check the semaphore before beginning a print job.

CPU A would read a copy of the semaphore into one of its internal registers. CPU A would then test the copy in its register to see if the printer is available. If the printer is available, CPU A would set the value in its register. Then CPU A would write the register's value (the new semaphore value) back to the semaphore's location in memory. This is sometimes referred to as a read-modify-write operation and the "modify" portion of the operation may be a test-and-set operation.

Once CPU A has written the new value of the semaphore back to memory, it may start writing to the printer, so long as the arbiter has still granted the buses to CPU A.

CPU B has to obey the same rules that CPU A does, so before CPU B writes to the printer, it must read and test the semaphore. CPU A has already written a new value back to the semaphore in memory, so when CPU B reads the semaphore, CPU B will see that the printer is not available. CPU B will go into a "spin loop," sometimes referred to as a "spin lock," re-reading the semaphore occasionally until it sees that the printer is once again available. This solves the mish-mash problem, or so it seems.

If CPU A and CPU B needed to write to the printer at the same time, they would both need to access the semaphore at the same time. The arbiter would only allow one of them to use the buses at a time. Assume CPU A has higher priority, so the arbiter grants the buses to CPU A, and CPU A reads the semaphore first. Let's start off assuming that the semaphore has a 0 in it to indicate that the printer is available. CPU A tests the semaphore and finds it to be 0, meaning the printer is available. CPU A then sets its copy of the semaphore (its general purpose register) to 1, to indicate that the printer is unavailable. While CPU A is testing and setting its copy of the semaphore, it does not need to use the buses, since the test and set operations occur entirely within its general purpose registers which are inside the microprocessor. During this time, CPU B is still requesting the buses, so the arbiter grants the buses to CPU B. CPU A hasn't had a chance to write the semaphore back to memory yet. CPU B reads a

copy of the semaphore and begins to test it. The semaphore in memory is still 0, so when CPU B tests it, CPU B thinks that the printer is available.

CPU A has determined that the printer is available and requests the buses from the arbiter so it can write the new value of the semaphore to memory. CPU A then requests the buses so it can begin writing to the printer. Meanwhile, CPU B has also determined that the printer is available because the semaphore was 0 when CPU B read it. CPU B believes that it owns the printer and we have the same mish-mash problem that we started with.

The intended operation was for CPU A to read the semaphore, test it, set it and write it back without interference from CPU B. These operations were intended to be indivisible, or "atomic" (yes, it's Greek). That's one of the purposes of LOCK#. By asserting LOCK#, CPU A prevents the arbiter from granting the buses to CPU B until after CPU A has written the updated semaphore to memory. After writing the semaphore, CPU A deasserts LOCK# and the arbiter can grant the buses to CPU B. Now, when CPU B reads the semaphore, it sees that the printer is not available and it goes into a spin loop.

If the programmer doesn't want the microprocessor to give up the buses during a series of instructions, each of the instructions should be prefaced by a LOCK prefix. Each time that the microprocessor sees a LOCK prefix in front of an instruction, it will assert the LOCK# output while the instruction is executed. The low on the LOCK# line will force the microprocessor's internal Hold request to remain deasserted and the HLDA output will also remain deasserted, preventing any bus master from stealing the buses.

The preceding discussion of atomic operations is just an example to illustrate the lock mechanism of the x86 microprocessors. The memory addresses of the semaphores and the meanings of the semaphores are system software issues. There are other ways of protecting access to shared resources other than using LOCK#. Also, it is rare for ISA machines to have multiple processors in them. The DMA controllers (which are non-intelligent bus masters) should not access semaphores since they cannot make decisions on their contents.

Ready Line

The READY# input allows slow access devices to stretch out the microprocessor's bus cycles to match their own slow response time. This subject was introduced in the chapter entitled "Introduction to the Bus Cycle" and is explored in more depth in the chapter entitled "Detailed View of the Bus Cycle."

Interrupt Lines

The 80286 microprocessor has two inputs used by external logic to interrupt the microprocessor's program execution. These two inputs are INTR and NMI.

The microprocessor's Interrupt Request input (INTR) is frequently referred to as maskable interrupt request. INTR is generated by the Intel 8259A Programmable Interrupt Controller when an external hardware device requires servicing (for example, the keyboard interface has a keystroke for the microprocessor).

INTR is called the maskable interrupt request input because the programmer has the ability to mask out, or ignore, this input. If the programmer has a section of critical code (series of instructions) that he or she doesn't want interrupted, the instructions can be prefaced with a CLI (Clear Interrupt Enable) instruction. Once the microprocessor executes this instruction, it will ignore the INTR input until recognition of external hardware interrupts is once again re-enabled by the programmer. When the critical series of instructions has been executed without interruption, the programmer should then execute the STI (Set Interrupt Enable) instruction, re-enabling recognition of external hardware interrupts on the INTR input line.

Since interrupts arriving on the INTR line will not be recognized while the programmer has interrupts disabled, the programmer shouldn't disable interrupt recognition for extended periods of time. If INTR is ignored for too long, important events can be missed by the microprocessor.

The Non-Maskable Interrupt request (NMI) input to the microprocessor is typically used by external logic to alert the microprocessor that a critical hardware or software failure has been detected (for example, a memory parity error). It's called the non-maskable interrupt request line because the programmer has no way of masking out recognition of NMI if it should become asserted. The microprocessor will service NMI interrupts immediately.

A complete explanation of both maskable and non-maskable interrupts can be found in the chapter entitled "The Interrupt Subsystem."

Processor Extension Interface Lines

Processor extension is another name for the numeric coprocessor. These four signal lines are used to connect the numeric coprocessor to the microprocessor.

Refer to figure 5-17. Processor Extension Request (PEREQ) is an output from the numeric coprocessor and an input to the microprocessor. It is used by the numeric coprocessor to ask the microprocessor to perform a memory transfer. Processor Extension Acknowledge (PEACK#) is an output from the microprocessor and an input to the numeric coprocessor. The microprocessor uses PEACK# to respond to the PEREQ issued by the numeric coprocessor. It informs the numeric coprocessor that the requested memory transfer is in progress.

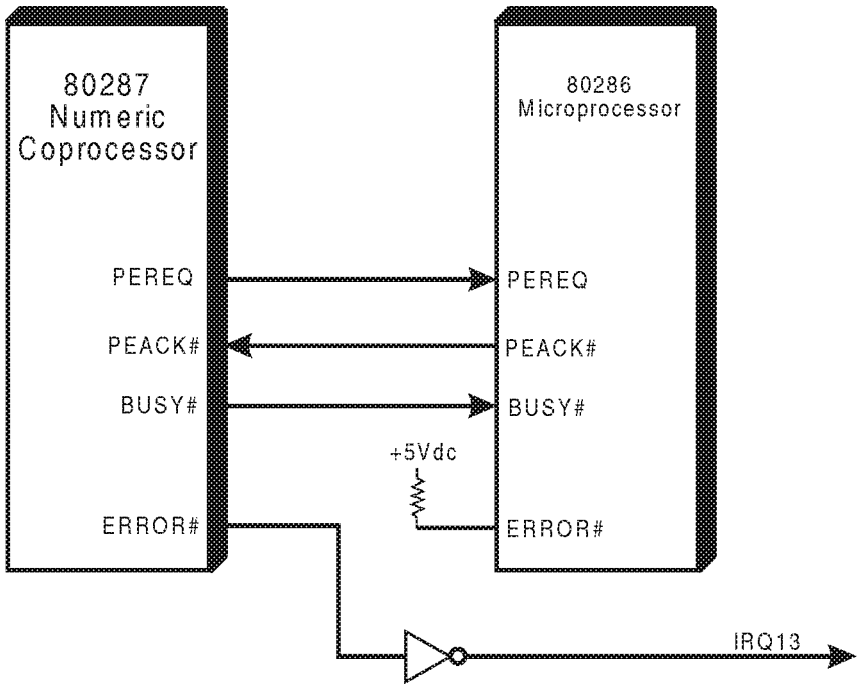


Figure 5-17. Relationship of the Microprocessor and the Numeric Coprocessor

BUSY# is an output from the numeric coprocessor to the microprocessor. The numeric coprocessor asserts BUSY# when it begins execution of an instruction received from the microprocessor. While BUSY# is asserted, the microprocessor shouldn't forward any more instructions to the numeric coprocessor.

ERROR# is an output from the numeric coprocessor to the microprocessor. It informs the microprocessor that the numeric coprocessor has incurred an error while executing an instruction. In ISA machines, the numeric coprocessor's ER-

ROR# output isn't connected to the ERROR# input of the microprocessor. Instead, IBM chose to attach the ERROR# output to IRQ13 (Interrupt Request Level 13). In other words, interrupts are used to report numeric coprocessor errors.

This subject is covered in more depth in the chapters entitled "The Numeric Coprocessor" and "The Interrupt Subsystem." Figure 5-17 illustrates the interface between the microprocessor and the numeric coprocessor.

The Clock Line

Frequently referred to as CLK2, this is the microprocessor's double-frequency clock input. Internally, the microprocessor then divides CLK2 by two to yield PCLK, the real heartbeat of the microprocessor.

The Reset Line

Among the possible RESET sources, the RESET input to the microprocessor is derived from the power supply's POWERGOOD output signal. In addition, RESET also fans out and is applied to many other system components, including cards plugged into the ISA slots. When the power supply is first turned on, its output voltages are not yet stable. The power supply will keep the POWERGOOD signal deasserted until power has stabilized. While POWERGOOD is deasserted, RESET will be asserted. When the power supply output voltages have stabilized, the POWERGOOD signal is asserted and RESET is deasserted.

While RESET is asserted, it has two effects on the microprocessor and other system components:

1. Keeps any activity from occurring until power has stabilized.
2. Presets the microprocessor and other system devices to a known state prior to letting them begin to do their job. This ensures that the machine will always start up the same way.

Refer to figure 5-18. In order to guarantee proper operation of the microprocessor, the RESET input must change state in synchronism with the double-frequency clock. The Reset Sync flip-flop in the illustration accomplishes this task. On each rising-edge of CLK2, the output of the flip-flop is set to the opposite state of the POWERGOOD input. During the period immediately following system power-up, POWERGOOD is deasserted and RESET will therefore be asserted (because the flip-flop is continuously clocking ones through to the RESET line). When the power supply output voltages have stabilized, however,

the POWERGOOD signal goes high and the flip-flop begins to continuously clock zeros onto the RESET line.

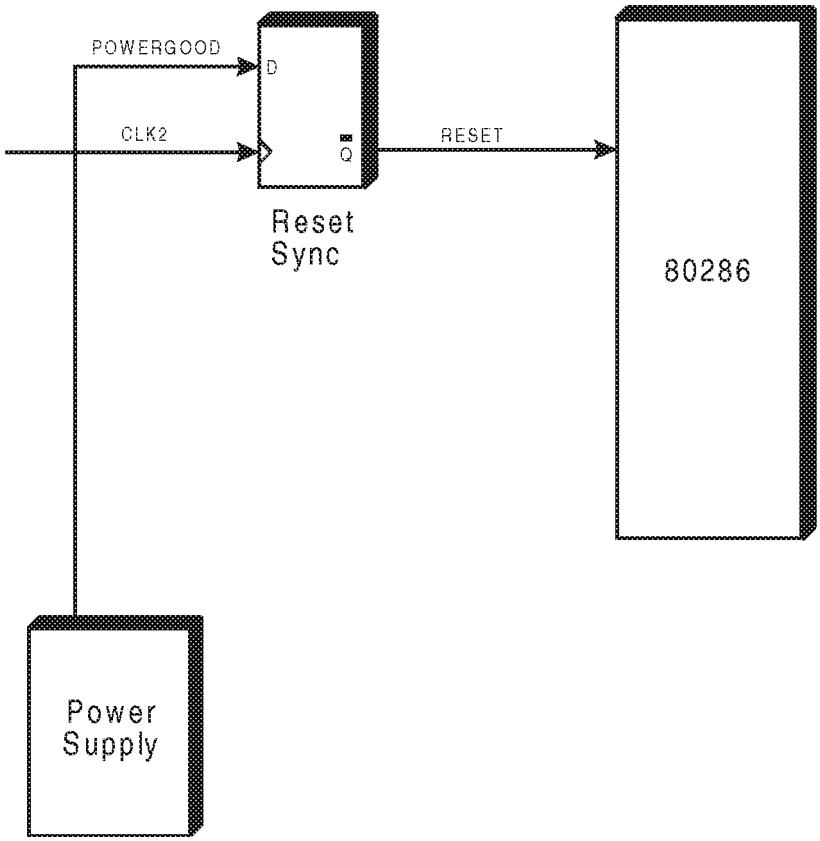


Figure 5-18. The Reset Logic

More information regarding RESET's effect on the microprocessor can be found in the chapter entitled "The Power-Up Sequence."

Protected Mode

This section provides a description of how the 80286 microprocessor forms physical memory addresses when protected mode is enabled. It does not pro-

vide an in-depth programmer's guide to all aspects of protected mode operation. For two reasons, that subject is outside the scope of this book:

- It is purely a programming issue and has absolutely nothing to do with the hardware interface between the microprocessor and the rest of the system.
- To provide an in-depth description would require a separate volume of sizable proportions. Since there are already a number of good books available on the subject, it would be an unnecessary duplication of effort.

Intro to Protected Mode and Multitasking Operating Systems

The 80286 microprocessor's protected mode was designed to facilitate the implementation of multitasking operating systems like OS/2 and UNIX. Unlike MS-DOS, which was designed to handle a single-task (applications program) at a time, a multitasking operating system is designed to run and keep track of multiple tasks.

A multitasking operating system will typically let a task run until either:

- a predetermined period of time has elapsed or
- the task requires something that isn't immediately available.

Assume that an applications program referred to as task A has not yet run for its allotted period of time, but it requires something that isn't immediately available. Good examples would be waiting for a keystroke or the completion of a disk read operation. Since these operations will take some time, it would be wasteful to have the microprocessor wait for the operation to complete. Instead, the operating system will store all of the microprocessor's registers in a special segment of memory, known as a Task State Segment (TSS). There is a separate TSS for every program that the microprocessor is running. This stored information will later allow the microprocessor to pick up where it had suspended Task A once the requested operation has completed.

Having stored task A's Task State, the microprocessor can then give control to a task which was previously suspended while waiting for completion of some other operation. We'll call this task B. The microprocessor loads all of the microprocessor's registers from the TSS belonging to task B. This TSS holds the information describing the microprocessor's state at the point when task B was suspended.

Task B will then run until either its allotted time has elapsed or it again needs something it has to wait for, at which point it will be suspended and control will pass to another task. A multitasking operating system can be viewed as a juggler who has to keep a number of balls in the air without dropping any.

With the help of the 80286's protected mode features, the operating system must:

1. Protect application programs from each other. In other words, don't allow one application program to play with another application program's data or code segments. Any attempt on the part of one application program to access another program's areas of memory should result in an Exception Interrupt which will be handled by the operating system software.
2. Protect itself from the application programs it is supervising.
3. Provide the interface between applications programs and I/O devices. It could be catastrophic if an application program manipulates an I/O device (for example, a disk) without the Operating System's knowledge. The operating system wouldn't know data on a disk or the state of an I/O device had been altered.

In order to prevent this possibility, any attempt on the part of an application program to execute an I/O read or write will cause an Exception Interrupt when the IOPL bit in the flag register is set. This informs the operating system of the attempt to access an I/O device. He may then choose to cast out the demon application program who has violated sacred ground.

In protected mode, segments can be described as:

1. belonging to (accessible by) all programs. This is referred to as a Global Segment.
2. belonging to a particular application program. This is referred to as a Local Segment because it's local to the currently running application program. Any attempt by a program other than the application program that "owns" this Local Segment or the operating system to access a Local Segment will cause an Exception Interrupt.
3. belonging to the operating system software. This is referred to as a System Segment. Any attempt by a program other than the operating system to access a System Segment will cause an Exception Interrupt.
4. read-only segments. Any attempt to write to a read-only segment will cause an Exception Interrupt.
5. execute-only code segments. Any attempt to access an execute-only segment for anything other than an instruction fetch will cause an Exception

Interrupt. Even an attempt to read the instructions as data (using MOV instructions) will cause this exception. This means that another program can't disassemble the instructions in a program's code segment to figure out what makes it tick.

6. read/write segments. This would be a normal data segment.
7. only accessible by programs with a privilege level equal to or greater than the segment's privilege level.

As will be seen in the next section, the operating system software maintains a group of tables in memory that describe:

- the Global segments that anyone can access. The Global Descriptor Table (GDT) resides in memory. Each entry contains a segment descriptor, describing a globally accessible segment.
- the Local segments that belong to a particular application program. The Local Descriptor Table (LDT) resides in memory. Each entry contains a segment descriptor, describing a segment belonging to a particular program.
- Interrupt Service Routines. The Interrupt Descriptor Table (IDT) resides in memory. Each entry contains an Interrupt Descriptor which describes the location and accessibility of the Interrupt Service Routine.

Segment Register Usage in Protected Mode

The microprocessor can be placed in protected mode by setting bit 0, the PE bit, in the MSW register. Prior to actually setting this bit, however, the programmer must write certain tables of information into memory. Failure to do so will cause all hell to break loose when protected mode is enabled.

With the one exception already described, it is impossible to address extended memory (memory above 1MB) while in real mode. In real mode, the segment registers can only be used to form a 5-digit address, thus limiting you to addresses in the range 000000h to 0FFFFFh (the lower 1MB). Since the 80286 microprocessor actually has 24 address lines, A[23:0], it is capable of generating memory addresses from 000000h to FFFFFFh (a 16MB address range). The 80286 must be switched into protected mode to do this, however.

When in protected mode, the value in the segment register is interpreted differently than when in real mode. In real mode, the value in the segment register (with the imaginary 0h on the end) was the actual start address of the respective segment (CS, DS, SS or ES) in memory. In protected mode, though, the

value placed in the segment register (usually referred to as the Segment Selector when in protected mode) actually contains the following information:

- Requester Privilege Level (RPL) field
- Table Indicator (TI) bit
- Table index field

Refer to figure 5-19. The RPL field indicates the privilege level of the program that is attempting to access the target segment. The Table Indicator (TI) bit points to one of two tables in memory:

- 0 = the Global Descriptor Table
- 1 = the Local Descriptor Table

The index portion of the value placed in the segment register (or segment selector) provides an index into the table indicated by the Table Indicator bit.

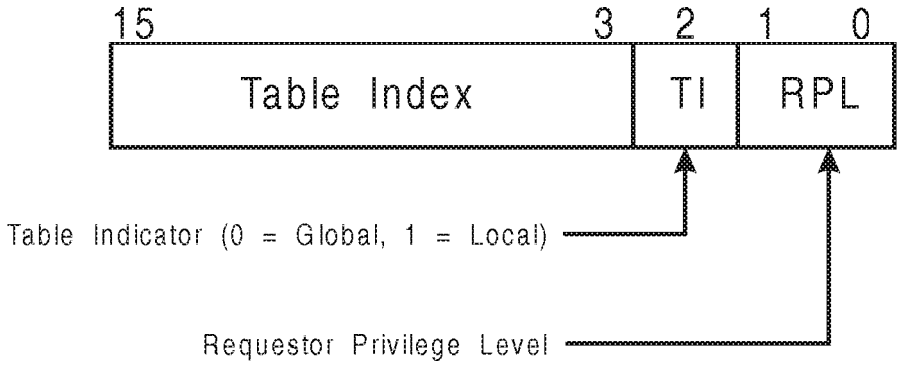


Figure 5-19. The Segment Selector

In addition to the segment registers, the 80286 also has three registers known as the Global Descriptor Table Register (GDTR), the Local Descriptor Table Register (LDTR) and the Interrupt Descriptor Table Register (IDTR). The operating system programmer must load the start memory addresses of each of the three tables into these registers.

Loading a new value into a segment register causes the 80286 microprocessor to automatically generate a series of four memory read bus cycles to get the 8-

byte segment descriptor from the indicated entry in the table identified by the TI bit. The microprocessor computes the start memory address of the 8-byte segment descriptor by multiplying the index by 8 (because each table entry is 8-bytes long) and adding the resulting offset to the contents of the respective Descriptor Table Register (either Global or Local).

The 8-byte segment descriptor read from memory is loaded into a special microprocessor register known as a descriptor cache register. There is a corresponding cache register for each of the four segment registers.

As an example, when a value is loaded into the CS register, the microprocessor uses the TI bit to identify the table to access (Global or Local), multiplies the index by eight and adds it to the table start address from the GDTR or LDTR. The 8-byte code segment descriptor is read into the microprocessor's code segment cache register. Only when the microprocessor has the segment descriptor can it add the offset supplied by IP register to the code segment's actual start address and generate the bus cycle to access the desired memory location in the code segment.

This rather involved process may seem tremendously inefficient, and would be if the microprocessor had to go through all of this every time it needed to access a memory location. In actuality, the microprocessor only has to read a segment descriptor from memory when a new value is placed in a segment register by the programmer. All subsequent memory accesses within the same segment happen quickly because all of the segment's descriptive information is already on-board the microprocessor chip in the cache register and can be used immediately in forming the actual physical memory address.

Refer to figure 5-20. A segment descriptor is 8 bytes in length and describes everything you ever wanted to know about the segment:

1. The 24-bit start address of the segment in memory. This means the programmer can specify a start memory address anywhere in the microprocessor's 16MB address range.
2. Unlike real mode, where the segment length is fixed at 64KB, the length of the segment can be specified as anywhere between 1 byte and 64KB.
3. Segment access rights byte. The bits in this byte allow the operating system to specify protection features associated with the segment being described.

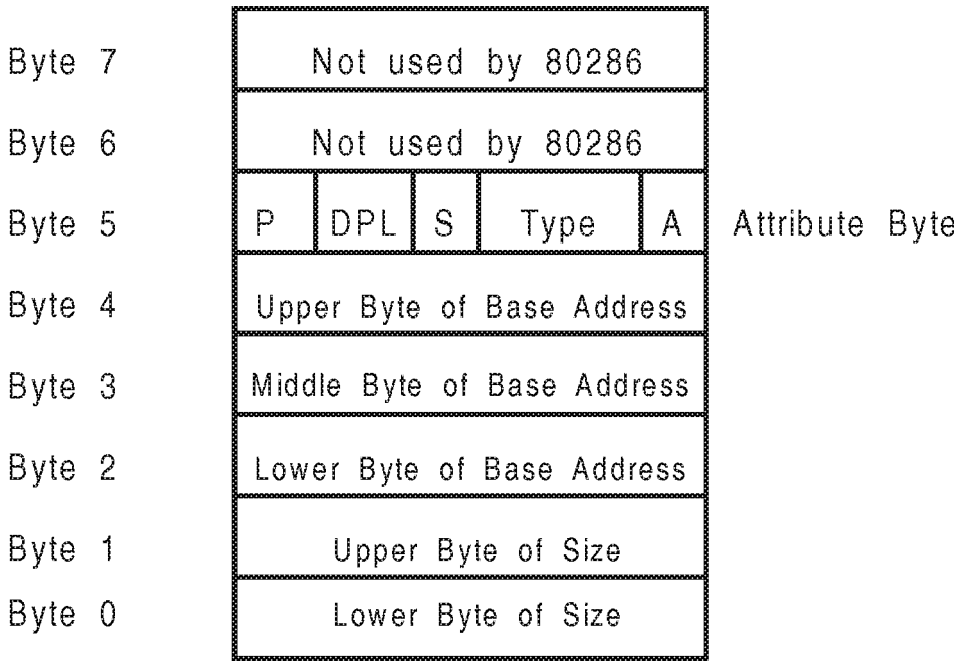


Figure 5-20. The Segment Descriptor

Bytes 0 and 1, the first two bytes in the descriptor, specify the size of the segment being described. The next three bytes supply the actual start address of the segment in memory. Since this is a 24-bit value, the start address may be anywhere in the 16MB memory address space of the 80286 microprocessor. The next byte is referred to as the attribute byte and defines the following characteristics of the segment:

Table 5-10. Attribute Byte Definition

Field Name	Purpose
A	Accessed bit. This bit will be 0 if the segment has not been accessed.
TYPE	This 3-bit field defines the type of segment. Using this field, the segment may be defined as: –an executable code segment (only instruction fetches may be performed from the segment). This prevents another program from reading these instructions so they can be disassembled. –a code segment that may be read as data or instructions. This code segment may be read by another program so it can be disassembled. –a read/writable data segment. –a read-only data segment.
S	S = 1 for a Data or code segment. S = 0 for a Control Descriptor.
DPL	Descriptor Privilege Level. The program attempting to access the segment must have a Requester Privilege Level (RPL) equal to or greater than the DPL specified here.
P	If 0, descriptor isn't valid (doesn't describe a segment of memory). If 1, descriptor is valid.

Note: An in-depth description of segment descriptors and other Protected Mode subjects is outside the scope of this book.

Chapter 6

The Previous Chapter

The first five chapters introduced the concepts of microprocessor communications and provide a detailed understanding of the 80286 microprocessor's interface signals.

This Chapter

ISA products usually have six separate source for reset, two of which are generated only by hardware and four that can be initiated by software commands. This chapter details the reset sources within typical ISA systems.

The Next Chapter

The next chapter explains the sequence of events that occur immediately after an ISA system is powered on.

The Power Supply Reset

The power supply provides the operating voltages necessary for system operation. When the power switch is first placed in the on position, it takes some time for the power supply's output voltages to reach their proper operating levels. If the system components were allowed to begin operating before the voltages stabilized, it would result in erratic operation.

Every PC power supply produces an output signal commonly called POWERGOOD. On the system board, the POWERGOOD signal is used to produce the RESET signal. During the period required for stabilization of the output voltages, the POWERGOOD signal is kept deasserted by the power supply. While POWERGOOD is deasserted, the RESET signal is kept asserted.

While RESET is asserted, it has two effects on the microprocessor and other system components:

1. Keeps any activity from occurring until power has stabilized.
2. Presets the microprocessor and other system devices to a known state prior to letting them begin to do their job. This ensures that the machine will always start up the same way.

When POWERGOOD becomes asserted, the RESET signal is deasserted and the microprocessor can begin to fetch and execute instructions. The first instruction is always fetched from the power-on restart address which is always located 16 locations from the very top of the memory address space. For the 8088 microprocessor, this would be FFFF0h, FFFFF0h for the 80286 and 80386SX, and FFFFFFF0h for the 80386DX, 80486 and Pentium microprocessors. This location contains the first instruction of the POST.

Reset Button

Some systems provide a momentary-contact switch which the user may press to force the RESET signal to be asserted to the microprocessor and to the rest of the system. The effect on the system is the same as deasserting POWERGOOD from the power supply, but the power supply is not actually turned off.

Shutdown Detect

When the microprocessor detects an exception while attempting to execute the handler for a prior exception, the two exceptions are generally handled serially. Certain combinations of exceptions cannot be handled serially, however. These exceptions are caused by protection violations in protected mode. For example, in protected mode, if an application attempts to access data that is outside its data segment, the microprocessor will start running the general protection fault exception handler. If, say, a stack overflow were to occur while the microprocessor ran the general protection fault exception handler, the microprocessor would not be able to handle the two faults serially. The microprocessor would invoke the double-fault exception instead.

If the microprocessor detects another exception while attempting to service the double-fault, it goes into a shutdown condition. This is sometimes referred to as a triple fault. The microprocessor indicates the shutdown condition to external logic by running the halt or shutdown bus cycle. Since the microprocessor has abnormally ceased to fetch and execute instructions, it can't run a program to inform the user that a severe error has occurred. Therefore, in an ISA machine, the shutdown detect logic outside the microprocessor detects the shut-

down and toggles the microprocessor's RESET line, causing the system to reboot.

See the chapters entitled "Detailed View of the 80286 Bus Cycle" and "Detailed View of the 80386 Bus Cycle" for additional information on the halt or shutdown bus cycle.

Hot Reset

MS-DOS was written specifically for the Intel 8088 microprocessor using 8088-specific instructions. Since the 8088 only has 20 address lines, it is incapable of generating a memory address greater than FFFFFh, or 1MB. Furthermore, protected mode wasn't introduced until the advent of the 80286 microprocessor. Consequently, MS-DOS can only address the lower 1MB and doesn't understand protected mode.

When an 80286 is powered up, it operates in real mode. In other words, it operates as if it were an 8088. This means that, although the 80286 has twenty-four address lines and can physically address up to 16MB of memory address space, it is limited to the lower 1MB when in real mode.

Most current applications programs are written to run under MS-DOS. Many of these programs require access to more memory space than allowed under MS-DOS. As an example, many Lotus 1-2-3 spreadsheets require very large amounts of memory space, well in excess of that allowed under MS-DOS and the 8088 microprocessor.

If a system is based on the 80286 microprocessor, the problem can be solved by switching the microprocessor into protected mode. This allows the microprocessor to access up to 16MB of memory space, but MS-DOS presents a problem. It doesn't understand protected mode and is therefore incapable of generating addresses above 1MB.

Special software that understands protected mode operation prepares the segment descriptor tables in memory prior to switching into protected mode. It also must save the address of the next instruction to be executed when the microprocessor returns to real mode so MS-DOS can continue to run at the point where it went to protected mode. The 80286 is then switched into protected mode and the extended memory above 1MB accessed (for example, to store spreadsheet data).
Once the extended memory access has been completed, the microprocessor must be switched back into real mode so the applications program can continue

to run under MS-DOS. Here's where the problem comes in. In order to switch the 80286 microprocessor from protected to real mode, the microprocessor must be reset.

To do this, the following actions must be taken:

1. The programmer stores an address pointer that points to the address of the real mode instructions that are responsible for restoring the system to its previous operating condition. This pointer is stored in locations 0040:0067 to 0040:006A. The contents of these locations will be used later when the processor returns to real mode to find the address in memory where the real mode instructions reside.
2. A special value (05h or 0Ah) is stored in the configuration CMOS RAM location (0Fh) to indicate the reason for the reset. Refer to table 6-1 for definitions of the reset byte in CMOS RAM.
3. The system has now been prepared to return to real mode. The Hot Reset command must now be issued to the keyboard/mouse interface.* This is accomplished by writing a FEh to the keyboard/mouse interface's command port at I/O address 0064h.
4. In response, the keyboard/mouse interface pulses its Hot Reset output one time, causing the hardware to generate a reset to the 80286.
5. When reset becomes deasserted, the microprocessor begins to fetch and execute instructions at the power-on restart address exactly as if a power-up had just occurred.
6. At the beginning of the POST, the programmer reads the value stored in the configuration RAM location to ascertain the reset's cause. In this case, the value (05h or 0Ah) indicates that it was caused by Hot Reset to get back to real mode and continue program execution.
7. POST then retrieves the previously stored real mode address pointer found at location 0040:0067 and jumps to the indicated address and picks up where it left off in real mode.

* Some manufacturers employ a Hot Reset "intercept" where external circuitry recognizes an FEh written to the keyboard controller and immediately generates a Fast Hot Reset. This provides a faster reset signal compared to the slower generation via a command to the keyboard controller. Later versions of Intel's microcontrollers (typically used as keyboard controllers) provide an internal intercept that also results in a fast Hot Reset.

Another method used to produce a fast Hot Reset is to cause the microprocessor to go into the shutdown condition by intentionally causing faults using software. This causes the shutdown detect logic to reset the microprocessor.

Alternate (Fast) Hot Reset

The Alternate or Fast Hot Reset command is found in some ISA systems that implement system control port A at I/O address 0092h, sometimes called the PS/2 compatibility port. Alternate Hot Reset performs the same function as the Hot Reset command. However, the microprocessor is reset more quickly using this method than when using Hot Reset. The Hot Reset command must be interpreted by the ROM-based program inside of the keyboard/mouse interface, while the Alternate Hot Reset command is executed by the hardware much more quickly.

Alternate Hot Reset is generated by setting bit 0 in system control port A to a one. This generates a pulse on the Alternate Hot Reset signal line which, in turn, causes a pulse on the Hot Reset signal. This resets the microprocessor. For a more extensive explanation of Hot Reset, see the previous section.

Ctrl-Alt-Del Soft Reset

When the control and alternate keys are depressed and held, followed by the delete key being depressed, three "make" codes for these keys are stored in the keyboard buffer in memory. When the sequence of "make" codes are detected by the keyboard interrupt service routine, the soft reset routine is invoked. The soft reset routine causes the system to:

1. Place the value 1234h into the reset flag location (0040:0072) in main memory. This value tells the system to skip a portion of the POST when rebooting.
2. Store a special value (00h)in configuration RAM to indicate that a Control/Alternate/Delete is causing the reset (also used for power-on reset).
3. Depending on the computer model and manufacturer one of two methods is typically used to reset the processor.
   - Write a FEh to port 64h, causing a CPU reset. When the reset is removed, the microprocessor begins to fetch and execute code from location FFFF0h, the beginning of POST.
   - Jump to location FFFF0h and begin program execution.
4. Near the beginning of the POST, the programmer reads the value stored in the configuration RAM location to ascertain the reset's cause. In this case, the value indicates that it was caused by Control/Alternate/Delete or power-on reset.

5. Next, the reset flag at location 0040:0072 is checked for the value 1234h. 1234h indicates a Ctrl-Alt-Del has caused the reset condition and as a result, only a subset of the POST is executed.

Table 6-1. CMOS Reset Code Definition

Value	Meaning
00h	Normal Power-Up reset or Ctrl-Alt-Del reset depending on the value of the Reset flag word in memory at location 0040:0072. When the reset flag contains the value 1234h, the POST memory test is skipped.
04h	POST is skipped. Causes DOS to be loaded from disk.
05h	POST is skipped and memory preserved. The 80286 microprocessor cannot be switched from protected mode into real mode without resetting the processor. This presents problems when the programmer wishes to temporarily switch to protected mode to access extended memory and then switch back into real mode to continue execution. When a program must temporarily leave real mode and enter protected mode, the programmer first stores a four byte memory address (pointer) into memory location 0040:0067. This pointer specifies the address at which real mode execution will resume. Having done this, the program may switch the processor into protected mode and access extended memory. Upon completing the protected mode function, the programmer must place the value 05h or 0Ah into the reset code byte in CMOS RAM and then reset the 80286 microprocessor. When reset goes away, the processor is in real mode and begins executing the POST. The POST code checks for a 05h or a 0Ah in the reset code byte, and, if present, jumps to the memory address specified in memory starting at location 0040:0067 to resume execution at the point where it left off in real mode. In addition, if the reset code byte value is 05h, an EOI command is issued to the interrupt controller.
09h	Block-move return. This value is placed in the reset code byte by a BIOS call to INT 15h to move a block of information between conventional and extended memory. In order to accomplish this move, the processor must switch into protected mode. On an 80286-based system, the INT 15h routine sets the reset code byte to 09h prior to resetting the processor to return to real mode after the block move. When reset, the processor starts executing the POST, checks the reset code byte value, and then performs an interrupt return, or IRET, instruction to return to the original program after completion of the call to the INT 15h BIOS routine.
0Ah	Jump to real mode pointer 0040:0067 without issuing EOI. See the explanation for reset code byte = 05h.

Chapter 7

The Previous Chapter

The previous chapter, "The Reset Logic," provided a detailed description of the possible sources for RESET.

This Chapter

This chapter provides a description of the ISA machine power-up sequence from the moment power is applied until the microprocessor begins to fetch and execute instructions.

The Next Chapter

The next chapter, "The 80286 System Kernel: the Engine," describes the support logic necessary to allow the 80286 microprocessor to communicate with 8- and 16-bit devices.

The Power Supply – Primary Reset Source

The power supply provides the operating voltages necessary for system operation. When the power switch is first placed in the on position, a period of time is required for the power supply's output voltages to reach their proper operating levels. If the system components are allowed to begin operation before the supply voltages have stabilized, erratic operation would result.

Figure 7-1 illustrates the power supply-related logic that produces the microprocessor's RESET signal. Every ISA PC power supply produces an output signal commonly called POWERGOOD. On the system board, the RESET signal is derived from the POWERGOOD signal. In order to guarantee proper operation of the microprocessor, the RESET input must change state in synchronism with the CLK2 signal (the microprocessor's double-frequency clock input). On each rising edge of CLK2, the state of the POWERGOOD input is latched by the flip-flop, inverted to its opposite state, and presented on the flip-flop's output, RESET. During the period required for stabilization of the output voltages, the

POWERGOOD signal is held deasserted by the power supply. While POWERGOOD is deasserted, the RESET signal is held asserted. The flip-flop in the figure provides a RESET output that is synchronized with CLK2 and is the invert of the POWERGOOD signal from the power supply.

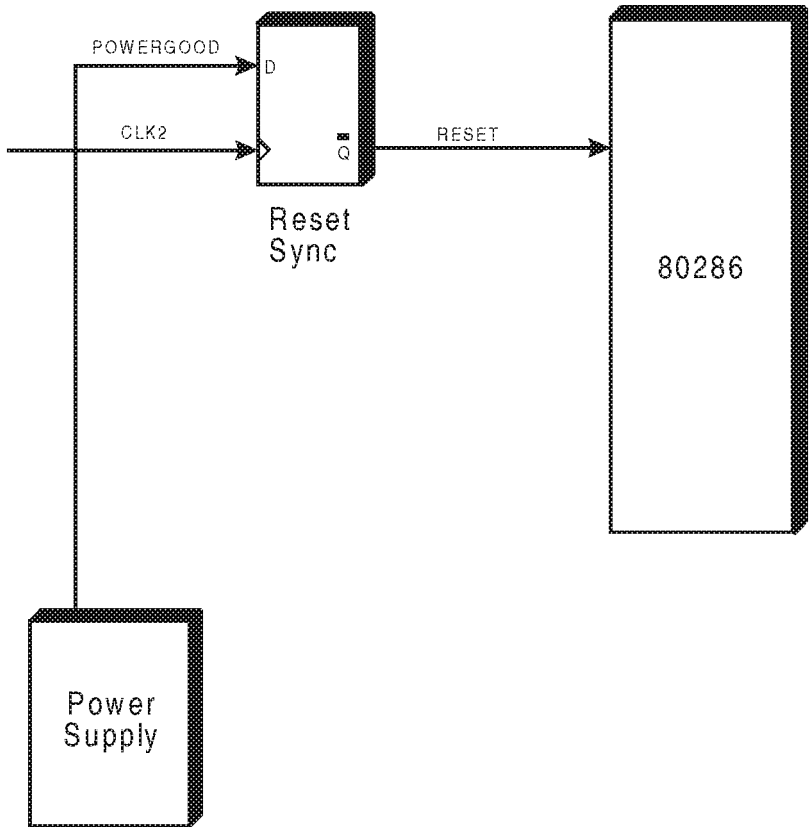


Figure 7-1. RESET Is Derived from POWERGOOD

While RESET is asserted, it has two effects on the microprocessor and other system components:

1. Keeps any activity from occurring until power has stabilized.
2. Presets the microprocessor and other system devices to a known state prior to letting them begin normal operation. This ensures that the machine will always start up the same way.

How RESET Affects the Microprocessor

Table 7-1 defines the values forced into the 8086 and 8088 microprocessor's registers when RESET is asserted.

Table 7-1. Values Preset Into 8086 and 8088 Microprocessor Registers by RESET

Register	Contents
FLAGS	0002h
MSW	FFF0h
IP	FFF0h
CS	F000h
DS	0000h
ES	0000h
SS	0000h

While RESET is asserted, the microprocessor cannot fetch and execute instructions and its outputs are set as illustrated in tables 7-2 and 7-3.

Table 7-2. State of 80386DX Outputs While RESET is Asserted

Pin Name	Pin State
LOCK#, D/C#, ADS#, A[31:2]	high
W/R#, M/IO#, HLDA, BE#[3:0]	low
D[31:0]	tri-state (floating)

Table 7-3. State of 80286 Outputs While RESET Is Asserted

Pin Name	Pin State
A[23:0], S0#, S1#, PEACK#, BHE#, LOCK#	high
M/IO#, COD/INTA#, HLDA	low
D[15:0]	tri-state (floating)

Processor Reaction When Output Voltages Stabilize

When the power supply output voltages have stabilized, the POWERGOOD signal is asserted and the logic on the system board responds by deasserting RESET. The deasserted level on RESET allows the microprocessor to begin functioning. It should be noted that the 386, 486 and Pentium microprocessors can perform a self-test when RESET becomes deasserted. Details of the processor self-test are discussed in the chapter entitled “The 80386DX and SX Micro-

processors" and in the MindShare Architecture Series books *80486 System Architecture* and *Pentium Processor System Architecture* published by Addison-Wesley. The microprocessor initiates the fetching and executing of instructions from memory, using the code segment (CS) and instruction pointer (IP) register contents to point to the memory location containing the first instruction.

While RESET was asserted, the machine status word (MSW) register had FFF0h forced into it. Since the protect enable bit, bit 0, is therefore 0, the microprocessor always begins operation in real mode. In real mode, the microprocessor always appends an extra 0h on the end of the CS register contents, F000h, resulting in a code segment start address of F0000h. It then adds the offset portion of the address, FFF0h, contained in the IP register, to the segment start address:

```
CS      F0000h
IP    + FFF0h
        FFFF0h = physical memory address
```

The resultant memory address, FFFF0h, is referred to as the power-on restart address. Since the address is formed exactly the same way every time the system is powered up, the 8086 or 8088 microprocessor always fetches its first instruction from the power-on restart address.

The First Bus Cycle

The microprocessor then initiates a memory read bus cycle to fetch the first instruction from the power-on restart address in memory. The power-on restart address is always located in the boot ROMs. This is because the first instruction must be located in ***non-volatile memory***. If the first instruction were fetched from RAM memory, the information returned would be junk. This is because RAM memory is ***volatile***.

The first instruction fetched from the power-on restart address is always the first instruction of the power-on self-test, or POST. The POST program is contained in the boot ROMs and is always the first program to be run.

Although they always begin operation in real mode and should therefore function as a fast 8086, the 286, 386, 486 and Pentium microprocessors have more than twenty address lines. When the 286 or higher microprocessors place the power-on restart address on A[19:0] of the address bus, the address lines above A19 are automatically set high. This results in address FFFFF0h being placed

on the 286 and 386SX address bus, rather than 0FFFF0h. The 386DX, 486 and Pentium processors place FFFFFFF0h on the address bus.

The processor keeps these upper address lines high until the program performs a jump to another code segment; in other words, until a new value is loaded into the CS register. A jump instruction that reloads both the CS and the IP registers is referred to as an "inter-segment jump" or a "far jump." The upper address lines are then set low and remain low unless the processor is switched into Protected Mode by the program. In practice, the first instruction executed from system ROM is always a jump instruction, thus a new CS value is immediately loaded when the first instruction is executed.

After the upper bits are set low, they will not be set high again while the processor remains in real mode. This means that the processor cannot access extended memory (memory above the lower megabyte) while in real mode. There is one exception to this rule. It is discussed in the chapter entitled "The 80286 Microprocessor" under the heading "Accessing Extended Memory in Real Mode."

Chapter 8

The Previous Chapter

The preceding chapters have explained how the 80286 microprocessor communicates with memory and I/O devices and how the 80286 interfaces with external devices. Sources of system reset and the power-up sequence have also been covered.

This Chapter

This chapter introduces and explains the external logic required by the 80286 microprocessor to communicate with 8- and 16-bit devices. Additional logic needed to synchronize memory and I/O devices of varying speeds is also covered. A thorough understanding of this chapter is crucial to understanding the 80386 kernel.

The Next Chapter

The next chapter provides a detailed look at the bus cycles that an 80286 can run. The chapter includes timing diagrams and explanations for read, write, and the halt or shutdown bus cycles.

The Bus Control Logic

In order to perform all of the actions required to complete a bus cycle, the 80286 microprocessor requires the aid of external logic. One of the major pieces of logic involved in this process is referred to as the bus control logic. Refer to figure 8-1.

Logic external to the microprocessor can detect the beginning of a bus cycle by looking at the 80286's bus cycle definition outputs. When either S0# or S1# is detected going low, this indicates the start of the bus cycle. This triggers the bus control logic's state machine that works with the microprocessor's Bus Unit to accomplish a data transfer during a bus cycle. The bus control logic uses CLK2, the double frequency clock, to define time slots for its state machine. At the

proper points during a bus cycle, the bus control logic performs the appropriate actions to accomplish the bus cycle.

The functions performed by the bus control logic are described in this chapter.

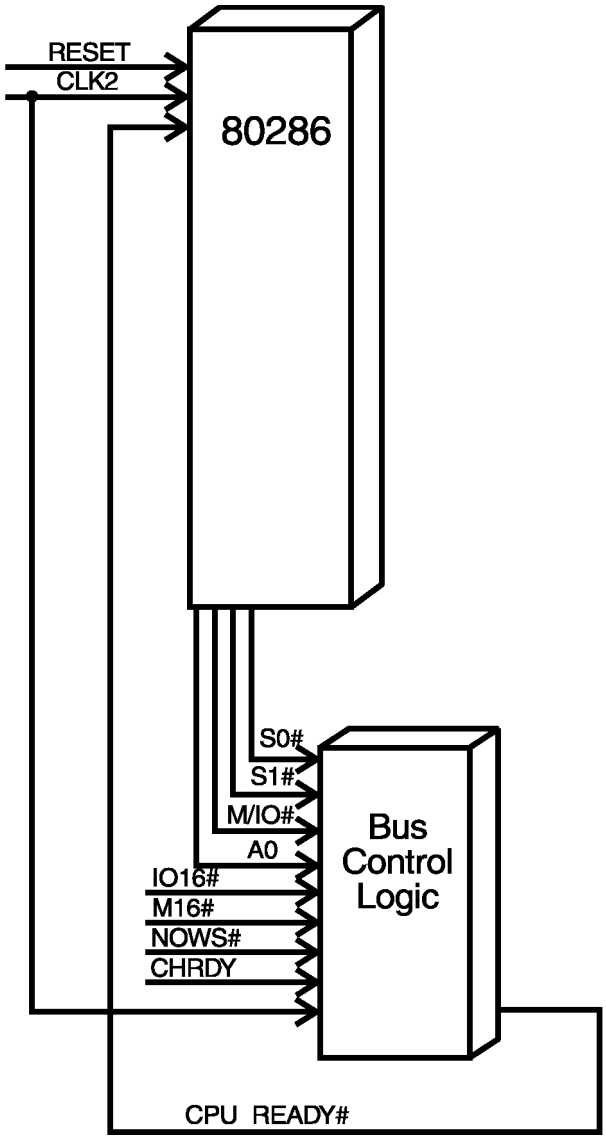


Figure 8-1. The Bus Control Logic

The Address Latch

Refer to figure 8-2. Intel dictates that every 80286-based system must incorporate an external device known as an address latch. When the microprocessor outputs the address onto its local address bus during a bus cycle, the bus control logic signal ALE (address latch enable) commands the address latch to hold (latch) the address and remember it. Once latched, the address latch outputs the address to the system on the system address (SA) bus, SA[19:1]. Address decoders throughout the system can then examine the latched address to see if the microprocessor is attempting to communicate with their respective device.

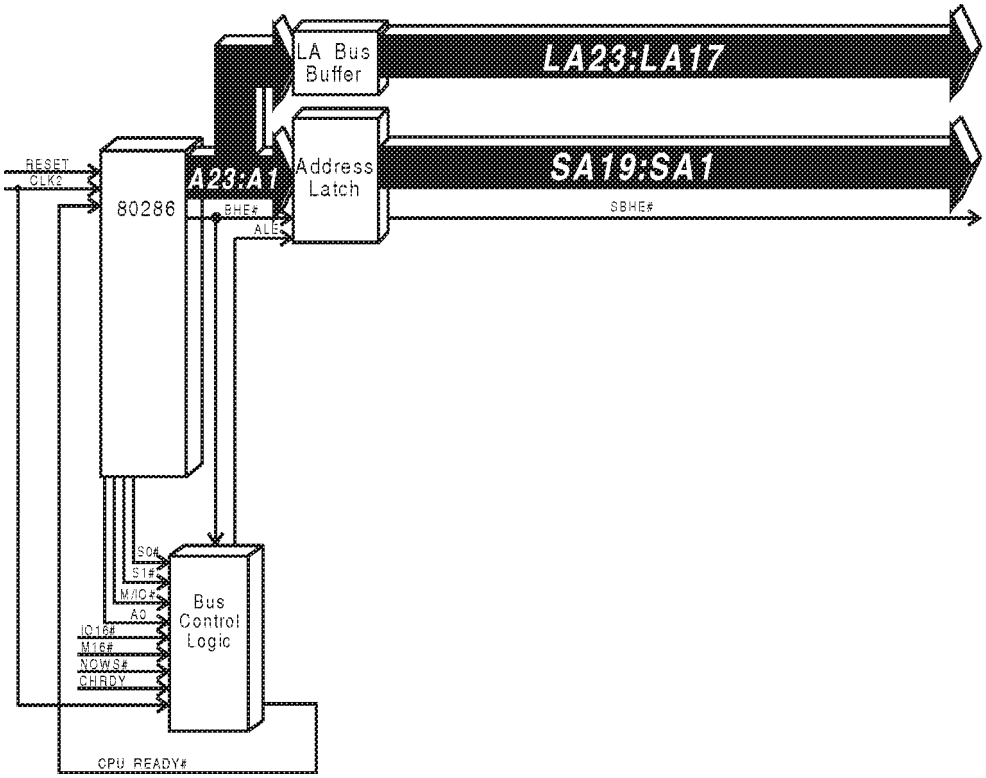


Figure 8-2. The Address Latch

The address latch and LA lines (latchable address lines) must be included to support the 80286's address pipelining capability. Notice that address lines A[23:20] are not latched. These address lines go directly to the LA bus through a buffer, along with address lines A[19:17]. From the buffer, the LA bus is connected directly to the 16-bit slots on the ISA bus. The purpose of the LA bus will be described later.

Notice that the address latch only latches A[19:1], but not A0 (address bit A0). The reason for this is clarified later. Rather than going to the address latch, A0 goes to the bus control logic where it is allowed to pass through onto SA0 at the same time as A[19:1] are latched and presented on SA[19:1].

Address Pipelining

When the 80286 microprocessor must perform back-to-back memory transfers, it uses address pipelining. This involves placing the address for the next cycle on the microprocessor's local address bus during the current bus cycle. This is possible because the current address will have been latched on the system address bus (SA Bus) and held for the duration of the current bus cycle. As a result, the address for the next bus cycle can be output by the microprocessor since the address for the current bus cycle has already been latched.

A portion of the next bus cycle's address, LA[23:17], goes directly to the ISA bus via an address buffer. LA[23:17] (the latchable address bus) are simply a buffered version of the uppermost bits of the microprocessor's address bus. 16-bit memory expansion boards are designed to decode the LA address lines. This allows 16-bit memory expansion boards to chip select their memory devices much earlier than would be possible if they waited for the latched address (SA lines).

At the end of address time of the next bus cycle, the bus control logic again pulses the address latch enable (ALE) line, commanding the address latch to latch address lines SA[19:0] and SBHE#. Since the high order bits (LA[23:17]) have already been used to generate the chip select, address decoding can be completed very quickly. Additional information regarding address pipelining can be found in the chapter entitled "Detailed View of the Bus Cycle."

Although the 80286 microprocessor pipelines out its entire 24-bit address, the ISA bus only provides seven of those lines, LA[23:17]. It would have been more expensive to include all 24 of the pipelined address lines and system designers are always looking for ways to reduce cost while maintaining or improving performance. If the designers had provided all 24 lines, the cost would have gone up but the performance would not have improved. There are two reasons for this: 1) the vast majority of bus cycles are memory reads since the processor is constantly fetching instructions and 2) the 4164 memory chip was popular when the PC/AT was designed. The 4164 has 64Kbits of information in it, but it's only one bit wide. Sixteen 4164's are needed to make a bank of memory for the 80286 microprocessor. Sixteen times 64Kbits yields 128KB, which is the smallest block of addresses that can be produced using LA[23:17]. With LA[23:17], the memory subsystem has enough information to select the proper bank of memory chips. Additional information about memory banks can be found in the chapter entitled "RAM Memory: Theory of Operation."

All I/O addresses are below 128K, so the LA bus always has zeros on it when I/O devices are addressed. Therefore, I/O devices can't benefit from the LA bus because the I/O address decoders don't see enough address lines to distinguish one I/O device from another. Although the I/O devices could, in theory, have benefited from seeing the rest of the pipelined address, that benefit would have been too slight to justify the increased cost.

The Data Bus Transceivers

Refer to figure 8-3. In addition to the address latch, Intel also requires the inclusion of two devices known as ***transceivers***.

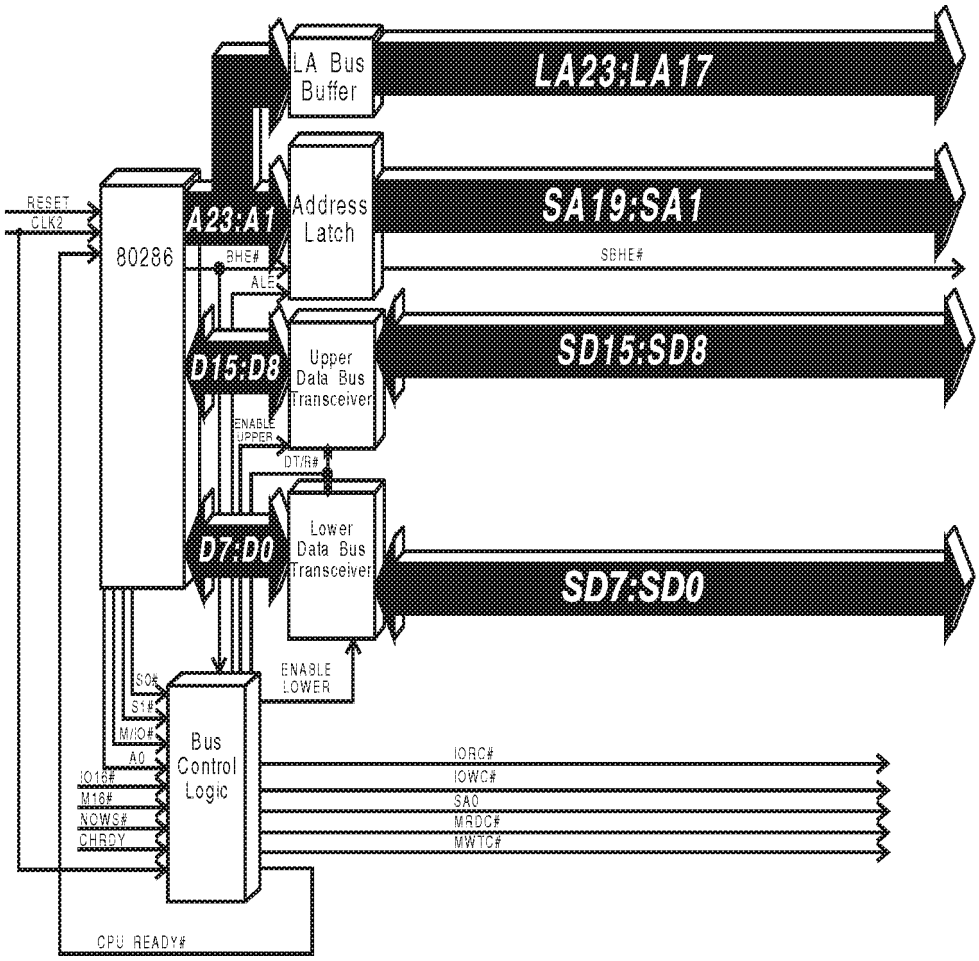


Figure 8-3. The Data Bus Transceivers

As can be seen in figure 8-3, each of the two data bus transceivers is connected between the CPU's local data bus and the system data (SD) bus. One of the transceivers is referred to as the lower data bus transceiver and the other as the upper data bus transceiver.

During a read bus cycle, the bus control logic places the transceivers into receive mode to pass data from right-to-left. In this way, the data flows from the currently-addressed device, over the system data bus, through the transceivers to the microprocessor. During a write bus cycle, the microprocessor drives data onto its local data bus. The transceivers are placed in transmit mode by the bus control logic, thereby allowing the data to flow onto the system data bus so it can be written to the currently-addressed device.

A transceiver has two control lines that allow the bus control logic to control its actions.

1. **DT/R#**. Data transmit or receive is set high to place the transceiver in transmit mode (left-to-right) during write bus cycles. It is set low during read bus cycles to place it in receive mode (right-to-left).
2. **DATA ENABLE**. When asserted (high), data enable allows the transceiver to actually pass the data in the direction selected by the state of the DT/R# control line. Note that the **DATA ENABLE** lines are called **ENABLE UPPER** for the upper data transceiver and **ENABLE LOWER** for the lower data transceiver.

At the proper moment during a bus cycle, the bus control logic always sets the direction line (DT/R#) to the appropriate level, and then asserts the appropriate data enable control line slightly afterwards to enable one or both of the data bus transceivers.

Data Bus Steering Logic

Refer to figure 8-4. Since 8-bit devices are not connected to the upper data path, any data transfer between the microprocessor and an 8-bit device must take place over the lower data path, one byte per bus cycle. The 80286 microprocessor, however, transfers data over the lower and upper data paths in accordance with the cardinal rules. Recall that the microprocessor transfers data between itself and odd locations using only the upper path. This means that when an odd location is involved in a transfer with an 8-bit device that external logic must "steer" the data between the upper and lower path. This ensures that both the microprocessor and the 8-bit device transfer data over the expected data path.

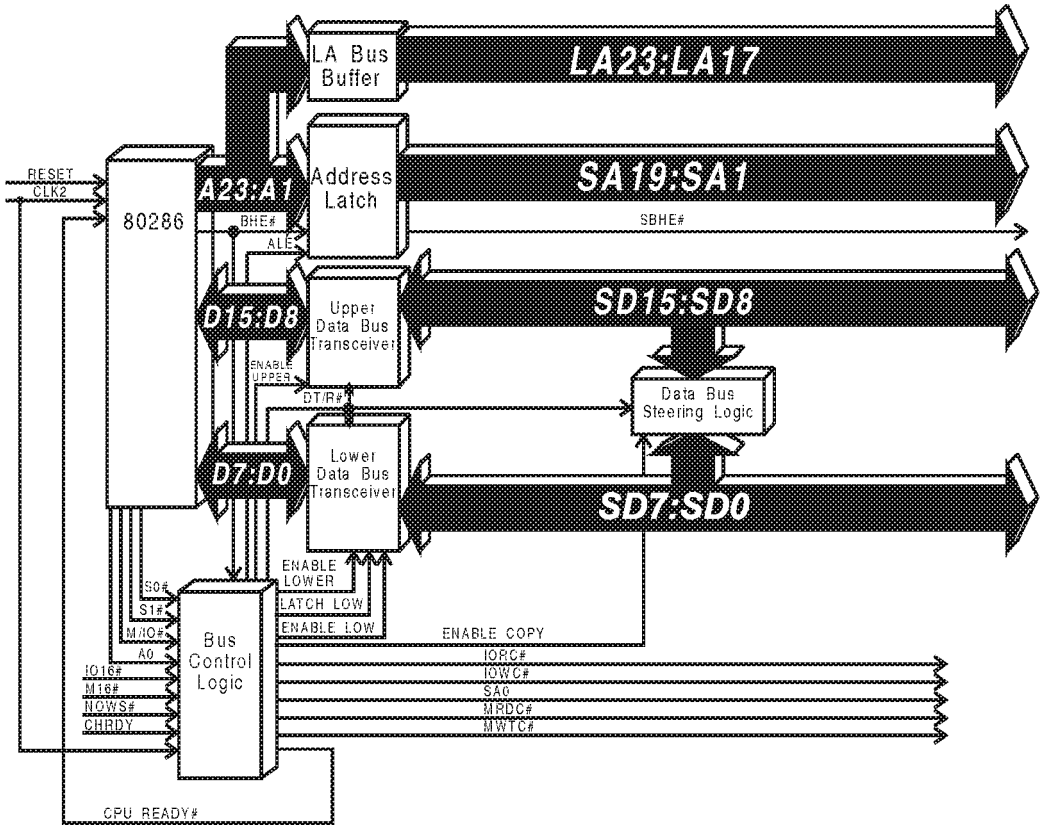


Figure 8-4. The Data Bus Steering Logic

Whenever an address decoder associated with a 16-bit device detects an address within its assigned range, in addition to chip-selecting its respective device it also asserts the ISA signal M16# (Memory size 16) or IO16# (I/O size 16). This tells the bus control logic that the microprocessor is communicating with a 16-bit device (as opposed to an 8-bit device). Note that the bus control logic samples either the M16# or IO16# signal depending on whether the bus cycle is referencing a memory or I/O location. Sampling the appropriate signal is important because both the M16# and IO16# signal may be asserted at the same time, since the generation of M16# is not qualified by whether the access is to a memory location or not.

Since two bytes can be transferred between the microprocessor and a 16-bit device during each bus cycle, data can be transferred twice as fast as it can with 8-bit devices and can be done without data path steering.

Circumstances requiring the use of the data bus steering logic are described in the following paragraphs.

Scenario One – Read Even-Addressed Location in 8-Bit Device

Assume that the microprocessor executes the following instructions:

```
MOV  DX,03F0    ;Floppy Status port to DX
IN   AL,DX      ;read Controller Status to AL
```

The first of these two instructions (MOV DX,03F0h) causes the value 03F0h to be placed in the microprocessor's DX register. This is the I/O address of the floppy disk controller's status register.

The second instruction (IN AL,DX) will cause the microprocessor to perform an I/O read bus cycle from the I/O address specified in the DX register (I/O location 03F0h). The data byte read from the status register is placed in the microprocessor's AL register.

While executing the second instruction, the microprocessor initiates an I/O read bus cycle, placing address 0003F0h on the address bus with BHE# deasserted and setting the bus cycle definition lines to the following:

M/IO#	=	0;	indicates an I/O operation
S0#	=	1;	deasserted, so it's not a write operation
S1#	=	0;	asserted, so it's a read operation

The bus control logic pulses the address latch enable (ALE) signal, causing the address to be latched into the address latch and presented on the system address (SA) bus. Address decoders throughout the system examine the address, but only the floppy disk controller logic is chip-selected. The bus control logic decodes the microprocessor's bus cycle definition outputs and asserts its IORC# (I/O Read Command line). Since it is a read operation, the floppy disk controller's status is placed onto the data bus by the floppy disk controller. The floppy disk controller is an 8-bit device, so the byte of status information is placed on the only part of the data bus available to an 8-bit device, SD[7:0].

Recall that, according to rule three, the microprocessor expects data read from even-addressed locations to come back over the lower data path. The bus control logic senses that the microprocessor is communicating with an 8-bit I/O device because IO16# is not asserted. Since the data from the floppy disk controller status register is already on the proper path (SD[7:0]), the bus control logic sets the lower data bus transceiver to receive mode (DT/R# = 0) and enables it to pass the data through (ENABLE LOWER = 1) to the microprocessor. The byte flows through the lower data bus transceiver onto the lower data path on the microprocessor's local data bus.

At the end of data time, CPU READY# is sampled asserted by the microprocessor. The microprocessor reads the data byte off the lower data path and the bus cycle ends. The status byte is placed in the microprocessor's AL register, completing the execution of the IN instruction.

In this example, everything worked just fine and no additional logic was required to complete the data transfer.

Scenario Two – 8-Bit Read from Odd-Addressed Location in 8-Bit Device

Assume that the microprocessor executes the following instruction:

```
IN   AL,61  ;read contents of Port 61 to AL
```

I/O port (address) 61h is an 8-bit I/O address that the programmer reads to ascertain the cause of a Non-Maskable Interrupt (NMI). The microprocessor will

initiate an I/O read bus cycle, placing address 000061h on the address bus with BHE# asserted and setting the bus cycle definition lines to the following states:

M/IO#	=	0;	indicates an I/O operation
S0#	=	1;	not asserted, so it's not a write operation
S1#	=	0;	asserted, so it's a read operation

The bus control logic pulses the address latch enable (ALE) signal, causing the address to be latched into the address latch and to be presented on the system address (SA) bus. Address decoders throughout the system examine the address, but only the port 61h logic is chip-selected. Since port 61h is an 8-bit device, the byte of information being read is placed on the only part of data bus available to an 8-bit device, SD[7:0].

According to cardinal rule number three, the microprocessor expects data read from odd-addressed locations to come back over the upper data path. The bus control logic determines that the microprocessor is communicating with an 8-bit I/O device because IO16# is not asserted. Since the data coming from port 61h is on the wrong path, something must be done. How would you fix this problem? All you have to do is copy the data to the proper data path. Another transceiver is added in between the upper and lower system data bus paths. This is referred to as the ***Hi/Lo Byte Copier***.

The bus control logic sets the upper path data bus transceiver to receive mode (DT/R# = 0) and enables it to pass the data through (ENABLE UPPER = 1) to the microprocessor. The low on DT/R# also sets up the Hi/Lo Byte Copier to pass data from the lower data path to the upper. The bus control logic then asserts the ENABLE COPY line. This enables the Hi/Lo Byte Copier to pass the data byte to the upper system data bus path, SD[15:8].

The byte flows through the upper data bus transceiver onto the upper data path on the microprocessor's local data bus. At the end of data time, CPU READY# is sampled asserted by the microprocessor. The microprocessor reads the data byte off the upper data path and the bus cycle ends. The byte is placed in the microprocessor's AL register, completing the execution of the IN instruction.

In this example, the additional logic referred to as the data bus steering logic was required to complete the data transfer. The data bus steering logic consists of:

1. Logic in the bus control logic that senses whether the bus cycle is to a memory or I/O location, and samples the appropriate "data size 16" signal line (either M16# or IO16#) so it can tell if the microprocessor is communicating with an 8- or 16-bit device.
2. Logic in the bus control logic that observes the A0 and BHE# outputs of the microprocessor to see what kind of transfer the microprocessor is attempting to perform (8-bit transfer with an even address; 8-bit transfer with an odd address; 16-bit transfer starting at an even address).
3. Hi/Lo Byte Copier.

Scenario Three – 8-Bit Write to Odd-Addressed Location in 8-Bit Device

Assume that the microprocessor executes the following instructions.

```
MOV  DX,03F5  ;set up I/O address 03F5h in DX
OUT  DX,AL    ;I/O write AL to I/O port 03F5h
```

When executing the OUT instruction, the microprocessor outputs the byte to be written to the odd address on the upper path of the local data bus (rule number three). The bus control logic sets the DT/R# control line high, placing the data bus transceivers into transmit (left-to-right) mode and asserts the ENABLE UPPER signal to enable the upper data bus transceiver. The byte flows from the upper path of the local data bus to the upper path of the system data (SD) bus. The bus control logic also decodes the microprocessor's bus cycle definition outputs and asserts its IOWC# (I/O Write command) line.

The bus control logic examines A0 and BHE# and determines that the microprocessor is performing a one byte transfer over the upper data path; and determines that the currently-addressed device is an 8-bit device by checking IO16#. This means that it must enable the Hi/Lo Byte Copier to copy the data byte from the upper to the lower data path so it can be written to the 8-bit device. To accomplish this, the bus control logic asserts its ENABLE COPY output.

Because DT/R# is set to transmit mode during write operations, the Hi/Lo Byte Copier copies the byte being output by the microprocessor on SD[15:8] down to SD[7:0] on the system data bus. This allows the data byte to get to the currently-addressed 8-bit device, which will place the byte into the odd-addressed location.

Scenario Four – 16-Bit Write to 8-Bit Device

Assume that the microprocessor executes the following instructions:

```
MOV  DX,03F4  ;set up I/O address 03F4h in DX
OUT  DX,AX    ;write AX to ports 3F4 and 3F5h
```

The microprocessor drives the two bytes from the AX register onto the two data paths, places I/O address 03F4h on the address bus and asserts BHE#. The bus control logic sets DT/R# high to set the data bus transceivers to pass from left to right. It also asserts ENABLE LOWER and ENABLE UPPER, allowing both transceivers to pass the two data bytes onto the SD bus.

The data bus steering logic recognizes that the current bus cycle is a 16-bit transfer by the lows on address bit A0 and BHE#. During this single bus cycle, the microprocessor will perform a 16-bit write transfer using both halves of the data bus simultaneously. When the bus control logic samples IO16# and finds it deasserted, however, it must perform two separate one-byte transfers over the lower data path.

Initially, A0 is allowed to flow through the bus control logic onto SA0 and the IOWC# line is asserted. The floppy disk controller is chip-selected and the data byte on the lower data path is written into location 03F4h in the floppy disk controller. After the bus control logic samples CHRDY asserted, it blocks A0 and forces SA0 to a one. This increments the address on the SA bus to 03F5h. The bus control logic also momentarily deasserts and then asserts the IOWC# line to fool the floppy controller into thinking another bus cycle has started. ENABLE LOWER is deasserted, blocking the data byte on the lower data path from getting through the lower data bus transceiver. ENABLE COPY is asserted and the data byte on the upper half of the SD bus is copied down to SD[7:0] so it can get to the floppy controller. The data byte is then written into location 03F5h in the floppy controller. When the bus control logic samples CHRDY asserted, it then sets CPU READY# asserted, allowing the microprocessor to end the bus cycle.

Although it may appear that this was two bus cycles, it was really one prolonged bus cycle that was stretched out by deasserting CPU READY# until the operations were completed.

Scenario Five – 16-Bit Read from 8-Bit Device

Assume that the microprocessor executes the following instructions:

```
MOV  DX,03F4    ;set up I/O address 03F4 in DX
IN   AX,DX      ;2 bytes from ports 3F4 & 3F5
```

The microprocessor places I/O address 03F4h with BHE# asserted on the address bus and sets the bus cycle definition lines to indicate that an I/O read bus cycle is in progress. The bus control logic sets DT/R# low to set the data bus transceivers to pass from right to left. It also asserts ENABLE LOWER and ENABLE UPPER, allowing both transceivers to pass the two data bytes from the SD bus to the CPU's local data bus.

The data bus steering logic recognizes that the current bus cycle is a 16-bit transfer by the lows on address bit A0 and BHE#. During this single bus cycle, the microprocessor will perform a 16-bit read transfer using both halves of the data bus simultaneously. When the bus control logic samples IO16# and finds it deasserted, however, it must perform two, separate one-byte transfers over the lower data path.

Initially, A0 is allowed to flow through the bus control logic onto SA0 and the IORC# line is asserted. The floppy disk controller is chip-selected and the data byte from location 03F4h in the floppy disk controller is driven onto SD[7:0]. After the bus control logic samples CHRDY asserted, it asserts LATCH LOW, causing the lower data bus transceiver to latch the data byte from location 03F4 and hold it in an internal register.

The bus control logic blocks A0 and forces SA0 to a one. This increments the address on the SA bus to 03F5h. The bus control logic also momentarily deasserts and then asserts the IORC# line to fool the floppy controller into thinking another bus cycle has started. ENABLE COPY is asserted and the data byte on the lower half of the SD bus is copied up to SD[15:8] so it can get to the microprocessor on the correct data path. When the bus control logic samples CHRDY asserted, it asserts ENABLE UPPER and ENABLE LOW simultaneously, causing the upper byte to pass through the transceiver to the microprocessor and the lower byte to be output to the microprocessor from the lower data bus transceiver's internal register.

The bus control logic then asserts CPU READY#, allowing the microprocessor to read the two bytes and end the bus cycle. The two bytes are then placed in the AX register, ending the execution of the IN instruction.

Although it may appear that this was two bus cycles, it was really one prolonged bus cycle that was stretched out by deasserting CPU READY# until the operations were completed.

Scenario Six – 8-Bit Read from 16-Bit Device

No intervention required by the data bus steering logic.

Scenario Seven – 16-Bit Read from 16-Bit Device

No intervention required by the data bus steering logic.

Scenario Eight – 8-Bit Write to 16-Bit Device

No intervention required by the data bus steering logic.

Scenario Nine – 16-Bit Write to 16-Bit Device

No intervention required by the data bus steering logic.

The Ready Logic

Access Time

The ***access time*** for a device is both device and manufacturer-dependent. This means that the access time for different types of RAM, ROM and I/O devices will differ. This being the case, you would expect each device type to have a READY# output pin that will be asserted when the device is ready to complete a data transfer. In reality, though, they have no such pin.

Figure 8-5 illustrates the concept of access time.

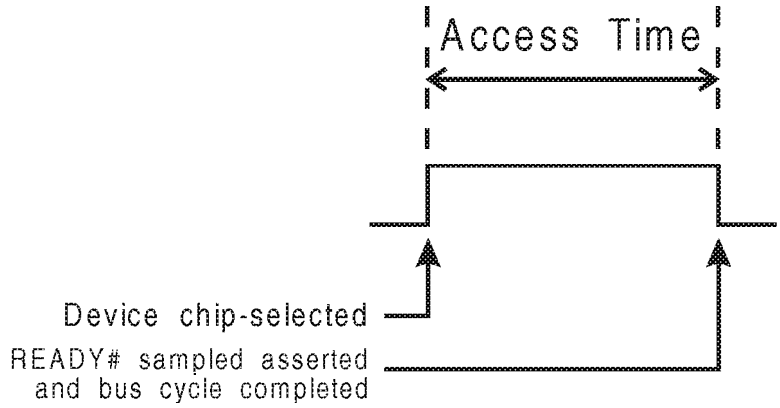


Figure 8-5. Access Time

Stretching the Transfer Time

When you begin to access a slow access device, the microprocessor's bus cycle must be stretched out to match the access time of the device. This is the purpose of the CPU READY# input on the microprocessor. Since there is no ready output from I/O or memory devices, some other method for deasserting the CPU READY# signal until the currently-addressed device is ready to complete the bus cycle had to be devised.

The Default Ready Timer

ISA systems use a single Default Timer for the different types of devices in the system. This default timer is located on the system board and is usually embedded within the bus control logic. Table 8-1 illustrates the number of wait states that are inserted into a bus cycle by the default timer when accessing I/O and memory.

Table 8-1. Number of Wait States—Default Ready Timer

Type of Device	Number of Wait States
16-bit RAM	1
16-bit I/O	1
8-bit RAM	4
8-bit I/O	4

It should be noted that the values stated in table 8-1 are based on an 80286 running at 8MHz. This was the speed of the processor in the version of the IBM PC/AT that became the industry standard. In other words, a bus cycle accessing an 8-bit device would cause the Default Ready Timer to deassert the 8MHz processor's CPU READY# input long enough to insert four wait states in the bus cycle. At 8MHz, a wait state is 125ns in duration. The total duration of the bus cycle would therefore consist of address time plus data time plus four wait states, for a total of 6 x 125ns = 750ns. A bus cycle accessing a 16-bit device would cause the Default Ready Timer to deassert the 8MHz microprocessor's CPU READY# input long enough to insert one wait state in the bus cycle. The total duration of the bus cycle would therefore consist of address time plus data time plus one wait state, for a total of 3 x 125ns = 375ns.

In later ISA machines, the processors are usually running substantially faster than 8MHz; however, the actual timing of the ISA bus is governed by the bus control logic to provide the same timing as an 80286 microprocessor running at 8MHz to 8.33MHz. This means that the actual number of wait states inserted in the microprocessor's bus cycle by the Default Ready Timer when communicating with 8- and 16-bit devices may be substantially greater than those stated in table 8-1.

This default timing is based on compatibility requirements for 8-bit boards designed for the original 8088-based IBM PC and on the speed of typical RAM chips at the time the IBM PC/AT was designed. The 8088 takes four ticks of its 4.77MHz clock to run a 0-wait-state bus cycle, so a 0-wait-state bus cycle takes 838ns (4 x 209.5ns = 838ns). Cards built for the original PC considered 838ns to be 0-wait-state performance, so the Default Ready Timer provides approximately that same delay (750nsec) as a default for 8-bit cards. The Default Ready Timer was also designed such that designers of ISA expansion boards could modify the default timing using custom ready timers as described in the following section.

Custom Ready Timers

Designers of ISA expansion boards may need to extend the default timing if the access time of their device is too long for the ISA defaults or they may want to shorten the default timing if their access time is shorter than the ISA default. Signals provided on the ISA bus permit designers flexibility in selecting devices with differing access times. The actual timing differs between 8- and 16-bit devices. Refer to figure 8-6. The following list shows the possible number of ISA wait states that can be achieved with both 8- and 16-bit devices.

- Shortened 8-bit device bus cycle: 1, 2 or 3 wait states
- Stretched 8-bit device bus cycle: more than 4 wait states
- Shortened 16-bit device ISA bus cycle (Memory only): 0 wait state
- Stretched 16-bit device ISA bus cycle: more than 1 wait state

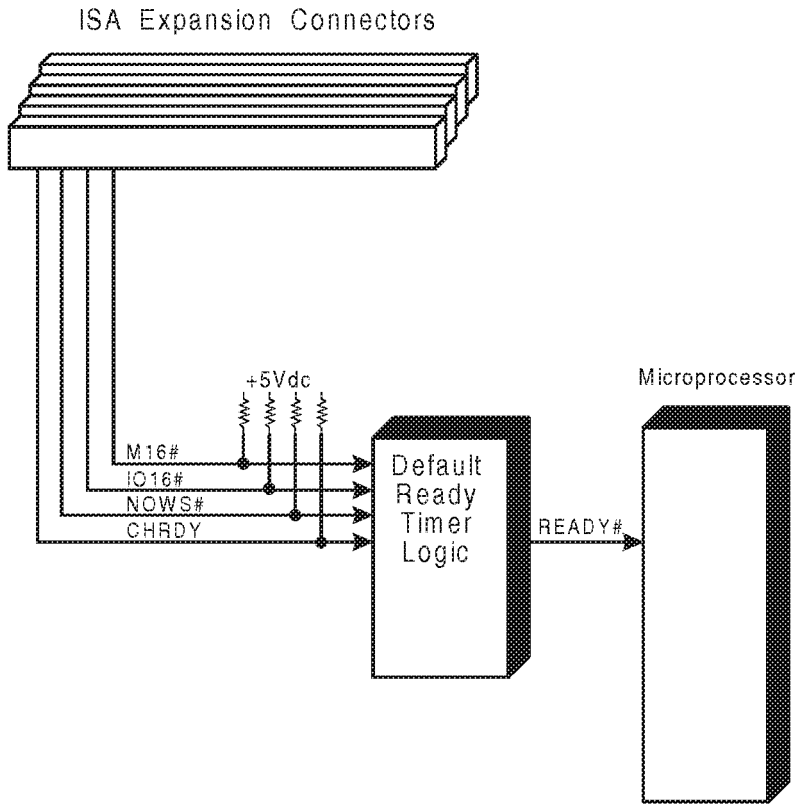


Figure 8-6. Custom Ready Timer Logic

Extending the Default Timing

When designing an ISA card, the access time for the device may be longer than the ready timeout provided by the Default Ready Timer. Designers can override the default timer and stretch out the bus cycle by the required number of wait states to match the access time of the device. This is implemented with the CHRDY signal (Channel Ready) on the ISA bus.

When the address decoder on an ISA card decodes an address assigned to its respective device, it not only chip-selects the device, but also triggers the custom ready timer on the card. When triggered, the timer forces the normally asserted CHRDY signal to become deasserted. When the default timer logic on the system board sees CHRDY become deasserted, the default timer is overridden and CPU READY# will not be asserted to the microprocessor until the CHRDY signal is seen to become asserted again. When the access time for the card has elapsed, the custom timer will then allow CHRDY to become asserted again. The default ready timer logic on the system board will then immediately assert CPU READY#, thus informing the microprocessor that it's OK to end the bus cycle. In this way, the designer of an add-in card can insert any number of additional wait states into a bus cycle when a particular device is addressed.

Shortening the Default Timing

The NOWS# (No Wait State) line provides a method to reduce the number of wait states associated with the default timing. When a fast ISA board is capable of delivering data faster than the default timing calls for, the board can assert the NOWS# line as soon as it recognizes it is being addressed; however, only 16-bit memory boards are capable of ending an ISA bus cycle in zero wait states. The default timer generates the CPU READY# signal as soon as it sees the NOWS# line become asserted, thereby ending the cycle earlier than the default time. Note that NOWS# doesn't necessarily or always mean ***no*** wait states; it means ***fewer*** wait states. Refer to the Chapter entitled "ISA Bus Cycles" for detailed information and timing requirements.

The IBM PC/AT was designed to permit zero-wait-state accesses only to 16-bit memory devices because only they can decode the LA (Latchable Address) lines. The LA lines are present on the ISA bus earlier than the latched System Address (SA) lines, meaning 16-bit memory devices can decode the address earlier than 16-bit I/O and 8-bit devices. Designers of 16-bit memory devices should latch the result of the current decode in the event of the LA lines being pipelined prior to the end of the current bus cycle. Failure to do so might cause the current device to be deselected when the new address is placed on the LA bus.

Chapter 9

The Previous Chapter

Previous chapters have covered how the microprocessor communicates with memory and I/O devices and have explained the support logic needed to allow the microprocessor to communicate with both 8- and 16-bit devices.

This Chapter

The exact timing of read and write bus cycles for both memory and I/O are covered in this chapter. Detailed explanations of the halt and shutdown bus cycles, along with the causes of each are also covered.

The Next Chapter

The next several chapters detail the 80386 microprocessor and the support circuitry required for the 80386 kernel.

Address and Data Time Revisited

The concept of the bus cycle was introduced in the chapter entitled "Introduction to the Bus Cycle." The microprocessor performs a bus cycle when it has to transfer information between itself and a memory or I/O location. The microprocessor's bus unit uses the address, data and control buses to address a device, indicate the type of transaction in progress and to transfer the data between the microprocessor and the currently addressed location.

The microprocessor uses the bus cycle definition lines to indicate the type of transaction. The types of bus cycles are indicated in table 9-1.

With the exception of the interrupt acknowledge bus cycle, this chapter provides a detailed description of each bus cycle type. The interrupt acknowledge bus cycle is described in the chapter entitled "The Interrupt Subsystem."

Table 9-1. 80286 Bus Cycle Definition

M/IO#	S1#	S0#	Bus Cycle Type
0	0	0	Interrupt Acknowledge
0	0	1	I/O Read
0	1	0	I/O Write
1	0	0	Halt or Shutdown
1	0	1	Memory Read
1	1	0	Memory Write

When it must perform a bus cycle, the microprocessor's bus unit leaves the idle state and enters address time. During this "tick" of PCLK, the microprocessor places the address and bus cycle definition on the buses. Address decoders throughout the system start decoding the address during this time slot.

Address time is always followed by data time. During this state, the microprocessor expects the data to be transferred between itself and the currently addressed device.

In reality, these aren't the state names Intel uses for the 80286 and 80386 bus unit states. Table 9-2 lists the state names used by Intel:

Table 9-2. 80286 and 80386 Bus Cycle State Names

Nickname	80286 Name	80386 Name
Address Time	Ts	T1
Data Time	Tc	T2

These states were renamed for clarity's sake because the names Ts and Tc provide little information regarding the actions that occur during these states. Intel calls address time Ts, meaning "Send Status" time, because one of the things that occurs during Ts is the output of the bus cycle definition on M/IO#, S0#, and S1#. Intel refers to S0# and S1# as the microprocessor's status outputs, hence the name "Send Status" time. Data time is called Tc, "Perform Command" time, because it's time to perform the command, or data transfer, that the microprocessor initiated.

The various actions that must take place during a bus cycle have been described in previous chapters. This chapter provides an in-depth look at the sequence and exact timing of these actions in relation to each other.

The Read Bus Cycle

With the exception of the M/IO# output, I/O and memory read bus cycles are identical. This section describes the exact timing of the actions performed during a read bus cycle. Figure 9-1 is a timing diagram illustrating a series of typical 80286 read bus cycles.

PCLK has been placed across the top of the timing diagram as a point of reference. Remember that each cycle of PCLK defines the duration of one bus unit state. Vertical dotted lines have been superimposed on the diagram to define each cycle of PCLK (and, therefore, each state). In addition, the name of each state is written across the top of the diagram.

A general idea of the information presented can be derived from the state names across the top. In this example, the sequence of states is as follows:

Tc	Data Time
Ts	Address Time
Tc	Data Time
Ts	Address Time
Tc	Data Time
Tc	Data Time
Ts	Address Time

Every bus cycle consists of at least an address time and data time pair. Additional data times (wait states) may be inserted in the bus cycle if the microprocessor samples its READY# input deasserted at the end of data time. Based on these rules, the states illustrated on the diagram can be interpreted as follows:

Tc	Data Time	end of previous bus cycle
Ts	Address Time	start of bus cycle A
Tc	Data Time	1st data time of bus cycle A
Ts	Address Time	start of bus cycle B
Tc	Data Time	1st data time of bus cycle B
Tc	Data Time	wait state inserted in bus cycle B
Ts	Address Time	start of next bus cycle

In other words, the diagram illustrates the end of a bus cycle followed by a 0-wait-state bus cycle (bus cycle A), a 1-wait-state bus cycle (bus cycle B) and the start of another bus cycle.

When reading a timing diagram, one scans from left-to-right (from earlier in time to later in time) looking for the signals that change first, second, and so on. Since PCLK should always be pulsing (changing), it's not necessary to note the fact that it changes.

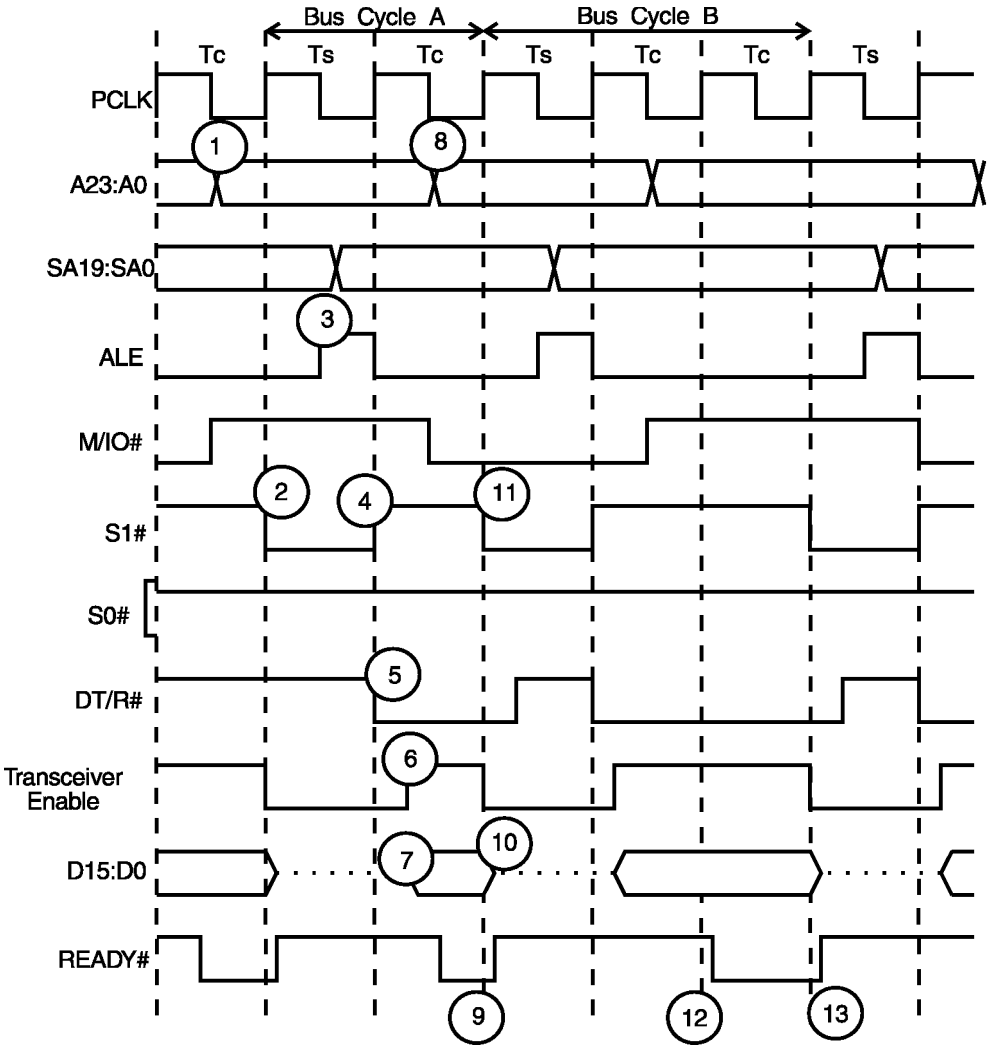


Figure 9-1. The Read Bus Cycle

Figure 9-2 shows the 80286 system engine. Referencing both the timing diagram and the functional block diagram will help tie the timing and functions together for a more complete understanding.

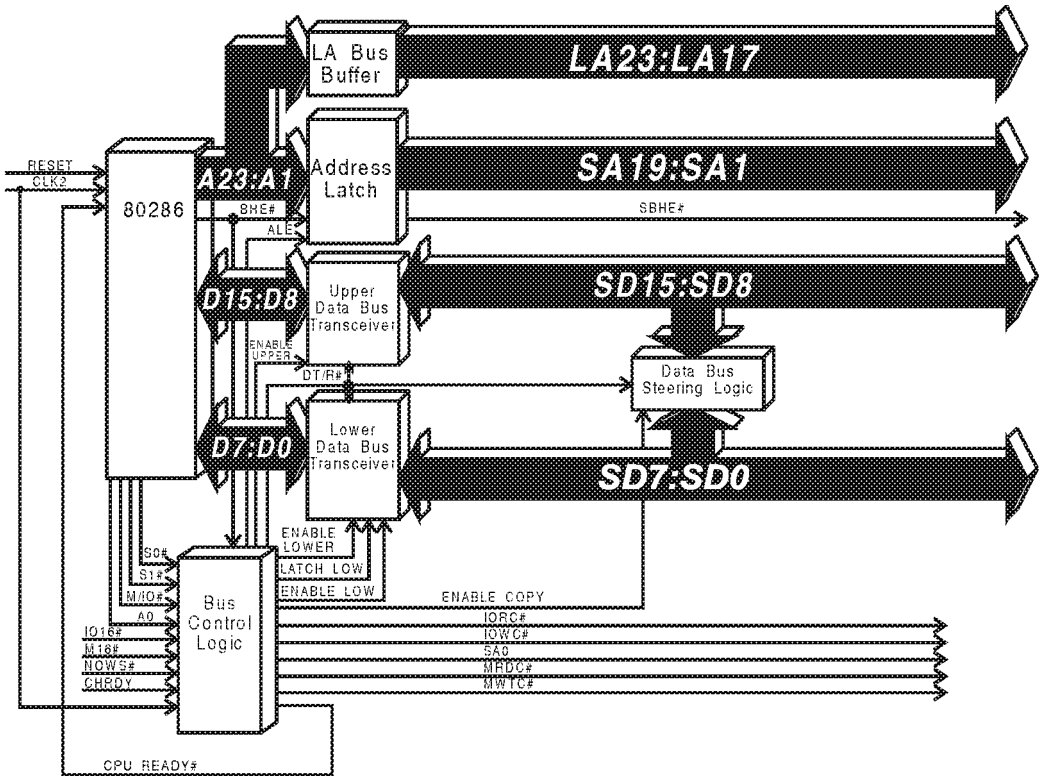


Figure 9-2. The 80286 System Engine

Bus Cycle A

Bus cycle A starts at reference point two, the beginning of the Ts time slot (address time). Scanning from left-to-right, however, the address and M/IO# change state prior to the start of bus cycle A. Halfway through the first Tc time in the previous bus cycle (at reference point 1), the microprocessor starts to output the address for the up-coming bus cycle (bus cycle A) before the actual start of bus cycle A. The address change is illustrated by the crossover point at reference point 1, the crossover indicating that the address starts to change to the new address at this point. This early address output is caused by the 80286's address pipelining capability. Refer to Figure 9-2.

Since the address for the current bus cycle has already been latched into the address latch, the microprocessor can safely output a new address onto its local address bus during the current bus cycle without interfering with the current bus cycle. Its address is already latched and being presented to the rest of the system on the System Address (SA) bus for examination. The latchable address (LA) bus is only used by 16-bit ISA memory boards and provides early decoding for these expansion devices. The early address change on the LA bus does not interfere with the current cycle because 16-bit boards are designed to decode the LA lines and latch the chip select for the current cycle. Since the chip select for the current bus cycle is latched, the next LA address can be decoded and latched once the current cycle is completed.

In addition to outputting the address for the up-coming bus cycle early, the microprocessor also sets the M/IO# line (reference point one) to the appropriate state for the up-coming bus cycle. This will be a one for a memory operation or a zero for an I/O operation.

At reference point two, bus cycle A actually begins. The 80286 outputs the remainder of the bus cycle definition on S0# and S1#. In the case of a read bus cycle, S1# will be asserted (low) and S0# will stay high. In addition, 16-bit expansion boards are free to examine the LA address to determine if it is their address.

At reference point three (the midpoint of Ts), the bus control logic asserts address latch enable (ALE) and keeps it asserted until the end of Ts. The address latch latches the address on the falling edge of ALE (when it goes low again). Once latched, the address is continuously presented on the system address (SA) bus where it can be examined by address decoders throughout the system. At this point, the microprocessor can safely output the address for the next bus

cycle without having any effect on the latched address. If the microprocessor does have another bus cycle to perform, it will start to output the address halfway through Tc (at reference point eight). In this example, the microprocessor is, in fact, performing back-to-back bus cycles, so you can see the address changing at reference point eight.

At reference point four, the microprocessor removes the asserted level from the S1# line. The bus control logic should have already decoded the microprocessor's bus cycle definition lines and determined the type of bus cycle currently in progress.

At reference point five, the bus control logic sets the DT/R# line (Data Transmit or Receive) line low because this is a receive (read) operation. This preconditions the Data Bus Transceivers to pass the data being read from the currently addressed device from right-to-left through the Data Bus Transceivers to the microprocessor. Slightly after setting the DT/R# line low (at reference point six), the bus control logic asserts ENABLE UPPER and/or ENABLE LOWER (transceiver enables), enabling one or both of the Data Bus Transceivers to actually pass the data. The bus control logic asserts one of its command outputs (IORC#, IOWC#, MRDC# or MWTC#). At this point (reference point seven), the data coming from the currently addressed device may begin to show up on the microprocessor's local data bus.

The microprocessor expects the currently addressed device to complete the data transfer during Tc. At the end of Tc, in bus cycle A, the microprocessor samples its READY# input asserted (low) at reference point nine. This tells the microprocessor that the currently addressed device is ready to end the bus cycle. Since this is a read operation, the microprocessor will read the data the device is supplying (at reference point ten) and end the bus cycle. This completes bus cycle A.

Bus Cycle B

Even though its address and M/IO# setting are output during bus cycle A (at reference point eight, due to pipelining), bus cycle B actually begins at reference point eleven with the output of S0# and S1#. Since this is another read bus cycle, S1# will be asserted (low) and S0# deasserted (high).

Bus cycle B proceeds the same as bus cycle A right up until the end of Tc time (reference point twelve). When the 80286 microprocessor samples READY# at this point, the currently addressed device hasn't yet asserted it to indicate its

readiness to end the bus cycle. Sampling READY# deasserted tells the microprocessor to "stretch" the bus cycle by inserting another Tc time (wait state).

The microprocessor leaves all signals exactly the same, waiting until the end of this extra Tc to sample READY# again. At the trailing edge (the end) of the extra Tc (reference point thirteen), the microprocessor samples READY# again. This time it is sampled asserted (low), telling the microprocessor that it can safely read the data from its local data bus and end the bus cycle.

Since another Ts time slot follows this bus cycle, another bus cycle is beginning.

The Write Bus Cycle

During a write bus cycle, just as with the read bus cycle, I/O and memory write bus cycles are identical with the exception of the M/IO# setting. This section describes the exact timing of the actions performed during a write bus cycle. Figure 9-3 is a timing diagram illustrating a typical 80286 write bus cycle.

A general idea of the information presented can be derived from the state names across the top. In this example, the sequence of states is as follows:

Tc	Data Time	end of previous bus cycle
Ts	Address Time	start of bus cycle
Tc	Data Time	end of bus cycle
Ts	Address Time	start of next bus cycle
Tc	Data Time	1st data time of bus cycle
Tc	Data Time	wait state
Ts	Address Time	start of next bus cycle

This timing diagram illustrates one 0-wait-state bus cycle bracketed by the end of the previous bus cycle and the start of the next bus cycle.

The address and M/IO# setting for the bus cycle is pipelined out early at reference point one. At the start of the bus cycle (reference point two), the microprocessor outputs the remainder of the bus cycle definition on S0# and S1#. S0# is asserted (low) to indicate a write operation and S1# is deasserted.

Because this is a transmit (write) operation, the bus control logic sets DT/R# (Data Transmit or Receive) high at reference point three. Since the quiescent (idle) state for the DT/R# line is high, no change is actually seen on this line, but it is nonetheless important that it is high at this point. This pre-conditions the Data Bus Transceivers to pass the data being output by the microprocessor

(on the local data bus) to the system data, or SD, bus. Immediately after setting DT/R# high, the bus control logic asserts ENABLE UPPER and/or ENABLE LOWER (transceiver enables), enabling one or both of the Data Bus Transceivers to pass the data onto the system data bus (reference point four). At the same time, the microprocessor begins to output the data to be written to the device (reference point five).

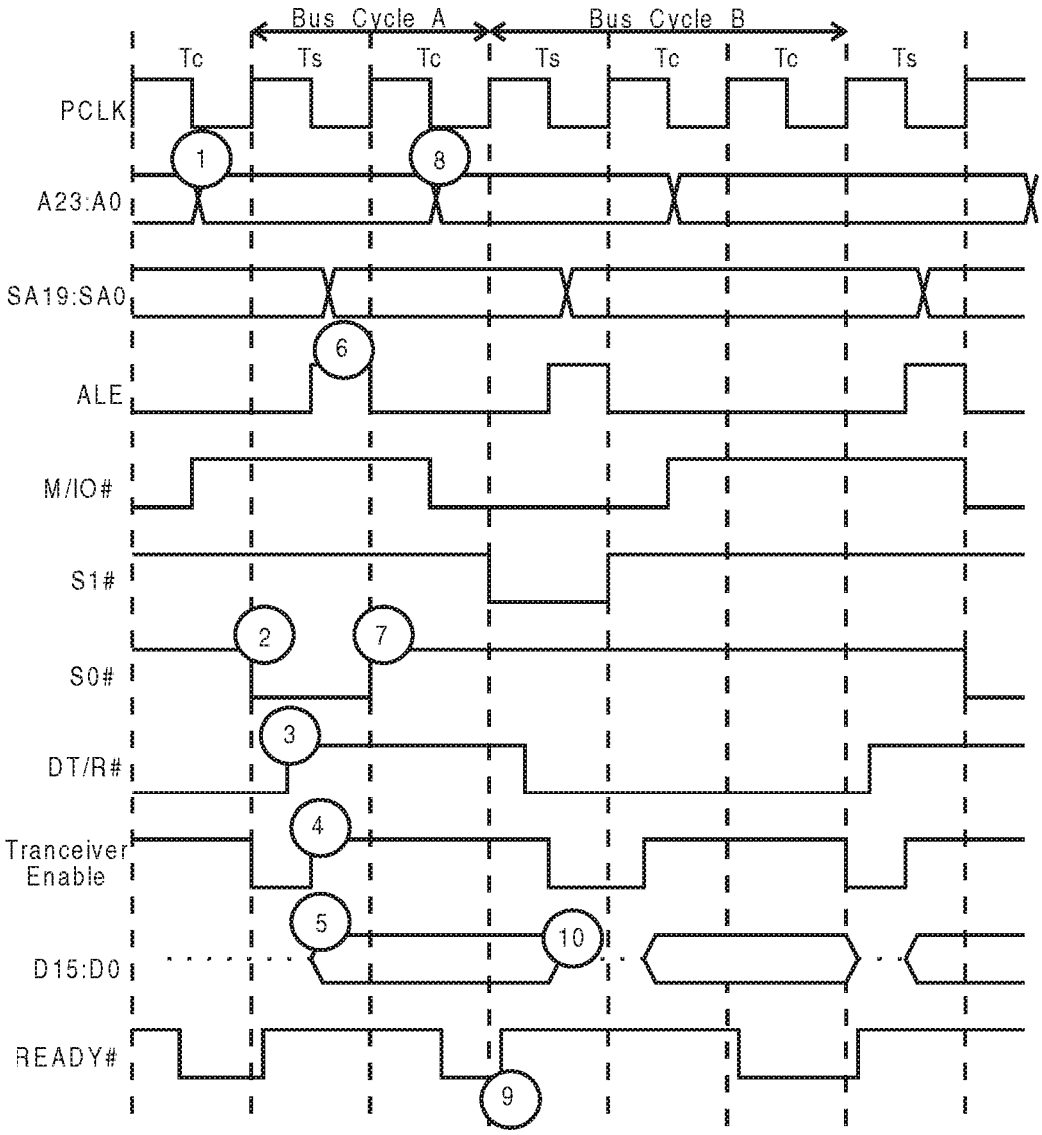


Figure 9-3. The Write Bus Cycle

Address latch enable (ALE) is pulsed by the bus control logic at reference point six, causing the address to be latched into the address latch (at the trailing edge of ALE). The latched address is now presented on the system address (SA) bus for examination by all address decoders in the system.

At reference point seven, the microprocessor deasserts the S0# line. The bus control logic and logic associated with the currently addressed device should have decoded the bus cycle definition lines by now to determine the type of bus cycle (write).

If the microprocessor is going to run another bus cycle immediately after this one, it will pipeline the next address out at reference point eight.

At the trailing edge of Tc (reference point nine), the microprocessor will sample its READY# input to see if the currently addressed device has accepted the data yet. In this example, READY# is sensed asserted (low), so the microprocessor ends the bus cycle.

It should be noted, however, that the bus control logic continues to keep ENABLE LOWER and/or ENABLE UPPER asserted, and the microprocessor continues to drive the data onto the local data bus for half a PCLK tick longer. The microprocessor placed the data on the data bus early (reference point five) and keeps it out there after the actual end of the bus cycle (reference point ten) for the reason described in the following paragraphs.

To guarantee that information is received correctly by the device being written to, every digital device capable of receiving data requires that a manufacturer-specified setup time and hold time be satisfied. In other words, the data must be present on the device's inputs for at least the specified setup time before the latch point and must remain present for at least the specified hold time after the latch point or the device may not receive the data correctly.

Since the microprocessor has no idea what the setup and hold times are for the individual device types it may be writing to, Intel's design takes worst-case setup and hold times into account by placing the data out very early in the bus cycle and keeping it out there as long as possible after the actual end of the bus cycle.

Continuing to drive the data onto the data bus for half of a PCLK tick into the next bus cycle will not disturb the next bus cycle. It takes time for an address decoder to detect an address and chip-select the next device. The fact that the data from the previous write bus cycle is still present on the data bus can't affect a device which hasn't even been chip-selected yet.

The Halt or Shutdown Bus Cycle

When the microprocessor executes a halt or shutdown bus cycle, the bus cycle definition lines and will reflect the following states:

- S1# and S0# are asserted (low).
- M/IO# is set high.

Notice there is no way to distinguish between a halt or shutdown during the bus cycle when looking only at the bus cycle definition lines. However, the state of address bit 1 (A1) indicates whether a halt or shutdown bus cycle is in progress.

- A1=1 for a halt
- A1=0 for a shutdown

During a halt or shutdown, the 80286 may service PEREQ (processor extension request) or HOLD requests. For further information on PEREQ and HOLD, refer to the chapters entitled "The Numeric Coprocessor" and "Direct Memory Access (DMA)" respectively.

Either NMI (non-maskable interrupt request) or RESET will force the 80286 out of either a halt or a shutdown. If interrupts are enabled, an INTR (maskable interrupt request) will also force the microprocessor out of a halt (but not out of shutdown). For information on interrupts, refer to the chapter entitled "The Interrupt Subsystem."

Halt

When the microprocessor executes a HLT instruction, it will initiate a halt or shutdown bus cycle and cease fetching and executing instructions. The microprocessor indicates it is running a halt bus cycle by placing the values on the bus cycle definition lines and address bit 1 as shown above.

It should be noted that S0# and S1# are only low during the bus cycle itself. The bus cycle ends when the microprocessor samples READY# asserted at the end of data time. After the bus cycle has ended, the halt condition can be recognized externally by the following signs:

- S1# and S0# are high.
- M/IO# is high.
- Address bit 1 is high.

These conditions are static — that is, they are constant, and therefore easily recognizable.

Shutdown

The indications of a shutdown are the same as those for a halt (shown above), with the exception that Address bit A1 is low.

When the 80286 microprocessor detects an exception while attempting to execute the handler for a prior exception, the two exceptions are generally handled serially. Certain combinations of exceptions cannot be handled serially, however. These exceptions are caused by protection violations in protected mode. For example, in protected mode, if an application attempts to access data that is outside its data segment, the microprocessor will start running the general protection fault exception handler. If, say, a stack overflow were to occur while the 80286 microprocessor ran the general protection fault exception handler, the microprocessor would not be able to handle the two faults serially. The microprocessor invokes the double-fault exception instead.

If the 80286 microprocessor detects another exception while attempting to service the double-fault, it goes into a shutdown condition. This is sometimes referred to as a triple fault. The microprocessor indicates the shutdown condition to external logic by running the halt or shutdown bus cycle. In an ISA machine, the shutdown detect logic outside the microprocessor detects the shutdown and toggles the microprocessor's RESET line, causing the system to reboot.

Chapter 10

The Previous Chapter

A detailed analysis of the 80286 microprocessor, the kernel logic necessary to interface it to external devices, and its bus cycle types were covered in prior chapters.

This Chapter

This chapter provides a detailed description of the 80386DX and SX microprocessors. It covers the functional units, processor self-test, enhancements to protected mode, virtual paging, virtual 8086 mode, and the processor's hardware interface to external devices.

The Next Chapter

The next chapter, "The 80386 System Kernel," defines the support logic necessary to interface the 80386DX and SX microprocessors to external devices.

Introduction

This chapter describes the 80386DX and SX microprocessors. The 80386DX and SX microprocessors are identical internally and only differ in their interface to external devices. This chapter is therefore divided into three parts:

- a discussion of the internal structure and operation of the 80386 microprocessor.
- a discussion of the 80386DX interface to external devices.
- a discussion of the 80386SX interface to external devices.

The 80386 Functional Units

General

Figure 10-1 illustrates the units that make up the 80386 microprocessor.

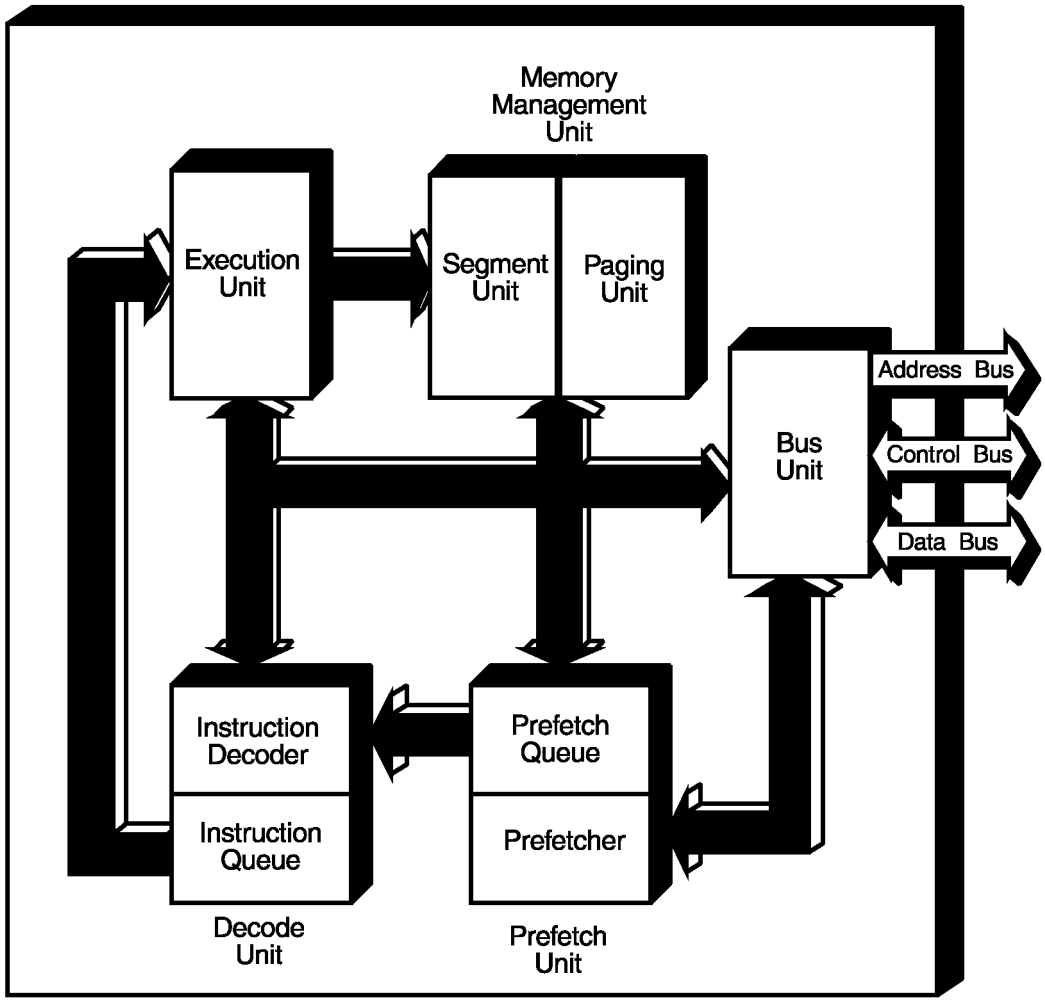


Figure 10-1. 80386 Microprocessor Block Diagram

The 80386 microprocessor consists of five functional units:

- Bus Unit. Handles communication with devices external to the microprocessor chip.
- Code Prefetch Unit. Fetches instructions from memory before the microprocessor actually requests them.
- Instruction Decode Unit. Decodes the instruction prior to passing it to the execution unit for execution.
- Execution Unit. Handles the actual execution of the instruction.
- Memory Management Unit (MMU). When the microprocessor must address a memory location, the MMU forms the physical memory address that is driven out onto the address bus by the bus unit during a bus cycle.

Code Prefetch Unit

The code prefetch unit capitalizes on the predictability of program execution. Statistically, program instructions are fetched from memory sequentially most of the time. Only jump instructions alter program flow, causing it to jump to another area of memory. Once the flow alteration has occurred, however, the program returns to sequential instruction processing until another jump instruction is encountered.

When the bus unit isn't performing bus cycles for the execution unit, the code prefetch unit uses the bus unit to fetch the next sequential instruction from memory and stores it in the 16-byte prefetch queue. The code prefetch unit always attempts to maximize the bus unit's throughput by fetching an entire doubleword (four bytes) at a time. The prefetched instructions are placed in the prefetch queue to await processing by the instruction decode unit.

Code prefetch requests are given a lower priority by the bus unit than requests from the execution unit. This ensures that the execution unit will not be stalled while waiting for a memory or I/O data transfer to complete. For all practical purposes, instruction prefetching reduces the time the execution unit spends waiting for the next instruction to zero.

Instruction Decode Unit

The instruction decode unit takes an instruction from the 16-byte prefetch queue, converts it into microcode and stores it in a three-deep decoded instruction queue for use by the execution unit. Any immediate data and offset associ-

ated with the instruction is also taken from the prefetch queue and stored in the decoded instruction queue.

Execution Unit

General

The execution unit executes each instruction received from the decoded instruction queue. It communicates with the other microprocessor functional units on an as needed basis in order to execute each instruction. The execution unit is comprised of three sub-units:

- The control unit's microcode and special parallel hardware speeds multiplies, divides and effective address calculation.
- The data unit contains the arithmetic logic unit, or ALU, eight general-purpose registers and a 64-bit barrel shifter. The data unit performs data operations requested by the control unit.
- The protection test unit checks for segmentation violations.

The Registers

The 80386 execution unit incorporates the following registers:

- The general registers
- The status and control registers
- The debug registers
- The test registers

The following sections describe each of these register groups.

General Registers

The general registers implemented in the 80386 microprocessor are a superset of those found in the earlier implementations of the X86 processor family. Figure 10-2 illustrates the general registers.

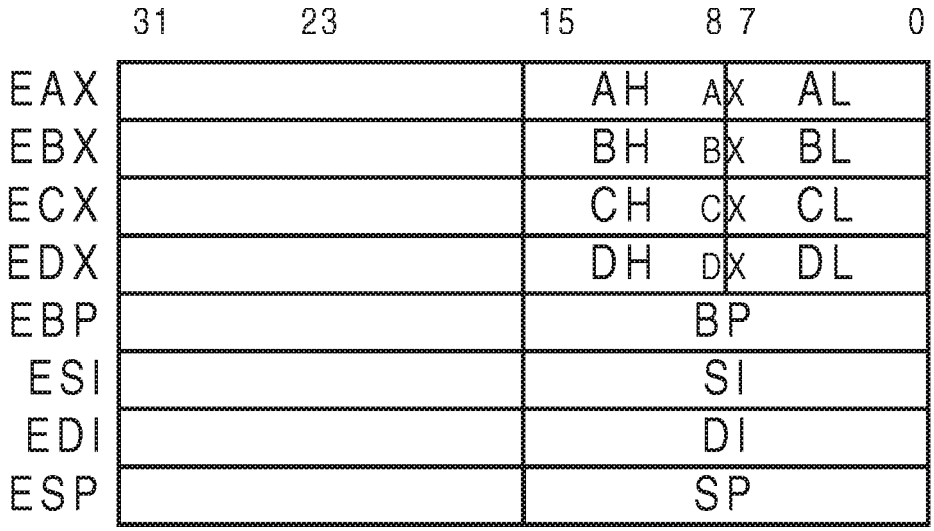


Figure 10-2. The 80386 General Registers

Extended versions of the AX, BX, CX, DX, BP, SI, DI and SP registers are now available to the programmer. The names of each of the extended registers starts with the letter "E". Referring to the EAX, EBX, ECX or EDX registers within a MOV instruction allows the programmer to specify a doubleword, or 32-bit, object to be moved. Extending the width of BP, SI, DI and SP registers to 32 bits permits the programmer to specify any memory address within the 4GB addressing range of the microprocessor without changing the contents of the segment registers.

Status, MSW and Instruction Registers

Figure 10-3 illustrates the 80386's status and instruction registers.

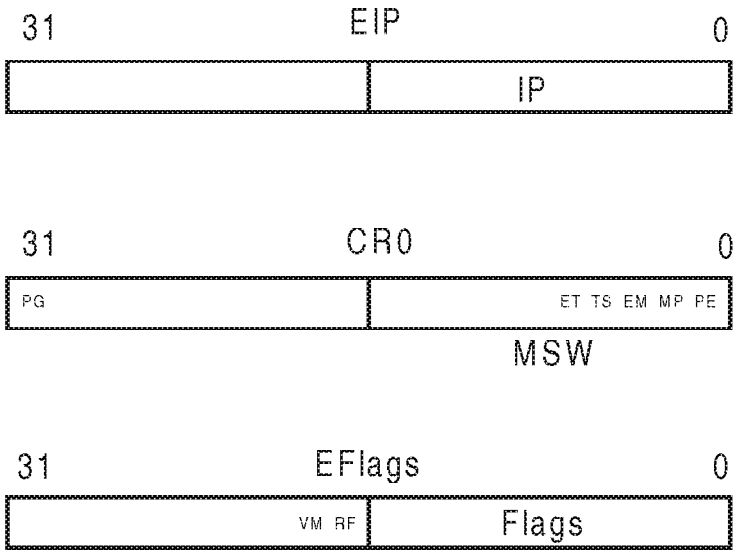


Figure 10-3. The 80386 Status and Instruction Registers

An extended version of the 80286 Flags register, the Eflags register, contains two new control bits:

- The RF, or resume flag, bit allows the programmer to disable debug exceptions so that the instruction can be restarted after a debug exception without immediately causing another debug exception.
- The VM, or virtual mode, bit allows the programmer to enable or disable virtual 8086 mode. Virtual 8086 mode is discussed later in this chapter.

Control register 0, or CR0, is a superset of the 80286 machine status word (MSW) register. CR0 contains two new bits not found in MSW. The extension type, or ET, bit (bit 4) indicates the type of numeric coprocessor present in the system (80287 or 80387). The ET bit is set to one when an 80387 coprocessor is detected. The paging bit, PG (bit 31), enables virtual paging when the bit is set to one. Virtual paging is described later in this chapter in the section entitled "Page Translation."

After reset, the 80386 MSW register has a value of 0000h in it. The 80286 MSW register has FFF0h in it after reset. Since the upper bits in the register aren't used by either processor, the difference has no effect.

Debug Registers

Figure 10-4 illustrates the 80386 debug registers. These registers support the implementation of both code and data breakpoints. In the case of a code breakpoint, the microprocessor monitors every instruction fetch for a match on the specified memory address. When a match is detected, a debug exception occurs and alerts the programmer. In the case of a data breakpoint, the microprocessor monitors for a data read from or a data write to the specified memory address defined by the programmer. When a match is detected, a debug exception occurs and alerts the programmer.

Linear Breakpoint Address 0	DR0
Linear Breakpoint Address 1	DR1
Linear Breakpoint Address 2	DR2
Linear Breakpoint Address 3	DR3
Reserved	DR4
Reserved	DR5
Breakpoint Status	DR6
Breakpoint Control	DR7

Figure 10-4. The 80386 Debug Registers

Debug register 7, or DR7, is the debug control register. It is used to define up to four breakpoints as being enabled or disabled. If enabled, a breakpoint may take one of three forms:

- break on instruction execution only.
- break on data writes only.
- break on data reads or writes, but not on instruction reads.

In addition, bits in DR7 may be used to define a breakpoint as local to a particular task or global in nature.

The debug status register, DR6, permits the programmer to determine which breakpoint condition has been met. DR4 and DR5 aren't used by the 80386. DR0 through DR3 are used to specify up to four breakpoint addresses.

For additional information about the debug registers, refer to Chapter 12 of Intel's *80386 Programmer's Reference Manual*, order number 230985.

Test Registers

The 80386 test registers are used to test the paging unit's translation lookaside buffer, or TLB. Figure 10-5 illustrates the two registers.

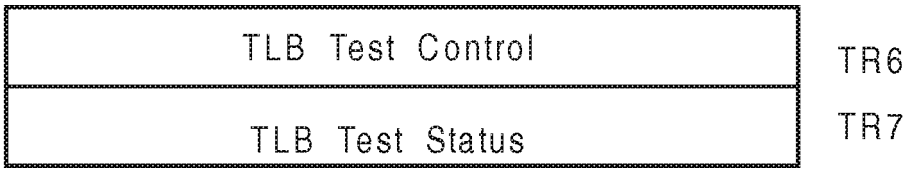


Figure 10-5. The 80386 Test Registers

TR6 is the command register for TLB accesses, while TR7 is the data register. Addresses and commands are written to the TLB using the command register, while data is read from or written to the TLB using the data register.

For further information about the test registers, refer to Chapter 12 of Intel's *80386 Hardware Reference Manual*, order number 231732.

Segmentation Unit

The microprocessor must access memory under the following circumstances:

- To fetch the next instruction for the code prefetch unit.
- When the execution unit must read data from memory to satisfy a MOV instruction.
- When the execution unit must write data to memory to satisfy a MOV instruction.
- To fetch a segment descriptor from memory when a segment register is loaded with a new value in protected mode.
- To fetch a page directory or page table entry from memory.

When the microprocessor must access memory, the segmentation unit forms the memory address by adding the offset within the target segment to the segment start address contained within the respective segment register. The resulting 32-bit memory address, known as the linear address, is then either submitted to the paging unit (if paging is enabled) or output onto the address bus during the resulting memory bus cycle.

Refer to figure 10-6. In real mode, all of the segment registers, including FS and GS, are available to the programmer. The FS and GS segment registers give the programmer the ability to create two new data segments in addition to the DS and ES data segments available in the earlier processors. The programmer also has access to the debug, control and test registers while in real mode.

A description of segment register usage in both real and protected modes can be found in the chapter entitled "The 80286 Microprocessor." The differences in protected mode operation on the 80286 and 80386 microprocessors are discussed later in this chapter.

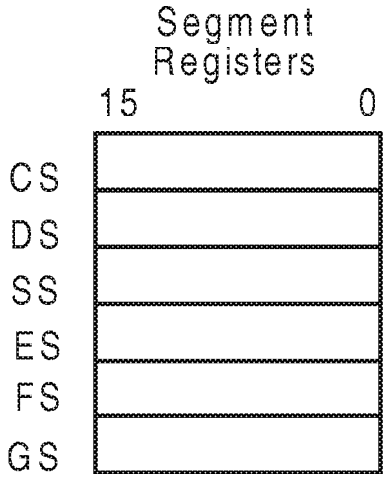


Figure 10-6. The 80386 Segment Registers

Paging Unit

A description of the paging unit can be found later in this chapter under the heading "Page Translation."

Bus Unit

The bus unit provides the interface between the microprocessor and external memory and I/O devices. When the microprocessor must read information from or write information to an external device, the bus unit uses the address, data, and control buses to run one or more bus cycles. The bus unit runs bus cycles when a request is received from:

- Execution unit to run an I/O or memory read or write.
- the code prefetch unit to read the next instruction from memory.
- the execution unit to fetch a segment descriptor from memory when a segment register is loaded with a new value in protected mode.
- the paging unit to fetch a page directory or page table entry from memory.

The bus unit also provides the interface between the microprocessor and the numeric coprocessor. A description of processor/coprocessor intercommunication can be found in the chapter entitled "Numeric Coprocessor."

Protected Mode

Figure 10-7 illustrates the segment descriptor format in the 80286 protected mode environment. Bytes 0 and 1 describe the length, or limit, of the segment. Since this is only a 16-bit field, the 80286 can only describe segments up to 64KB in length. Bytes 2, 3 and 4 supply the start address of the segment, allowing the segment to start anywhere in the 16MB memory space of the 80286 microprocessor. Byte 5 is the attribute byte and describes various characteristics of the segment. Bytes 6 and 7 are not used by the 80286.

Figure 10-8 illustrates the segment descriptor format in the 80386 protected mode environment. Bytes 0 and 1 and the lower four bits of byte 6 specify the length of the segment. With a 20-bit field, a segment length of from one byte to 1Mbyte (1,048,576) can be specified. In addition, bit 7 of byte 6 is used to further define the segment length. Known as the granularity, or G, bit, it indicates how the limit (segment length) field is to be interpreted. If G = 0, the length is in bytes. This allows a segment to be anywhere from one byte to 1MB long. If G = 1, the length is specified as the number of 4KB pages of memory. This allows a segment to be anywhere from 4KB to 4GB in length.

Bytes 2, 3, 4 and 7 are used to specify the start address of the segment, allowing the segment to be located anywhere in the 4GB address range of the microprocessor.

A description of the protected mode concept can be found in the chapter entitled "The 80286 Microprocessor."

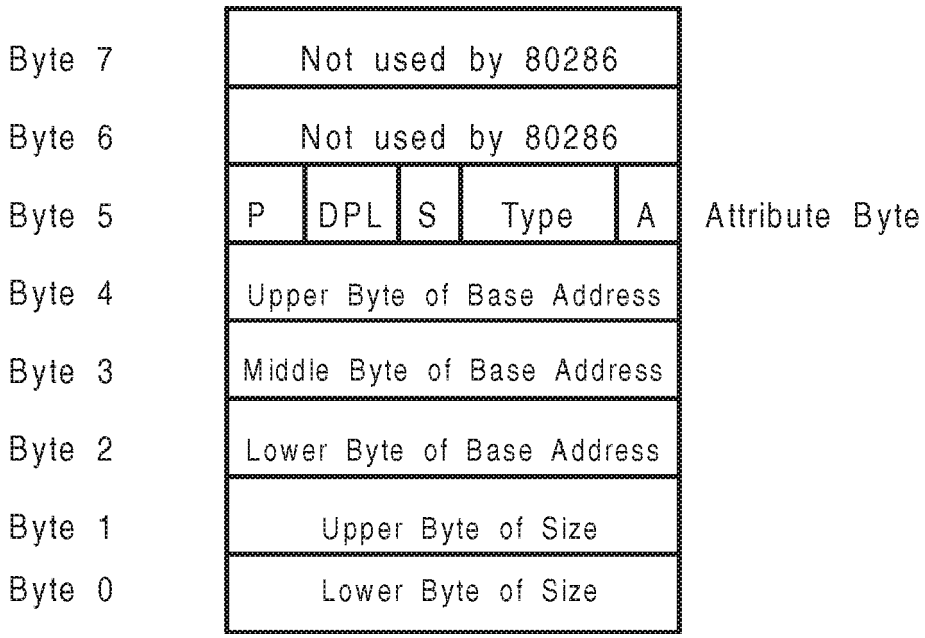


Figure 10-7. The 80286 Segment Descriptor Format

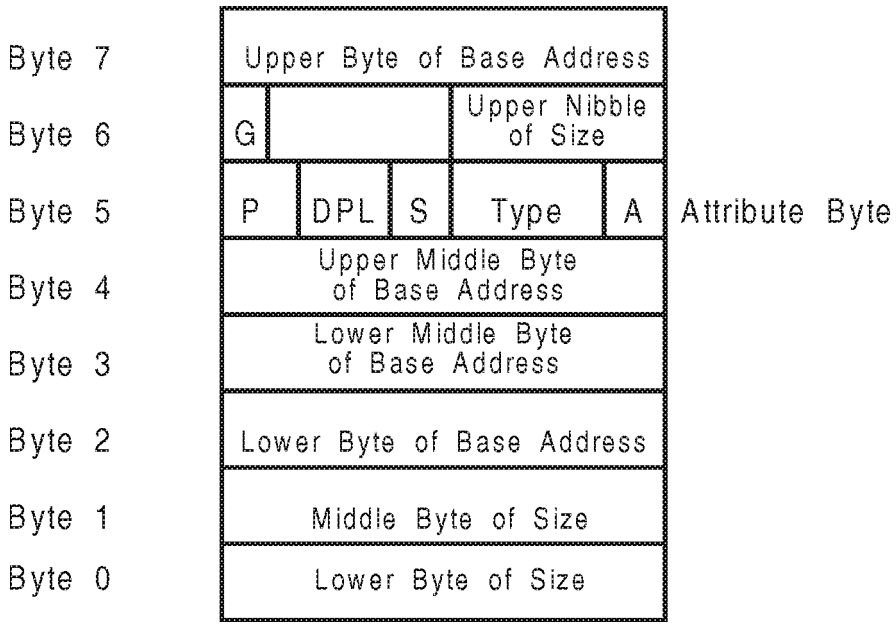


Figure 10-8. The 80386 General Segment Descriptor Format

Page Translation

Virtual Paging

The 80386 microprocessor incorporates logic to facilitate the implementation of virtual paging (also referred to as demand paging) in an operating system.

Paging is enabled by setting the PG bit in CR0 (bit 31). When enabled, the address produced by the segmentation unit, called the linear address, is used as an input to the paging unit. The linear address output by the segmentation unit is used by the paging unit to identify a page table, a page described within that table, and an offset within that page.

Before describing the function performed by the paging unit, the concept of the page should be defined.

The 80386 paging system provides an indexing system to keep track of the location and status of up to 1,048,576 four kilobyte pages (a total of 4GB) of pro-

gram and/or data that "belongs" to a task (applications program). At a given moment in time, a page of program and/or data can be:

- on disk or
- on disk and in main memory

When paging is enabled, the paging unit uses the linear address produced by the segmentation unit to locate:

- the page directory entry that points to the respective page table.
- the page table entry that describes the page containing the location the programmer is attempting to access.
- the offset within the main memory page.

As illustrated later in this text, each 32-bit page table entry contains a page present bit that defines whether or not the desired page is currently resident in main memory. If it is, the entry also defines the start address of the desired page in main memory. The lower part of the linear address from the segmentation unit is then used to identify the offset of the desired location in the target main memory page and a bus cycle is generated to access the desired main memory location. If the entry indicates the desired page isn't currently present in main memory, a page fault (exception 14) occurs, causing the 80386 to jump to the page fault handler program. In addition, the microprocessor also pushes a 16-bit error code onto the stack.

In the page fault handler, the operating system can then read CR2 (see figure 10-9) to ascertain the actual 32-bit linear address that caused the page fault. The linear address can be used to locate the page table entry that defined the page as "not present." Since only one bit of the 32-bit entry was used to define the page as not present, the operating system can use the other 31 bits to define a disk address where the actual 4KB page of code and/or data can be found on the disk. The page fault handler can then locate an unused page in main memory and read the desired page into the respective main memory page. The page table entry will then be updated to mark the page present in main memory and point to the page's start address in main memory. The actual bus request to access the desired code or data is then initiated.

When a task begins execution, CR3 (see figure 10-9) is automatically loaded with the start address of the page directory in main memory. Only the upper 20 bits, 31:12, of the page directory start address are actually found in CR3 (the least-significant 12 bits of the start address are assumed to be 0).

Figure 10-9. Control Registers CR2 and CR3

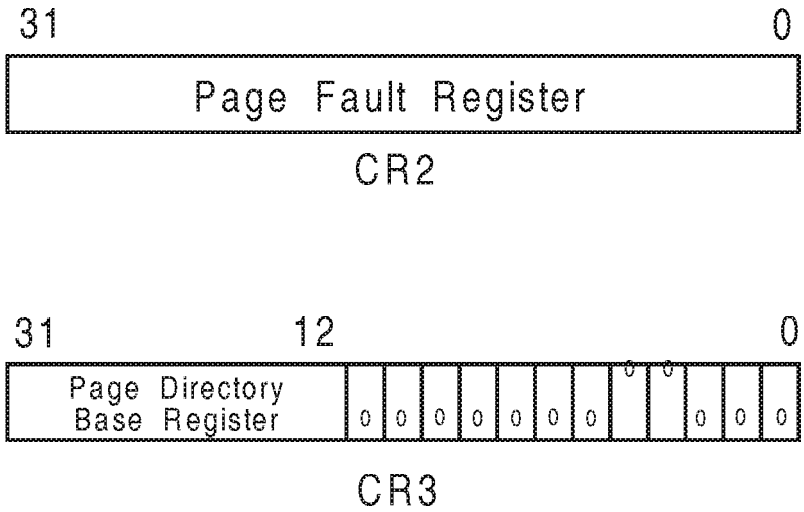


Figure 10-10 illustrates the page directory, page tables, and the translation of the segmentation unit's linear address output to the physical memory address. The page directory starts at the location specified in CR3 and is one page (4KB) in size. It can contain up to 1K (1,024) 32-bit page directory entries, each consisting of 32 bits. Bits 31:22 of the linear address identify which of the 1,024 32-bit page directory entries to use. A page directory entry has the format illustrated in table 10-1. It should be noted that page directory and page table entries have the same format.

Only the upper 20 bits, 31:12, of the page table start address are actually found in bits 31:12 of the page directory entry (the least-significant 12 bits of the start address are assumed to be 0). The other bits in the page table entry are defined in table 10-1.

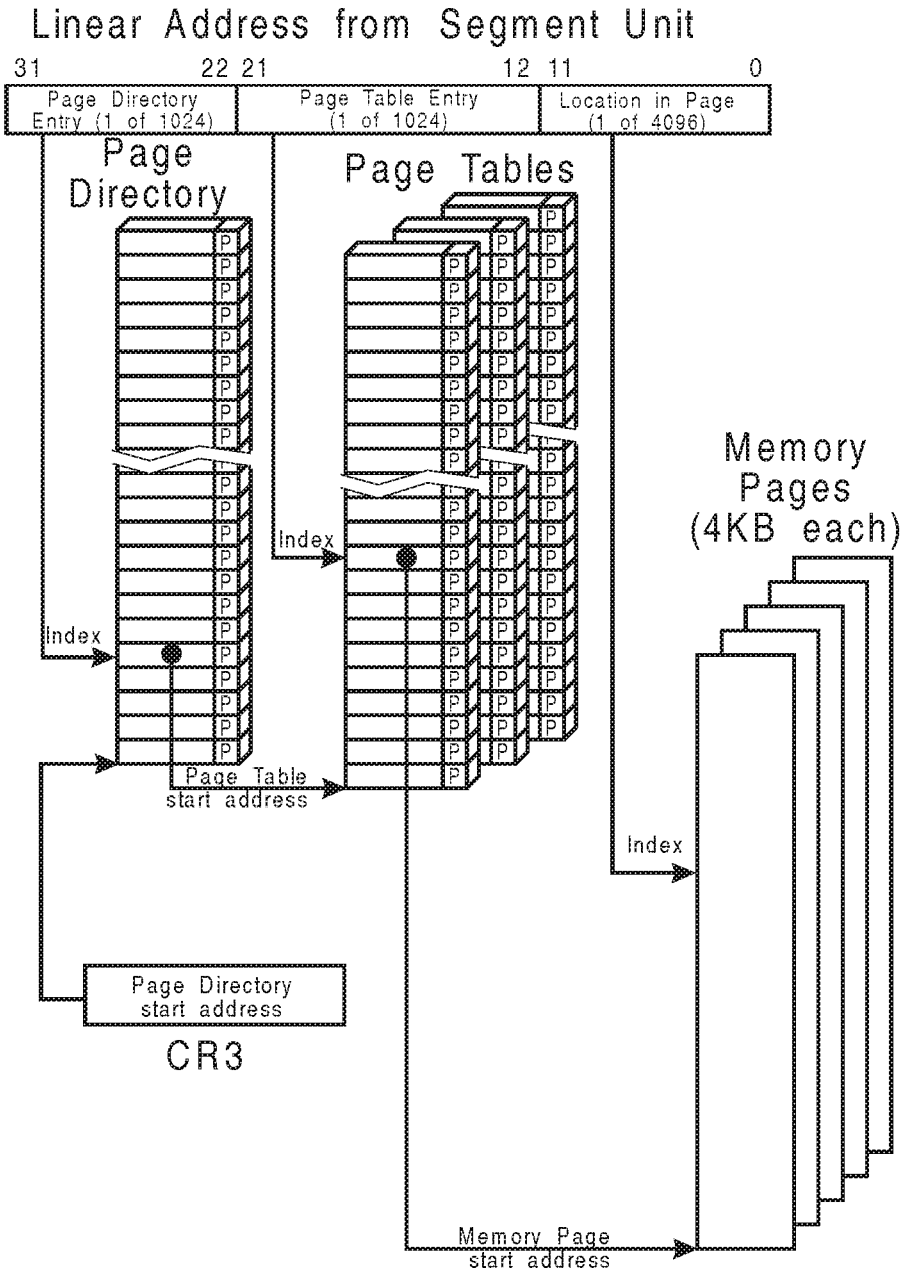


Figure 10-10. Translation of Linear Memory Address to Physical Memory Address

Table 10-1. Format of Page Table Entry Attribute Bits

Bit(s)	Designator	Definition
0	P	Page Present bit. Set to one if the target page is currently resident in main memory. Cleared to zero if the page containing the target page table must be read from disk.
1	R/W	Read/Write bit. When set to one, it is permitted to both read and write the target page. Cleared to zero, the page can only be read.
2	U/S	User/Supervisor bit. A one in this bit indicates that the page is accessible by both the operating system and applications programs. Cleared to zero, it indicates that the page may only be accessed by the operating system.
4:3	none	Intel reserved, do not use (defined on 80486).
5	A	Accessed bit. Set to one by hardware before a read or write access to the target page. The operating system can use the state of this bit to determine whether the page has been used (accessed). If not, the operating system may choose to read a different page from disk into this 4KB page of main memory.
6	D	Dirty bit. This bit is undefined in a page directory entry. In a page table entry, the dirty bit is set before a write to the page described by the page table entry. When set, it indicates that the operating system should insure that the page is written back to disk to maintain coherency between disk and main memory.
8:7	none	Intel reserved, do not use.
11:9	none	These three bits aren't used by the 80386 microprocessor. They may be used by the systems programmer in a programmer-specified fashion.

As stated above, the page directory entry describes the page which contains the page table. The upper 20 bits (31:12) point to the start address of the page table.

Bits 21:12 of the linear address are then used to index into the page table pointed to by the page directory entry. Starting at this address, the page table can contain up to 1,024 32-bit entries. The format of the page table entry is iden-

tical to that of the page directory entry described in table 10-1. If the page is present in memory (P = 1), bits 31:12 of the page table entry point to the actual start address of the page in main memory. Bits 11:0 of the linear address provide the offset within the main memory page. The start address of the page and the offset within the page are then combined to form the physical memory address that will be driven onto the address bus.

As stated earlier, if the page table describing the page or the page itself isn't currently in memory, a page fault will be generated and the page can then be read from disk into an available page of main memory.

Figure 10-11 shows another view of the page directory, page tables and memory pages. The page directory and page tables form a tree that provides the start addresses of the memory pages. Notice that if the present bit, the least significant bit of a page directory entry, is set to one, then that entry contains the start address of one of the page tables. If the bit is cleared to zero, the entry is not currently in use and the entry doesn't point to anything. Likewise, if the present bit of a page table entry is one, that entry contains the start address of a memory page. Remember that the bits of the linear address are used to select the page directory and page table entries, as shown in figure 10-10.

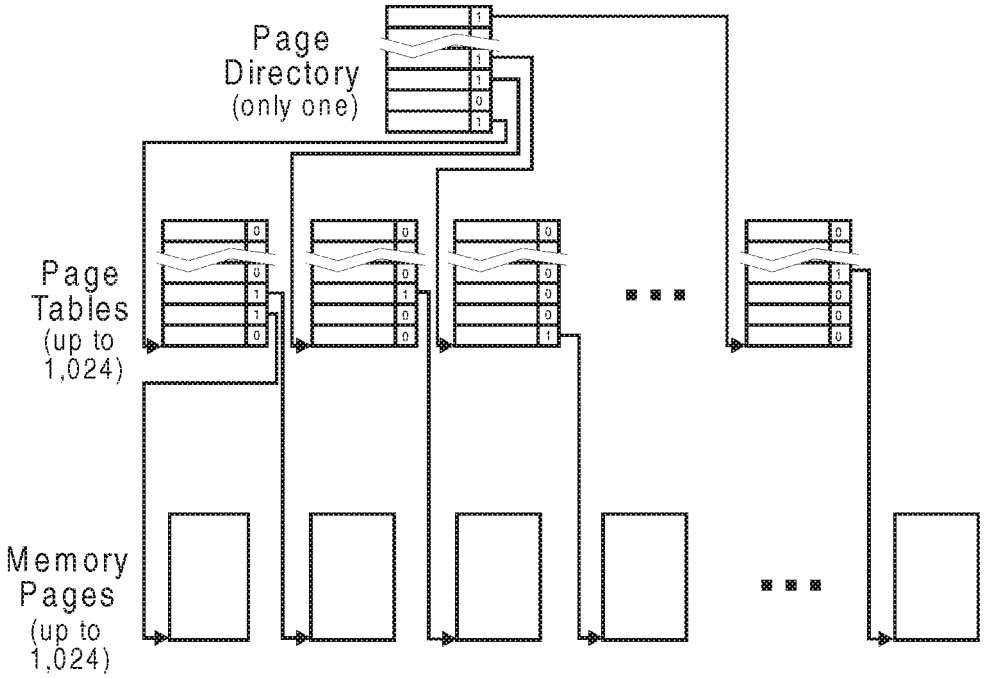


Figure 10-11. Another View of the Page Directory, Page Tables and Memory Pages

Translation Lookaside Buffer

Performance would degrade substantially if the microprocessor had to access two levels of memory-based tables each time memory had to be accessed. To solve this problem, the 80386 microprocessor maintains an on-chip cache of the most recently accessed page table entries. This cache is called the translation lookaside buffer (TLB).

Refer to figure 10-12. The TLB is a four-way set-associative 32-entry page table cache (set-associative caches are described in the chapter entitled "Cache Memory Concepts"). The TLB automatically keeps the most-recently-used page table entries on-board the processor. The 32-entry TLB coupled with a 4KB page size results in coverage of up to 128KB of memory addresses.

For many multi-tasking operating systems, the TLB will exhibit a hit rate of about 98%. This means that the processor will only have to access the two-level page structure on 2% of all memory references.

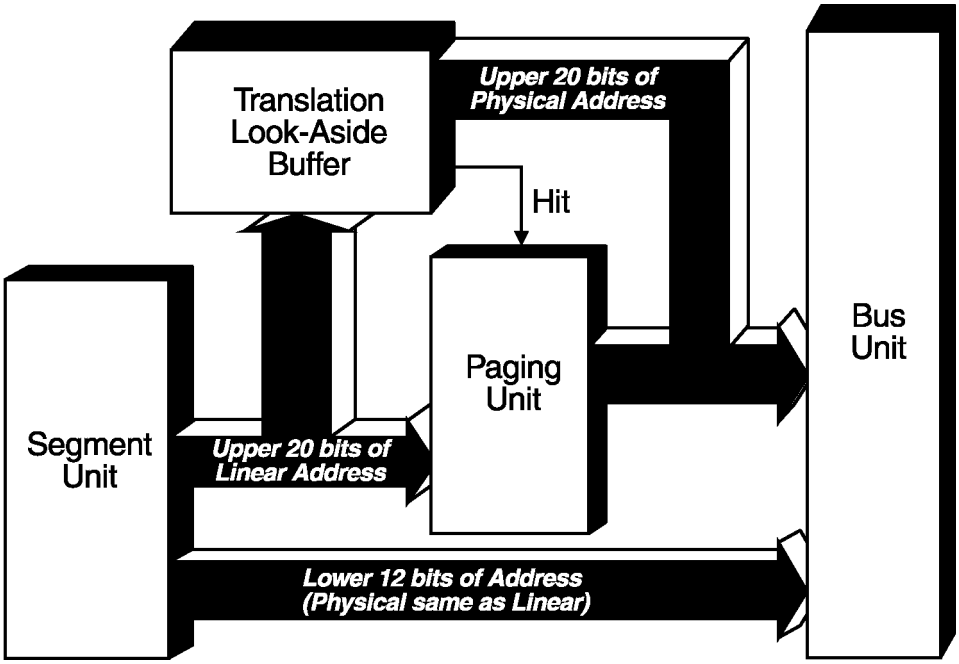


Figure 10-12. The Translation Look-Aside Buffer

Reading a new entry into the TLB (referred to as a TLB Refresh) is a two step process handled by the 80386 microprocessor hardware. The sequence of data cycles to perform a TLB refresh are:

- Read the correct page directory entry, as pointed to by the page base register (CR3) and the upper ten bits of the linear address. Optionally perform a locked read/write to set the accessed bit in the page directory entry. The page directory entry will actually get read twice if the 80386 microprocessor needs to set any of the bits in the entry. If the page directory entry changes between the two reads, the data returned from the second read will be used.
- Read the correct entry in the page table and place the entry in the TLB. Optionally perform a locked read/write to set the accessed and/or dirty bits in the page table entry. Again, note that the page table entry will actually get read twice if the 80386 microprocessor needs to set any of the bits in the entry. If the page table entry changes between the two reads, the data returned from the second read will be used.

Note that the TLB contains page table entries only. It does not contain page directory entries. Nevertheless, if there is a miss on the TLB, the microprocessor must access both the page directory entry and the page table entry in order to refresh the TLB. This is because the page directory entry contains the start address of the page table. Since both tables are being accessed on a TLB miss, page faults can be signaled from either the page directory read or the page table read.

The paging unit hardware receives a 32-bit linear address from the segmentation unit. The upper 20 linear address bits are compared with four entries in the TLB (the entries in the four-way-associated set) to determine if there is a match. If there is a match (a hit), then the 32-bit physical address is calculated and will be placed on the address bus.

Virtual-8086 Mode

The 80386 microprocessor can execute 8086 code in either of two modes:

- Real mode
- Virtual 8086 mode

Virtual 8086 Mode establishes an 8086 execution environment within the protected mode environment of the 80386. Where real mode governs everything that the 80386 does, virtual 8086 mode can be applied to selected 80386 tasks. When executing a virtual 8086 mode task, the processor behaves like an 8086, but upon a switch to a normal task, the processor operates as an 80386. Thus, virtual 8086 mode enables an operating system to support the execution of 8086, 80286 and 80386 tasks concurrently.

The VM flag bit in the Eflags register, which is loaded from the tss (task state segment), defines the running task's virtual processor as an 8086 or as an 80386. When the 80386 loads its registers from a TSS whose VM flag bit is set, the processor enters virtual 8086 mode. When, on a subsequent task switch, the processor loads its registers from a TSS where the VM bit is clear, it leaves virtual 8086 mode. Thus, on a task-by-task basis, the processor emulates an 80386 or an 8086 according to the value of the VM bit. The 80386 also leaves virtual 8086 mode when it raises an exception or is interrupted, making the full resources of the architecture available to interrupt and exception handlers. On return from a handler invoked while in virtual 8086 mode, the 80386 automatically re-enters virtual 8086 mode when the Eflags register is reloaded from the stack.

Because the address space of an 8086 is one megabyte, the logical addresses generated by a virtual 8086 mode task fall into the first megabyte of the 80386 linear address space. Multiple virtual 8086 mode tasks could interfere with each other, since they would all share the low megabyte of the linear address space. An operating system can use 80386 paging to relocate the linear address spaces of virtual 8086 mode tasks to different areas of the physical address space. Using paging in this way not only prevents interference between virtual 8086 mode tasks, but enables a virtual memory operating system to swap the pages of virtual 8086 mode tasks just as if they were 80386 tasks.

A virtual 8086 mode task may execute a program that was written for execution on a single-task personal computer. Such a program can contain instructions that are potentially disruptive if executed in a multi-tasking environment. For example, allowing a virtual 8086 mode task to execute the clear interrupt enable (CLI) instruction, thereby disabling recognition of interrupts, could bring the entire system to a halt. To prevent such disruptions, the 80386 raises an exception when a virtual 8086 mode task attempts to execute an I/O or interrupt-related instruction. These instructions are referred to as "sensitive" because their execution depends on (is sensitive to) the state of the I/O privilege level bits in the Eflag register.

I/O instructions are prevented from being executed when in virtual 8086 mode on the basis of an I/O permission map included as part of the TSS. When a real mode program attempts to access an I/O location, the processor checks the address against entries within the I/O permission map. Each bit in the permission map represents a single I/O location. If the corresponding bit for an I/O location is set, access is allowed. The first bit in the map is assigned to I/O address zero, the second bit to I/O location one, and so forth up to 64Kbits (8KB). In this way, all of the possible 64KB of I/O addresses can be mapped to allow or deny access.

Preventing the execution of such instructions protects the rest of the system from a virtual 8086 mode task, but does not satisfy the virtual 8086 mode task's need to execute the instructions. The solution is to simulate the "sensitive" instructions in an operating system procedure known as a virtual machine monitor. When an exception handler is invoked, it can inspect the VM flag bit in the Eflags image on the stack to determine if the source of the exception is a virtual 8086 mode task; if so, the exception handler can call the virtual machine monitor which can simulate the instruction and return to the virtual 8086 mode task. Note that a virtual machine monitor program simulates only a few 8086 instructions and that both the simulated instructions and those the 80386 exe-

cutes directly benefit from the much higher performance of the 80386 compared to the 8086.

Working together, the 80386 and a virtual machine monitor implement the full 8086 instruction set, and paging can provide each virtual 8086 mode task with its own protected address space. However, most 8086 programs need additional resources provided by an operating system and peripheral hardware. An example of the former type of resource is a file system; an example of the latter is a display controller manipulated directly by an applications program. These resources may not exist in the same form in the 80386-based system as they did in the system for which the 8086 program was designed. To simplify the job of providing these resources in a different environment, the 80386 can trap operating system and peripheral references made by virtual 8086 mode tasks.

For example, most 8086 operating systems use the INT (Interrupt) instruction to implement operating system calls. If a virtual 8086 mode task's current privilege level is less privileged than the task's IOPL (I/O privilege level), the 80386 raises an exception when the virtual 8086 mode task attempts to execute an INT instruction. The virtual machine monitor can then translate the 8086 operating system call into a call on the 80386 operating system.

Peripheral references made by a virtual 8086 mode task using IN and OUT instructions are always checked against the I/O permission map, regardless of the IOPL. The 80386 paging facility can be used to redirect references to memory-mapped peripherals to other addresses, if necessary. Such references can also be trapped by marking the corresponding pages as read-only (to trap writes), or not-present (to trap both reads and writes).

Automatic Self-Test

The 80386 microprocessor has the ability to perform a self-test of its three major logic arrays and to verify the contents of its internal microcode control ROM. The automatic self-test is initiated by asserting the microprocessor's BUSY# input during initialization (while the RESET signal is asserted). The test result is stored in the EAX register, and will be non-zero if an error was incurred. The self-test provides 100% coverage of single-bit faults, which statistically comprise a high percentage of total faults.

The RESET input to the 80386 must remain asserted for at least 80 CLK2 periods if the self-test is to be performed. On the high-to-low transition of RESET, the BUSY# input is sampled. If asserted, the microprocessor initiates the self-test which will take approximately 2^{20} + 60 CLK2 cycles to complete. When the

test completes, the microprocessor initiates program execution without examining the test results.

80386DX External Interface

This section provides a detailed description of the signal lines the 80386DX microprocessor uses to communicate with the rest of the system. Many of these signal lines serve the same purpose as on the 80286 microprocessor. These signals are listed in table 10-2.

Table 10-2. Signals Common to the 80286 and 80386DX Microprocessors

Signal Name	Description
CLK2	Double Frequency clock input
M/IO#	Memory or I/O output
LOCK#	Lock output
READY#	Ready input
HOLD	Hold Request input
HLDA	Hold Acknowledge output
PEREQ	Processor Extension Request input
BUSY#	Processor Extension Busy input
ERROR#	Processor Extension Error input
INTR	Interrupt Request input
NMI	Non-Maskable Interrupt Request input
RESET	Reset input

The remainder of the 80386DX signals have been grouped into the following categories:

- Address Bus
- Data Bus
- Control Bus

The Address Bus

The 80386DX address bus consists of two sets of signal lines:

- the address bus, consisting of 30 signal lines designated A[31:2].
- the byte enable bus, consisting of the 4 signal lines designated BE#[3:0].

As demonstrated later in this chapter, the 80386DX uses the address bus, A[31:2], to identify a group of four locations, known as a doubleword.

The 80386DX uses the byte enable lines to identify one or more of the four locations it actually wishes to perform a data transfer with.

Internally, the 80386DX microprocessor actually generates 32-bit addresses. A 32-bit address bus gives it the ability to address any one of 4GB (4 gigabytes; 4,096 megabytes) individual memory locations. When the microprocessor attempts to place this 32-bit address on the address bus, however, the two least-significant address bits, A0 and A1, get stripped off. This is because these two address lines don't physically exist on an 80386DX microprocessor.

When designing an 80386DX based system, one should always assume that, whenever the microprocessor outputs an address during a bus cycle, A0 and A1 are always 0. Let's examine the result. Table 10-3 illustrates a number of addresses being output by the 80386DX. Remember that address bits 0 and 1 are stripped off during address output and are always assumed to be 0.

Table 10-3. Example Addresses Output by an 80386DX

Address to be Output (in hex)	Address Placed on Address Bus (in hex)
00000000	00000000
00000001	00000000
00000002	00000000
00000003	00000000
00000004	00000004
00000005	00000004
00000006	00000004
00000007	00000004
00000008	00000008
00000009	00000008
0000000A	00000008
0000000B	00000008
0000000C	0000000C
0000000D	0000000C
0000000E	0000000C
0000000F	0000000C
00000010	00000010
00000011	00000010
00000012	00000010
00000013	00000010

The result of always forcing A0 and A1 to 0 is that the 80386DX microprocessor is only capable of outputting every fourth address. It is physically incapable of addressing any of the intervening addresses. When the 80386DX outputs an address, it is really identifying a group of four locations, called a doubleword, starting at the address presented on the address bus.

In addition to identifying the doubleword address on A[31:2], the microprocessor also asserts one or more of the byte enable lines to indicate which of the four locations it really wants to communicate with during the current bus cycle. Asserting a byte enable line also identifies which of the 80386DX's four data paths will be used to communicate with the identified location(s) in the doubleword.

The BE0# line is associated with the first location in the group of four (doubleword) and with the lowest data path, D[7:0]. BE1# is associated with the second location in the doubleword and with the second data path, D[15:8]. BE2# is associated with the third location in the doubleword and with the third data path, D[23:16]. BE3# is associated with the fourth location in the doubleword and with the fourth data path, D[31:24]. Table 10-4 illustrates this relationship.

Table 10-4. Relationship of 80386DX Byte Enables, Data Paths and Locations in the Currently-Addressed Doubleword

Byte Enable	Data Path Used	Location Addressed In Doubleword
BE0#	D[7:0]	first
BE1#	D[15:8]	second
BE2#	D[23:16]	third
BE3#	D[31:24]	fourth

Figure 10-13 illustrates this same relationship. A[31:2] identify the doubleword. The asserted byte enable lines identify the location(s) within the addressed doubleword and the data path(s) to be used when communicating with them.

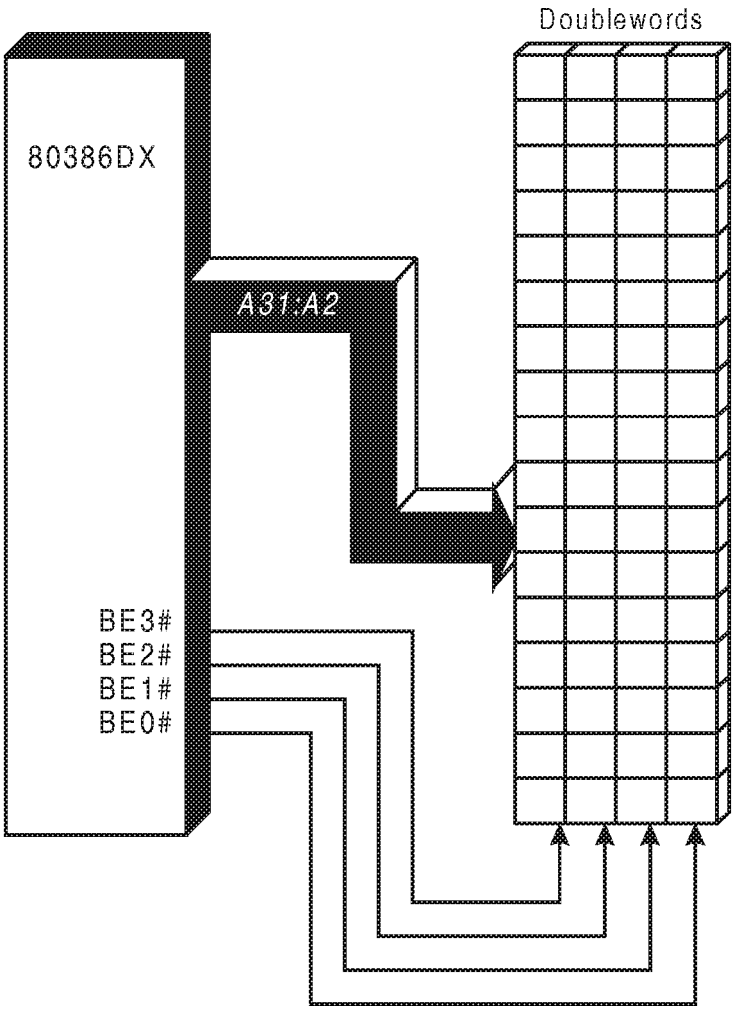


Figure 10-13. 80386DX Address/Byte Enable Relationship

Table 10-5 summarizes the relationship between the address and byte enables by showing the address specified by the programmer (in both binary and hex) along with the address and byte enables that the microprocessor outputs.

Table 10-5. Relationship of Addresses and Byte Enables Output by an 80386DX

Address to be Accessed (binary) A31	A30	- - -	A5	A4	A3	A2	A1	A0	Address to be Accessed (in hex)	Address Placed on Address Bus (in hex)	Byte Enable
X	X	- - -	X	X	0	0	0	0	XXXXXXX0	XXXXXXX0	BE0# = 0
X	X	- - -	X	X	0	0	0	1	XXXXXXX1	XXXXXXX0	BE1# = 0
X	X	- - -	X	X	0	0	1	0	XXXXXXX2	XXXXXXX0	BE2# = 0
X	X	- - -	X	X	0	0	1	1	XXXXXXX3	XXXXXXX0	BE3# = 0
X	X	- - -	X	X	0	1	0	0	XXXXXXX4	XXXXXXX4	BE0# = 0
X	X	- - -	X	X	0	1	0	1	XXXXXXX5	XXXXXXX4	BE1# = 0
X	X	- - -	X	X	0	1	1	0	XXXXXXX6	XXXXXXX4	BE2# = 0
X	X	- - -	X	X	0	1	1	1	XXXXXXX7	XXXXXXX4	BE3# = 0
X	X	- - -	X	X	1	0	0	0	XXXXXXX8	XXXXXXX8	BE0# = 0
X	X	- - -	X	X	1	0	0	1	XXXXXXX9	XXXXXXX8	BE1# = 0
X	X	- - -	X	X	1	0	1	0	XXXXXXXA	XXXXXXX8	BE2# = 0
X	X	- - -	X	X	1	0	1	1	XXXXXXXB	XXXXXXX8	BE3# = 0
X	X	- - -	X	X	1	1	0	0	XXXXXXXC	XXXXXXXC	BE0# = 0
X	X	- - -	X	X	1	1	0	1	XXXXXXXD	XXXXXXXC	BE1# = 0
X	X	- - -	X	X	1	1	1	0	XXXXXXXE	XXXXXXXC	BE2# = 0
X	X	- - -	X	X	1	1	1	1	XXXXXXXF	XXXXXXXC	BE3# = 0

Table 10-6 illustrates several example addresses and a description of how they are interpreted by external logic.

Table 10-6. 80386DX Addressing Examples

Address on A[31:2]	BE3#	BE2#	BE1#	BE0#	Location(s) addressed	Data Path
00001000h	1	1	1	0	00001000h	0
FA026504h	0	0	1	1	FA026506h, FA026507h	2, 3
00000108h	0	0	0	0	00000108h to 0000010Bh	0,1,2,3
01AD0F0Ch	1	0	0	1	01AD0F0Dh, 1AD0F0Eh	1, 2

In the first example, the microprocessor is identifying the doubleword starting at location 00001000h. By asserting the BE0# line, it indicates its intention to communicate with the first location in the doubleword using the first data path (D[7:0]).

In the second example, the microprocessor is identifying the doubleword starting at location FA026504h. By asserting the BE2# and BE3# lines, it indicates its

intention to communicate with the third and fourth locations in the doubleword using the third (D[23:16]) and fourth (D[31:24]) data paths.
In the third example, the microprocessor is identifying the doubleword starting at location 00000108h. By asserting all four of the byte enable lines, it indicates its intention to communicate with all four locations in the doubleword using all four data paths.

In the fourth example, the microprocessor is identifying the doubleword starting at location 01AD0F0Ch. By asserting the BE1# and BE2# lines, it indicates its intention to communicate with the second and third locations in the doubleword using the second (D[15:8]) and third (D[23:16]) data paths.

The 80386DX microprocessor is capable of performing 8-, 16-, 24- and 32-bit transfers. When asserting multiple byte enables, it may only assert adjacent byte enables, however. Table 10-7 shows the valid and invalid combinations of byte enables that can be generated during a bus cycle.

Table 10-7. 80386DX Byte Enable Combinations

BE3#	BE2#	BE1#	BE0#	Transfer Description
1	1	1	1	invalid
1	1	1	0	8-bit transfer with one location
1	1	0	1	8-bit transfer with one location
1	1	0	0	16-bit transfer with two locations
1	0	1	1	8-bit transfer with one location
1	0	1	0	invalid
1	0	0	1	16-bit transfer with two locations
1	0	0	0	24-bit transfer with three locations
0	1	1	1	8-bit transfer with one location
0	1	1	0	invalid
0	1	0	1	invalid
0	1	0	0	invalid
0	0	1	1	16-bit transfer with two locations
0	0	1	0	invalid
0	0	0	1	24-bit transfer with three locations
0	0	0	0	32-bit transfer with four locations

Table 10-8 shows the address and byte enables generated in the bus cycle caused by execution of the indicated instruction. All examples assume that the data segment register has 0000h in it.

Table 10-8. Example Instructions and Resultant Addresses

Instruction	Address	BE3#	BE2#	BE1#	BE0#	Transfer Type
MOV AL,[0100]	00000100h	1	1	1	0	8-bit transfer
MOV AL,[0101]	00000100h	1	1	0	1	8-bit transfer
MOV BL,[0102]	00000100h	1	0	1	1	8-bit transfer
MOV AH,[0103]	00000100h	0	1	1	1	8-bit transfer
MOV AX,[0100]	00000100h	1	1	0	0	16-bit transfer
MOV AX,[0101]	00000100h	1	0	0	1	16-bit transfer
MOV AX,[0102]	00000100h	0	0	1	1	16-bit transfer
MOV EAX,[0100]	00000100h	0	0	0	0	32-bit transfer

With regard to the last table entry, the EAX register is the 80386's 32-bit version of the AX register. It stands for extended AX register.

A 24-bit transfer occurs if the programmer attempts to perform a transfer that crosses a doubleword address boundary. As an example, assume the DS register contains 0000h and the programmer attempts to execute the following instruction:

```
MOV [0101],EAX
```

This tells the 80386DX to write the four bytes in the 32-bit EAX register to memory starting at location 00000101h. To do this, the microprocessor must write the lower three bytes in the EAX register to locations 00000101h to 00000103h in the doubleword starting at 00000100h. It must then write the most-significant byte in the EAX register to the first location (00000104h) in the doubleword starting at location 00000104h. Because it can't address two doublewords at the same time, this causes the microprocessor to generate two back-to-back bus cycles.

During the first bus cycle, the microprocessor places the doubleword address, 00000104h, on the address bus. This identifies the doubleword starting at memory location 00000104h. Since the programmer instructed the microprocessor to write the four bytes in the EAX register to memory locations 00000101h to 00000104h, the microprocessor now asserts only BE0#. This tells external logic that the microprocessor wants to communicate with the first location in the doubleword, but not the upper three. The most-significant byte in the EAX register is driven out onto the lowest data path, D[7:0], by the microprocessor and is written into memory location 00000104h. This completes the second bus cycle and also the execution of the MOV instruction.

After the first bus cycle, the microprocessor automatically places address 00000101h on the address bus, but A0 and A1 are stripped off on the way out. The resultant address placed on the address bus is 00000100h. This identifies the doubleword starting at memory location 00000100h. Since the programmer instructed the microprocessor to write the four bytes in the EAX register to memory starting at location 00000101h, the microprocessor asserts BE1#, BE2# and BE3# (but not BE0#) during the first bus cycle. This tells external logic that the microprocessor wants to communicate with the upper three locations in the doubleword (00000101h to 00000103h), but not the first one (00000100h). The least-significant three bytes in the EAX register are driven out onto the upper three data paths by the microprocessor and are written into memory locations 00000101h to 00000103h.

The 80386DX microprocessor is inefficient when attempting transfers that cross doubleword address boundaries. To obtain optimum data throughput (transfer speed), Intel therefore urges programmers to perform data transfers that don't cross doubleword address boundaries.

The Data Bus

The 80386DX microprocessor has a 32-bit data bus. It is more correct, however, the think of it as having four 8-bit data paths:

- Data Path 0 is comprised of D[7:0]
- Data Path 1 is comprised of D[15:8]
- Data Path 2 is comprised of D[23:16]
- Data Path 3 is comprised of D[31:24]

Refer to figure 10-14. Every doubleword consists of four locations and starts at an even address that is divisible by four. The microprocessor communicates with the first location over data path 0 (D[7:0]), the second over data path 1 (D[15:8]), the third over data path 2 (D[23:16]) and the fourth over data path 3 (D[31:24]).

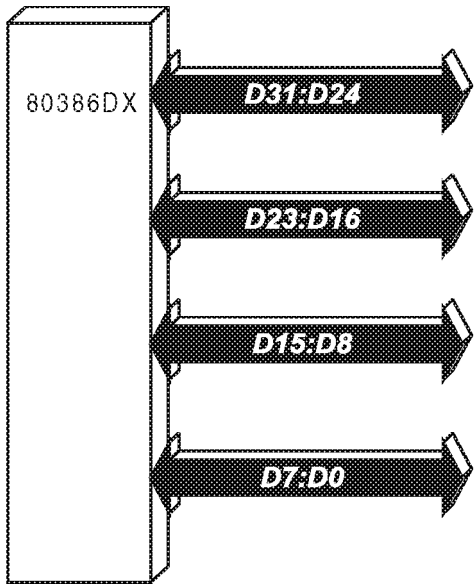


Figure 10-14. The 80386DX Data Bus

The Control Bus

The control bus consists of all the 80386DX signal lines other than the address bus, byte enables and the data bus. These signal lines can be separated into the following subcategories:

- Bus cycle definition outputs
- Clock input
- Reset input
- Ready input
- Bus mastering lines
- Interrupt inputs
- Processor extension lines
- Address status output
- Pipelining control input
- Bus size 16 input

The following paragraphs only describe the 80386DX signal lines that are different from the 80286's control bus lines.

Bus Cycle Definition Outputs

To read data from or write data to an external location, the 80386DX bus unit must run a bus cycle. During the bus cycle, the microprocessor indicates the type of bus cycle by placing the proper pattern on the bus cycle definition signal lines. Table 10-9 shows each type of bus cycle and the respective pattern which will be output onto the bus cycle definition lines.

Table 10-9. 80386 Bus Cycle Definition

M/IO#	D/C#	W/R#	Bus Cycle Type
0	0	0	Interrupt Acknowledge
0	1	0	I/O Read
0	1	1	I/O Write
1	0	0	Memory Code (instruction) Read
1	1	0	Memory Data Read
1	1	1	Memory Data Write
1	0	1	Halt or Shutdown

Processor Extension Lines

With one exception, the 80386 microprocessor uses the same signal lines as the 80286 microprocessor to interface to the numeric coprocessor. The PEACK# (processor extension acknowledge) signal line has been eliminated and the READY# line is now connected to the 80387 coprocessor as well as to the 80386 microprocessor. A detailed description of the processor extension signal lines can be found in the chapter entitled "Numeric Coprocessor."

Address Status Output

At the start of a bus cycle, during "Address Time," the 80386 microprocessor places the address on the address bus and the bus cycle definition on the control bus. It also asserts the address status (ADS#) output to indicate that a valid address and bus cycle definition are present on the buses.

Pipelining Control Input

Discussion of 80286 address pipelining can be found in the chapter entitled "Detailed View of the Bus Cycle." The 80386 handles address pipelining differently than the 80286. The 80386 microprocessor has an input called NA# (next address). The following actions occur during a bus cycle:

1. The microprocessor outputs the address of the device it wishes to communicate with.
2. The device's address decode logic decodes the address on the address bus and chip-selects its device.
3. Logic associated with the addressed device latches the decoded chip-select and lower address bits on the address bus.
4. Since the addressed device's logic has now latched the address information, it no longer needs the address the microprocessor is presenting on the bus. It then asserts the 80386's NA# (next address) signal, alerting the microprocessor that it can place the address for the next bus cycle onto the address bus early (during the current bus cycle). This is a variation on the address pipelining scheme used by the 80286 microprocessor. Permitting the microprocessor to place the address for the upcoming bus cycle onto the address bus early allows the next device to be addressed to see its address early. This allows the designer to use relatively slow access devices. The early decoding of the address and selection of the addressed device allows the slow device to be ready to complete the bus cycle earlier than if it hadn't received the address for decoding until the microprocessor had caused the address to be latched into the external address latch.

When addressed devices allow the microprocessor to place the next address on the bus early for a number of back-to-back bus cycles, the 80386 microprocessor is using the buses to their fullest potential. These back-to-back, pipelined bus cycles provide the highest bandwidth (transfer rate) on the buses.

Dynamic Bus Sizing (BS16#)

The 80386DX microprocessor's bus size 16 (BS16#) input is used to inform the 80386DX that the currently addressed device is a 16-bit device rather than a 32-bit device. When BS16# is sampled asserted, the 80386DX performs data transfers with the addressed device using only the lower two data paths, D[7:0] and D[15:8]. This feature is known as dynamic bus sizing. In cases where the 80386DX requires the transfer of more than 16 bits, the microprocessor will automatically generate additional bus cycles to fulfill the data transfer request if BS16# is sampled asserted.

As an example, consider the actions caused by the execution of the following instruction (assume DS = 0000h and the processor is in real mode):

```
MOV EAX,[1F08]
```

Assume that the addressed device is a 16-bit device. The programmer is requesting the transfer of a doubleword (the EAX register is four bytes in size) starting at memory location 00001F08h. At the beginning of the first bus cycle, the microprocessor places address 00001F08h on the address bus (A[31:2]) and asserts all four byte enable outputs. When the currently addressed device's address decoder detects the address, it asserts BS16#, informing the microprocessor that the currently addressed device is only capable of communicating over the lower two data paths.

The contents of location 00001F08h is placed on D[7:0] and the contents of location 00001F09h is placed on D[15:8] by the addressed 16-bit memory device. At the end of the bus cycle, when READY# is sampled asserted, the microprocessor reads the two data bytes and places them in the lower two bytes of the EAX register.

The microprocessor then automatically begins a second bus cycle with the same address (00001F08h) on the address bus and BE2# and BE3# asserted. The addressed 16-bit device's address decoder once again asserts BS16#, and the contents of 00001F0Ah and 00001F0Bh are transferred back to the microprocessor over the two lower data paths. At the end of the bus cycle, the microprocessor reads the two data bytes from the two lower data paths and places them into the upper two bytes of the EAX register, completing the overall transfer. More examples of dynamic bus sizing can be found in the chapter entitled "The 80386 System Kernel."

80386SX External Interface

Interface Signal Differences

Functionally, the 80386SX is identical to the 80386DX microprocessor. The 80386SX microprocessor's interface to external devices differs from that of the 80386DX in the structure of its address and data buses. Table 10-10 details the differences.

Table 10-10. Interface Signals That Differentiate the 80386SX from the 80386DX Microprocessor

Signal(s)	Description
Address Bus	Unlike the DX address bus consisting of A[31:2] and the four byte enable lines, the SX address bus consists of A[23:0] and BHE#. In other words, its address bus and its addressing scheme is identical to that of the 80286 microprocessor.
Data Bus	Like the 80286 microprocessor, the SX only has two data paths, versus four for the DX microprocessor.
BS16#	Since the SX does not have a 32-bit data bus, the BS16# input isn't necessary.

A0 or BLE#

The A0 output of the 80386SX is called BLE#. This is because the 80386SX uses the same rules as the 80286 regarding the usage of its two data paths. This being the case, the 80386SX uses its lower data path to communicate with even-addressed locations. Since all even addresses have A0 set to a zero, a low on A0 indicates that the SX will use its lower data path during a bus cycle. A0 can therefore be thought of as BLE#, or bus low enable.

Addressing Scheme, Data Bus Width Ramifications

Consider the following instruction:

```
MOV  EAX,[0100]
```

Assume that the processor is in real mode and the DS register currently is set to 0000h. When executed, this instruction will instruct the microprocessor to transfer four bytes of information from memory starting at location 0100h into the microprocessor's EAX register. Where the 80386DX microprocessor would only have to run one bus cycle to address all four locations and transfer all four bytes simultaneously, the 80386SX must run two bus cycles to accomplish the four byte move.

First, the address 000100h is placed on the address bus, A[23:0], and the BHE# signal is asserted. The 16-bit memory will interpret this as a read from location 000100h and the next sequential odd location, 000101h. The addressed memory will place the contents of the even address, 000100h, on the lower data path

and that of the odd address, 000101h, on the upper data path. At the end of the bus cycle, the microprocessor inputs the two bytes and places them in the lower two locations in the target register, AL and AH. Half of the requested data has now been moved from memory into the target register.

Now, the 80386SX microprocessor initiates a second bus cycle, placing the address 000102h on the address bus, A[23:0], and asserting the BHE# signal again. The 16-bit memory will interpret this as a read from location 000102h and the next sequential odd location, 000103h. The addressed memory will place the contents of the even address, 000102h, on the lower data path and that of the odd address, 000103h, on the upper data path. At the end of the bus cycle, the microprocessor inputs the two bytes and places them in the upper two locations in the target register. All of the requested data has now been moved from memory into the target register.

It should also be noted that the 80386SX microprocessor is limited to a maximum of 16MB of installed memory. This is the maximum amount of memory that can be addressed with a 24-bit address bus.

Throughput and Compatibility Considerations

Due to a couple of factors, the 80386SX microprocessor has substantially slower throughput then an 80386DX.

- Two data paths versus four. This means that, at the same clock rate, the 80386DX has twice the throughput as that of the SX. This is true even if the DX is running 80286 code that doesn't attempt any 32-bit reads or writes, because the instruction prefetcher always fetches four bytes at a time.
- The 80386DX microprocessor is available in substantially higher clock rates than the SX. Where the SX is available in clock rates up to 25MHz, the DX is available in clock rates up to 40MHz.

The 80386SX and DX microprocessors are both capable of running any code written for the 8086, 80286 or 80386 microprocessors. The 80286 microprocessor, on the other hand, cannot run code written for the 80386 microprocessor.

Chapter 11

The Previous Chapter

The previous chapter, "The 80386DX and SX Microprocessors," provided a detailed description of these processors.

This Chapter

This chapter provides a detailed description of the kernel support logic necessary to interface the 80386DX and SX processors to the rest of the system. Sample instructions are used to illustrate the actions of the processor and support logic when performing various types of transfers. Interfacing issues are covered for the 80386DX with and without dynamic bus sizing implemented.

The Next Chapter

The next chapter, "Detailed View of the 80386 Bus Cycle," provides a detailed description of bus cycle timing when the 80386 is performing a read or write transfer. The other types of bus cycles are discussed as well.

Introduction

This chapter covers the essential logic that must surround an 80386 microprocessor in order to allow it to communicate with 8-, 16- and 32-bit devices. This is known as the kernel logic. Three variations are covered:

- The 80386SX system kernel
- The 80386DX system kernel with dynamic bus sizing implemented
- The 80386DX system kernel without dynamic bus sizing implemented

The 80386SX System Kernel

Figure 11-1 illustrates the kernel logic surrounding an 80386SX microprocessor. It should be immediately apparent that it is virtually identical with the 80286 kernel logic detailed in the chapter entitled "The 80286 System Kernel: the Engine." The only differences are the renamed bus cycle definition signals, W/R# and D/C#, and the ADS# signal is used to trigger the bus control logic at the start of a bus cycle. In an 80286-based kernel, the falling edge of S0# or S1# is used to trigger the bus control logic at the start of a bus cycle.

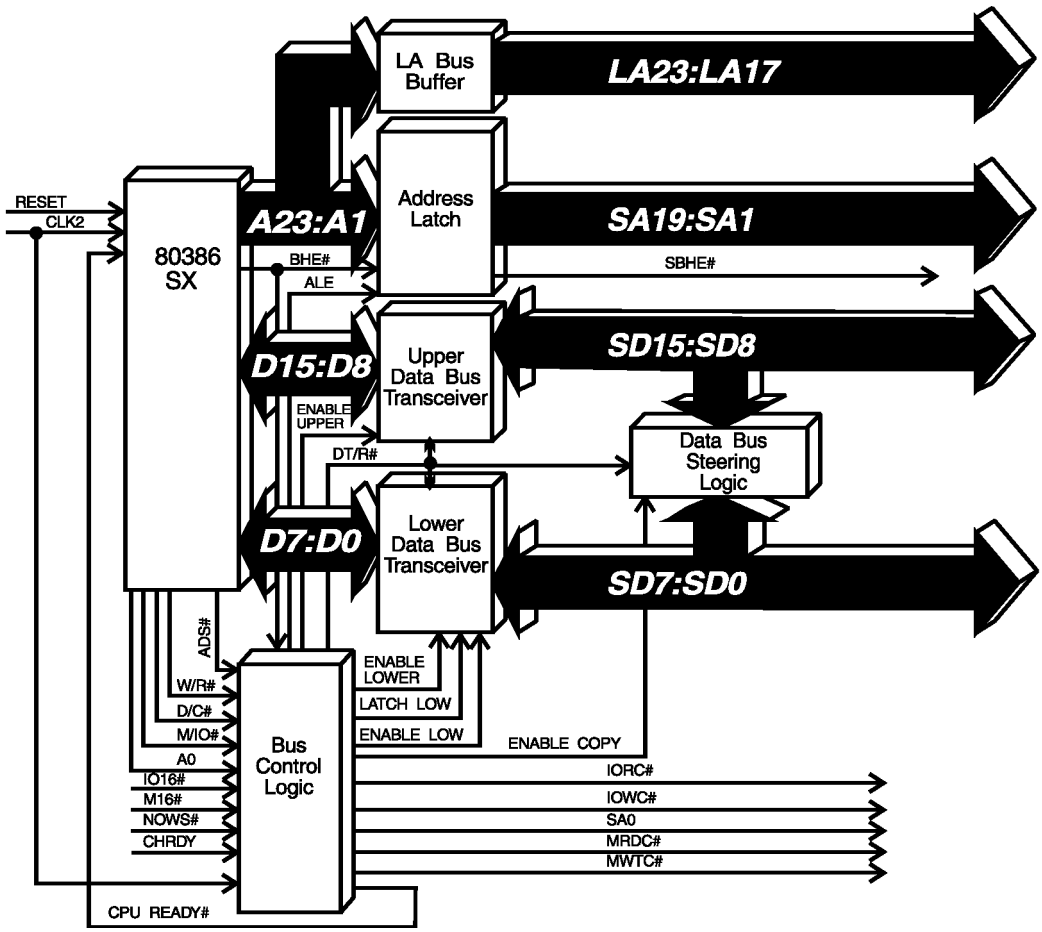


Figure 11-1. The 80386SX-Based Kernel

The 80386DX System Kernel

80386DX System Kernel with Dynamic Bus Sizing

Introduction

Figure 11-2 illustrates the kernel logic surrounding an 80386DX microprocessor. In this implementation, the designer has taken advantage of the 80386DX's dynamic bus sizing capability by hooking up the microprocessor's BS16# input.

If the device addressed during a bus cycle is a 16-bit memory or I/O device, it will assert either the M16# or IO16# signal. An asserted level on either of these signals causes an asserted level to be placed on the microprocessor's BS16# input. When the microprocessor samples BS16# asserted at the end of T1, dynamic bus sizing is activated and multiple bus cycles are performed as necessary to transfer the object. The transfer takes place only over the lower two data paths when communicating with a 16-bit device. A detailed description of dynamic bus sizing can be found in the chapter entitled "The 80386DX and SX Microprocessors."

When the 80386DX microprocessor initiates a bus cycle, the target doubleword address is output onto A[31:2] and the appropriate byte enable outputs are asserted. 32-bit devices recognize this address format, but 8- and 16-bit devices require additional address information. In addition to A[31:2], 8-bit devices require A0 and A1, while 16-bit devices require A0, A1 and BHE#. Since the 80386DX microprocessor is incapable of producing these signals, the bus control logic must convert the microprocessor's byte enable outputs into these three signals.

Every doubleword consists of four locations. The first of these locations is always an even address, the second an odd address, the third even and the fourth odd. If the microprocessor intends to communicate with the first location in the doubleword, an even address, it asserts its BE0# output. When communicating with the second location in the doubleword, an odd address, it asserts its BE1# output, and so on for the third and fourth locations.

When a bus cycle is initiated, the bus control logic examines the microprocessor's byte enable outputs and determines the locations within the target doubleword that the microprocessor is communicating with. The following paragraphs describe the interaction between the microprocessor and 8-, 16- and 32-bit devices. Figure 11-2 illustrates the kernel of an 80386DX-based system with dynamic bus sizing implemented. In all of the example instructions, it is

assumed that the microprocessor is in Real Mode and the Data Segment (DS) register contains 0000h.

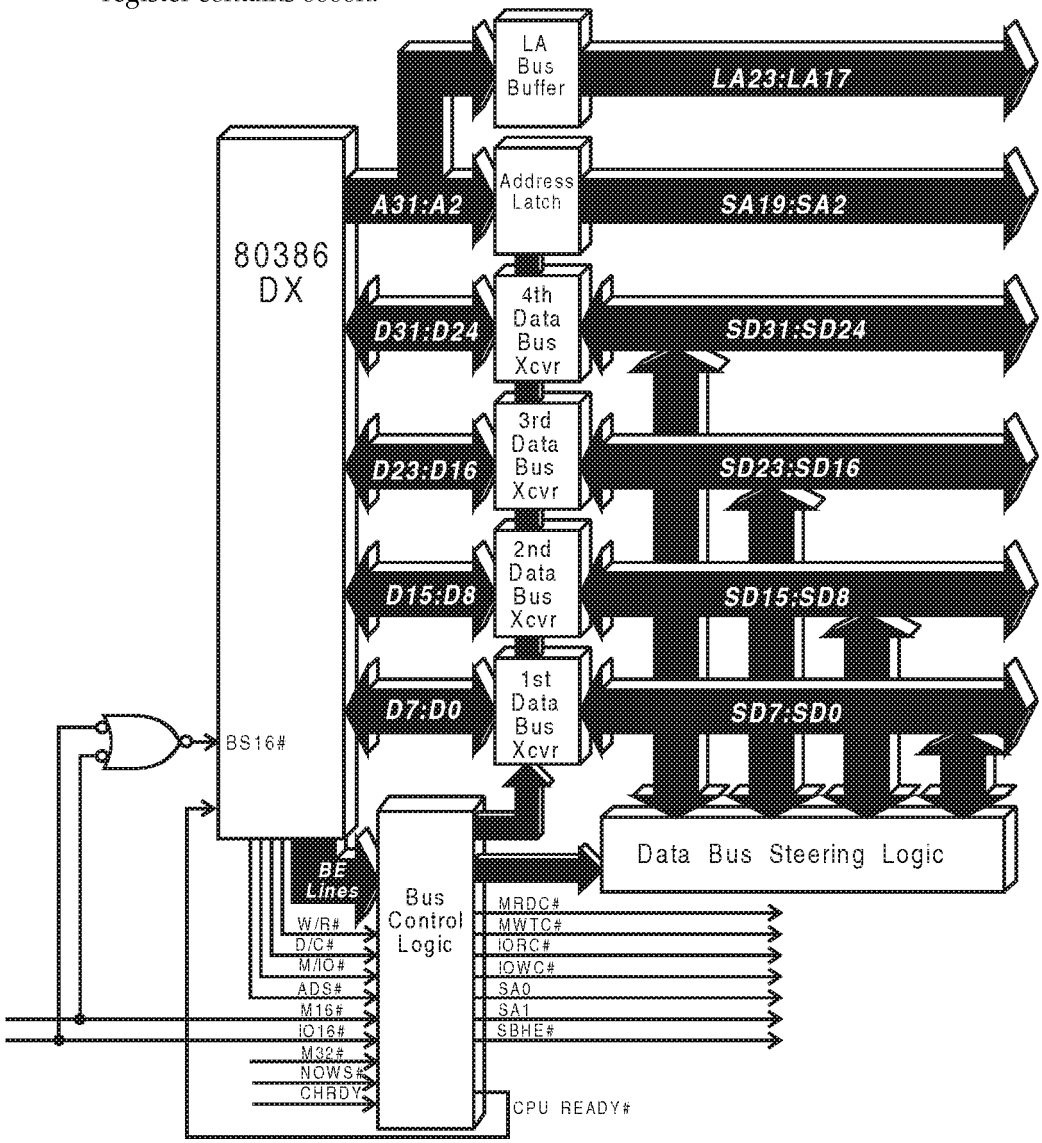


Figure 11-2. The 80386DX-Based Kernel with Dynamic Bus Sizing Implemented

Reading from an 8-Bit Device

One-Byte Read from an 8-Bit Device

In this example, the following instruction is addressing an 8-bit device.

```
MOV  AL,[0100]
```

The microprocessor must read the contents of memory location 00000100h and place the resulting byte in the AL register. Address 00000100h is placed on A[31:2] and BE0# is asserted. In this way, the microprocessor indicates that it wishes to communicate with location 00000100h in the doubleword that starts at location 00000100h. When the microprocessor is addressing either of the first two locations in a doubleword, the bus control logic sets SA1 to 0. The bus control logic also sets SA0 to a 0 because the asserted level on BE0# indicates that the microprocessor is addressing the first even address in the target doubleword. BE1# is deasserted, indicating that the microprocessor isn't addressing the odd location immediately following the addressed even location. As a result, the bus control logic deasserts SBHE#.

Since the target device is an 8-bit device, M16# remains deasserted. As a result, the microprocessor finds BS16# deasserted when sampled at the mid-point of the processor's first data time. This indicates to the microprocessor that it is communicating with a 32-bit device. Note that the 80386DX microprocessor can only sense that it is communicating with 16-bit or 32-bit devices and has no knowledge of 8-bit devices. If BS16# is asserted, the 80386DX knows it is communicating with a 16-bit device. If BS16# isn't asserted, the 80386DX assumes that it is communicating with a 32-bit device.

The 80386DX assumes it is communicating with a 32-bit device and runs a single bus cycle. Since the target device is an 8-bit memory device, it asserts neither M16# nor M32#. The bus control logic samples M16# and M32# at the end of ISA address time and finds them deasserted. Note that the bus control logic does not sample IO16# because it has previously sampled M/IO# and knows that a memory location is being accessed.

Sensing neither M16# nor M32# asserted, the bus control logic determines that it is communicating with an 8-bit memory device and that the data will therefore be sent back over the lower data path. The 8-bit device sees address 00000100h on the address bus and the MRDC# line asserted and returns the requested byte over the lower data path. The bus control logic gates the byte through the lower data bus transceiver to the microprocessor and asserts the

CPU READY# line. When the microprocessor samples CPU READY# asserted, it reads the byte from the lower data path and places it in the target register, AL, completing the execution of the MOV instruction.

Two-Byte Read from an 8-Bit Device

In this example, the following instruction is addressing an 8-bit memory device.

```
MOV  AX,[0100]
```

The microprocessor must read the contents of memory locations 00000100h and 00000101h and place the resulting word in the AX register. Address 00000100h is placed on A[31:2] and BE0# and BE1# are asserted. In this way, the microprocessor indicates that it wishes to communicate with locations 00000100h and 00000101h in the doubleword that starts at location 00000100h. When the microprocessor is addressing either of the first two locations in a doubleword, the bus control logic sets SA1 to 0. The bus control logic also sets SA0 to a 0 because the asserted level on BE0# indicates that the microprocessor is addressing the first even address in the target doubleword. Since BE1# is also asserted, the microprocessor is also addressing the odd location immediately following the addressed even location. As a result, the bus control logic asserts SBHE#.

Since the target device is an 8-bit device, M16# remains deasserted. As a result, the microprocessor finds BS16# deasserted when sampled at the mid-point of the processor's first data time. This indicates to the microprocessor that it is communicating with a 32-bit device. Note that the 80386DX microprocessor can only sense that it is communicating with 16-bit or 32-bit devices and has no knowledge of 8-bit devices. If BS16# is asserted, the 80386DX knows it is communicating with a 16-bit device. If BS16# isn't asserted, the 80386DX assumes that it is communicating with a 32-bit device.

The 80386DX assumes it is communicating with a 32-bit device and runs a single bus cycle. Since the target device is an 8-bit memory device, it asserts neither M16# nor M32#. The bus control logic samples M16# and M32# at the end of ISA address time and finds them deasserted. Note that the bus control logic does not sample IO16# because it has previously sampled M/IO# and knows that a memory location is being accessed.

Sensing neither M16#, IO16#, nor M32# asserted, the bus control logic determines that it is communicating with an 8-bit device and that the data will therefore be returning over the lower data path. This also means that the device can

only return one byte at a time. The 8-bit device sees address 00000100h on the address bus and the MRDC# line asserted and returns the requested byte over the lower data path. The bus control logic latches the byte into the lower data bus transceiver. It then increments the address by placing a 1 on the SA0 line. The MRDC# line is momentarily deasserted and then reasserted again to trick the addressed device into thinking a second bus cycle has begun. The device then places the contents of memory location 00000101h on the lower data path. The bus control logic causes the data bus steering logic to copy the byte to the second data path. All of the requested information has now been assembled, so the bus control logic simultaneously enables the lower data bus transceiver to output the first byte to the microprocessor on the lower data path, while enabling the second data bus transceiver to pass the other byte through to the microprocessor on the second data path.

It then asserts the CPU READY# line. When the microprocessor samples CPU READY# asserted, it reads the two bytes from the two lower data paths and places them in the target register, AX. The byte from location 00000100h is placed in AL and that from 00000101h in AH.

Four-Byte Read from an 8-Bit Device

In this example, the following instruction is addressing an 8-bit memory device.

```
MOV  EAX,[0100]
```

The microprocessor must read the contents of memory locations 00000100h through 00000103h and place the resulting doubleword in the EAX register. Address 00000100h is placed on A[31:2] and all four byte enables are asserted. In this way, the microprocessor indicates that it wishes to communicate with locations 00000100h through 00000103h in the doubleword that starts at location 00000100h. When the microprocessor is addressing either of the first two locations in a doubleword, the bus control logic sets SA1 to 0. The bus control logic also sets SA0 to a 0 because the asserted level on BE0# indicates that the microprocessor is addressing the first even address in the target doubleword. Since BE1# is also asserted, the microprocessor is also addressing the odd location immediately following the addressed even location. As a result, the bus control logic asserts SBHE#.

Since the target device is an 8-bit device, M16# remains deasserted. As a result, the microprocessor finds BS16# deasserted when sampled at the mid-point of the processor's first data time. This indicates to the microprocessor that it is

communicating with a 32-bit device. Note that the 80386DX microprocessor can only sense that it is communicating with 16-bit or 32-bit devices and has no knowledge of 8-bit devices. If BS16# is asserted, the 80386DX knows it is communicating with a 16-bit device. If BS16# isn't asserted, the 80386DX assumes that it is communicating with a 32-bit device.

The 80386DX assumes it is communicating with a 32-bit device and runs a single bus cycle. Since the target device is an 8-bit memory device, it asserts neither M16# nor M32#. The bus control logic samples M16# and M32# at the end of ISA address time and finds them deasserted. Note that the bus control logic does not sample IO16# because it has previously sampled M/IO# and knows that a memory location is being accessed.

Sensing neither M16# nor M32# asserted, the bus control logic determines that it is communicating with an 8-bit memory device and that the data will therefore be returning over the lower data path. This also means that the device can only return one byte at a time. The 8-bit device sees address 00000100h on the address bus and the MRDC# line asserted and returns the requested byte over the lower data path. The bus control logic latches the byte into the lower data bus transceiver. It then increments the address by placing a 1 on the SA0 line. The MRDC# line is momentarily deasserted and then reasserted again, tricking the addressed device into thinking a second bus cycle has begun. The device then places the contents of memory location 00000101h on the lower data path. The bus control logic causes the data bus steering logic to copy the byte to the second data path and commands the second data bus transceiver to latch and hold the byte.

Two of the four requested bytes have now been assembled and the bus control logic must address the next two bytes that have been requested. BE2# is asserted, indicating that the microprocessor is requesting the byte from the second even address in the doubleword. The bus control logic therefore sets SA0 to a 0. Since this location is in the second word of the doubleword, the bus control logic sets SA1 to a 1. The microprocessor has also asserted BE3#, indicating that it wants the contents of the odd location immediately following the even location addressed by BE2#. As a result, the bus control logic asserts SBHE#.

The bus control logic again deasserts and then reasserts the MRDC# line to trick the 8-bit device into thinking another bus cycle has been initiated. The 8-bit device now sees address 00000102h. It places the requested byte on the lower data path and the bus control logic commands the data bus steering logic to copy it to the third data path, where it is latched by the third data bus transceiver.

The bus control logic then increments the address by one by setting SA0 to a 1, and again deasserts and then reasserts the MRDC# line to trick the 8-bit device into thinking another bus cycle has been initiated. The 8-bit device now sees address 00000103h. It places the requested byte on the lower data path and the bus control logic commands the data bus steering logic to copy it to the fourth data path.

All of the requested information has now been assembled, so the bus control logic simultaneously enables the first, second and third data bus transceivers to pass the first three bytes to the microprocessor on the three lower data paths, and also enables the fourth data bus transceiver to pass the other byte through to the microprocessor on the fourth data path.

It then asserts the CPU READY# line. When the microprocessor samples CPU READY# asserted, it reads the four bytes from all four data paths and places them in the target register, EAX. The byte from location 00000100h is placed in AL and that from 00000101h in AH, etc. This completes the execution of the MOV instruction.

Writing to an 8-Bit Device

One-Byte Write to an 8-Bit Device

In this example, the following instruction is addressing an 8-bit memory device.

```
MOV  [0100],AL
```

The microprocessor must write the byte in the AL register to memory location 00000100h. Address 00000100h is placed on A[31:2] and BE0# is asserted. In this way, the microprocessor indicates that it is addressing location 00000100h in the doubleword that starts at location 00000100h. The data byte from the AL register is driven out onto data path 0, D[7:0], by the microprocessor starting at the midpoint of T1. When the microprocessor is addressing either of the first two locations in a doubleword, the bus control logic sets SA1 to 0. The bus control logic also sets SA0 to a 0 because the asserted level on BE0# indicates that the microprocessor is addressing the first even address in the target doubleword. Since BE1# is deasserted, the microprocessor isn't addressing the odd location that immediately follows the addressed even location. As a result, the bus control logic deasserts SBHE#.

Since the target device is an 8-bit device, M16# remains deasserted. As a result, the microprocessor finds BS16# deasserted when sampled at the mid-point of the processor's first data time. This indicates to the microprocessor that it is communicating with a 32-bit device. Note that the 80386DX microprocessor can only sense that it is communicating with 16-bit or 32-bit devices and has no knowledge of 8-bit devices. If BS16# is asserted, the 80386DX knows it is communicating with a 16-bit device. If BS16# isn't asserted, the 80386DX assumes that it is communicating with a 32-bit device.

The 80386DX assumes it is communicating with a 32-bit device and runs a single bus cycle. Since the target device is an 8-bit memory device, it asserts neither M16# nor M32#. The bus control logic samples M16# and M32# at the end of ISA address time and finds them deasserted. Note that the bus control logic does not sample IO16# because it has previously sampled M/IO# and knows that a memory location is being accessed.

Sensing neither M16# nor M32# asserted, the bus control logic determines that it is communicating with an 8-bit memory device and that the data must therefore be sent to the device over the lower data path. The 8-bit device sees address 00000100h on the address bus and the MWTC# line asserted and accepts the byte from the lower data path. The bus control logic asserts the CPU READY# line. When the microprocessor samples CPU READY# asserted, it ends the bus cycle. This completes the execution of the MOV instruction.

Two-Byte Write to an 8-Bit Device

In this example, the following instruction is addressing an 8-bit memory device.

```
MOV   [0100],AX
```

The microprocessor must write the two bytes in the AX register to memory locations 00000100h and 00000101h. Address 00000100h is placed on A[31:2] and BE0# and BE1# are asserted. In this way, the microprocessor addresses locations 00000100h and 00000101h in the doubleword that starts at location 00000100h. The two bytes from the AX register are driven out onto the two lower data paths starting at the midpoint of T1. The contents of the AL register is driven out onto D[7:0], while the contents of AH is driven out onto D[15:8]. When the microprocessor is addressing either of the first two locations in a doubleword, the bus control logic sets SA1 to 0. The bus control logic also sets SA0 to a 0 because the asserted level on BE0# indicates that the microprocessor is addressing the first even address in the target doubleword. Since BE1# is also

asserted, the microprocessor is also addressing the odd location immediately following the addressed even location. As a result, the bus control logic asserts SBHE#.

Since the target device is an 8-bit device, M16# remains deasserted. As a result, the microprocessor finds BS16# deasserted when sampled at the mid-point of the processor's first data time. This indicates to the microprocessor that it is communicating with a 32-bit device. Note that the 80386DX microprocessor can only sense that it is communicating with 16-bit or 32-bit devices and has no knowledge of 8-bit devices. If BS16# is asserted, the 80386DX knows it is communicating with a 16-bit device. If BS16# isn't asserted, the 80386DX assumes that it is communicating with a 32-bit device.

The 80386DX assumes it is communicating with a 32-bit device and runs a single bus cycle. Since the target device is an 8-bit memory device, it asserts neither M16# nor M32#. The bus control logic samples M16# and M32# at the end of address time and finds them deasserted.

Sensing neither M16# nor M32# asserted, the bus control logic determines that it is communicating with an 8-bit memory device and that the data must therefore be transferred over the lower data path. This also means that the device can only accept one byte at a time. The bus control logic gates the two bytes through the two lower data bus transceivers. The 8-bit device detects address 00000100h on the address bus and the MWTC# line asserted, accepts the byte from the lower data path and writes it into location 00000100h. The bus control logic then increments the address by placing a 1 on the SA0 line. The MWTC# line is momentarily deasserted and then reasserted again, tricking the addressed device into thinking a second bus cycle has begun. The bus control logic also disables the lower data bus transceiver and commands the data bus steering logic to copy the second byte from the second data path to the first. The device detects a write to address 00000101h and accepts the data byte from the lower path and writes it into location 00000101h.

All of the requested information has now been written to the target device, so the bus control logic asserts the CPU READY# line. When the microprocessor samples CPU READY# asserted, it ends the bus cycle, completing the execution of the MOV instruction.

Four-Byte Write to an 8-Bit Device

In this example, the following instruction is addressing an 8-bit memory device.

```
MOV   [0100],EAX
```

The microprocessor must write the contents of the EAX register to memory locations 00000100h through 00000103h. Address 00000100h is placed on A[31:2] and all four byte enables are asserted. In this way, the microprocessor indicates that it is addressing locations 00000100h through 00000103h in the doubleword that starts at location 00000100h. The four bytes from the EAX register are driven out onto the four data paths starting at the midpoint of T1. When the microprocessor is addressing either of the first two locations in a doubleword, the bus control logic sets SA1 to 0. The bus control logic also sets SA0 to a 0 because the asserted level on BE0# indicates that the microprocessor is addressing the first even address in the target doubleword. Since BE1# is also asserted, the microprocessor is also addressing the odd location immediately following the addressed even location. As a result, the bus control logic asserts SBHE#.

Since the target device is an 8-bit device, M16# remains deasserted. As a result, the microprocessor finds BS16# deasserted when sampled at the mid-point of the processor's first data time. This indicates to the microprocessor that it is communicating with a 32-bit device. Note that the 80386DX microprocessor can only sense that it is communicating with 16-bit or 32-bit devices and has no knowledge of 8-bit devices. If BS16# is asserted, the 80386DX knows it is communicating with a 16-bit device. If BS16# isn't asserted, the 80386DX assumes that it is communicating with a 32-bit device.

The 80386DX assumes it is communicating with a 32-bit device and runs a single bus cycle. Since the target device is an 8-bit memory device, it asserts neither M16# nor M32#. The bus control logic samples M16# and M32# at the end of ISA address time and finds them deasserted.

Sensing neither M16# nor M32# asserted, the bus control logic determines that it is communicating with an 8-bit memory device and that the data must therefore be transferred over the lower data path. This also means that the device can only accept one byte at a time. All four data bus transceivers are enabled by the bus control logic, allowing all four data bytes to be driven onto the four System Data Bus paths (SD bus) by the microprocessor.

The 8-bit device detects address 00000100h on the address bus and the MWTC# line asserted, accepts the byte from the lower data path and writes it into location 00000100h. The bus control logic then increments the address by placing a 1 on the SA0 line. The MWTC# line is momentarily deasserted and then reasserted again, tricking the addressed device into thinking a second bus cycle has begun. The bus control logic also disables the first data bus transceiver and enables the data bus steering logic to copy the byte from the second path to the first. The device then accepts the data byte from the lower path and writes it into memory location 00000101h.

Two of the four requested bytes have now been transferred and the bus control logic must address the next two locations in the doubleword. BE2# is asserted, indicating that the microprocessor is writing to the second even address in the doubleword. The bus control logic therefore sets SA0 to a 0. Since this location is in the second word of the doubleword, the bus control logic also sets SA1 to a 1. The microprocessor has also asserted BE3#, indicating that it is writing to the odd address immediately following the even location addressed by BE2#. As a result, the bus control logic asserts SBHE#.

The second data bus transceiver is disabled and the bus control logic commands the data bus steering logic to copy the byte from the third data path to the first. It again deasserts and then reasserts the MWTC# line, tricking the 8-bit device into thinking another bus cycle has been initiated. The 8-bit device now detects address 00000102h, accepts the byte on the lower data path and writes it into location 00000102h. The bus control logic then increments the address by setting SA0 to a 1, again tricking the 8-bit device into thinking another bus cycle has been initiated by deasserting and reasserting the MWTC# line. The 8-bit device now detects address 00000103h. The third data bus transceiver is disabled and the data bus steering logic is commanded to copy the byte from the fourth data path to the first data path. The addressed device accepts the fourth and final data byte and writes it into memory location 00000103h.

All of the requested information has now been written to the target device, so the bus control logic then asserts the CPU READY# line. When the microprocessor samples CPU READY# asserted, it ends the bus cycle, completing execution of the MOV instruction.

Reading from a 16-Bit Device

Two-Byte Read from a 16-Bit Device

In this example, the following instruction is addressing a 16-bit memory device.

```
MOV  AX,[0102]
```

The microprocessor will read the contents of memory locations 00000102h and 00000103h and place the resulting word in the AX register. Address 00000100h is placed on A[31:2] and BE2# and BE3# are asserted. In this way, the microprocessor indicates that it is addressing locations 00000102h and 00000103h in the doubleword that starts at location 00000100h. When the microprocessor is addressing either of the last two locations in a doubleword, the bus control logic sets SA1 to 1. The bus control logic sets SA0 to a 0 because the asserted level on BE2# indicates that the microprocessor is addressing an even address in the addressed doubleword. Since BE3# is also asserted, the microprocessor is also addressing the odd location immediately following the addressed even location. As a result, the bus control logic asserts SBHE#.

Since the target device is a 16-bit memory device, it asserts M16#. As a result, the microprocessor finds BS16# asserted when sampled at the mid-point of the processor's first data time. This indicates to the microprocessor that it is communicating with a 16-bit device. The bus control logic samples M16# at the end of ISA address time and finds it asserted. Since this is a read bus cycle, both the microprocessor and the bus control logic then expect the data to come back over the two lower data paths.

The 16-bit device sees address 00000102h on the address bus with SBHE# asserted and the MRDC# line asserted and returns the two requested bytes over the two lower data paths. The bus control logic passes the bytes through the two lower data bus transceivers to the microprocessor.

The bus control logic then asserts the CPU READY# line. When the microprocessor samples CPU READY# asserted, it reads the two bytes from the two lower data paths and places them in the target register, AX. The byte from location 00000102h is placed in AL and that from 00000103h in AH. This completes execution of the MOV instruction.

Four-Byte Read from a 16-Bit Device

In this example, the following instruction is addressing a 16-bit memory device.

```
MOV  EAX,[0100]
```

The microprocessor will read the contents of memory locations 00000100h through 00000103h and place the resulting doubleword in the EAX register. Address 00000100h is placed on A[31:2] and all four byte enables are asserted. In this way, the microprocessor indicates that it is addressing locations 00000100h through 00000103h in the doubleword that starts at location 00000100h. When the microprocessor is addressing either of the first two locations in a doubleword, the bus control logic sets SA1 to 0. The bus control logic sets SA0 to a 0 because the asserted level on BE0# indicates that the microprocessor is addressing the first even address in the target doubleword. Since BE1# is also asserted, the microprocessor is also addressing the odd location immediately following the addressed even location. As a result, the bus control logic asserts SBHE#.

Since the target device is a 16-bit memory device, it asserts M16#. As a result, the microprocessor finds BS16# asserted when sampled at the mid-point of the processor's first data time, indicating to the microprocessor that it is communicating with a 16-bit device. The bus control logic samples M16# at the end of ISA address time and finds it asserted. Since this is a read bus cycle, both the microprocessor and the bus control logic then expect the data to come back over the two lower data paths. This also means that the device can only return two bytes at a time. Since the microprocessor has requested four bytes, it must run two back-to-back bus cycles to get all four bytes.

The 16-bit device detects address 00000100h on the address bus with SBHE# and the MRDC# line asserted and returns the first two of the requested bytes over the two lower data paths. The bus control logic passes both bytes through the lower two data bus transceivers to the microprocessor and asserts CPU READY#. When the microprocessor samples CPU READY# asserted at the end of the current data time, it reads the two bytes from the bus and stores them in the lower two bytes of the target EAX register. This completes the first of the two bus cycles necessary to fulfill the request. Although the microprocessor had asserted all four byte enable lines, it expected just two bytes on the lower data paths because BS16# was sampled asserted.

Two of the requested four bytes have now been read and the microprocessor initiates a second bus cycle using the same address. The same doubleword is

addressed, 00000100h on A[31:2], but only BE2# and BE3# are now asserted. BE2# is asserted, indicating that the microprocessor is requesting the byte from the second even address in the doubleword. The bus control logic therefore sets SA0 to a 0. Since this location is in the second word of the doubleword, the bus control logic sets SA1 to a 1. The microprocessor has also asserted BE3#, indicating that it must also read the contents of the odd address immediately following the even location addressed by BE2#. As a result, the bus control logic asserts SBHE#.

When the microprocessor and bus control logic sample their respective size inputs, the 16-bit memory device being addressed has asserted M16#. The asserted level on the microprocessor's BS16# input indicates that the 16-bit device will return the two requested bytes over the lower two data paths, rather than the upper two as expected. The asserted level on the M16# input to the bus control logic indicates that the addressed device is a 16-bit memory device and the requested data will be transferred over the lower two data paths.

The bus control logic asserts the MRDC# line. The 16-bit device now sees address 00000102h and SBHE# asserted and places the two requested bytes on the two lower data paths. The bus control logic passes the two bytes through the lower two data bus transceivers to the microprocessor and asserts the CPU READY# line. When the microprocessor samples CPU READY# asserted at the end of the current data time, it reads the two bytes from the data bus and stores them in the upper two bytes of the target EAX register. This ends the second bus cycle, completing the execution of the MOV instruction.

Writing to a 16-Bit Device

Two-Byte Write to a 16-Bit Device

In this example, the following instruction is addressing a 16-bit memory device.

```
MOV   [0102],AX
```

The microprocessor must write the contents of the AX register (2 bytes) to memory locations 00000102h and 00000103h. Address 00000100h is placed on A[31:2] and BE2# and BE3# are asserted. In this way, the microprocessor indicates that it is addressing locations 00000102h and 00000103h in the doubleword starting at location 00000100h. When the microprocessor is addressing either of the last two locations in a doubleword, the bus control logic sets SA1 to 1. The bus control logic sets SA0 to a 0 because the asserted level on BE2# indicates that the microprocessor is addressing the second even address in the

target doubleword. Since BE3# is also asserted, the microprocessor is also addressing the odd location immediately following the addressed even location. As a result, the bus control logic asserts SBHE#.

The two bytes from AX register are not only driven onto the two upper data paths by the 80386DX microprocessor, but are also duplicated on the lower two paths.

Since the target device is a 16-bit memory device, it asserts M16#. As a result, the microprocessor finds BS16# asserted when sampled at the mid-point of the processor's first data time. The bus control logic samples M16# at the end of ISA address time and finds it asserted. This indicates to both the microprocessor and the Bus Controller Logic that they are communicating with a 16-bit device.

Sensing M16# asserted, the bus control logic determines that it is communicating with a 16-bit memory device and that the data must therefore be transferred to the device over the two lower data paths. The two lower data bus transceivers are enabled, allowing the two bytes from the AX register to be driven out onto SD[7:0] and SD[15:8]. The 16-bit device sees address 00000102h on the address bus with SBHE# asserted and the MWTC# line asserted and accepts the two bytes from the two lower data paths.

All of the information has now been written to the target device, so the bus control logic asserts the CPU READY# line. When the microprocessor samples CPU READY# asserted, it ends the bus cycle, completing execution of the MOV instruction.

Four-Byte Write to a 16-Bit Device

In this example, the following instruction is addressing a 16-bit memory device.

```
MOV   [0100],EAX
```

The microprocessor must write the contents of the EAX register (4 bytes) to memory locations 00000100h through 00000103h. Address 00000100h is placed on A[31:2] and all four byte enable lines are asserted. In this way, the microprocessor indicates that it is addressing locations 00000100h through 00000103h in the doubleword starting at location 00000100h. When the microprocessor is addressing either of the first two locations in a doubleword, the bus control logic sets SA1 to 0. The bus control logic also sets SA0 to a 0 because the asserted level on BE0# indicates that the microprocessor is addressing the first even address in the target doubleword. Since BE1# is also asserted, the microproces-

sor is also addressing the odd location immediately following the addressed even location. As a result, the bus control logic asserts SBHE#. The four bytes from EAX register are driven onto the four data paths by the 80386DX microprocessor.

Since the target device is a 16-bit memory device, it asserts M16#. As a result, the microprocessor finds BS16# asserted when sampled at the mid-point of the processor's first data time, indicating to the microprocessor that it is communicating with a 16-bit device. The bus control logic samples M16# at the end of ISA address time and finds it asserted.

Sensing M16# asserted, the bus control logic determines that it is communicating with a 16-bit memory device and that the data must therefore be transferred to the device over the two lower data paths. All four data bus transceivers are enabled, allowing the four bytes from the EAX register to be driven out onto the four SD paths. The 16-bit device sees address 00000100h on the address bus with SBHE# asserted and the MWTC# line asserted and accepts the two bytes from the two lower data paths.

The bus control logic asserts CPU READY# to the microprocessor. When the microprocessor samples CPU READY# asserted at the end of the current data time, its asserted state allows the microprocessor to end the first bus cycle. Two of the four bytes from the EAX register have been written to the target 16-bit device.

The second bus cycle is automatically initiated by the microprocessor. Address 00000100h is placed on A[31:2] and BE2# and BE3# are asserted. In this way, the microprocessor indicates that it is addressing locations 00000102h and 00000103h in the doubleword starting at location 00000100h. When the microprocessor is addressing either of the last two locations in a doubleword, the bus control logic sets SA1 to 1. The bus control logic sets SA0 to a 0 because the asserted level on BE2# indicates that the microprocessor is addressing the second even address in the target doubleword. Since BE3# is also asserted, the microprocessor is also addressing the odd location immediately following the addressed even location. As a result, the bus control logic asserts SBHE#.

The two upper bytes from EAX register are not only driven onto the two upper data paths by the 80386DX microprocessor, but are also duplicated on the lower two paths.

Since the target device is a 16-bit memory device, it asserts M16#. As a result, the microprocessor finds BS16# asserted when sampled at the mid-point of the

processor's first data time, indicating to the microprocessor that it is communicating with a 16-bit device. The bus control logic samples M16# at the end of ISA address time and finds it asserted.

Sensing M16# asserted, the bus control logic determines that it is communicating with a 16-bit memory device and that the data must therefore be transferred to the device over the two lower data paths. The two lower data bus transceivers are enabled, allowing the two upper bytes from the EAX register to be driven out onto SD[7:0] and SD[15:8]. The 16-bit device sees address 00000102h on the address bus with SBHE# asserted and the MWTC# line asserted and accepts the two bytes from the two lower data paths.

All of the information has now been written to the target device, so the bus control logic asserts the CPU READY# line. When the microprocessor samples CPU READY# asserted at the end of the current data time, it ends the bus cycle, completing execution of the MOV instruction.

Reading from a 32-Bit Device

In an ISA system, the only 32-bit device is the system board RAM memory. When addressed, it asserts the M32# signal to indicate that it is a 32-bit device. In this example, the following instruction is addressing a 32-bit memory device.

```
MOV  EAX,[0100]
```

The microprocessor must read the contents of the memory locations 00000100h through 00000103h into the EAX register. Address 00000100h is placed on A[31:2] and all four byte enable lines are asserted. In this way, the microprocessor indicates that it is addressing locations 00000100h through 00000103h in the doubleword starting at address 00000100h. When the microprocessor is addressing either of the first two locations in a doubleword, the bus control logic sets SA1 to 0. The bus control logic also sets SA0 to a 0 because the asserted level on BE0# indicates that the microprocessor is addressing the first even address in the target doubleword. Since BE1# is also asserted, the microprocessor is also addressing the odd location immediately following the addressed even location. As a result, the bus control logic asserts SBHE#.

Since the device is a 32-bit memory device, it asserts M32#. When the microprocessor checks its BS16# input at the mid-point of the first data time, it is sensed deasserted. When the bus control logic checks M16# and M32# at the end of address time, M32# is sensed asserted. Since this is a read bus cycle, both the microprocessor and the bus control logic then expect the data to come

back over all four data paths. This also means that the device can return four bytes at a time. Since the microprocessor has requested four bytes, it must only run one bus cycle to get all four bytes.

The bus control logic sets MRDC# asserted. When the 32-bit memory detects address 00000100h with all four byte enables asserted, it sends the contents of all four locations in the doubleword back to the microprocessor over the four data paths. The bus control logic enables all four data bus transceivers, allowing the four bytes of data to flow through to the microprocessor.

The bus control logic then asserts CPU READY#. When the microprocessor samples CPU READY# asserted at the end of the current data time, it reads the four bytes from the four data paths, places them in the EAX register, and ends the bus cycle. This completes the execution of the MOV instruction.

Writing to a 32-Bit Device

In an ISA system, the only 32-bit device is the system board RAM memory. When addressed, it asserts the M32# signal to indicate that it is a 32-bit device. In this example, the following instruction is addressing a 32-bit memory device.

```
MOV   [0100],EAX
```

The microprocessor must write the contents of the EAX register to memory locations 00000100h through 00000103h. Address 00000100h is placed on A[31:2] and all four byte enable lines are asserted. In this way, the microprocessor indicates that it is addressing locations 00000100h through 00000103h in the doubleword starting at address 00000100h. The four bytes from the EAX register are driven out onto the four data paths by the microprocessor.

When the microprocessor is addressing either of the first two locations in a doubleword, the bus control logic sets SA1 to 0. The bus control logic sets SA0 to a 0 because the asserted level on BE0# indicates that the microprocessor is addressing the first even address in the target doubleword. Since BE1# is also asserted, the microprocessor is also addressing the odd location immediately following the addressed even location. As a result, the bus control logic asserts SBHE#.

Since the device is a 32-bit memory device, it asserts M32#. When the microprocessor checks its BS16# input at the mid-point of the first data time, it is sensed deasserted. When the bus control logic checks M16# and M32# at the end of address time, M32# is sensed asserted. Since this is a write bus cycle, both the microprocessor and the bus control logic then expect the data to be

transferred over all four data paths. Since the microprocessor is communicating with a 32-bit device, only one bus cycle is necessary.

The bus control logic asserts MWTC# and enables all four data bus transceivers, allowing the four data bytes to pass through onto the System Data (SD) bus. When the 32-bit memory detects address 00000100h with all four byte enables and the MWTC# line asserted, it accepts the four data bytes and stores them in the four locations within the target doubleword. The bus control logic asserts CPU READY#. When the microprocessor samples CPU READY# asserted at the end of the current data time, it ends the bus cycle, completing execution of the MOV instruction.

80386DX System Kernel without Dynamic Bus Sizing

Introduction

Figure 11-3 illustrates the kernel logic surrounding an 80386DX microprocessor. In this implementation, the designer has not taken advantage of the 80386DX microprocessor's dynamic bus sizing capability (BS16# is deasserted by a pull-up resistor to Vcc). The data bus steering logic is used to handle communication with 8-, 16- and 32-bit devices.

The kernel of a DX-based system without dynamic bus sizing operates identically to a kernel with this feature, except when the microprocessor is communicating with 16-bit devices. The examples for communication with 8- and 32-bit devices described in the section entitled, "The 80386DX System Kernel with dynamic bus sizing Implemented," are also true for the DX-based system without dynamic bus sizing.

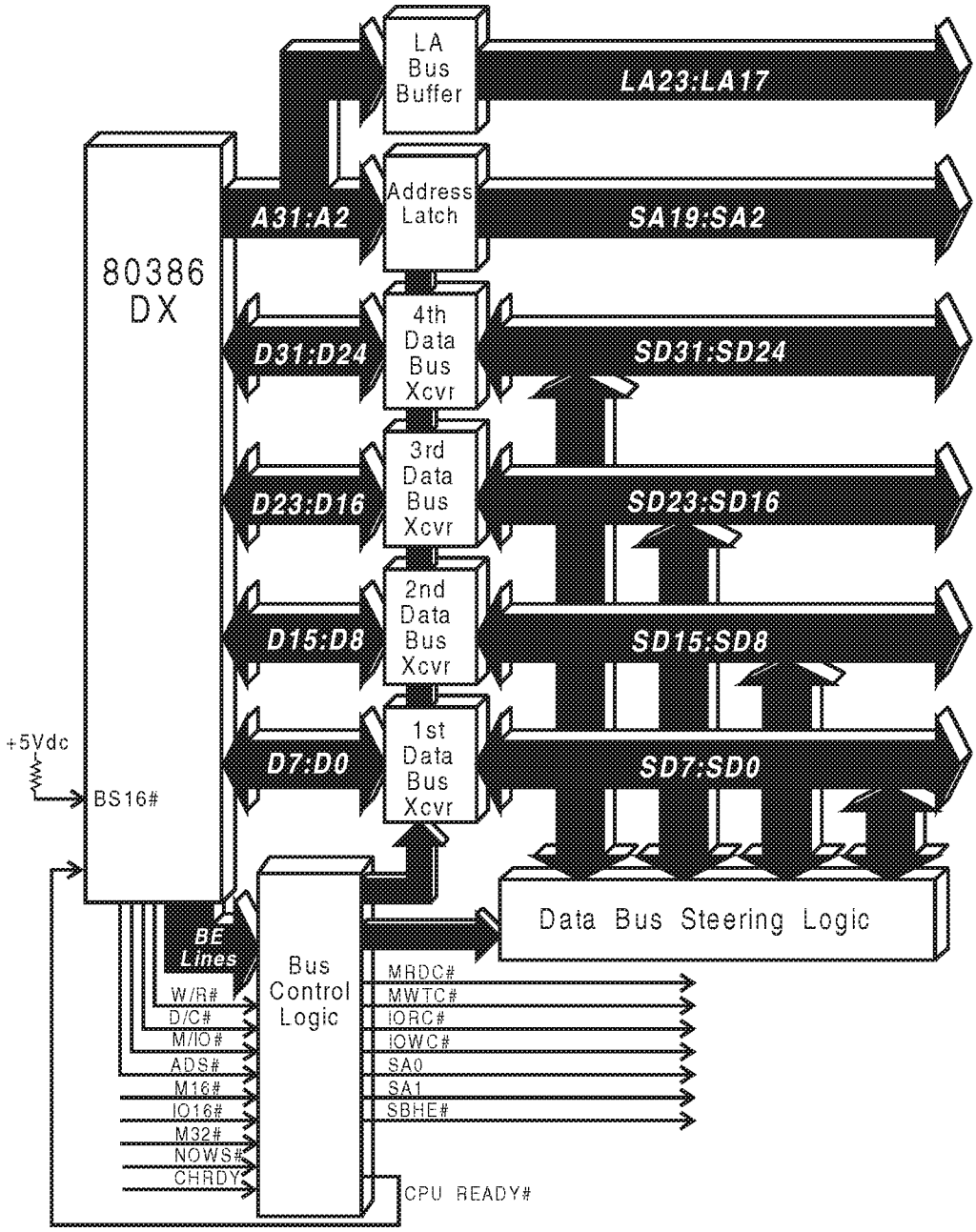


Figure 11-3. The 80386DX-Based Kernel without Dynamic Bus Sizing Implemented

Reading from a 16-Bit Device

Two-Byte Read from a 16-Bit Device

In this example, the following instruction is addressing a 16-bit memory device.

```
MOV  AX,[0102]
```

The microprocessor will read the contents of memory locations 00000102h and 00000103h and place the resulting word in the AX register. Address 00000100h is placed on A[31:2] and BE2# and BE3# are asserted. In this way, the microprocessor indicates that it is addressing locations 00000102h and 00000103h in the doubleword that starts at location 00000100h. When the microprocessor is addressing either of the last two locations in a doubleword, the bus control logic sets SA1 to 1. The bus control logic sets SA0 to a 0 because the asserted level on BE2# indicates that the microprocessor is addressing an even address in the addressed doubleword. Since BE3# is also asserted, the microprocessor is also addressing the odd location immediately following the addressed even location. As a result, the bus control logic asserts SBHE#.

Since the device is a 16-bit memory device, it asserts M16#. When the microprocessor checks its BS16# input at the mid-point of the processor's first data time, it is sensed deasserted (because of the pull-up to Vcc). When the bus control logic checks M16# at the end of address time, it is sensed asserted. Note that the bus control logic does not sample IO16# because it has previously sampled M/IO# and knows that a memory location is being accessed.

Sampling BS16# deasserted indicates to the microprocessor that it is communicating with a 32-bit device, while the bus control logic recognizes that the target is really a 16-bit memory device when it samples M16# asserted. In this read bus cycle, the microprocessor expects the two requested bytes to come back over the upper two data paths, while the bus control logic expects the data to come back over the two lower data paths.

The 16-bit device detects address 00000102h on the address bus with SBHE# and the MRDC# line asserted and returns the two requested bytes over the two lower data paths. The bus control logic commands the data bus steering logic to copy the two data bytes from the lower two to the upper two data paths. The bus control logic then enables the two upper data bus transceivers, passing the bytes through the two upper data bus transceivers to the microprocessor.

The bus control logic then asserts the CPU READY# line. When the microprocessor samples CPU READY# asserted, it reads the two bytes from the two upper data paths and places them in the target register, AX. The byte from location 00000102h is placed in AL and that from 00000103h in AH, completing execution of the MOV instruction.

Four-Byte Read from a 16-Bit Device

In this example, the following instruction is addressing a 16-bit memory device.

```
MOV  EAX,[0100]
```

The microprocessor will read the contents of memory locations 00000100h through 00000103h and place the resulting doubleword in the EAX register. Address 00000100h is placed on A[31:2] and all four byte enables are asserted. In this way, the microprocessor indicates that it is addressing locations 00000100h through 00000103h in the doubleword that starts at location 00000100h. When the microprocessor is addressing either of the first two locations in a doubleword, the bus control logic sets SA1 to 0. The bus control logic sets SA0 to a 0 because the asserted level on BE0# indicates that the microprocessor is addressing the first even address in the target doubleword. Since BE1# is also asserted, the microprocessor is also addressing the odd location immediately following the addressed even location. As a result, the bus control logic asserts SBHE#.

Since the device is a 16-bit memory device, it asserts M16#. When the microprocessor checks its BS16# input at the mid-point of the processor's first data time, it is sensed deasserted (because of the pull-up to Vcc). When the bus control logic checks M16# at the end of ISA address time, it is sensed asserted. Note that the bus control logic does not sample IO16# because it has previously sampled M/IO# and knows that a memory location is being accessed.

Sampling BS16# asserted indicates to the microprocessor that it's communicating with a 32-bit device. The bus control logic, on the other hand, detects that it is communicating with a 16-bit memory device when it samples M16# asserted. Since this is a read bus cycle and the microprocessor thinks it's communicating with a 32-bit device, the it expects all four data bytes to be returned to it simultaneously using all four data paths. The bus control logic expects the data to come back over only the two lower data paths. This also means that the device can only return two bytes at a time.

Because the microprocessor has requested four bytes and will only run one bus cycle because it thinks it's communicating with a 32-bit device. The bus control logic must trick the 16-bit device into thinking two separate bus cycles have been generated by the microprocessor.

The 16-bit device detects address 00000100h on the address bus with SBHE# and the MRDC# line asserted and returns the first two of the requested bytes over the two lower data paths. The bus control logic causes these two bytes to be latched into the lower two data bus transceivers. Seeing that the upper two byte enable lines are also asserted, the bus control logic sets SA1 to 1, SA0 to 0, and asserts SBHE#. It also deasserts and then reasserts the MRDC# line to simulate a second bus cycle.

The 16-bit device now detects address 00000102h with SBHE# and MRDC# asserted and returns the contents of 00000102h and 00000103h on the lower two data paths. The bus control logic then causes the data bus steering logic to copy the two bytes to the upper two paths. It then enables the upper two data bus transceivers to pass the upper two bytes to the microprocessor and simultaneously commands the lower two data bus transceivers to pass the two previously latched bytes to the processor on the lower two data paths.

The bus control logic then asserts CPU READY#. When the microprocessor samples CPU READY# asserted at the end of the current data time, it reads the four bytes from the bus and stores them in the EAX register. This completes the bus cycle and completes the execution of the MOV instruction.

Writing to a 16-Bit Device

Two-Byte Write to a 16-Bit Device

In this example, the following instruction is addressing a 16-bit memory device.

```
MOV   [0102],AX
```

The microprocessor must write the contents of the AX register (2 bytes) to memory locations 00000102h and 00000103h. Address 00000100h is placed on A[31:2] and BE2# and BE3# are asserted. In this way, the microprocessor indicates that it is addressing locations 00000102h and 00000103h in the doubleword starting at location 00000100h. When the microprocessor is addressing either of the last two locations in a doubleword, the bus control logic sets SA1 to 1. The bus control logic sets SA0 to a 0 because the asserted level on BE2# indicates that the microprocessor is addressing the second even address in the

target doubleword. Since BE3# is also asserted, the microprocessor is also addressing the odd location immediately following the addressed even location. As a result, the bus control logic asserts SBHE#.

The two bytes from the AX register are not only driven onto the two upper data paths by the 80386DX microprocessor, but are also duplicated on the lower two paths.

Since the device is a 16-bit memory device, it asserts M16#. When the microprocessor checks it BS16# input at the mid-point of the processor's first data time, it is sensed deasserted (because of the pull-up to Vcc). When the bus control logic checks M16# at the end of ISA address time, it is sensed asserted. Note that the bus control logic does not sample IO16# because it has previously sampled M/IO# and knows that a memory location is being accessed. Sampling BS16# deasserted indicates to the microprocessor that it's communicating with a 32-bit device, while the bus control logic detects that it's communicating with a 16-bit memory device when sampling M16# asserted.

Sensing M16# asserted, the bus control logic determines that it is communicating with a 16-bit memory device and that the data must therefore be transferred to the device over the two lower data paths. Because the 80386DX duplicates the data from the upper paths on the lower paths, the two lower data bus transceivers are enabled, allowing the two bytes from the AX register to be driven out onto SD[7:0] and SD[15:8]. The 16-bit device sees address 00000102h on the address bus with SBHE# asserted and the MWTC# line asserted and accepts the two bytes from the two lower data paths.

All of the information has now been written to the target device, so the bus control logic asserts the CPU READY# line. When the microprocessor samples CPU READY# asserted, it ends the bus cycle, completing execution of the MOV instruction.

Four-Byte Write to a 16-Bit Device

In this example, the following instruction is addressing a 16-bit memory device.

```
MOV   [0100],EAX
```

The microprocessor must write the contents of the EAX register (4 bytes) to memory locations 00000100h through 00000103h. Address 00000100h is placed on A[31:2] and all four byte enable lines are asserted. In this way, the microprocessor indicates that it is addressing locations 00000100h through 00000103h in the doubleword starting at location 00000100h. When the microprocessor is addressing either of the first two locations in a doubleword, the bus control logic sets SA1 to 0. The bus control logic also sets SA0 to a 0 because the asserted level on BE0# indicates that the microprocessor is addressing the first even address in the target doubleword. Since BE1# is also asserted, the microprocessor is also addressing the odd location immediately following the addressed even location. As a result, the bus control logic asserts SBHE#. The four bytes from EAX register are driven onto the four data paths by the 80386DX microprocessor.

Since the device is a 16-bit memory device, it asserts M16#. When the microprocessor checks its BS16# input at the mid-point of the processor's first data time, it is sensed deasserted (because of the pull-up to Vcc). As a result, the microprocessor thinks that it's communicating with a 32-bit device and can transfer the entire doubleword in one bus cycle. However, when the bus control logic checks M16# at the end of address time it is sensed asserted. This indicates that it is communicating with a 16-bit device. This means that the bus control logic must trick the 16-bit device into thinking two separate bus cycles occur, while the microprocessor only performs one.

Sensing M16# asserted, the bus control logic determines that it is communicating with a 16-bit memory device and that the data must therefore be transferred to the device over the two lower data paths. All four data bus transceivers are enabled, allowing the four bytes from the EAX register to be driven out onto the System Data (SD) bus. The 16-bit device sees address 00000100h on the address bus with SBHE# asserted and the MWTC# line asserted and accepts the two bytes from the two lower data paths.

The bus control logic now deasserts and then reasserts the MWTC# line to trick the 16-bit device into thinking another bus cycle has begun. In addition, the bus control logic alters the address by setting SA1 to 1, SA0 to 0 and by asserting

SBHE#. This is done because the microprocessor must also write the upper two bytes into the doubleword and the 16-bit device must see the address for these two locations. The bus control logic disables the lower two data bus transceivers and commands the data bus steering logic to copy the upper two bytes to the two lower paths. The 16-bit device then accepts the two bytes and writes them into the addressed locations, 00000102h and 00000103h.

The bus control logic asserts CPU READY# to the microprocessor. When the microprocessor samples CPU READY# asserted at the end of the current data time, its asserted state allows the microprocessor to end the bus cycle. The four bytes from the EAX register have been written to the target 16-bit device, completing execution of the MOV instruction.

Chapter 12

The Previous Chapter

The chapters prior to this one provided a detailed description of the 80386DX and SX microprocessors, as well as the basic structure and operation of the kernel logic associated with these two variations of the 80386 microprocessor.

This Chapter

This chapter provides a detailed description of the 80386 bus cycle. Timing diagrams are used to illustrate the sequence of events during the 80386 microprocessor's communication with external devices.

The Next Chapter

Part 2, "Memory Subsystems," follows immediately after this chapter, providing a detailed description of RAM memory, cache architecture and ROM memory. Since the processor spends more of its time accessing memory then any other subsystem, it is imperative that the linkage between memory and the processor be optimized to provide the fastest possible throughput. The chapter on cache provides a detailed look at the most common method of achieving this optimization.

The Bus Cycle Types

The 80386 microprocessor is capable of running precisely the same seven types of bus cycles that the 80286 microprocessor can run:

- Memory instruction read
- Memory data read
- Memory data write
- I/O data read
- I/O data write
- Interrupt Acknowledge
- Halt or Shutdown

Address Pipelining Overview

The subject of address pipelining was discussed in the chapter entitled "The 80386DX and SX Microprocessors." Unlike the 80286 microprocessor, however, the 80386 microprocessor can turn address pipelining on and off on a bus-cycle-by-bus-cycle basis. This is accomplished by asserting or deasserting the processor's NA#, next address, input. During the current bus cycle, the 80386 microprocessor samples NA# to determine whether or not to output the address for the upcoming bus cycle early. If NA# is sampled deasserted, the processor will not output the address and bus cycle definition for the upcoming bus cycle until the bus cycle has actually begun. If, on the other hand, NA# is sampled asserted during the current bus cycle, the processor will place the address and bus cycle definition for the upcoming bus cycle on the bus during the current bus cycle. The following discussion of the read and write bus cycles assumes that pipelining is disabled — the NA# input is sampled deasserted during each bus cycle. The section entitled, "Address Pipelining Example," provides a detailed discussion of bus cycle timing when the 80386 microprocessor is using address pipelining.

Memory or I/O Read Bus Cycle

Figure 12-1 illustrates the sequence of events during a memory or an I/O read bus cycle. The following numbered steps can be cross-referenced to the numbered referenced points in the timing diagram. The 80386's address time is referred to as T1, while data time is referred to as T2. The small, dark rectangles identify the points at which a signal is sampled.

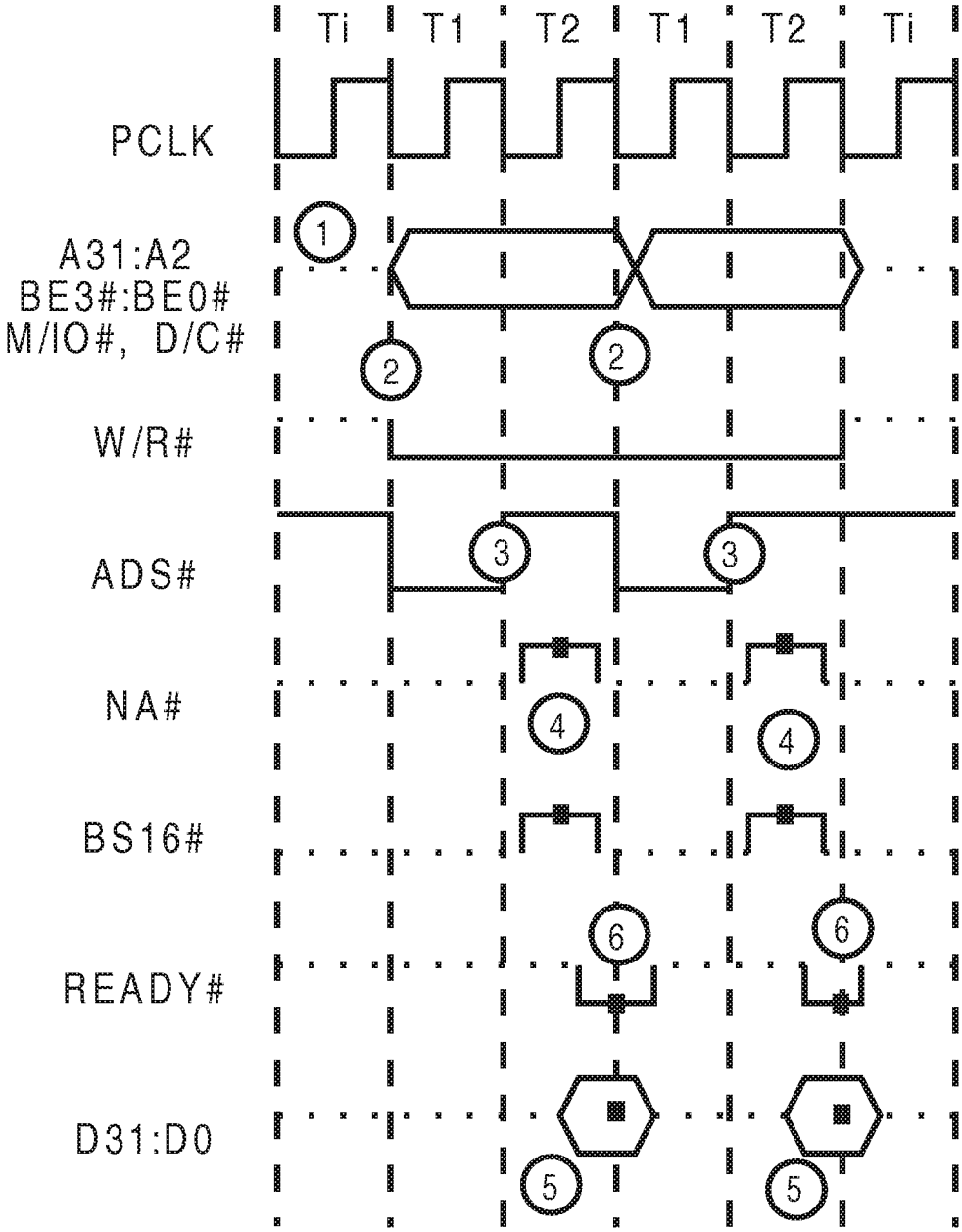


Figure 12-1. Non-Pipelined Read Bus Cycle

1. Because the processor's bus unit is in the idle state prior to this bus cycle, the address for the next bus cycle is not output early. Intel documentation shows that the 80386's internal clock (PCLK) is different than the 80286's PCLK. The 80386's PCLK is inverted when compared to the 80286's.
2. The processor places the memory or I/O address on A[31:2] and asserts the appropriate byte enable lines at the start of T1. The microprocessor also indicates the bus cycle type on its bus cycle definition lines and asserts ADS# to indicate the start of the bus cycle and the presence of a valid address and bus cycle definition.
3. At the end of T1, the microprocessor deasserts the ADS# signal.
4. At the midpoint of T2, the microprocessor samples its BS16# and NA# inputs. In this example, NA# is sampled deasserted, causing the microprocessor to maintain the address for the current bus cycle on the address bus until the bus cycle ends. In this example, BS16# is sampled deasserted, indicating that the currently addressed device is a 32-bit device.
5. The currently addressed device begins to drive the requested data onto the data bus. The exact moment this occurs is dependent upon the addressed device's access time.
6. At the end of T2, the microprocessor samples its READY# input to determine if the currently addressed device is ready to end the bus cycle. In this example, READY# is sampled asserted. The microprocessor reads the requested data from the data bus and ends the bus cycle. The microprocessor immediately initiates the next bus cycle, placing the new address and bus cycle definition on the buses and asserting ADS# again.

   If READY# had been sampled deasserted, the microprocessor inserts a wait state into the bus cycle to provide additional time for the addressed device to respond to the request for data. The wait state consists of an additional T2 time.

Memory or I/O Write Bus Cycle

Figure 12-2 illustrates the sequence of events during a memory or an I/O write bus cycle. The following numbered steps can be cross-referenced to the numbered referenced points in the timing diagram. The 80386's address time is referred to as T1, while data time is referred to as T2. The small, dark rectangles identify the points at which a signal is sampled.

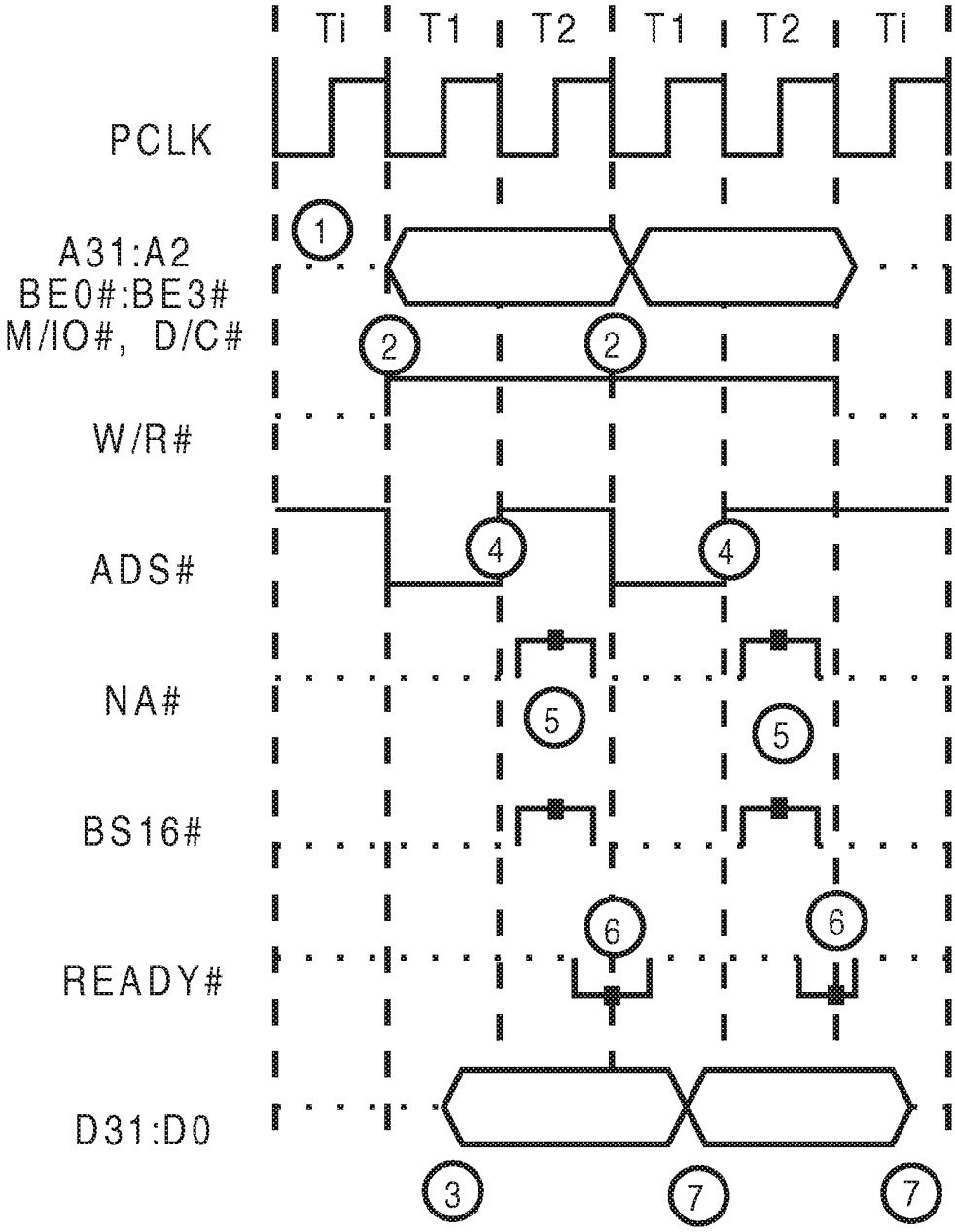


Figure 12-2. Non-Pipelined Write Bus Cycle

1. Because the processor's bus unit is in the idle state prior to this bus cycle, the address for the next bus cycle is not output early. Intel documentation shows that the 80386's internal clock (PCLK) is different than the 80286's PCLK. The 80386's PCLK is inverted when compared to the 80286's.
2. The processor places the memory or I/O address on A[31:2] and asserts the appropriate byte enable lines at the start of T1. The microprocessor also indicates the bus cycle type on its bus cycle definition lines and asserts ADS# to indicate the start of the bus cycle and the presence of a valid address and bus cycle definition.
3. The microprocessor begins to drive the data to be written onto the data bus at the midpoint of T1.
4. At the end of T1, the microprocessor deasserts the ADS# signal.
5. At the midpoint of T2, the microprocessor samples its BS16# and NA# inputs. In this example, NA# is sampled deasserted. This causes the microprocessor to maintain the address for the current bus cycle on the address bus until the bus cycle ends. In this example, BS16# is sampled deasserted, indicating that the currently addressed device is a 32-bit device.
6. At the end of T2, the microprocessor samples its READY# input to determine if the currently addressed device is ready to end the bus cycle. In this example, READY# is sampled asserted, indicating that the currently addressed device has accepted the data being written by the microprocessor. The microprocessor immediately initiates the next bus cycle, placing the new address and bus cycle definition on the buses and asserting ADS# again. If READY# had been sampled deasserted, the microprocessor inserts a wait state into the bus cycle to provide additional time for the addressed device to respond to the request for data. The wait state consists of an additional T2 time.
7. The microprocessor continues to drive the write data onto the data bus until the midpoint of T1 in the next bus cycle, thus ensuring that it satisfies the hold time for the target device.

Address Pipelining Example

Figure 12-3 illustrates the transition from non-pipelined to pipelined operation. Table 12-1 defines the states used in the example. The small, dark rectangles identify the points at which a signal is sampled.

1. The microprocessor's bus unit is idle because no internal bus cycle requests are pending. The prefetch queue is full and the currently executing instruction doesn't require a bus cycle. Intel documentation shows that the 80386's internal clock (PCLK) is different than the 80286's PCLK. The 80386's PCLK is inverted when compared to the 80286's.

2. At the start of T1, the microprocessor initiates a bus cycle to fulfill an internal bus cycle request from the prefetcher or the execution unit. The microprocessor drives the address and bus cycle definition onto the buses and asserts ADS#. W/R# is set high, indicating that this is a write bus cycle. Address decoders begin to see the new address shortly after the start of T1. Address pipelining isn't in effect because the bus unit is just coming out of the idle state. The address can't be placed on the address bus during the prior bus cycle because there isn't one.
3. At the midpoint of T1, the microprocessor begins to drive the write data on the data bus.
4. ADS# is deasserted at the end of T1. Address decoders may latch the address or their chip select on the rising edge of ADS#.
5. At the midpoint of T2, the microprocessor samples its BS16# and NA# inputs. At this point, the deasserted state of BS16# indicates that the microprocessor is communicating with a 32-bit device. The deasserted state on NA# indicates that the currently addressed device does not support address pipelining. As a result, the microprocessor will not place the address and bus cycle definition for the next bus cycle on the buses until the end of the current bus cycle.
6. The microprocessor samples READY# asserted at the end of T2, indicating that the currently addressed device has accepted the write data and is ready to end the bus cycle. In response, the microprocessor ends the bus cycle, but continues to drive the data onto the data bus until the midpoint of T1 in the next bus cycle to ensure that the device successfully latches the data. The microprocessor initiates the next bus cycle at the start of T1. This bus cycle is a read, as indicated by the low on W/R#.
7. At the midpoint of T2, the microprocessor samples BS16# and NA#. BS16# is again deasserted, indicating that the currently addressed device is a 32-bit device. NA# is asserted, however, indicating that the currently addressed device supports address pipelining. The microprocessor must therefore output the address and bus cycle definition for the upcoming bus cycle early (during this bus cycle).

   If READY# is sampled asserted at the end of T2, however, NA# is ignored because the asserted READY# causes the bus cycle to terminate.
8. READY# is sampled deasserted, causing the microprocessor to insert a wait state (T2p) in the bus cycle.
9. At the leading edge of T2p, the microprocessor places the address and bus cycle definition for the next bus cycle on the buses and asserts ADS#. W/R# is set high, indicating that the next bus cycle is a write. The pipelining of the address and bus cycle type allows the device to be addressed in the

next bus cycle to begin its access early so it will be ready to end the bus cycle earlier.

10. At the end of T2p, the microprocessor samples READY# asserted and reads the data from the data bus, ending the current bus cycle. The microprocessor deasserts ADS# and the microprocessor's bus unit enters the T1p state. This is the first PCLK cycle of a pipelined bus cycle.

    ADS# is only asserted for one PCLK cycle when the microprocessor outputs a new address and bus cycle type onto the buses. Since the address and bus cycle type for a pipelined bus cycle are output during T2p of the previous bus cycle, ADS# isn't asserted during T1p.
11. At the midpoint of T1p, the microprocessor begins to drive the write data onto the data bus. At this point, the microprocessor samples NA# asserted, indicating that the currently addressed device supports address pipelining. BS16# is sampled deasserted, indicating that the device addressed in the current bus cycle is a 32-bit device.
12. At the leading edge of T2p, the microprocessor outputs the address and bus cycle type for the upcoming bus cycle and asserts ADS#. W/R# is set low, indicating that the next bus cycle is a read.
13. At the end of T2p, the microprocessor samples READY# asserted, indicating that the device addressed in the current bus cycle is ready to end the write transfer. The microprocessor ends the bus cycle, but continues to drive the write data onto the data bus to ensure the device latches the data correctly. The microprocessor deasserts ADS#.
14. The next bus cycle begins with T1p.
15. At the midpoint of T1p, the microprocessor samples NA# asserted, indicating that the currently addressed device supports address pipelining. In this case, however, the microprocessor's bus unit doesn't have another bus cycle to run, so the next address and bus cycle type are not pipelined out early. The microprocessor samples BS16# deasserted at the midpoint of T1p, indicating that the device addressed in the current bus cycle is a 32-bit device.
16. The microprocessor's bus unit enters the T2i state. This is data time and the bus unit is preparing to go back to the idle state after this bus cycle completes. At the start of T2i, the microprocessor ceases to drive the address and bus cycle type.
17. At the end of T2i, the microprocessor samples READY# asserted, indicating that the device addressed in the current bus cycle is ready to end the bus cycle. The microprocessor reads the data from the data bus and ends the bus cycle.
18. The microprocessor's bus unit enters the idle state until the prefetcher or the execution unit request another bus cycle.

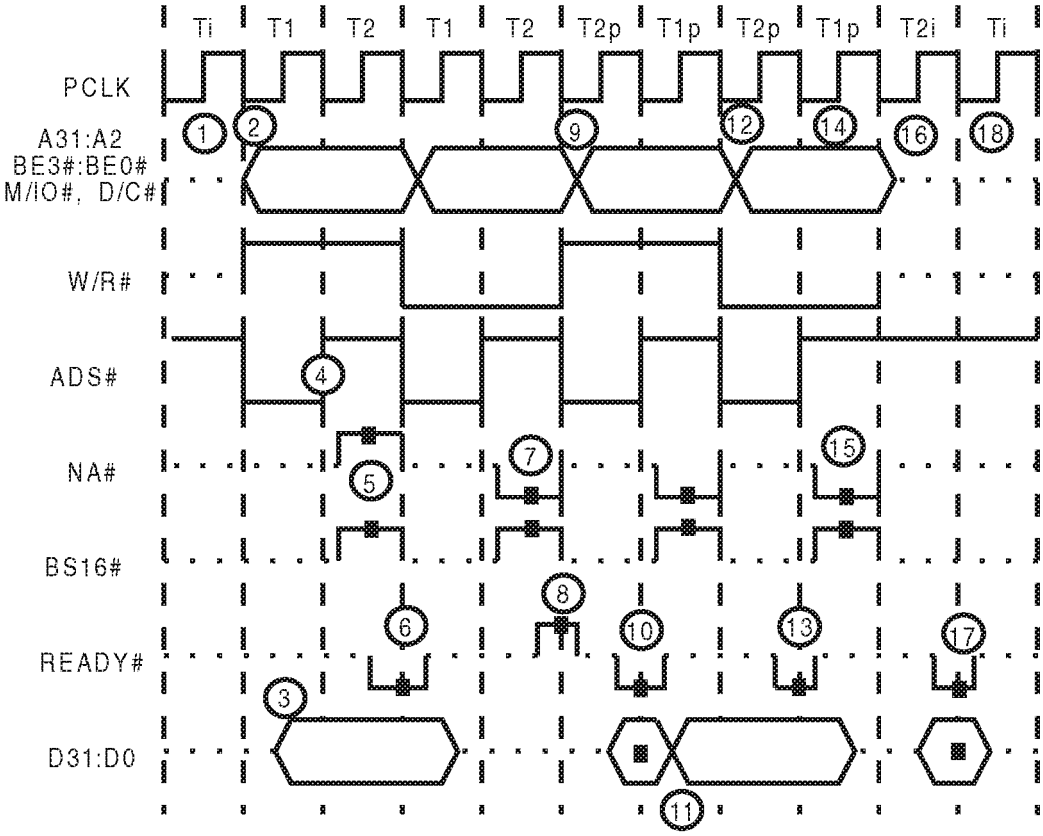


Figure 12-3. Transition from Non-Pipelined to Pipelined Operation

Table 12-1. 80386 Bus Unit States

State	Description
Ti	The **idle state**. The bus unit remains in the idle state when it has no bus cycles to run (the prefetch queue is full and the currently executing instruction does not require a bus cycle).
T1	**Address time**. The microprocessor places the address and bus cycle type on the buses and asserts ADS#. If pipelining is in effect, the address is output prior to T1. If this is a write bus cycle, the microprocessor also begins to drive the data onto the data bus at the midpoint of T1.
T2	**Data time**. Time period allocated to allow data to be transferred between the microprocessor and the target device. NA# is sample at the T2 midpoint to determine if the processor should initiate early address output (address pipelining). BS16# is also sampled at the T2 midpoint to determine if the microprocessor is communicating with a 16- or 32-bit device. READY# is sampled at the end of T2 to determine if the currently addressed device is ready to end the bus cycle.
T1p	**Address time in a pipelined bus cycle**. NA# is sampled at the T1p midpoint to determine if the next address and bus cycle definition should be output early (at the start of T2p). BS16# is also sampled at the midpoint of T1p to determine if the currently addressed device is a 16- or 32-bit device. If this is a write bus cycle, the microprocessor begins to drive the data onto the data bus at the midpoint of T1p.
T2p	**Data time in a pipelined bus cycle**. The address and bus cycle definition for the upcoming bus cycle are placed on the buses at the start of T2p. ADS# is asserted for the duration of T2p. READY# is sampled at the end of T2p to determine if the currently addressed device is ready to end the bus cycle.

Interrupt Acknowledge Bus Cycle

The interrupt acknowledge bus cycle is discussed in the chapter entitled "The Interrupt Subsystem."

Halt or Shutdown Bus Cycle

The processor runs the halt or shutdown bus cycle under either of the following circumstances:

- The HLT (HALT) instruction causes the microprocessor to cease fetching and executing instructions.
- The microprocessor detects a fault while running the double-fault handler.

When the microprocessor executes the HLT instruction, it runs the halt or shutdown bus cycle to indicate its intention to halt to external logic. In an ISA machine, this has no effect on external logic.

When the 80386 microprocessor detects an exception while attempting to execute the handler for a prior exception, the two exceptions are generally handled serially. Certain combinations of exceptions (including page faults and what Intel refers to as "contributory exceptions") cannot be handled serially, however, so the 80386 microprocessor invokes a double-fault exception instead. If the microprocessor detects an exception while attempting to service the double-fault, it goes into a shutdown condition. It indicates this to external logic by running the halt or shutdown bus cycle. In an ISA machine, the shutdown detect logic outside the microprocessor detects the shutdown and toggles the microprocessor's RESET line, causing the system to reboot.

The contributory exceptions are mostly caused by protection violations in protected mode. For example, in protected mode, if an application attempts to access data that is outside its data segment, the microprocessor will start running the general protection fault exception handler. If, say, a stack overflow were to occur while the microprocessor ran the general protection fault exception handler, the microprocessor would not be able to handle the two faults serially. The microprocessor would invoke the double-fault exception instead. If the microprocessor detected another exception while attempting to service the double-fault, it would go into a shutdown condition. This is sometimes referred to as a triple fault.

The halt or shutdown bus cycle is indicated by ADS# being asserted and:

- M/IO# and W/R# driven high.
- D/C# driven low.

In addition, the microprocessor asserts BE0# to indicate a shutdown, or asserts BE2# to indicate a halt. The address bus outputs are driven low.

During a halt or shutdown, the 80386 may service PEREQ (processor extension request) or HOLD requests. For further information on PEREQ and HOLD, refer to the chapters entitled "Numeric Coprocessor" and "Direct Memory Access (DMA)" respectively.

Either NMI (non-maskable interrupt request) or RESET will force the 80386 out of either a halt or a shutdown. If interrupts are enabled, an INTR (maskable interrupt request) will also force the microprocessor out of a halt (but not out of shutdown). For information on interrupts, refer to the chapter entitled "The Interrupt Subsystem."

Part 2

Memory Subsystems

Part 2 covers the major memory subsystems that can be found in ISA systems. The memory chapters cover specific devices along with implementations of these devices found within the ISA environment.

Part 2 includes the following chapters:

- Chapter 13: RAM Memory: Theory of Operation
- Chapter 14: Cache Memory Concepts
- Chapter 15: ROM Memory

Two of the memory subsystems, "RAM memory" often referred to as system memory, and the "system ROM" are found in all ISA systems. The other subsystem covered, "cache memory," is typically found in 80386 or higher systems running at 20MHz and above.

Chapter 13

The Previous Chapter

Part 1 of the book provided a detailed description of microprocessor communications and the kernel support logic.

This Chapter

Many memory designs can be found in ISA systems. These designs are based on two broad classifications of memory devices: dynamic RAM and static RAM. The primary emphasis of any memory design revolves around performance and cost, but other factors such as physical space available, power consumption, and external support circuitry play important roles in determining the design of a RAM subsystem. This chapter details the most commonly found memory types and memory implementations.

The Next Chapter

Chapter 14 explains the concept of cache memory.

Dynamic RAM (DRAM) Memory

Dynamic RAM (DRAM) memory is the most commonly used RAM device in PCs due to its lower cost when compared to static RAM, or SRAM. Though its cost is less than SRAM, DRAM chips are much slower. Many innovations have occurred in DRAM designs to increase speed. As a result, much of this chapter focuses on the various types of DRAM and associated subsystem design.

A DRAM memory chip contains many storage locations. Depending on the manufacturer and the chip type, a DRAM memory chip may contain 64K, 256K, 1M, 4M, or more storage locations. Internally, the storage locations are laid out in rows and columns.

As it is with any other device in the system, DRAM is accessed by the current bus master. The current bus master is the device that currently owns the ad-

dress, data and control buses. It is the device that runs the bus cycle that is currently under way and it broadcasts the address. It may be the system microprocessor, one of the DMA controllers or a 16-bit bus master plugged into the ISA bus. In order to identify the specific location that the current bus master wants to communicate with, the DRAM chip must receive the address in two steps:

- the row address
- the column address

DRAM must always be addressed in that order: row, then column. The intersection of the row and column defines the exact location the current bus master wants to communicate with. The state of the DRAM's read/write pin, high or low, defines the type of memory operation. The data bus is connected to the DRAM chip so that the data can be transferred from the DRAM chip to the current bus master during a read or from the current bus master to the DRAM chip during a write.

DRAM Addressing Sequence

The following paragraphs describe the exact sequence of events that must occur when addressing a DRAM memory location. Refer to figure 13-1.

1. First, the row address must be routed to the DRAM's address pins on the memory address bus.
2. The RAS# (row address strobe) line must then be asserted by the DRAM addressing logic. This causes the row address present on the DRAM's address inputs to be latched into the row latch inside the DRAM. The DRAM's row address decoder begins to decode the latched row address, selecting one row line to be activated.
3. The DRAM addressing logic then delays a fixed amount of time before giving the column address to the DRAM chip. This is called the RAS/CAS delay and is described later in this chapter.
4. After the RAS/CAS delay has elapsed, the DRAM addressing logic routes the column portion of the address to the DRAM's address inputs on the memory address bus and asserts CAS# (column address strobe). This causes the column address to be latched into the column address latch inside the DRAM chip.
5. The DRAM chip now has all the information it needs to identify the location that the current bus master is attempting to communicate with. The DRAM's internal row address decoder has already selected the specified row line. The DRAM's internal column address decoder now selects and

activates the specified column line. The point at which the row and the column lines intersect identifies the storage location that the current bus master is attempting to communicate with.

6. The state of the DRAM's read/write input pin indicates whether the current bus master is attempting to read the information stored in the specified storage location or write information into that location.

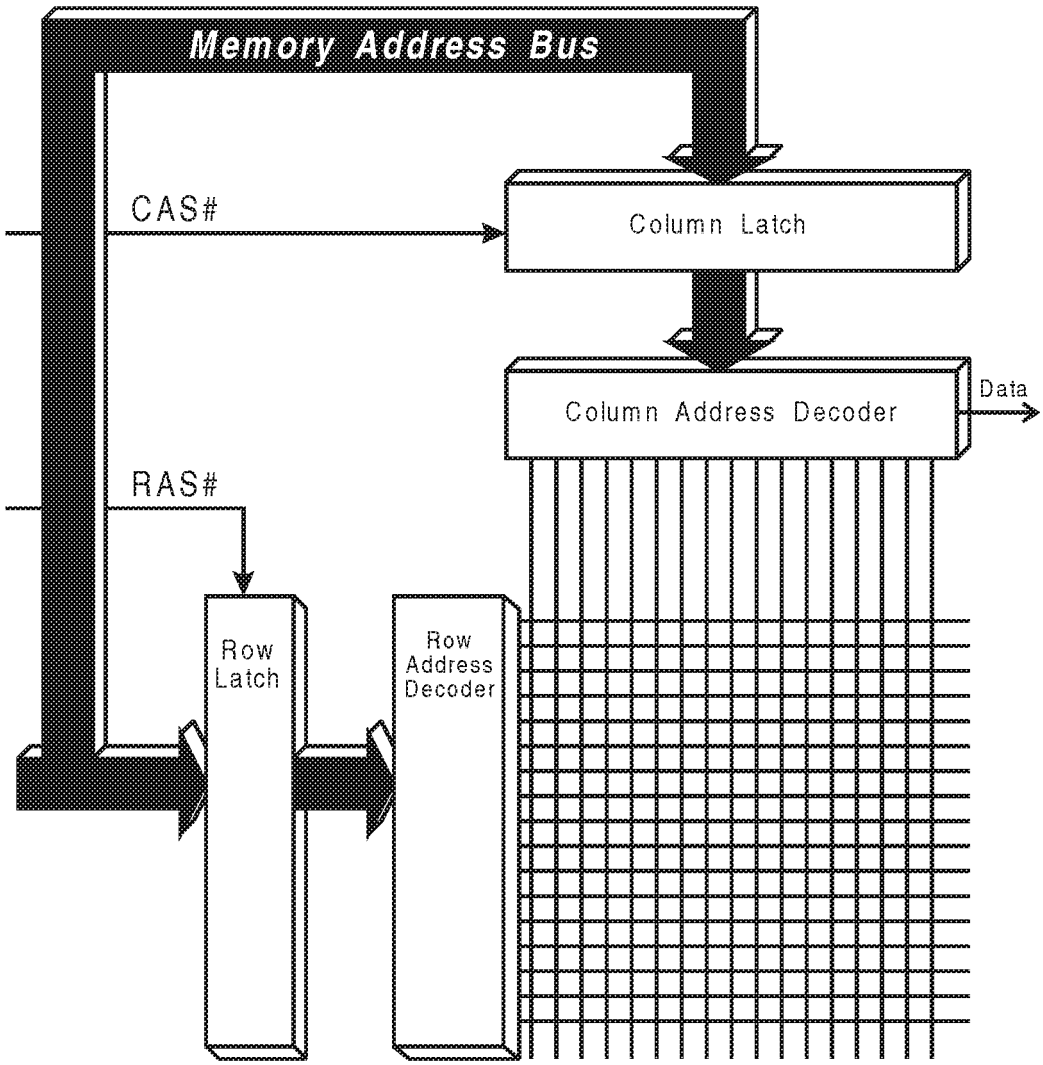


Figure 13-1. Block Diagram of a DRAM Memory Chip

If a read operation is in progress, the contents of the indicated location are placed on the data bus by the DRAM so they can be read by the current bus master. If a write operation is in progress, the information that the current bus master is presenting on the data bus is written into the specified storage location, completing the bus cycle.

Row and Column Address Source

The fact that the DRAM expects to see the address presented in two steps, row and column, presents a special problem. The DRAM expects the address in a form other than that output by the current bus master. This necessitates the inclusion of some type of logic to translate the address being output by the current bus master into a row and column address. This address translation logic is commonly referred to as the DRAM addressing logic.

Bus masters always output 24- (the 80286 and 80386SX microprocessors) or 32-bit (the 80386DX, 80486 and Pentium microprocessors) addresses onto the address Bus. From the DRAM addressing logic's perspective, the address is treated as three packets of information. Refer to figure 13-2.

- The upper-most bits on the address bus are used by the DRAM address decoder to determine if the address currently on the bus is within the address range assigned to the DRAM.
- The lower portion of the address is divided into two packets of equal size:
    - The row address
    - The column address

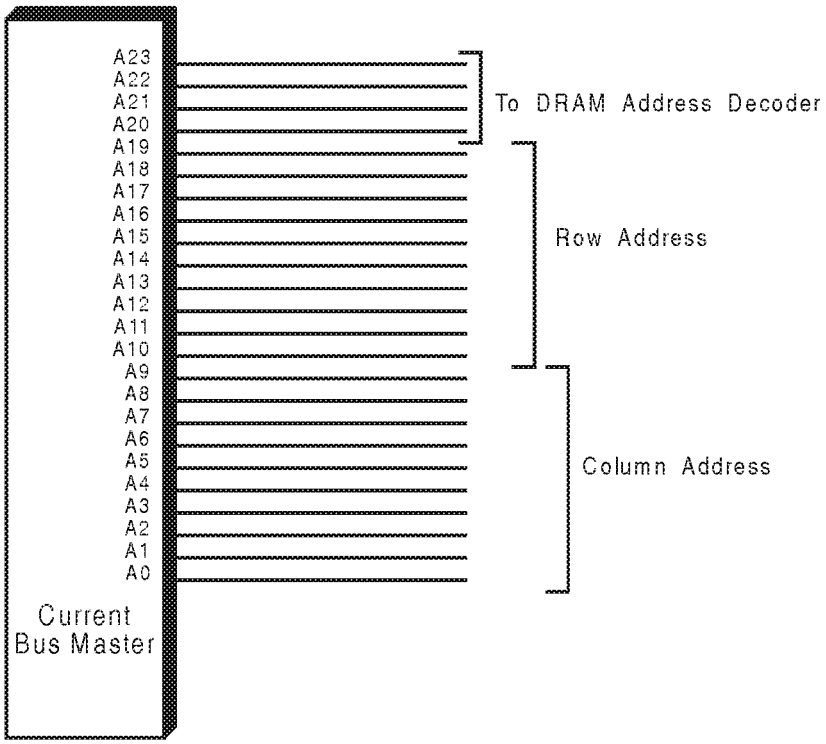


Figure 13-2. The Address Output by the Current Bus Master

DRAM Addressing Logic

As stated earlier, the steps involved in addressing DRAM must occur in an exact sequence and at specified intervals. A special timer device is included in the DRAM addressing logic, sequencing the actions of the DRAM addressing logic at the proper intervals. This timer is usually implemented using a time delay device.

The time delay device has one input called a trigger and a series of outputs. Once triggered by an asserted level on its trigger input, the time delay device asserts each of its outputs at precise intervals. As an example, the time delay device shown in figure 13-3 has a ten nanosecond (10ns) resolution. 10ns after it is triggered, its first output is asserted; 10ns later the second output is asserted, etc. By tapping off selected outputs of the time delay device, the designer can choose the exact moment in time (relative to the trigger point) for a specific event to occur.

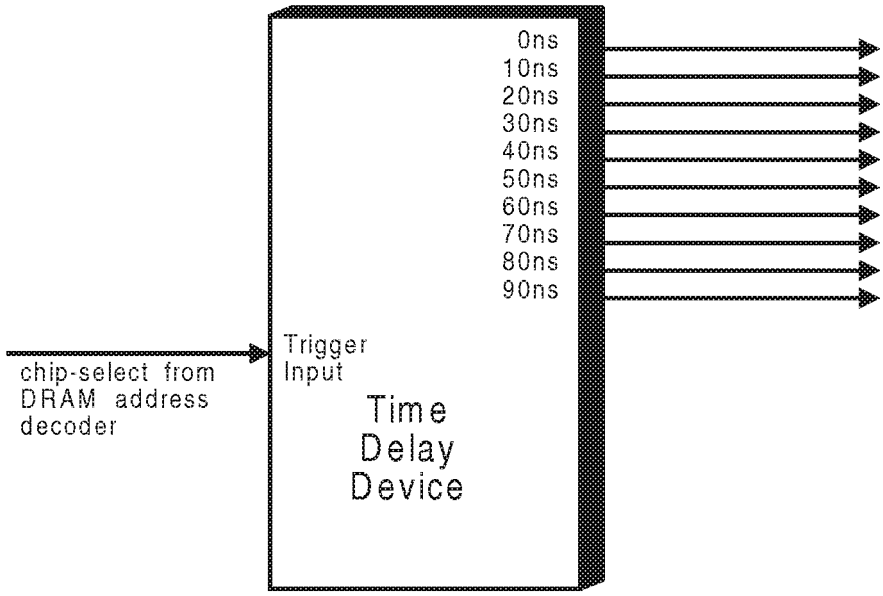


Figure 13-3. The Time Delay Device

The major components of the DRAM addressing logic are:

- the DRAM address decoder
- the time delay
- the DRAM addressing multiplexer, or MUX
- the RAS/CAS generation logic

Refer to figure 13-4. The DRAM address multiplexer (MUX) has two sets of inputs and one set of outputs. It has only one control line, called a select line, that controls its function, and this line can be set either high or low. A low selects the "A" set of inputs to be gated through to the MUX's outputs. When the select line is high, the "B" set of inputs is selected and gated through to the MUX's outputs. In other words, a MUX is nothing more than a switch, in this case a two position switch, that can select one of two groups of input signals to gate through to its output lines.

The upper-most bits on the address bus are connected to the DRAM address decoder so it can determine if the address currently on the bus falls within those assigned to this DRAM.

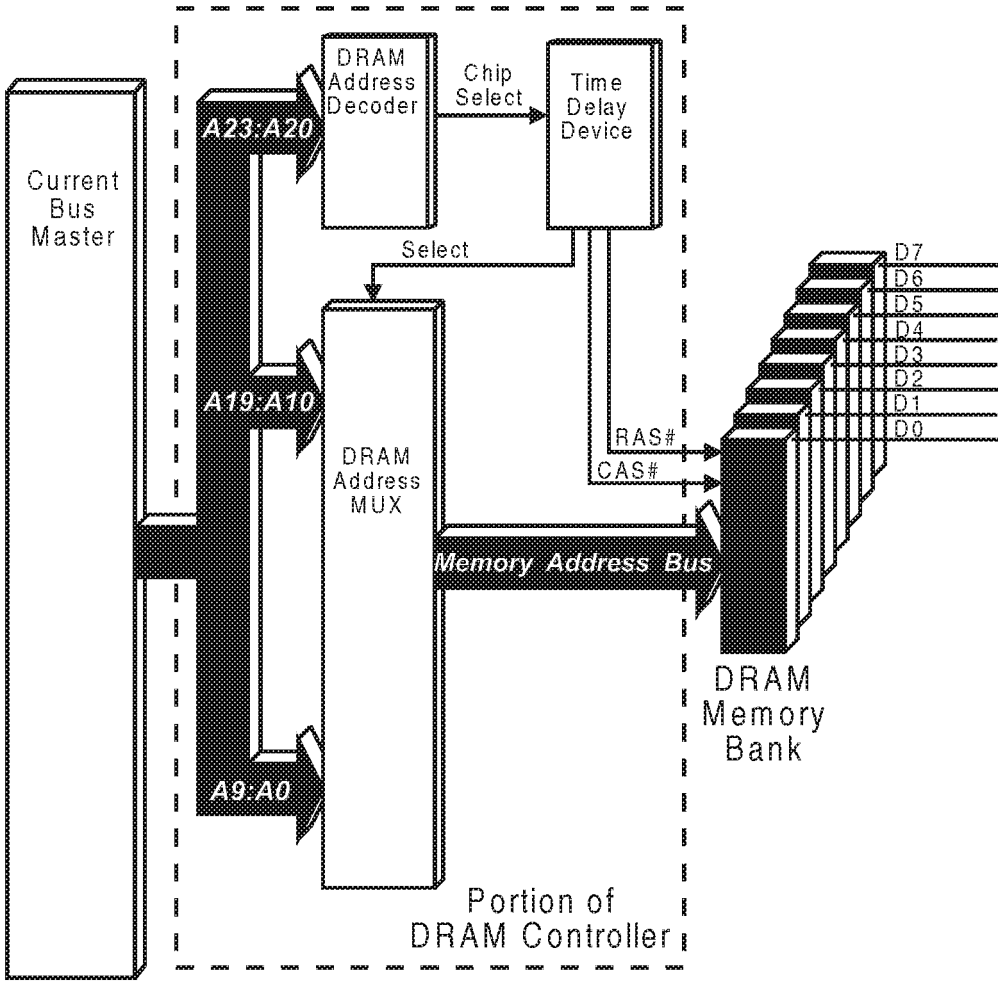


Figure 13-4. Block Diagram of DRAM Addressing Logic

The lower bits on the address bus are connected to the two sets of inputs on the DRAM address multiplexer. One half of these bits are connected to the MUX's "A" set of inputs, and the other half to its "B" set of inputs. Initially, the select line on the MUX is set to zero, selecting the "A" set of inputs to be gated through to the MUX's outputs. The outputs of the MUX are connected to the memory address bus, which is attached to the address inputs of the DRAM.

Detailed Description of DRAM Addressing Sequence

The following paragraphs describe the sequence of events during the addressing of a DRAM memory location. Refer to the block diagram in figure 13-4 and the timing diagram illustrated in figure 13-5.

1. **DRAM address decode**. At the beginning of the bus cycle, the current bus master places the address on the address bus, and the bus cycle definition on the bus cycle definition lines. The DRAM address decoder determines that it is a memory address by the high on the M/IO# line, and decodes the upper-most bits on the address bus to determine if the address is within the range of addresses assigned to the DRAM. If it is, the DRAM address decoder's chip-select output is asserted, providing a trigger to the time delay device.
2. The **row address**. When initially triggered, the time delay outputs are all deasserted, causing the MUX's select input to be set low. This selects the MUX's "A" set of inputs, address bits A[19:10], to be gated onto the memory address bus. This portion of the address provides the row address to the DRAM memory.
3. **Row address strobe** (RAS#). In the example DRAM addressing logic (figure 13-4), 10ns is allowed to permit the row address to settle on the memory address bus prior to latching it into the DRAM chips. 10ns after the time delay was triggered by the chip-select, the first output of the time delay is asserted (high), causing RAS# (row address strobe) to be asserted. The high-to-low transition on the RAS# line causes the DRAM's internal row address latch to latch the row address presented to it on the memory address bus.
4. **RAS/CAS delay**. For correct operation, the designer must delay a DRAM-specific amount of time prior to presenting the column address to the DRAM chips. If the column address is presented too soon, the DRAM chips will not function correctly. This delay between the presentation of the row address and the presentation of the column address is known as the RAS/CAS delay. The exact duration of the delay is specific to the type of DRAM chips being used.
5. The **column address**. In the example DRAM addressing logic, 30 nanoseconds after the trigger, the time delay's second output is asserted, setting the MUX's select line to a one. This selects the MUX's "B" set of inputs, A[9:0], to be gated onto the memory address bus. This portion of the address provides the column address to the DRAM memory.
6. **Column address strobe** (CAS#). In the example logic, 10ns is allowed to pass to allow the column address to settle on the memory address bus prior to latching it into the DRAM chip. 10ns later, the time delay's third output

is asserted, causing CAS# (column address strobe) to be asserted (low). The high-to-low transition on CAS# causes the DRAM chip's internal column address latch to latch the column address being presented to it on the memory address bus.

7. **Read or write**? Internally, the DRAM chip has now latched the row and column addresses and activates one row and one column line. The point at which they intersect defines the location the current bus master is attempting to access. The DRAM addressing logic uses the S0# and S1# (80286-specific lines; W/R# if an 80386, 80486 or Pentium processor-based system) bus cycle definition lines (or signals derived from them) to determine if a read or write operation is in progress. The read/write control input of the DRAM chip is set accordingly.
8. The **data transfer**. Assuming a read operation is in progress, the contents of the addressed DRAM storage location are placed on the data bus so they may be read by the current bus master. If a write operation, the data sent to the DRAM by the current bus master is written into the addressed storage location.

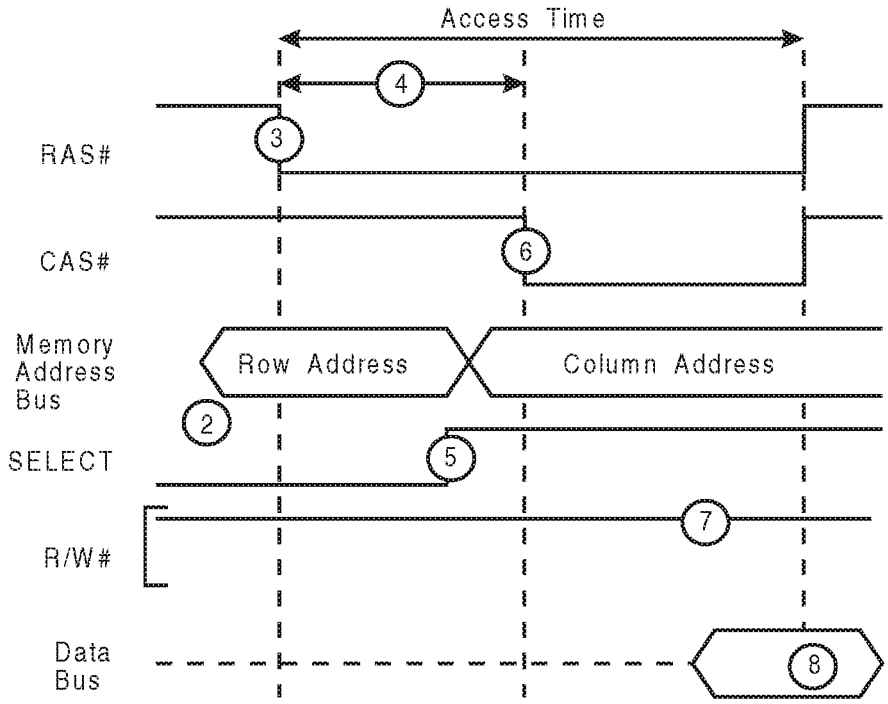


Figure 13-5. DRAM Addressing Timing Diagram

How Data is Stored in DRAM

A typical DRAM location can only store one bit of information, a one or a zero. This one-bit storage location is referred to as a bit cell and consists of a capacitor and a transistor. As an example, an electrical charge is placed on the capacitor to store a one, while the capacitor is discharged to store a zero in the location.

The most important operational characteristic of a capacitor is its tendency to lose a charge, or drain, over a period of time. The discharge time is related partly to the size of the capacitor. The capacitors in DRAM cells are quite small. They have a capacity of around 100 femtofarads, or 10^{-14}F. When a bit is stored in a DRAM bit cell by placing a charge on its capacitor, therefore, the capacitor discharges in a relatively short period of time and loses the bit stored in it. This would appear to be a highly inefficient form of storage.

DRAM Refresh

To ensure that information stored in DRAM memory isn't lost, the bit cells containing charges must be periodically recharged to a good level. The process of rejuvenating DRAM locations with information stored in them is known as "refreshing" the DRAM. DRAM memory is called "dynamic" because it must be refreshed on a dynamic, ongoing basis or it will lose the information stored in it.

Storage locations within a row in a typical DRAM chip are refreshed by activation of the row line. When a row line is activated with the DRAM's read/write pin in the read state, every bit cell (every storage location) that contains a charge along the row is automatically recharged up to a good level. In other words, it is refreshed.

Since a row line is activated whenever any location along the row is addressed, one way to refresh DRAM memory would be to create a program that reads from one DRAM location within each row on an on-going basis. This would cause the activation of every row line within every DRAM chip and therefore refresh every location. This method for refreshing DRAM isn't feasible because the microprocessor would be tied up refreshing DRAM memory all of the time and would not have any time left over to do any useful work.

Refresh Logic and RAS-only Refresh

DRAM refresh is actually handled by logic separate from the microprocessor known as the refresh logic. In ISA machines, the refresh logic consists mainly of a refresh row counter and timer #1, the refresh timer. Refer to figure 13-6. When the system is first powered up, the system's RESET signal is asserted until the power has come up and stabilized. The asserted level on reset causes the refresh row counter to be preset to a count of zero (row zero). Timer #1, the refresh timer, is designed to assert REFRESH TRIGGER once every 15 microseconds.

When the power has stabilized and RESET becomes deasserted, the system begins to run, meaning the microprocessor begins to fetch and execute instructions. Fifteen microseconds after the system begins to run, the refresh timer asserts the REFRESH TRIGGER signal.

In order to refresh a row of DRAM memory, the refresh logic must send a row address to the DRAM and set the DRAM's read/write control line to the read state. To do this, the refresh logic must run a special memory read bus cycle. In other words, the refresh logic must become bus master before running the refresh bus cycle. The numbered list that follows figure 13 - 6 describes the sequence of events involved in DRAM refresh.

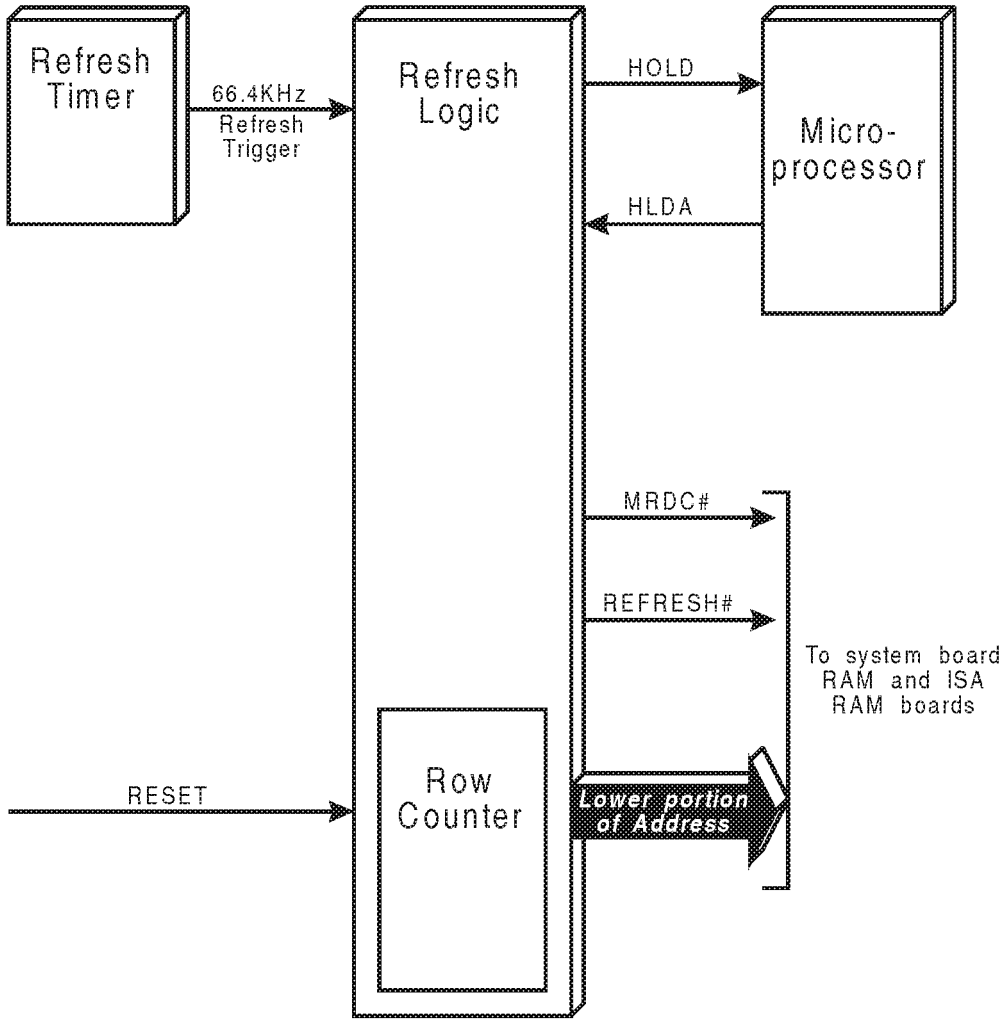


Figure 13-6. The Refresh Logic

1. The refresh timer sends a REFRESH TRIGGER to the refresh logic on the system board.
2. The refresh logic sends a Hold Request (HOLD) to the microprocessor to request the buses and awaits HLDA from the CPU.
3. The refresh logic asserts the ISA bus cycle definition line MRDC# to indicate that a memory read bus cycle is in progress.

4. The refresh row counter outputs its current row number (initially, row zero) onto the lower portion of the address bus. At the same time, the row counter is internally incremented by one (to row one).
5. The ISA REFRESH# signal informs every memory card in the system that a refresh bus cycle is in progress. Consequently, on every memory card, the row address from the refresh row counter is applied to the "A" set of inputs of the card's DRAM Address MUX. The row address, currently row zero, is gated through every memory card's MUX onto its memory address bus. The card's DRAM addressing logic then asserts RAS# to latch the row address into every DRAM chip in the system simultaneously.
6. Since the ISA bus is indicating a memory read bus cycle is in progress, the read/write pin on every DRAM chip will be set to the read state.
7. This set of actions causes the exact same row line (initially, row zero) within every DRAM chip in the system to be activated at the same time, thereby refreshing every location along that row in every DRAM chip.
8. After completion of the refresh bus cycle, the refresh logic returns bus ownership to the system board microprocessor by deasserting HOLD.
9. The microprocessor reconnects itself to the buses and deasserts HLDA to indicate that it is bus master again. Normal operation (program execution) then resumes and continues until another fifteen microseconds has elapsed. At that time, the refresh timer asserts REFRESH TRIGGER again to request another refresh bus cycle.
10. When the refresh logic has once again been granted bus mastership, it again outputs the contents of the refresh row counter (incremented by one during the previous refresh bus cycle) onto the portion of the address bus that supplies the row address to the DRAMs.

In this way, the next sequential row in every DRAM chip in the system is refreshed. This action continues once every fifteen microseconds as long as the system is powered on. When the last row within the DRAM chips has been refreshed, the row counter rolls over to all zeros again and the process starts over at the first row in the DRAM chips.

The actions described earlier in step 5 are for what's called RAS-only refresh. There are three other ways that the DRAM addressing logic can handle a refresh bus cycle: CAS-before-RAS refresh, hidden refresh and self refresh.

CAS-before-RAS Refresh

A DRAM chip capable of doing CAS-before-RAS refresh (CBR refresh) has an internal refresh row counter, so it does not need to receive a row address from the refresh logic. To refresh a CBR refreshed DRAM, the DRAM controller asserts CAS# first, then RAS#, without regard to the row or column address on the memory address bus. The DRAM chip responds by refreshing the row pointed to by its internal refresh row counter and then incrementing that row counter.

Hidden Refresh

Hidden refresh is just a variation of CBR refresh. Normally, an access to a DRAM chip is terminated by deasserting both RAS# and CAS#. When CAS# is deasserted at the end of a read cycle, the DRAM chip stops driving the data out on its data pin. However, if CAS# is left asserted when RAS# is deasserted at the end of the access, then a DRAM chip capable of hidden refresh will get ready to refresh the row pointed to by its refresh row counter. The DRAM chip will actually do the refresh if RAS# becomes asserted again while CAS# is still asserted. Simultaneously, the DRAM chip will continue to drive the data from the previously accessed bit cell onto the data bus, as long as CAS# remains asserted. The DRAM is thus providing data and doing refresh simultaneously, thus the refresh appears to be hidden.

Self-Refresh

Each of the preceding types of refresh requires intervention by the system board to tell the DRAM chip to do the refresh. Self-refreshing DRAMs are different. They have the ability to do refreshes based on their internal refresh row counters and based on their own internal refresh timer, without waiting for a signal from the system board. Self-refreshing DRAMs are used in lieu of static RAM for low power, battery-backed storage.

To understand self-refreshing DRAM, it may help to understand what the alternatives are for battery-backed storage. Often, SRAM or a variation of ROM called non-volatile RAM (NVRAM) are used to store critical data immediately during a power failure. SRAM is well suited to this task due to its infinitesimally small current drain when it is not being accessed. The contents of the SRAM can be maintained by a small battery. NVRAM requires no power at all to maintain its contents since it is actually SRAM and EEPROM put together (EEPROM is described in the chapter entitled "ROM Memory"). Neither the SRAM nor the NVRAM need to be refreshed, so no logic has to be running on

the system board to maintain the SRAM or NVRAM contents while the main power supply is off.

Standard DRAM requires refresh logic to be running, and it is thus not as well suited for battery-backed applications as SRAM due to this complexity and due to the power drain caused by the standard refresh logic. Self-refreshing DRAM eliminates this complexity and is thus competitive with SRAM for battery-backed use.

Self-refreshing DRAM initiates its refresh operation when it sees CAS# asserted followed by RAS#, just as with a CBR-refresh DRAM. If CAS# and RAS# both remain asserted for a long time (like a dog's life, a long time is relative), typically 100 microseconds (forever in computer time), then the DRAM's internal timer takes over and begins self-refresh. The self-refreshing DRAM continues to refresh itself until it sees RAS# and CAS# deasserted.

As noted above, RAS-only refresh, CBR-refresh and hidden refresh require a REFRESH TRIGGER from the system board. In an ISA system, timer #1 provides that trigger. The refresh interval of timer #1 is programmable, allowing a programmer to change the refresh interval to something other than fifteen microseconds. This practice is dangerous, however, because alteration of the refresh interval raises the possibility that some DRAM memory locations will have entirely discharged by the time the refresh logic gets around to refreshing their respective rows again. In other words, data in DRAM memory may be lost.

It should be noted that a refresh bus cycle may also be triggered by an ISA bus master. In the event that an ISA bus master has control of the buses for a period longer than 15 microseconds, it can trigger a refresh cycle by asserting the REFRESH# signal. When the refresh controller detects that the REFRESH# signal is asserted, it will run a refresh cycle. During the refresh cycle the ISA bus master will not use the buses. See the chapter entitled "ISA Bus Masters" for more details.

Destructive Read: Pre-Charge Delay and Cycle Time

When a DRAM location is read, the DRAM must have some way of determining if the location contains a charge (if it contains a one or a zero). The only way it can ascertain the presence of a charge is to discharge the bit cell. In other words, DRAM chips use a destructive read process. It must discharge the capacitor used to store a bit in order to determine if there was in fact a bit stored in that location. A storage device that can only be read one time would be useless.

Every DRAM chip contains logic that recharges a DRAM location back to a good level after the destructive read takes place. The location is only recharged if it is determined that it originally held a charge. It takes an appreciable amount of time to recharge a DRAM location back to a good level again. This period of time is known as the pre-charge delay.

Refer to figure 13-7. This timing diagram illustrates the pre-charge delay immediately following the access time. If a bus master attempts to read data from any memory location during the pre-charge period, the data received may very well be garbage (because the DRAM logic is tied up recharging the row last read from).

To ensure that the data read from the DRAM is good data, the hardware should not allow a bus master to access the DRAM again until the pre-charge delay has elapsed. The bus master can be held off by keeping its READY# line deasserted until both the access time and pre-charge delay has elapsed. When READY# is then asserted, the bus master is allowed to complete the first transfer and can immediately and safely initiate a subsequent transfer.

When a DRAM chip manufacturer specifies the access time on a DRAM (for example, 60ns, 70ns, 80ns or 100ns access time), it doesn't paint an entirely accurate picture. Although the first access to a 60ns DRAM chip can be safely accomplished in 60ns, the hardware must delay initiation of a subsequent access until the DRAM's pre-charge delay has elapsed. Since the pre-charge delay is typically equal to the access time for the DRAM, the access time should be doubled to ascertain how often the DRAM chip may be safely accessed. As an example, a DRAM chip with a rated access time of 60ns could only be accessed safely once every 120ns (60ns access time + 60ns pre-charge delay). The overall period of time consisting of the access time and the pre-charge delay is known as the cycle time, and this factor should be considered when designing a memory system using DRAM chips.

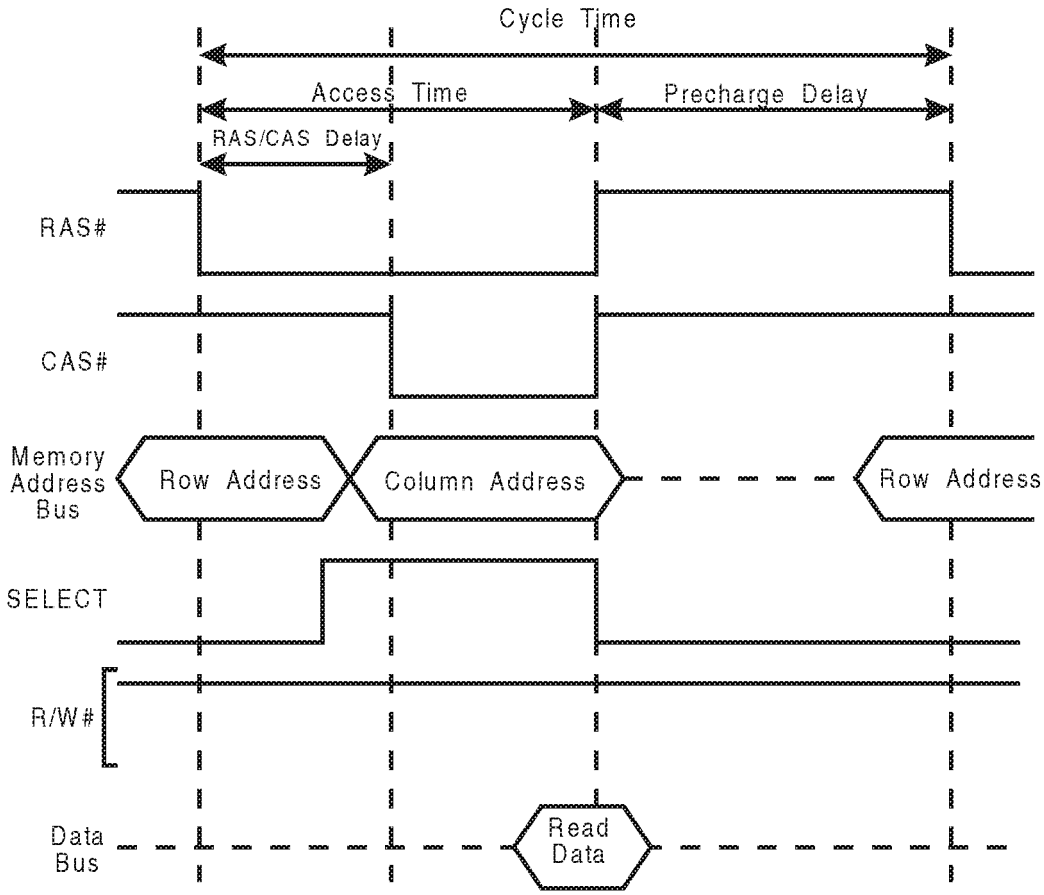


Figure 13-7. Pre-Charge Delay and Cycle Time

DRAM Bank

Refer to figure 13-8, the DRAM bank. The typical DRAM chip used in personal computers stores one bit of information per location. Since the current bus master expects to be able to read or write at least one byte of information at a time, the ability to store and retrieve one bit per location appears relatively useless.

To create a usable design, the engineer combines a series of eight DRAM chips so that, taken collectively, they can store one byte of information per location. This is referred to as a bank of DRAM. The designer connects the memory address bus in parallel to all eight DRAM chips. The RAS# and CAS# lines are

also connected to all eight DRAM chips, as well as the read/write control line. This give the bus master the ability to address the exact same location in all eight DRAM chips simultaneously.

The designer connects each individual data bus line to the data input/output pin on a separate DRAM chip within the bank of DRAM.

When the bus master performs a memory write operation and places the address on the address bus, the DRAM addressing logic performs the necessary breakdown of the address into the row and column addresses and selects the exact same row and column within each of the eight DRAM chips within the bank. The read/write pin on the DRAM chips is set to the write state and each of the eight data bits that make up the byte to be written into the memory location is presented to its respective DRAM chip within the bank. The eight data bits are then written into exactly the same location within the eight individual DRAM chips.

When the bus master performs a memory read operation to read a byte of information from DRAM, it simultaneously addresses precisely the same row and column location within each of the eight DRAM chips. The bit stored in that location within each of the eight DRAM chips is driven out onto the respective data bus line by the individual DRAM chip. From the bus master's perspective, it's communicating with DRAM memory that can store one byte of information per location.

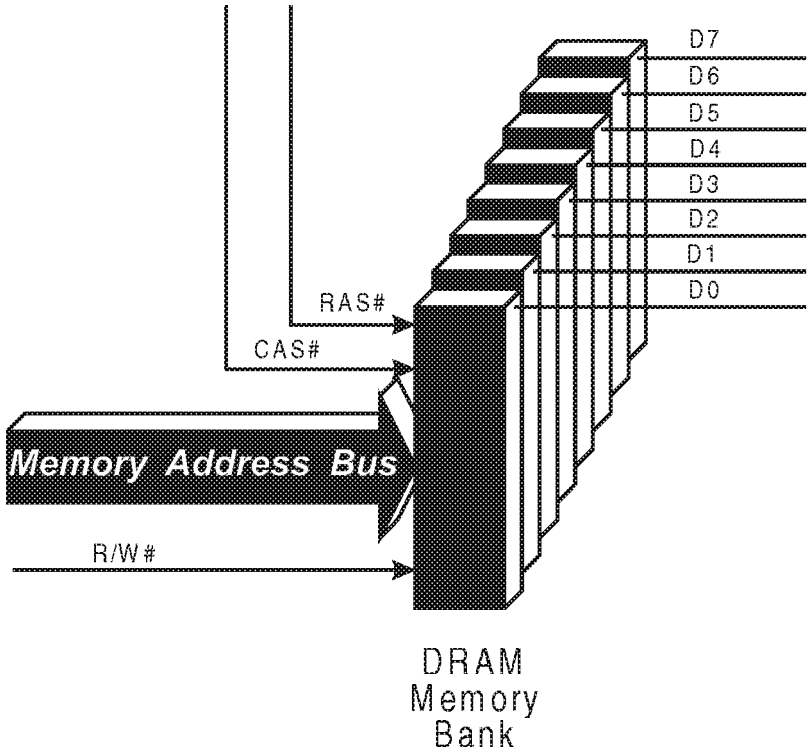


Figure 13-8. The DRAM Bank

DRAM Bank Width

The original IBM PC and PC/XT were based on the Intel 8088 microprocessor, which had an 8-bit data bus. A bank of DRAM in those machines consisted of eight DRAM chips plus a parity bit (parity is discussed later in this chapter).

Refer to figure 13-9. The IBM PC/AT (or any PC based on the 80286 or 80386SX microprocessors) has a 16-bit data bus, so a bank of DRAM consists of 16 DRAM chips, divided into two halves. All even-addressed locations are located in the lower half (connected to the lower data path, D[7:0]), while all odd-addressed locations are in the upper half (connected to the upper data path, D[15:8]). Each half consists of eight data DRAMs and a parity DRAM.

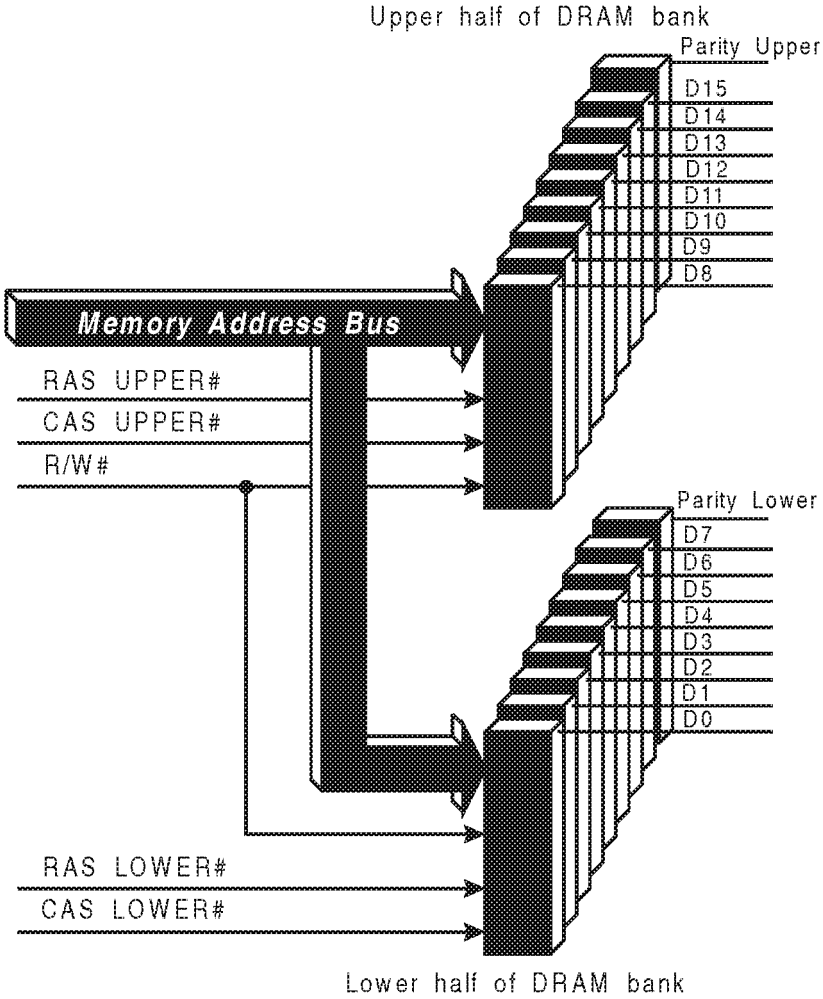


Figure 13-9. 16-Bit DRAM Bank

Refer to figure 13-10. Systems based on the 80386DX or 80486 microprocessor have a 32-bit data bus, so each bank of DRAM consists of thirty-two data DRAM chips, plus four parity DRAMs.

In other words, a bank of DRAM is sized to match the width of the data bus so the microprocessor can transfer the maximum amount of information per bus cycle.

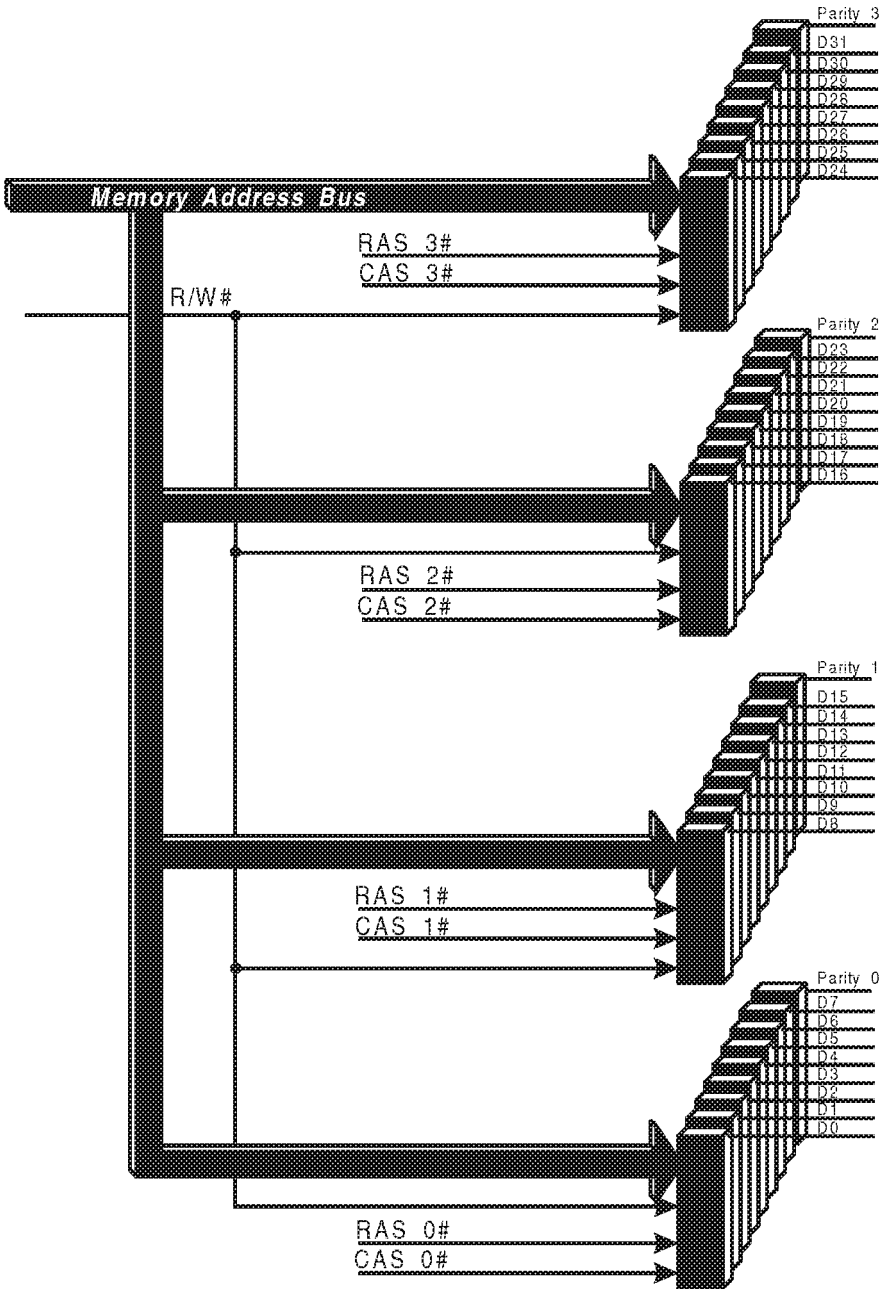


Figure 13-10. 32-Bit DRAM Bank

DRAM Error Detection and Correction

The tiny capacitors used to store data in DRAM are susceptible to soft errors. Soft errors are data bits that were initially stored correctly but which have subsequently lost their charge due to electrical and cosmic interference or due to improper refresh. It's possible for the bus master to read incorrect data or instructions from DRAM that has suffered a soft error. If this happens, the bus master may execute the wrong instruction or make a bad decision on data without realizing that an error has occurred.

Crosstalk and noise on the data bus can cause data that was stored and retrieved correctly by the DRAM chip to become garbled between the DRAM chip and the bus master. This isn't necessarily the fault of the DRAM chip. As it is with soft errors, the bus master may execute the wrong instruction or make a bad decision on data.

Parity memory and error-checking-and-correcting memory are two approaches to solving these problems.

DRAM Parity

The DRAM memory subsystem used in most ISA products incorporates parity generation and checking logic. Consider the DRAM bank pictured in figure 13-11. This bank of DRAM consists of eight DRAM chips and an additional, ninth DRAM chip referred to as the parity DRAM. All of the address lines, as well as the RAS# and CAS# lines and the read/write control line are connected in parallel to all nine DRAM chips. The eight data lines are connected to the eight data DRAM chips. The data bus is not only connected to the data DRAM chips, but also to the parity generator/checker chip.

When the bus master writes a byte of information to the DRAM chips, it is presented to the DRAM chips and to the parity generator/checker. During a memory write operation, parity is generated and written into the parity DRAM chip. Parity is not checked for correctness during memory write operations, only during memory reads (as described further on).

When the data byte is presented to the parity generator/checker chip during a write operation, it checks the number of one bits in the data byte. If the parity generator is using "even parity" and the number of one bits in the data byte is even, then the parity generator places a zero on the PARITY line going to the parity DRAM chip. The parity bit and the data byte are simultaneously written

into the same address within the parity DRAM and the data DRAM. That means that a total of nine bits of information is written into the addressed memory location during a write operation.

If the parity generator is using "even parity" and it had not detected an even number of one bits in the data byte being written to the data DRAM, it would place a one on the PARITY line to the parity DRAM. The total number of one bits in the 9-bit pattern written into the addressed memory location is therefore always an even number of one bits. In other words, the parity generator's job is to force the 9-bit pattern written into memory to always consist of an even number of one bits.

When the data is later read back from memory, the nine bits of information are driven out onto the data bus and the parity line by the data and parity DRAMs. The data byte is sent back to the bus master over the data bus and is also presented to the parity generator/checker, now acting as a parity checker, rather than a parity generator. In addition to the 8-bit data byte being presented to the parity checker, the parity bit is also being presented to it. The parity checker examines the 9-bit pattern and checks it to see if it contains an even number of one bits (which it always should if even parity is being used).

If the 9-bit pattern doesn't contain an even number of one bits, an error has occurred and the parity check logic generates a parity error. The parity error causes a non-maskable interrupt (NMI) request to the microprocessor, and the microprocessor is forced to jump to the NMI interrupt service routine. In this routine, the programmer performs an I/O read from system control B at I/O address 0061h to ascertain the cause of the NMI. If bit 7 is set to a 1, a parity check was encountered while reading from system board DRAM memory. The NMI routine then reports a DRAM parity error on the screen and halts the system. The parity error is not recoverable because the parity generator/checker can't correct the parity error. It only knows that one or more bits in a byte are bad, but it can't determine which bit is bad. Nevertheless, it is better for the system to halt before further (possibly worse) damage is done.

The preceding discussion assumes that the parity generator/checker is using even parity. If, instead, the parity generator/checker is using odd parity, then it will force the 9-bit pattern written into memory to always consist of an odd number of one bits. Likewise, on a read bus cycle, the parity checker examines the 9-bit pattern and checks it to see if it contains an odd number of one bits.

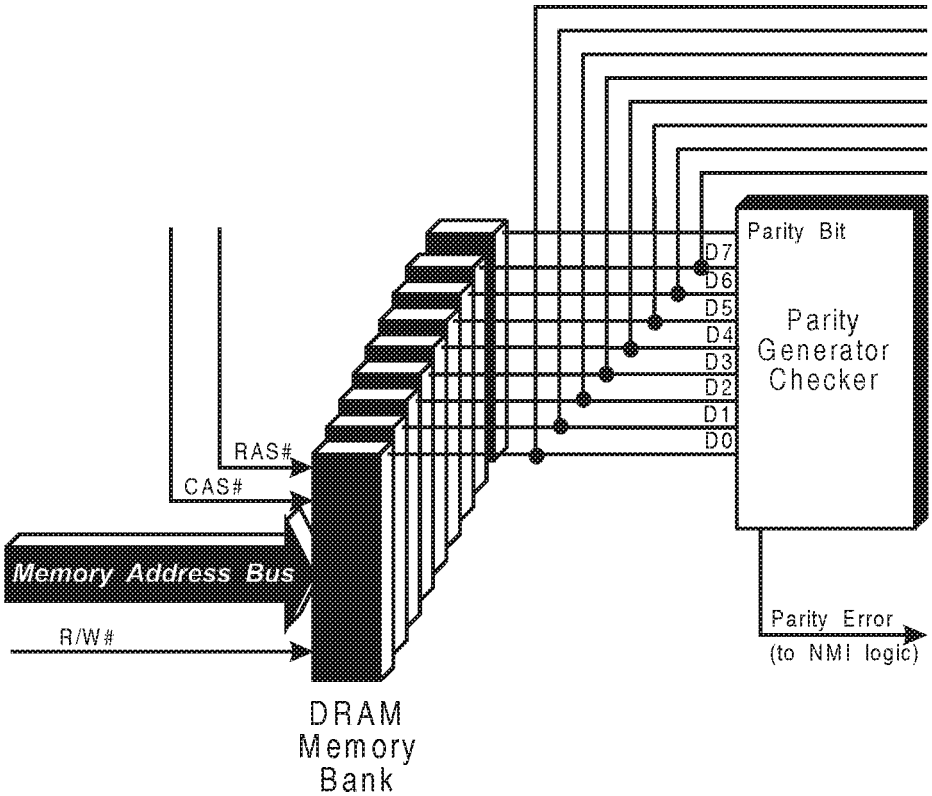


Figure 13-11. The System Board DRAM Parity Logic

Since parity memory systems have 9 bits for every 8-bit byte of data, they require one-eighth or about 13% more memory (and cost) than non-parity memory systems. Since the amount of DRAM in modern PC's is much greater than it was in the original PC (about 100 to 2,000 times more), system designers must consider the cost of parity versus its error catching benefit. Modern DRAMs protect the capacitors in a semiconductor "well" which deflects the radiation that caused soft errors in older DRAMs. Also, improved board manufacturing and component handling techniques have reduced the manufacturing-induced failure rates on DRAMs. As a result, many systems do not use parity memory. But then, consider that PCs are depended on more heavily today than they were over a decade ago and there is a lot more memory being used.

Error-Checking-and-Correcting Memory

On very high-end PCs such as file servers for reservation systems ("mission critical" systems), error detection and correction is vital. Users of these systems do not want the system to crash when a soft error occurs. These systems may have hundreds of megabytes of memory in them, so the chances for a soft error are greater than on a typical PC.

Since the parity generator/checker can only detect errors and cannot correct them, the mission critical systems use error-checking-and-correcting (ECC) memory. ECC memory may be wider than 9 bits. When the bus master writes data to ECC memory, the ECC memory controller encodes the bytes before storing them. There are more bits in the ECC memory than there are in the original data byte, so some of the possible bit patterns that could exist in ECC memory do not correspond to data bytes. Those bit patterns are invalid. If the ECC memory controller subsequently reads one of these invalid patterns from the ECC memory, it knows that an error has occurred because it did not write that pattern into memory. The ECC memory controller can then usually correct the error by finding the valid bit pattern that it "closest" to the invalid bit pattern.

Page-Mode DRAM and Its Variations

Some ISA systems incorporate page mode DRAM in one of its various forms. The following sections explain the advantages of page-mode, enhanced page-mode, burst-mode (nibble-mode), static column page-mode memory and synchronous DRAM.

Page Mode DRAM

Page mode DRAM is just a variation on standard DRAM memory. In most respects, its operation is identical. As with standard DRAM, sequential page mode DRAM locations are laid out or distributed sequentially along a row. In other words, location 0 in a page mode DRAM chip is located in row 0, column 0; location 1 is located in row 0, column 1; location 2 at row 0, column 2, etc.

If the currently executing program is performing sequential memory accesses, sequential locations within a row are accessed. Since the majority of program execution is sequential in nature (in-line code fetches), program execution very often proceeds along a row of memory. At the end of a row in a page mode

DRAM chip, the next sequential location is in column 0 at the beginning of the next row. Subsequent locations are located along that row.

As an example, a 1Mbit (1 megabit) page mode DRAM chip consists of a 1,024-by-1,024 row/column matrix. This means that each row contains 1,024 locations. The microprocessor could perform 1,024 memory accesses within the row before having to cross over into the next row. Page mode DRAM is called by that name because the designer considered one row of RAM memory to be a page of DRAM memory.

In a standard DRAM chip, a memory location is addressed in the following fashion:

1. The row address is placed on the memory address bus.
2. The RAS# line is asserted.
3. RAS/CAS delay to ensure proper operation.
4. The column address is placed on the memory address bus.
5. The CAS# line is asserted.

At that point, the DRAM chip has the row and the column address and can therefore identify the addressed memory location. The data transfer can be performed.

Refer to figure 13-12. To take full advantage of the operational characteristics of page mode DRAM chips, some external support logic is necessary. This consists of a row latch and a row comparator.

The first access to a page mode DRAM is identical to that for a standard DRAM. The row address is placed on the memory address bus and RAS# is asserted. When RAS# is asserted in a page mode DRAM system, two things occur:

- The DRAM chips latch the row address.
- The row address is latched into the row latch that is external to the DRAM chips.

The output of the row latch is applied to one side of the row comparator. After the RAS/CAS delay has elapsed, the DRAM addressing logic places the column address on the memory address bus and asserts CAS#. This causes the column address to be latched by the DRAM. Up to this point, the page mode DRAM addressing scheme is identical to that for standard DRAM.

One other ingredient that is necessary for the page mode DRAM architecture to work efficiently is the address pipelining capability of the 80286, 80386 and Pentium microprocessors. These microprocessors have the ability to place the address for the upcoming bus cycle on the address bus during the current bus cycle.

The row comparator compares the row address currently held by the row latch to the portion of the address bus that contains the row address for the next upcoming DRAM access. If equal, it means that the upcoming DRAM access is going to be within the same row (or page) within the page mode DRAM chips. This is called a page hit. When the row comparator detects that they are equal, it informs the DRAM addressing logic. The DRAM addressing logic responds by keeping RAS# asserted at the end of the current bus cycle.

The continued assertion of RAS# tells the page mode DRAM chips to remember the previously latched row address because the upcoming access to going to be within the same row. Since the page mode DRAM chip already has the row address, the new column address can be immediately gated through the DRAM addressing MUX onto the memory address bus and CAS# can be asserted. This commands the DRAM chips to latch the new column address.

When sequential accesses fall within the same row of page mode DRAM memory, it isn't necessary to wait for the RAS/CAS delay to elapse before giving the DRAM chip the next column address. Since the RAS/CAS delay is approximately 50% of the overall access time for the DRAM chips, the access time can effectively be cut in half by eliminating this delay.

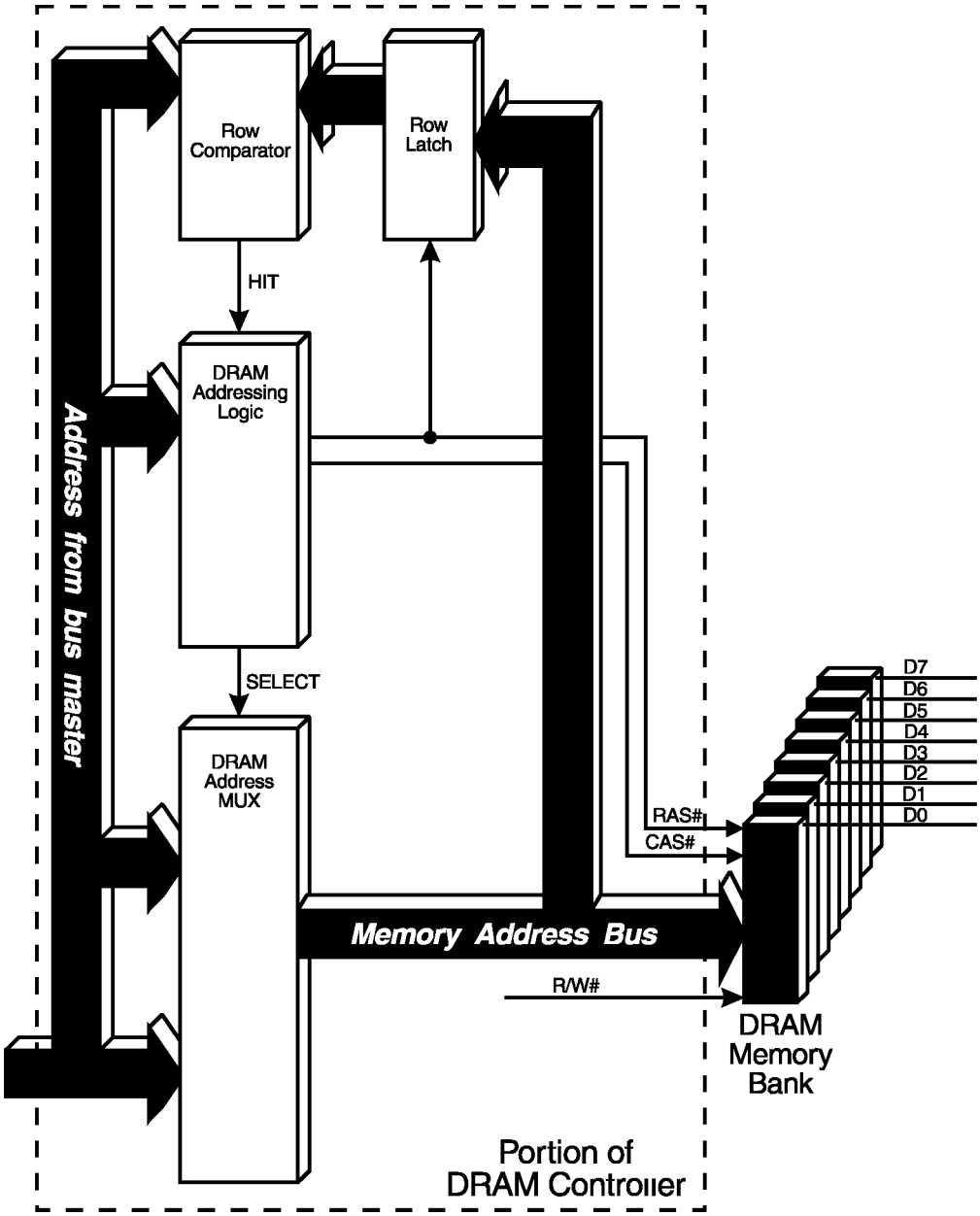


Figure 13-12. Page Mode DRAM Support Logic

Refer to figure 13-13. As an example, page mode DRAM chips rated at 100ns per access would require 100ns for the first access, but subsequent sequential accesses within the same row could be performed in approximately 50ns, substantially improving the transfer rate. As long as subsequent, back-to-back memory reads are within the same row, the RAS# line stays asserted throughout multiple accesses, while new column addresses appear on the memory address bus and CAS# is asserted once for each new column address.

When the row comparator detects that the upcoming bus cycle will access a row other than the one internally latched within the DRAM chips, it must instruct the DRAM addressing logic that a miss has occurred and the page mode DRAM chips must therefore forget the currently latched row address. This is accomplished by deasserting RAS#. This is called a page miss.

The next access is exactly the same as a standard DRAM access. The new row address must be sent to the DRAMs and latched, followed by the RAS/CAS delay and the latching of the column address. If, however, subsequent accesses past that point are within the same row, then the memory logic can cut the access time in half by doing away with the RAS/CAS delay again.

Page mode DRAM takes advantage of one very important characteristic of any microprocessor. What is the microprocessor always doing on an ongoing basis? The answer is fetching instructions. Since the majority of program execution is sequential in nature (in-line code fetches), program execution very often proceeds along a row of memory. A series of memory reads from locations located within the same row can take place quickly because the DRAM access time is cut approximately in half by keeping RAS# asserted thus eliminating the RAS/CAS delay.

Anything that can be done to increase the rate at which instructions are sent to the microprocessor has a substantial impact on the overall performance of a PC. In addition, any program that tends to access large blocks of data from page mode DRAM memory sequentially will take advantage of this architecture as well.

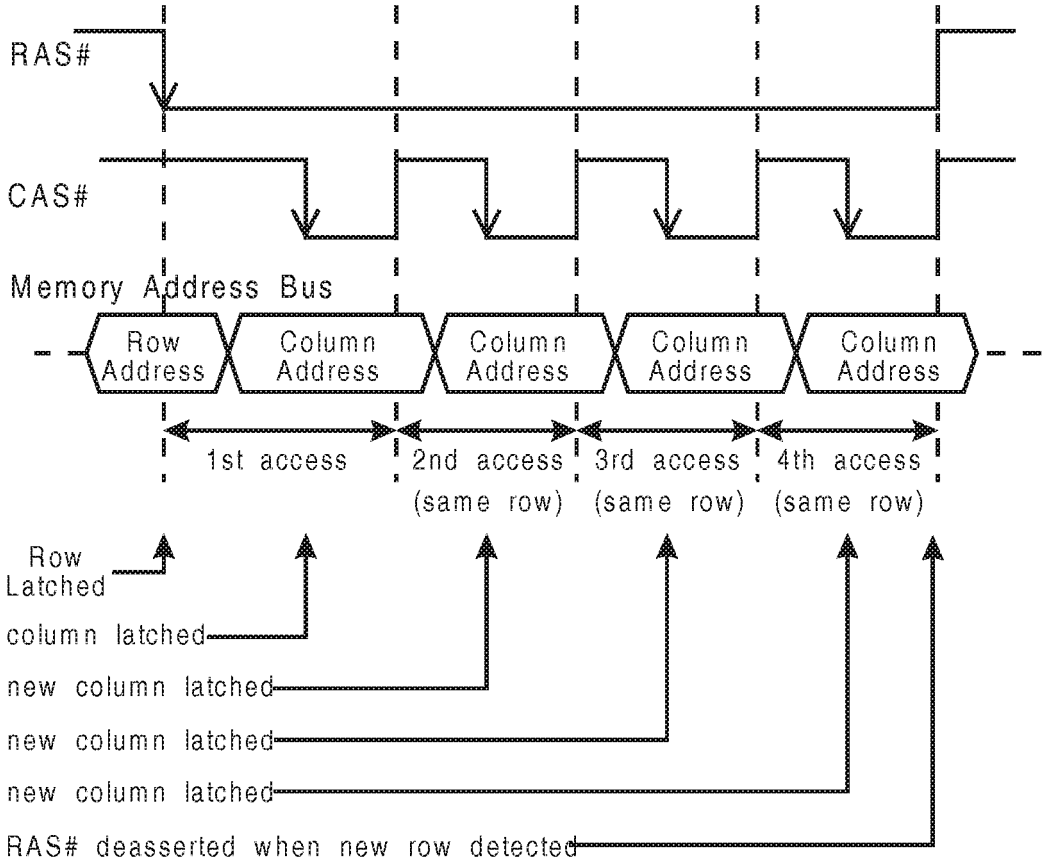


Figure 13-13. Page Mode DRAM RAS/CAS Timing

Enhanced Page Mode DRAM

Enhanced page mode DRAM provides faster access time than standard page mode DRAM by beginning the column address decode as soon as a new column address appears on the memory address bus, even before CAS# is asserted. The column address latch is a transparent latch, so that its input (the column address) is immediately available to its output (the column address decoder) while CAS# is deasserted. Since the column address decoder starts its decode early, the DRAM's access time is improved. As with standard page mode DRAM, when CAS# is subsequently asserted, the column address latch latches the column address.

Burst and Nibble Mode DRAM

Burst mode and nibble mode DRAMs are designed to provide optimum performance when the cache is reading or writing lines of information from main RAM. The next chapter describes caches and defines cache lines in detail, but briefly, a cache line is the minimum information that the cache can store. Cache lines are typically many bytes in length, requiring that the cache controller perform multiple accesses to main DRAM in order to read an entire cache line. Each of those cache accesses are to DRAM locations that are adjacent to one another. In other words, the DRAM locations are contiguous.

Normally, an x86 microprocessor bus cycle consists of one address time (Ts or T1) followed by one or more data times or wait states (Tc or T2). The maximum amount of information that can be transferred during the bus cycle is limited to the width of the microprocessor's data bus. For example, if the cache controller needed 16 bytes of information (so the cache line size is 16 bytes) and if the microprocessor's data bus were four bytes wide, then the microprocessor would have to run four bus cycles to fill a cache line. Each of those bus cycles would have an address time. But since all 16 of the bytes are contiguous with one another, some time could be saved by not broadcasting the addresses of all of the bytes. In a burst bus cycle, as used by the 80486 microprocessor, the starting address of the cache line is broadcast during the address time (T1) and 16 bytes come back, a doubleword at a time, in the following data times, without any further T1 times. (The memory subsystem still has the ability to insert wait states in the transaction.) The memory subsystem must have its own address counter internally so that it can calculate the address of the second and subsequent doublewords that the cache is requesting. The Pentium processor uses a burst bus cycle as well, but its cache line size is larger than the 80486's.

Burst and nibble mode DRAM take advantage of the fact that the cache accesses several contiguous locations in rapid succession. Refer to figure 13-14. As with standard DRAM, burst and nibble mode DRAM accept a row address and a starting column address from the DRAM controller. Burst and nibble mode DRAM have internal column counters, so they do not need to see new column addresses during the burst. This eliminates the setup and hold time for the second and subsequent column addresses and thus improves the DRAM access time when compared to standard page mode DRAM. CAS# is toggled to cause the DRAM to increment its internal column counter and to access that column.

The second and subsequent column addresses that the DRAM generates internally must match the column portion of the address that the cache controller is

producing. The burst mode DRAM must be designed to work with the cache controller or the burst mode DRAM must be programmed for a column address sequence that matches the sequence used by the cache controller. The length of the burst is cache-dependent and the burst mode DRAM must likewise be designed or programmed to generate as many column addresses as the cache controller does.

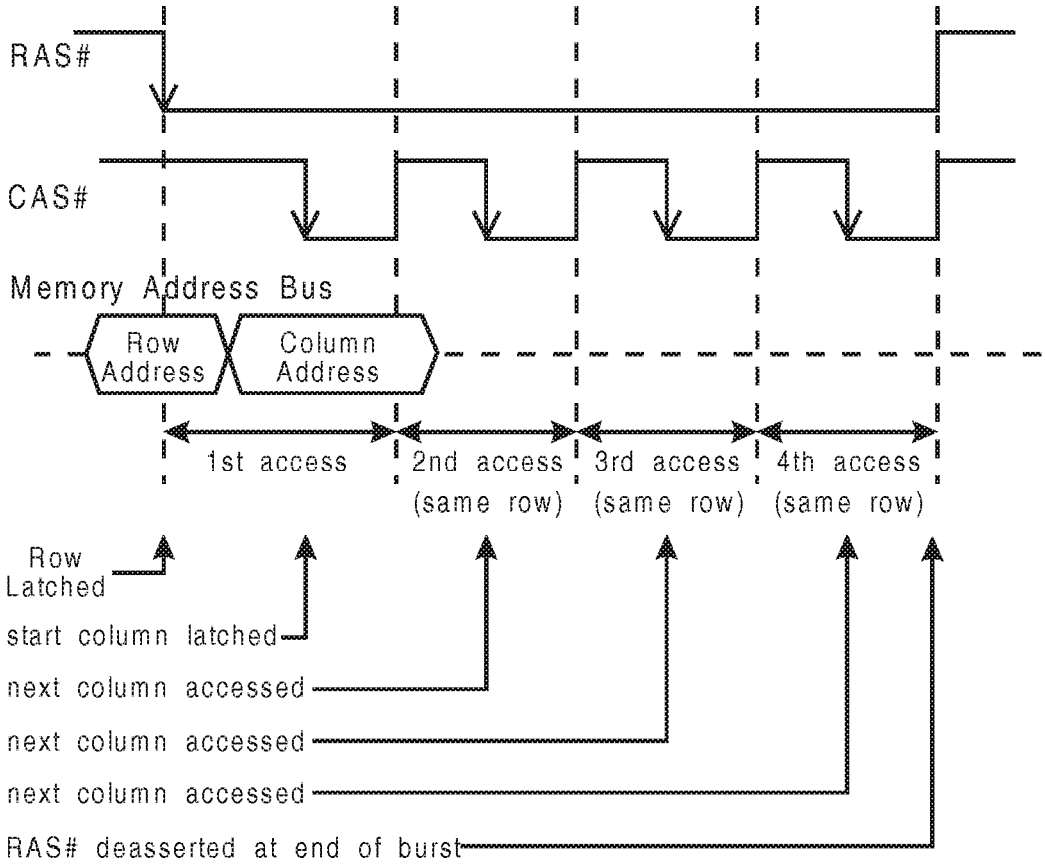


Figure 13-14. Burst Mode Access Timing

Notice that burst mode and nibble mode DRAM lack the ability to do random accesses along the current row. Standard page mode DRAM can provide information from any column in the currently selected row, but burst and nibble

mode DRAM can only provide information from the contiguous columns that were selected with the starting column number.

The access time improvement in burst and nibble mode DRAM comes from saving the setup and hold time on the second and subsequent column addresses, at the expense of no longer being able to do random page mode accesses along the currently selected row. The next section describes static column page mode DRAM, which retains the ability to do random accesses along the same row and has access times similar to burst mode DRAM.

Static Column RAM (SCRAM)

Static column RAM, or SCRAM, probably has one of the most misleading names in the industry. Many people believe that static column RAM is a form of static RAM. It is not. Static RAM is discussed later in this chapter. Static column RAM is another form of DRAM memory. It is extremely similar to page mode DRAM, so much so that they are frequently mixed up in trade articles. Just as with page mode DRAM, SCRAM is structured with sequential memory locations distributed along a row. SCRAM also requires the same external support logic as page mode RAM. That is, it requires a row latch and a row comparator and takes advantage of address pipelining.

The only difference between the two technologies lies in how SCRAM recognizes that a new column address has been presented to it. Whereas page mode DRAM requires that CAS# be asserted each time a new column address is presented to it, SCRAM is slightly more intelligent.

Refer to figure 13-15. When the SCRAM chip detects RAS# and CAS# remaining asserted from one access to another, it recognizes that it will not be receiving a new row address, only a column address. It then observes the memory address bus. When it sees the value on the address lines begin to change, it interprets this as a new column address appearing and waits a predetermined amount of time for the column address to stabilize. It then generates its own CAS internally, latching the new column address. The name static column RAM stems from the fact that the column address strobe line, CAS#, stays static asserted for multiple accesses within the same row.

It is possible to design a memory subsystem that accommodates either SCRAM or page mode DRAM chips. This is possible because, although the SCRAM chip doesn't need to have CAS# reasserted each time a new column address is presented to it, it still operates properly if CAS# is reasserted. In other words, if a

memory subsystem is designed for page mode DRAM, it would also work with SCRAM.

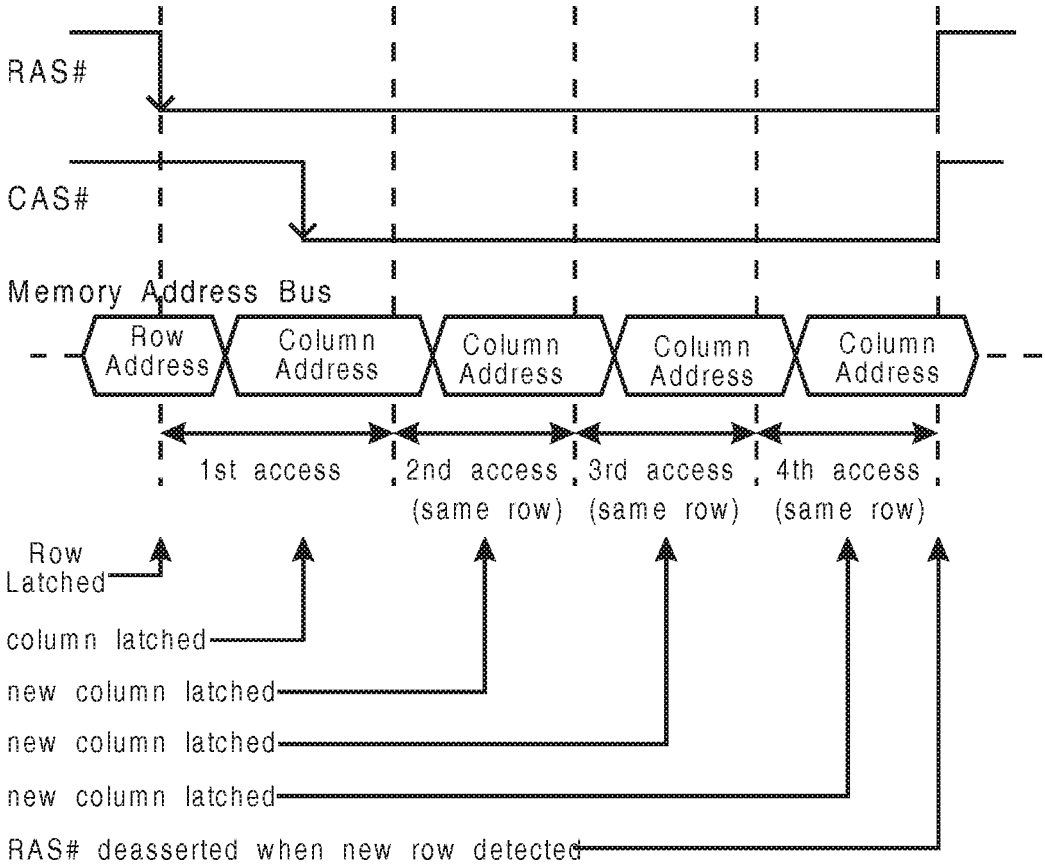


Figure 13-15. SCRAM Access Timing

Synchronous DRAM

Synchronous DRAM isn't a type of DRAM by itself. Instead, it is a way that the DRAM chip interfaces its internal storage array to its pins. The preceding sections described asynchronous DRAM interfaces which do not require a clock. Synchronous interfaces are based on a clock so that the memory address and control signals (RAS# and CAS#) are only valid at appropriate edges of the clock.

Synchronous DRAM interfaces can simplify the design of the interface between the DRAM chips and the microprocessor because the microprocessor is a synchronous (clocked) device. To achieve optimum performance with the microprocessor, the DRAM must be synchronized to the microprocessor, and thus to the microprocessor's clock. For asynchronous DRAM, the system designer must provide logic external to the DRAM to synchronize it to the microprocessor. Synchronous DRAM can be run from the same clock that runs the microprocessor, thus reducing the amount and complexity of the external logic.

Synchronous DRAM interfaces are often used on burst mode DRAM.

Interleaved Memory Architecture

Interleaved memory architecture takes advantage of the sequential nature of program execution. The problem addressed by the interleaved memory architecture is the period of time the microprocessor is forced to wait when performing back-to-back accesses to the same DRAM chip. This wait is necessary because the DRAM chip is charging back up again after a destructive read. This is referred to as the pre-charge delay.

When an interleaved memory architecture is used, what would ordinarily be one bank of DRAM memory is split up into two banks. As an example, refer to figure 13-16. In an 80386DX- or 80486-based system, the microprocessor is capable of fetching a doubleword (four bytes) per bus cycle. The memory address decode logic is designed so that the doubleword consisting of the first four memory locations, locations 00000000h through 00000003h, is physically located in DRAM bank A. The second doubleword, consisting of locations 00000004h through 00000007h, is physically located in DRAM bank B. The next doubleword is physically located in DRAM bank A and consists of locations 00000008h through 0000000Bh, while the following doubleword is physically located in DRAM bank B and consists of locations 0000000Ch through 0000000Fh, and so on.

Assume that the 80386DX microprocessor has just fetched an instruction from the doubleword starting at location 00000000h in DRAM bank A. That means that the microprocessor will more than likely fetch the next instruction from the doubleword starting at memory address 00000004h in DRAM bank B. While the microprocessor is performing the second instruction prefetch from the doubleword in bank B, the DRAMs that make up bank A are going through precharge delay and charging back up again.

When the microprocessor completes the fetch from the doubleword in bank B, the next instruction fetch is more than likely going to be from the next doubleword in bank A starting at location 00000008h. While the microprocessor is fetching the third doubleword from bank A, the DRAMs that make up bank B will be charging back up again.

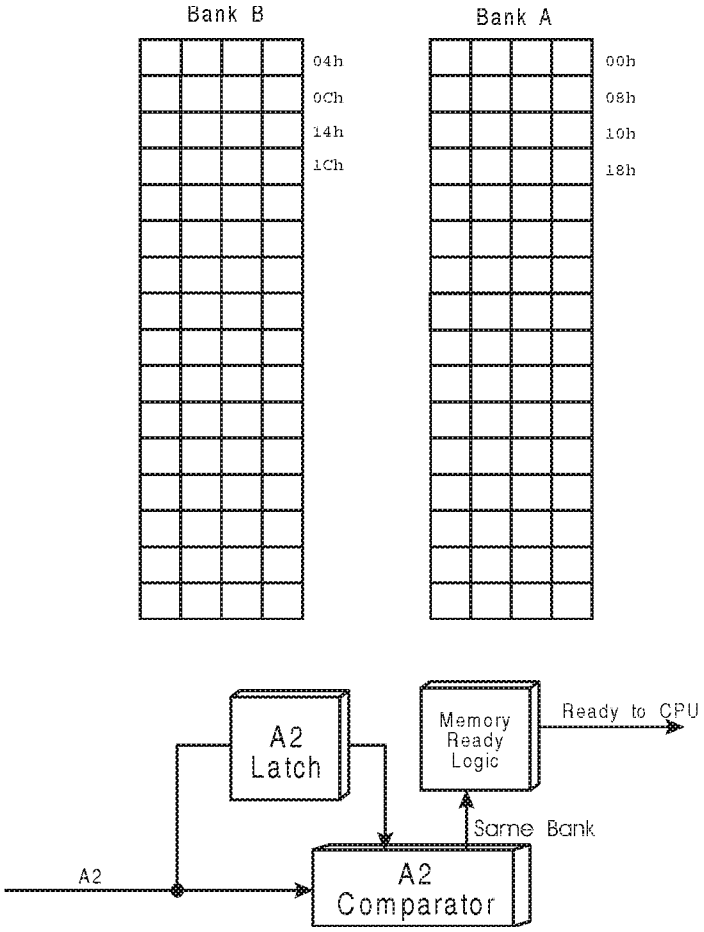


Figure 13-16. Interleaved Memory Architecture

The microprocessor is able to perform back-to-back read operations (in this case, instruction fetches) without waiting for the pre-charge delay. The only times that the microprocessor will have to wait for the pre-charge delay to elapse will be when the next memory access is within the same bank as the

previous access. This occurs, for instance, when a jump instruction instructs the microprocessor to fetch its next instruction from a location residing in the same bank that the previous instruction was read from.

It also occurs when the currently executing instruction instructs the microprocessor to access a memory location within the same bank as the previous access.

Refer to figure 13-16. Support logic within the interleaved memory subsystem keeps track of which bank was accessed last. Using address pipelining, the subsystem can detect that the upcoming access is from the same bank or the other bank. If from the other bank, then the memory logic doesn't have to allow for the pre-charge delay and can let the microprocessor access it immediately. Address bit A2 indicates which bank is to be accessed in the upcoming bus cycle. The support logic remembers which DRAM bank is currently being accessed via the A2 Latch. The A2 Comparator compares this to the state of A2 for the upcoming access (using address pipelining). The timeout value for the Memory Ready Logic is adjusted accordingly.

On the other hand, if the upcoming access is within the same bank, the memory logic keeps the READY# line deasserted until the pre-charge delay has elapsed. The memory logic then asserts READY#, letting the current bus cycle end and letting the microprocessor safely begin the next bus cycle within the same bank.

If page mode DRAM is used in the interleaved banks, then the pre-charge delay may be avoided even if the current bus master accesses the same bank consecutively. Of course, accesses to the same bank would have to be to the same row in order to avoid the pre-charge delay. By combining page mode DRAM and interleaving, the system designer can eliminate most of the pre-charge delays.

Static RAM (SRAM)

Unlike DRAM, static RAM, or SRAM, memory doesn't require refreshing at regular intervals in order to maintain the information stored in memory. This is because, unlike a DRAM bit cell which uses a capacitor and a transistor to store information, SRAM uses single-bit latches known as D flip-flops. A typical SRAM bit cell consists of four or six transistors. A four transistor design costs less and a six transistor design consumes less power.

Once a bit of information has been written into one of these latches (or flip-flops), the SRAM chip continues to hold the data as long as power is applied to the SRAM chip. The information doesn't degrade (discharge) over a period of

time as it does with DRAM. SRAM is called static because, once the information is written into it, it remains static, or steady, without being refreshed at regular intervals.

Another important characteristic of SRAM is its relatively short access time. This is mainly due to the fact that, rather than providing it the address in two steps (as with DRAMs), the SRAM receives the entire address at once. Once chip-selected, the SRAM uses the address on its address inputs to identify the addressed memory location and the data transfer takes place.

Although SRAM is typically much faster than DRAM memory, it is cost-prohibitive to populate an entire system with it. SRAM chips typically cost ten times more than DRAMs, are physically larger, consume more power and generate more heat. All of these factors, taken collectively, substantially add to the overall cost of a system. In the faster 80386, 80486 and Pentium processor-based systems, limited amounts of high-speed SRAM are used as cache memory to improve the microprocessor's overall transfer rate when addressing memory. This subject is covered in the chapter entitled "Cache Memory Concepts."

As with DRAM, SRAM can be designed to operate either asynchronously or synchronously. Likewise, SRAM can be designed to use a burst mode, which simplifies the interface of SRAM used in caches, since caches usually operate synchronously, in burst mode.

Chapter 14

The Previous Chapter

Earlier chapters described the 80386DX microprocessor and DRAM memory. The 80386DX microprocessor is capable of performing data transfers at substantially higher clock rates than the 80286 microprocessor. Combine the increased performance of the 80386DX microprocessor with the relatively slow access time of DRAM memory and a problem presents itself. Whenever the microprocessor attempts to access memory, wait states will be inserted into the bus cycles to accommodate the DRAM's slow access time. Since the microprocessor accesses memory far more than any other system element (to fetch instructions), processor performance would suffer severe degradation.

This Chapter

In this chapter, cache memory is discussed. This is the most common method used to achieve superior memory performance while holding down the cost of implementation. Various memory cache strategies found in 80386-, 80486- and Pentium processor-based systems are discussed, along with advantages and disadvantages of each.

The Next Chapter

The next chapter discusses the types and configurations of ROM memory typically found in ISA systems.

The Problem

When the IBM PC was introduced, it was based on the Intel 8088 microprocessor running at 4.77MHz. The 8088 takes four ticks of its 4.77MHz PCLK to run a 0-wait-state bus cycle, so a 0-wait-state bus cycle takes 838ns (4 x 209.5ns = 838ns). Virtually any DRAM or ROM on the market at the time could respond well within this period of time, so wait states really weren't an issue.

As processor speeds have rapidly increased, the speed of economical dynamic RAM (DRAM) has increased at a slower pace. Thus, in order to gain the maximum performance benefit of the faster and more powerful processors, faster and more expensive static RAM devices must be used. Static RAM is easily capable of providing zero-wait-state performance; however, for several reasons, this solution is seldom considered:

- SRAM is typically ten times more expensive than DRAM memory.
- SRAM chips are typically larger than DRAMs, requiring more real estate for the same amount of memory.
- During normal operation, SRAMs consume more power than DRAMs and may require a more powerful, and therefore more expensive, power supply.
- During normal operation, SRAMs generate more heat than DRAMs and may therefore require a larger cooling fan.

To be fair, it's true that SRAM consumes less power than DRAM when it is not being accessed. But why put of expensive SRAM in a system and then not access it repetitively at high speed?

The Solution

The solution chosen by most manufacturers of fast 80386-, 80486- and Pentium processor-based PCs is cache memory. Implementation of a cache memory subsystem is an attempt to achieve acceptable system cost and a high percentage of 0-wait-state bus cycles when accessing DRAM memory.

A cache memory subsystem is implemented by populating main memory with relatively slow access, cheap DRAM, while also incorporating a relatively small amount of high cost, fast access SRAM into the system. The SRAM memory is referred as cache memory and a cache memory controller attempts to maintain copies of frequently accessed information read from DRAM memory in the cache.

Refer to figure 14-1. The cache memory controller keeps track of the information it has copied into the cache memory. When the cache memory controller sees the microprocessor initiate a memory read bus cycle, it checks to determine if it has a copy of the requested information in cache memory. If a copy is present, it immediately reads the information from the cache, sends it back to the system board microprocessor over the microprocessor's data bus, and asserts the microprocessor's READY# signal. This is known as a read hit. The ac-

cess can be completed in 0 wait states because the information is fetched from the fast access cache SRAM.

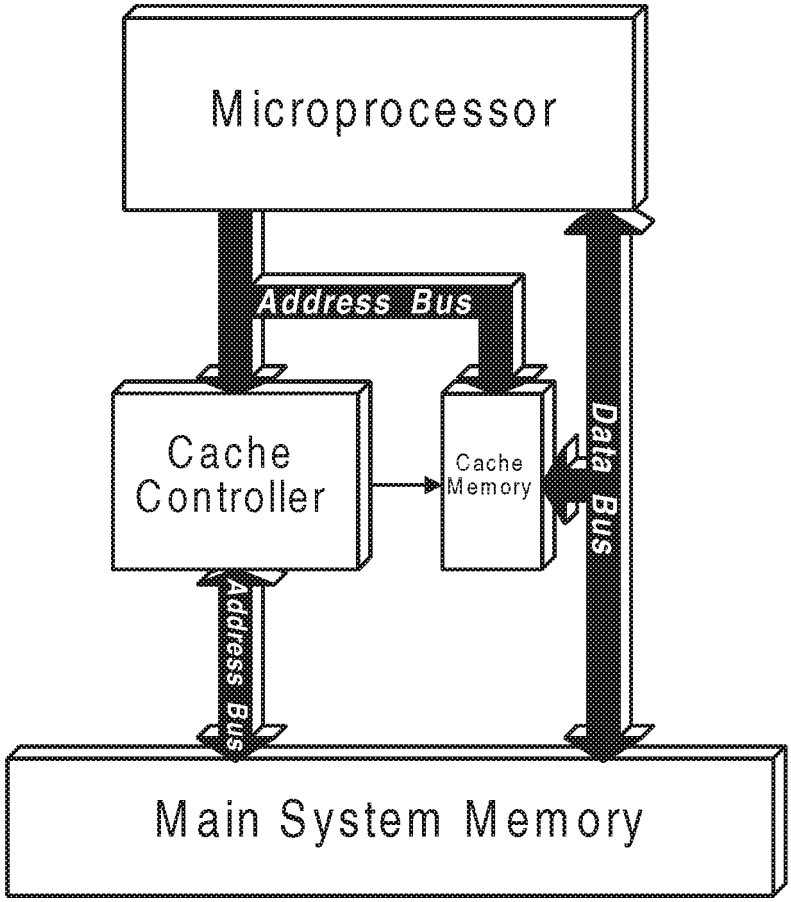


Figure 14-1. The Cache Memory Concept

If, on the other hand, the cache memory controller determines that it does not have a copy of the requested information in cache, the information must be read from DRAM memory. This is known as a read miss and, due to the slow access time of the DRAM memory, results in wait states. The requested information is sent from DRAM back to the microprocessor to fulfill the request. In addition, the information is also copied into the cache memory by the cache memory controller. The cache controller updates an internal directory that keeps track of information stored in cache memory.

On the surface, it may not be evident how this can speed up the microprocessor's memory accesses. However, most programs, such as Lotus 1-2-3, contain program loops that are executed many times. A program loop is a series of instructions that must be executed a number of times in succession to produce the desired results.

As an example, Lotus 1-2-3 uses a program loop to recalculate spreadsheet cells. Depending on the number of calculations to be made during a recalc, this program loop may have to be executed hundreds or even thousands of times.

Assume that the cache memory is empty — it contains no copies of information from DRAM memory. The first time the program loop is executed, the following series of events takes place:

1. The microprocessor initiates a memory read bus cycle to fetch the first instruction from memory.
2. The cache memory controller uses the memory address to determine if a copy of the requested information is already in the cache memory. Since the cache is initially empty, a copy doesn't exist and a read miss occurs.
3. The cache memory controller initiates a memory read bus cycle on its own buses to fetch the requested information (the instruction) from DRAM memory. This will take one or more wait states, depending on the access time of the DRAM.
4. The information from DRAM memory (the instruction) is sent back to the microprocessor and the cache memory controller also copies it into the cache memory and updates a directory within the controller to reflect the presence of the new information. No advantage was realized during this memory read because the information had to be read from slow DRAM memory.
5. Upon completion of the first instruction in the program loop, the microprocessor initiates a series of memory read bus cycles to fetch the remaining instructions in the program loop. If the cache is sufficiently large, all of the instructions in the program loop will be resident in cache memory after the last instruction in the loop has been read from DRAM memory and copied into the cache. No advantage was realized during the first execution of this program loop.
6. The last instruction in the program loop is an instruction to jump to the beginning of the loop and start over again. In other words, the microprocessor starts fetching and executing instructions at the memory address of the first instruction in the program loop again.
7. When the microprocessor initiates the memory read bus cycle to fetch the first instruction again, the cache memory controller checks its cache direc-

tory and detects the presence of the requested information in cache memory. The cache memory controller immediately reads the information from the fast access SRAM that makes up the cache and sends it back to the microprocessor with 0 wait states incurred.

The microprocessor is thus able to fetch and begin executing the instruction faster than it was able to the first time through the loop. This proves true for the remaining instructions in the program loop as well. The second and subsequent times through the program loop, the microprocessor incurs no wait states in fetching the instructions, so the program runs substantially faster.

Principles of Locality

The term used to explain the characteristics of programs which run in relatively small loops in consecutive memory locations is the "locality of reference." As shown in the previous example, cache subsystems work because most programs that run on PCs require the same information from consecutive memory locations over and over again. The locality of reference principle is comprised of two components: temporal locality and spatial locality.

Temporal Locality

Since programs run in loops, the same instructions must be fetched from memory on a frequent and continuing basis, meaning that programs tend to use the most recently used information again. The older the information in cache, the less likely it is to be used again. This is known as the principle of "temporal" locality. How well a cache subsystem performs is tied directly to how often the software requests the same information from memory. If a program were intentionally designed to never access information more than one time, the cache subsystem would provide absolutely no performance increase.

Spatial Locality

Programs and the data they request tend to reside in consecutive memory locations. This means that programs are likely to need code or data that are close to or adjacent to locations already in use. This characteristic is called the principle of spatial locality.

Cache Performance

Performance of cache subsystems depends primarily on the frequency of cache hits. Performance is usually stated in terms of a percentage of cache hits or hit rate. Hit rate can be expressed as:

$$\text{Hit Rate \%} = \frac{\text{Cache Hits}}{\text{Total Memory Requests}} \times 100\%$$

Since cache hits only occur on subsequent accesses to the same location, performance of cache systems then can be linked directly to the way in which software runs or how well it satisfies the principle of locality of reference. If the program runs in a fairly small area of memory and repeats or loops, a high percentage of accesses to memory will be cache hits. If on the other hand, the program runs long strings of non-looping code, every access will be a cache miss. Fortunately, most programs run in a looping fashion and therefore cache hits occur a high percentage of the time (approximately 85 to 95%).

Several factors contribute to the cache hit rate in addition to the principle of locality of reference. These include the cache's architecture, size of the cache memory and the cache memory organization, which are discussed later in this chapter.

Overall System Performance

In addition to providing fast access to the memory subsystem, cache subsystems can also provide performance advantages by improving bus utilization. Bus utilization is of particular concern when multiple processors and bus masters are used. The percentage of the time that any bus master occupies the system bus is called its bus utilization. If the total bus utilization of all bus master devices add up to over 100%, performance would drop due to exceeding the bandwidth of the bus. Well designed cache subsystems reduce the amount of time processors spend on the system bus by making access to the bus as short as possible (zero wait states).

An additional benefit related to bus utilization, called bus concurrency, is possible when look-through cache architectures are used. Bus concurrency means that a processor can gain access to memory from its cache memory, while another bus master accesses main memory over the system bus at the same time. Look-through cache designs permit concurrency because they decouple the processor's local bus from the main system bus. See the section on look-through cache designs for additional information.

Cache Consistency

In order for cache subsystems to work properly, the contents of cache and main memory must be exact duplicates. There are several possible instances in which locations stored in cache memory are updated in cache or in main memory while the duplicate copy remains unchanged. When either the cache memory or main memory is updated, the duplicate copy will contain old information called "stale" data. Cache systems must be designed to eliminate potential cache consistency problems. See the section on cache consistency for details.

Components of a Cache Subsystem

Cache subsystems are comprised of three basic parts as shown in figure 14-2:

- The cache Memory (High speed Static RAM)
- The cache management logic
- The cache memory directory (tag RAM)

Cache Memory

The cache memory is comprised of SRAM chips that can provide the processor with data at zero wait states. Cache memory is accessed when a copy of an address being accessed in main memory is found in the cache memory directory by the cache management logic.

Some literature refers to the cache memory as the "data cache RAM" to distinguish it from the cache memory directory or "tag RAM." But beware that "data cache RAM" is easily confused with a "data cache" (which would be half of a split-cache system, described in the "Cache Architectures" section).

The size and organization of the memory cache varies according to design and is covered in the section entitled "Cache Organization and Size."

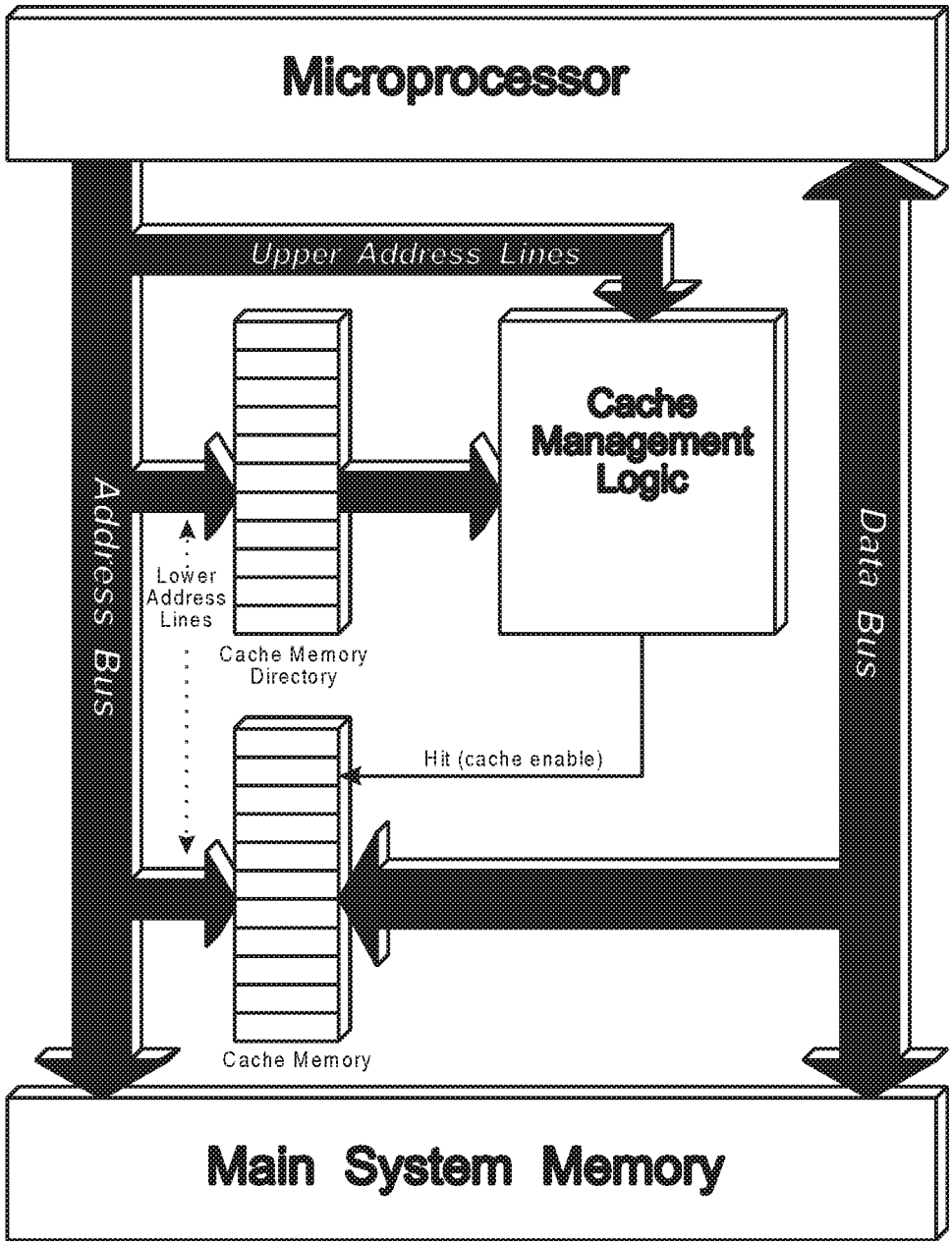


Figure 14-2. Cache Subsystem Components

Cache Management Logic

During a read access to main memory, the cache management logic compares the memory address output by the processor with the cache memory directory which contains a record of which locations are stored in cache memory. If a match occurs, the cache management logic enables cache memory to output information from the addressed location, and the bus cycle is completed in zero wait states. If a match does not occur, the requested information is supplied by main memory. The information is sent back to the processor from main memory and also stored in cache memory so that the latest information is now available in cache.

Cache Memory Directory

The cache memory directory, also called tag RAM, contains a listing of all the memory addresses that have copies stored in cache memory. Each cache location (sometimes referred to as a set) has a corresponding entry in the cache directory as shown in figure 14-3. The contents of the directory are compared to the memory address from the processor to determine if a copy of the requested information is contained in cache memory. The contents of the directory are updated each time new information is stored in cache memory.

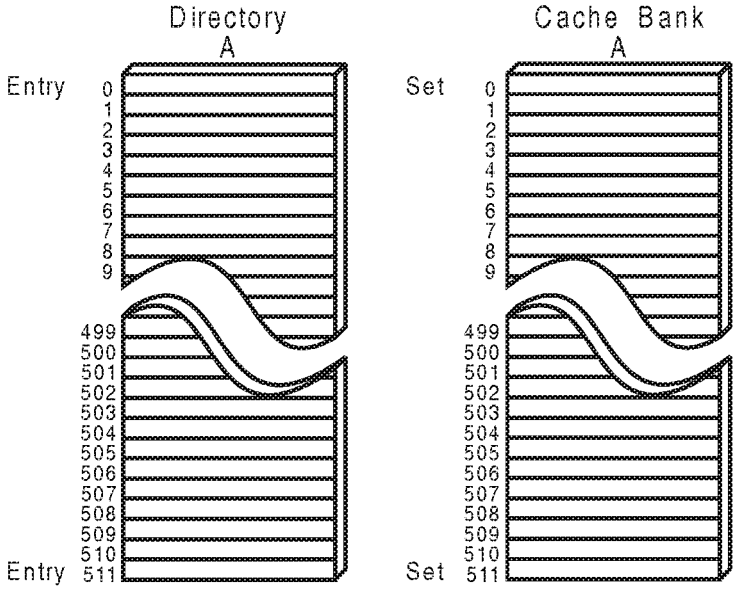


Figure 14-3. Relationship of Cache Directory to the Cache Memory

Intro to Cache Architecture, Coherency, Write Policies and Organization

The next several sections describe various parameters which specify caches. These parameters and some of their possible values are listed below. A cache may be designed by selecting one parameter from each of the lists. For example, a cache could be designed as an 8KB, look-through, level 1, split, write-back policy, two-way set-associative cache with a 32-byte line size, that maintains coherency by snooping the system bus. An example of that type of cache is the data cache in the Pentium processor. There are three lists for Architecture because three independent parameters define the physical layout (architecture) of the cache.

Architecture
- Look-through
- Look-aside

Architecture
- Level 1
- Level 2
- Level 3, etc.

Architecture
- Combined (unified)
- Split (dedicated)

Coherency Protocol (ways of maintaining coherency)
- Flushing
- Snooping (bus watching)
- Software-enforced coherency

Write Policy
- Write-through policy
- Buffered write-through policy
- Write-back policy

Organization or Set Associativity
- fully-associative cache
- direct-mapped cache
- two-way set-associative cache

- four-way set-associative cache, etc.

Line Size
- bytes per line

Cache Size
- KB of data SRAM

Addressing
- physical
- virtual

Although each of these parameters are independent of one another, it is possible to select values that result in impractical or inefficient cache designs. For example, a small (1KB), look-through cache statistically degrades (instead of improving) overall system performance due to the lookup penalty imposed by the high miss rate of the small cache.

The cache parameters are defined and described in detail in the sections that follow.

Cache Architectures

Two basic architectures are used in today's systems:

- Look-through cache designs
- Look-aside cache designs

Each type of cache architecture has its advantages and disadvantages. The following sections detail both types of architectures and analyzes their benefits and liabilities.

Look-Through Cache

Performance in look-through cache designs is typically higher overall than for look-aside designs. Figure 14-4 illustrates the look-through cache design. As its name implies, the look-through cache "looks through" its internal cache to determine if a copy of the requested information is in the cache. If found, information is sent back to the processor in zero wait states. Notice that the memory request from the processor is not automatically transferred to the system bus. This keeps the system bus free for use by other bus masters. Only on cache misses will a request be passed to the system bus. Look-through cache

misses will a request be passed to the system bus. Look-through cache designs isolate the processor's local bus from the system bus, thereby reducing the use of the system bus and permitting concurrent operations to take place.

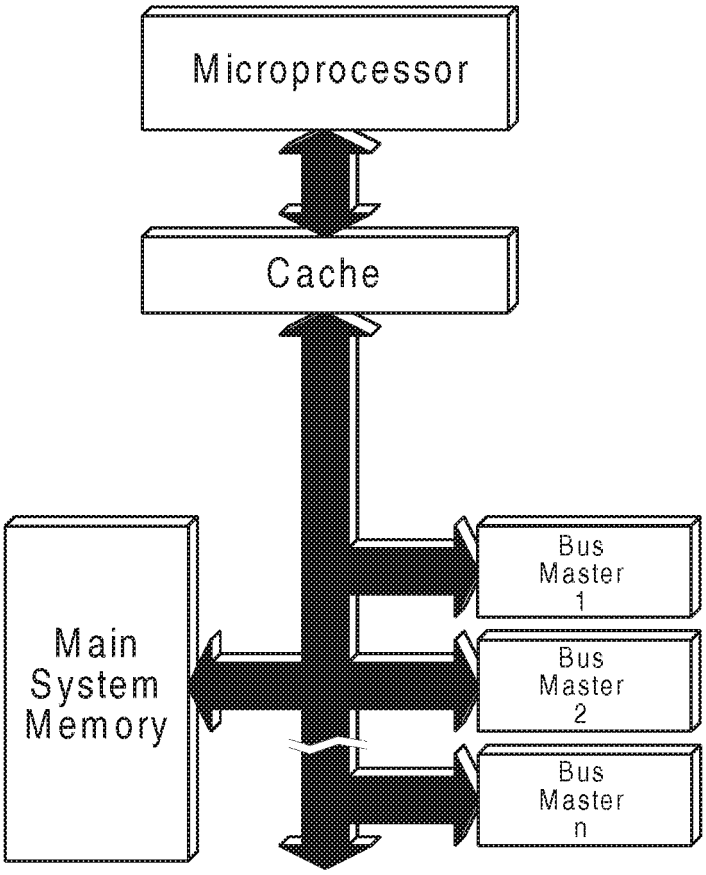


Figure 14-4. Look-Through Cache Architecture

A closer look at the look-through design reveals buffers on the address and data buses that isolate the processor and system buses. Refer to figure 14-5. These buffers are controlled by the cache management logic. If a request from the processor is a cache hit, information is accessed directly from cache memory and the processor and system bus remain isolated. If the requested information is not in cache (a read miss), the cache management logic will enable

the buffers allowing the request to be sent over the system bus to get the requested information from main memory. The information is passed through a latch in the data path back to the processor, as well as into the memory cache.

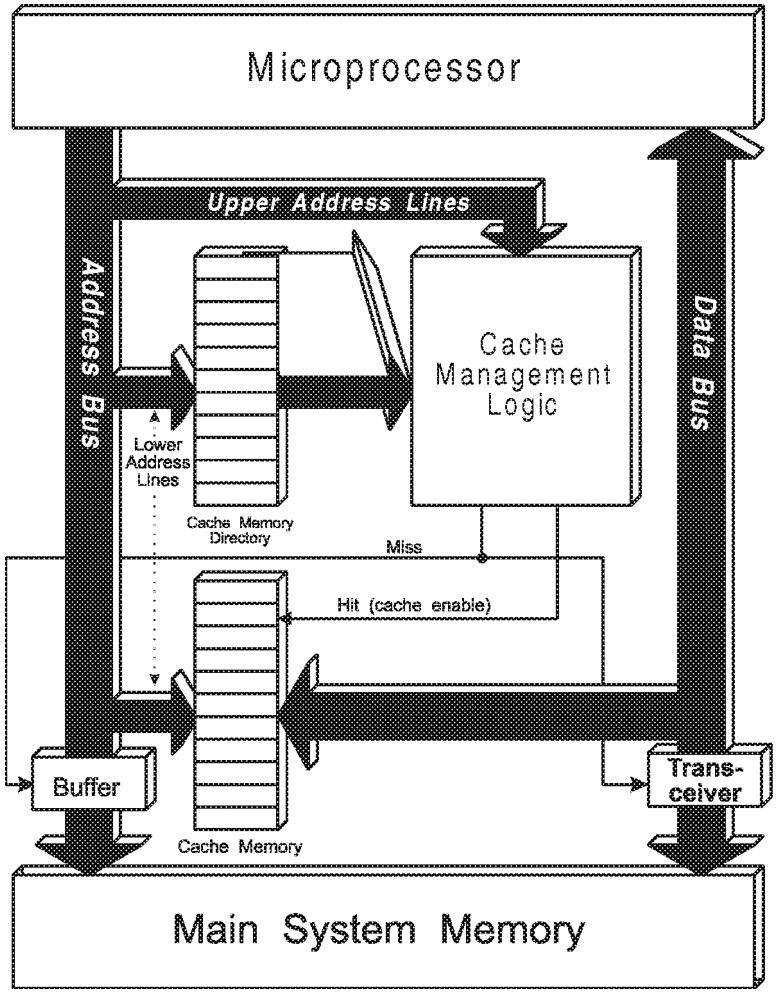


Figure 14-5. Look-Through—Concurrent Bus Operation

A major advantage of look-through designs is that they permit two bus masters to operate simultaneously in the same system. The processor can be running zero-wait-state operations out of the cache memory, while a bus master such as

the DMA controller or refresh logic can use the system bus to access main memory. These two operations can be run at the same time without affecting each other and are typically referred to as system concurrency.

To expansion devices, look-through cache controllers appear as if they are the main microprocessor. In single processor systems, when a bus master needs access to main memory it requests use of the system bus directly from the cache controller. The cache controller will grant the use of the system bus by sending an acknowledge to the bus master just as the microprocessor would normally do.

Since look-through cache controllers isolate the system bus and processor bus and since they can be designed to run their own bus cycles to main memory, they also provide performance advantages during write operations. During memory writes, look-through cache designs can provide zero-wait-state operation for write misses. During write operations, many look-through designs will trick the microprocessor into thinking the information was written within zero wait states, but in fact no write need have taken place at all. The look-through cache subsystem memorizes the entire write operation so that it can complete the memory write to main memory later. This is called a posted write or buffered write.

In summary, look-through designs provide performance benefits by:

- reducing system bus utilization since most memory accesses come from cache memory, leaving the system bus free for other bus master's use.
- allowing system concurrency, where both the processor and another bus master can perform bus cycles at the same time, and
- completing write operations in zero wait states using posted writes.

The primary disadvantages of look-through designs are:

- The local CPU's memory requests go first to its cache subsystem to determine if there is a copy of the target location is in its cache memory. This lookup process delays the request to main memory in the event that the memory request is a cache miss. This delay is commonly referred to as the "lookup penalty."
- look-through caches are more complex and difficult to design and implement.
- look-through designs are costly.

Look-Aside Cache

Refer to figure 14-6. Look-aside caches do not isolate the processor bus from the system bus as do look-through caches. When the processor begins a bus cycle all devices in the system see the address as they do in non-cache systems. The cache controller monitors each memory request to see if cache memory contains a copy of the requested information. The cache controller terminates the bus cycle in zero wait states if the operation is a hit, and informs the main memory subsystem to abort the request. If the operation is a miss, the bus cycle completes in normal fashion from main memory.

Note that even when a memory request is a cache hit, main system memory will start the access thus creating an unnecessary memory precharge cycle in many cases. This means that other bus masters in the system needing access to memory must wait until the precharge cycle has completed.

The major advantages of look-aside cache designs are:

- Response time on cache miss cycles. Cache miss cycles complete faster in look-aside caches because the bus cycle is already in progress to main memory and no lookup penalty is incurred.
- Simplicity of design. Look-aside designs need only monitor one address bus, whereas, look-through designs must interface with both the processor bus and system bus.
- Lower cost of implementing due to their simplicity.

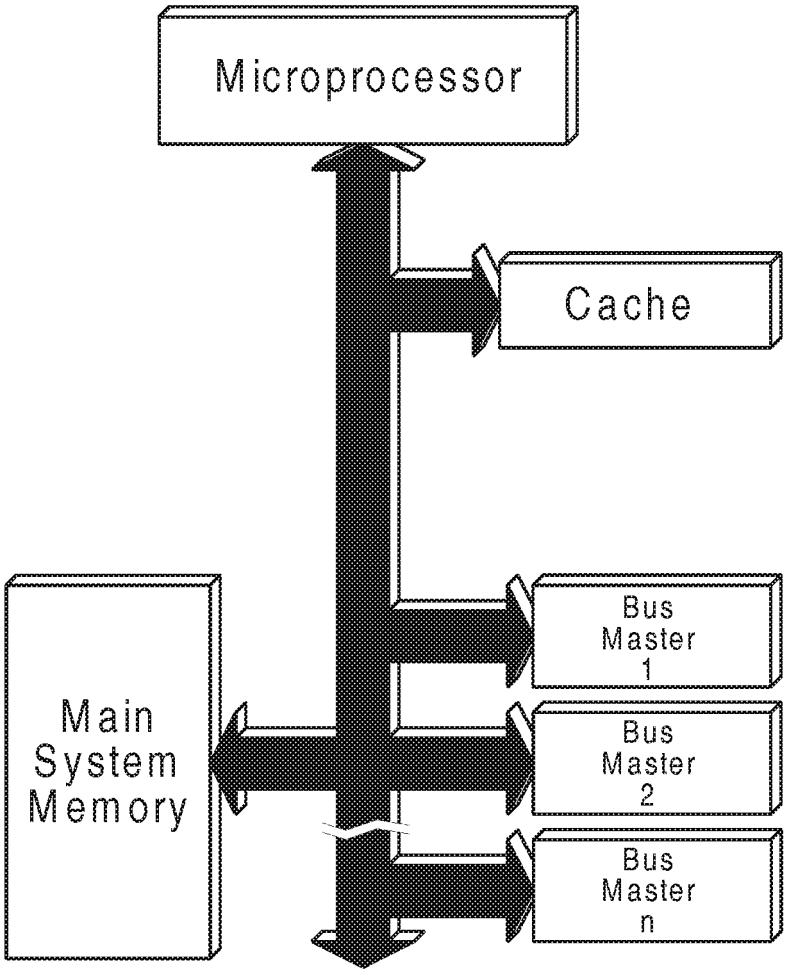


Figure 14-6. Look-Aside Cache Architecture

The disadvantages of look-aside cache designs are:

- System bus utilization is not reduced. Each access to main memory goes to both the cache subsystem and main memory.
- All memory requests, whether a hit or miss, result in the start of a memory cycle in main memory. This causes a precharge cycle to begin which prevents other devices from accessing main memory until the precharge time has expired.

- Concurrent operations are not possible since all masters reside on the same bus.

First- and Second-Level Caches

Some systems use two levels of cache to improve overall system performance. Figure 14-7 shows the relationship between first- and second-level caches in typical 80386- and 80486-based systems.

First-level caches provide the processor with the most often used code and data. These caches are typically small (4KB to 64KB). An example of this type of cache is the internal cache used in the 486.

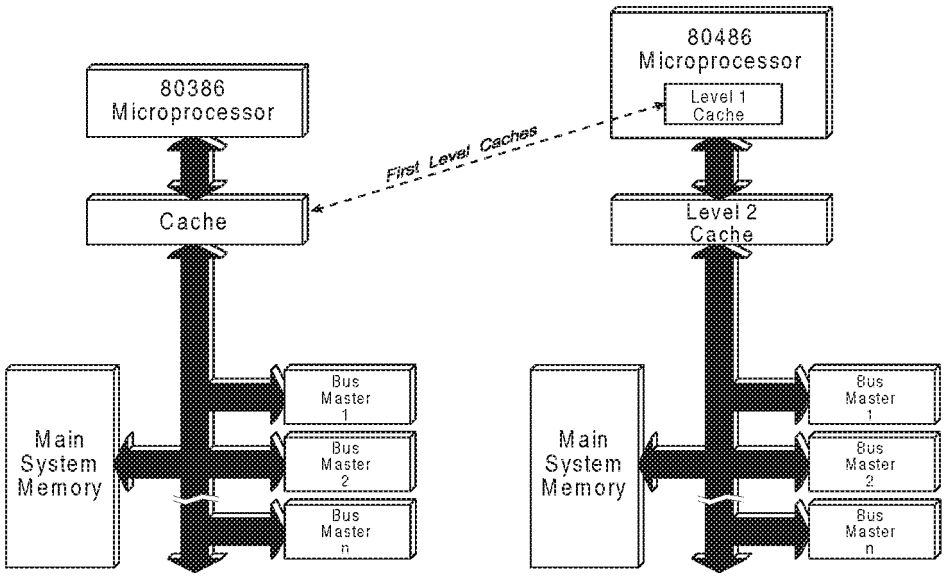


Figure 14-7. First- and Second-Level Caches

Second-level caches are sometimes added between the first-level cache and main memory. These caches are usually much larger than first-level caches (64KB to 512KB) because they hold all the information contained in the first-level cache, as well as information that the first-level cache is unable to retain. Second-level caches improve overall performance since the first-level cache can

get information from the second-level cache at zero wait states even on most misses.

Most modern microprocessors include a level-one cache on the same silicon die as the microprocessor core. The second-level cache is typically a separate component or group of components on the motherboard.

Caches are managed by the cache management logic using hardware algorithms that are very similar to the algorithms used by the operating system when it manages virtual memory (see "Page Translation" in the chapter entitled "The 80386DX and SX Microprocessors"). The cache contains the most recently accessed lines from main memory while main memory contains the most recently accessed pages from mass storage. Due to this similarity, main DRAM is sometimes referred to as the level three cache in computer mainframe designs, since main DRAM caches the contents of mass storage.

Combined (Unified) and Split (Dedicated) Caches

There are two entities in the microprocessor which need efficient access to memory: the instruction prefetcher and the execution unit. Unless the microprocessor is halted, the prefetcher is constantly making sure that its prefetch queue is full of instruction bytes for the instruction decoder to decode. While the prefetcher is fetching bytes, the execution unit may need access to memory to read or write data as the result of executing a MOV instruction. The microprocessor's bus unit gives priority to the execution unit since it is the execution unit that actually gets work done in the microprocessor. The prefetcher is merely a performance enhancement tool. But if the execution unit is very fast and if it is performing many memory accesses, the execution unit may execute all of the prefetched instructions, thus emptying the prefetch queue, before the prefetcher has had a chance to access memory. When the prefetch queue empties out, the execution unit will starve, thus stalling the processor until the prefetcher can once again access memory. The access to memory becomes a bottleneck.

Even with a cache, there's still a bottleneck since the prefetcher and the execution unit are still contending for access to the cache. To eliminate this bottleneck, the code that the prefetcher needs may be put in a dedicated code cache while the data that the execution unit needs may be put in a dedicated data cache. As long as there are separate interfaces to these two caches, the bottleneck may be avoided. Both the prefetcher and the execution unit may be served simultaneously from their own caches. Caches that are designed this way are referred to as split or dedicated caches and the two caches are called the in-

struction cache and the data cache. Sometimes, the instruction cache is referred to as the code cache. The Pentium processor has dedicated internal code and data caches while the 486 microprocessor has a combined internal cache. After a miss on the internal cache, the microprocessor's external bus becomes the bottleneck for access to external memory, so it is rare to see dedicated external caches except on microprocessors that have two sets of address buses and two sets of data buses, such as high-speed digital signal processors.

The ability to access code and data simultaneously is a significant advantage of split caches vs. combined caches, but that doesn't mean that split caches are always better. Unified or combined caches have the ability to dynamically reallocate space to store more code and less data, or to store more data and less code, depending on what information the microprocessor is actually accessing during a specific period of time. The combined cache decides what to keep in the cache by using a least-recently-used algorithm (described later in this chapter) based on the principles of locality (described earlier in this chapter). The combined cache doesn't know or care whether the information that it stores represents code or data, so it does not have to base its decision on whether the requests are coming from the prefetcher or the execution unit.

Split caches lack the ability to dynamically reallocate space to store more code or more data, and they run the risk of the toll-booth problem. If you live in a city with toll roads, you may have noticed that lines form when there are dedicated lanes, say for trucks only, or for exact change only, while other lanes, such as for "receipts given," will be empty. That means that some lanes have too much capacity (they are empty) and others lack capacity (they are backed up). The same can happen with dedicated caches. If all the toll lanes provided all the same services, then the lines would even out and overall throughput would increase. So why do they have dedicated lanes? To provide priority to certain classes of traffic. That's exactly what the dedicated caches do — they provide priority service to their specific entities in the microprocessor. But combined caches typically provide the ability to store more code and data overall since every location in the cache may be used for either code or data.

Cache Consistency (Coherency)

Loss of cache consistency occurs when the copy of a location in cache no longer matches the contents of the information stored in main memory. Cache consistency (also known as cache coherency) problems can result from either the cache memory being updated while the main memory location is not, or main memory being updated while the cache memory is not. In each of these instances, some action must be taken to ensure that the conflict is resolved.

It is imperative that any microprocessor or bus master that is accessing main memory be guaranteed access to the latest copy of information that is, or should be, stored in main RAM.

Causes of Cache Consistency Problems

Loss of cache consistency occurs when the information stored in cache and the corresponding location in main memory do not match. Consistency problems occur under two conditions:

- Cache memory is updated while main memory is not.
- Main memory is updated while cache memory is not.

In the situations above, either cache memory or main memory contains stale data. In the first case, stale data is caused by a cache write hit within the cache memory. In the second case, a bus master wrote to main memory. Stale data in the cache must be updated before the microprocessor accesses the cache. Stale data in main RAM must be updated before another bus master accesses that location in main RAM. Note that this does not necessarily mean that the cache and main RAM must always be exact copies of one another. It does mean that none of the microprocessors or bus masters should ever be allowed to access stale data. To detect a coherency problem, the cache controller must monitor (snoop) the system bus to detect memory read and write operations to locations contained in cache memory.

Consistency problems are handled differently depending on the cache's write policy.

Write Policy

When a cache write hit occurs, the cache memory is updated. Cache memory then contains the latest information and main memory contains outdated information. The location in main memory is said to have "stale" information and the cache location is said to be "dirty," or modified, since it no longer is an exact duplicate of its corresponding location in main memory. To correct this cache consistency problem, the corresponding main memory location must be updated to reflect the change made in cache. If main memory is not updated, another bus master could receive stale data when it reads from a memory location that has not been updated. Three write policies commonly used to prevent this type of consistency problem are:

- write-through
- buffered write-through
- write-back

These three write policies are covered in the following sections.

Write-Through Cache Designs

Many look-through cache controllers pass each write operation on to main memory. This ensures that main memory always contains valid data. This is called a write-through policy. This implementation is very simple and effective, but results in poor performance since each write operation must access slow main memory.

Buffered Write-Through Designs

A variation of the write-through policy is the buffered or posted write-through operation. The buffered write-through cache has the advantage of providing zero-wait-state write operations for both hits and misses. When a write occurs, a buffered look-through cache tricks its associated processor into thinking the information was written in zero wait states, but the write to memory has not taken place yet. The look-through cache controller stores the entire write operation in a buffer so that it can complete the memory write to main memory later.

If there are two back-to-back memory write bus cycles, the cache controller inserts wait states into the second bus cycle until the first memory write has actually been completed. The second bus cycle is then posted and the CPU's READY# line is asserted. Since memory writes typically occur as single occurrences, a large percentage of memory writes are completed at zero wait states.

In a buffered write-through cache, another bus master is not permitted use of the buses until the write-through is completed, thereby ensuring that the bus master receives the latest information from main memory.

The write-through operations described above require use of the system bus. As a result, when the write-through to memory is taking place, bus masters are prevented from accessing main memory. Note, however, that the cache consistency problem only occurs when a bus master reads from a location in main memory that has not yet been updated by the cache controller. The frequency of this type of occurrence is very small. In fact, the memory location will likely be updated numerous times by the CPU before another bus master reads that

particular location. As a result, write-through designs, as well as buffered-write-through designs, update main memory each time a memory write is performed, though the need for such action may not be required immediately.

In summary, cache coherency problems can occur when the cache memory is updated on a write hit, but the corresponding location in main memory is not updated. Problems would occur if another bus master read from a location in main memory that contains stale data. The write-through policy prevents this potential consistency problem by automatically passing each write operation on to main memory, thereby ensuring that main memory always has the latest data.

Write-Back Cache Designs

Write-back designs improve overall system performance by updating memory only when it is necessary, thereby keeping the system bus free for use by other processors and bus masters. Main memory is updated (written to) only when another bus master accesses a main memory location that contains stale data or when a cache location that contains modified information is about to be overwritten to make room for a new cache line.

Cache memory locations are marked as modified in the cache directory when they are updated and the update is not propagated through to memory. When another master is reading from main memory, the cache subsystem must monitor, or snoop, the system bus to check for reads or writes to main memory locations that have modified data in cache memory.

Write-back cache designs are more complicated to implement because they must make decisions on when to update main memory.

Bus Master/Cache Interaction

When another device in the system requires the use of the buses, it must become bus master. In a system that doesn't incorporate a cache controller, the device requesting bus mastership asserts HOLD to seize the buses from the microprocessor. In a system incorporating a look-through cache, however, the requesting device must force the cache controller, rather than the microprocessor, to relinquish control of the system buses. To do this, the requester asserts the cache controller's HOLD input to request control of the buses. In response, the controller disconnects from the system bus and asserts HLDA (Hold Acknowledge) to grant the system bus to the requester. The new bus master can then use the system bus to perform one or more bus cycles to transfer data.

Since a bus master can read from and write to main memory, cache consistency problems can occur under two circumstances:

- **Writes to main memory**—When bus masters write to main memory they update locations that may also be cached by the cache controller. In these instances, main memory is updated and the data in cache memory becomes stale. The cache controller must monitor writes to main memory to detect potential coherency problems.
- **Reads from main memory (with write-back cache)**—When bus masters read from main memory in write-back cache implementations, they may read from a location that contains stale data. That is, the location is in cache and has been updated in cache but not in main memory. If the bus master is allowed to read the data from main memory, stale data would be received. To detect this potential coherency problem, write-back caches must also monitor reads from main memory.

The following section discusses the process of monitoring the buses to detect potential cache consistency problems.

Bus Snooping/Snarfing

Bus snooping is the method used by cache subsystems to monitor main memory accesses made by another bus master. The cache controller monitors (snoops) the system bus when another bus master is performing a bus cycle to main memory. If the location being accessed in main memory is also contained in the cache memory, it is termed a cache snoop hit. In such cases, cache memory or main memory may contain stale data. Two conditions arise that create the need to snoop the buses:

1. A bus master writes to a location in main memory. Assume that the cache controller has relinquished control to the DMA controller so that it may transfer a block of data from a disk controller into main memory. It is quite possible that the DMA controller will alter the contents of main memory locations that have already been copied into the cache. This means that the cache and main memory would no longer be coherent. In other words, the information in the cache will not accurately reflect the actual information in main memory. This is a potentially dangerous situation because the microprocessor may now fetch stale information from the cache and make bad decisions based on it.

   Cache subsystems handle this situation differently depending on how they are designed and depending on their write policies.

   a) Write-through policy caches may invalidate the line in cache. The next time that particular location is accessed by the cache's local microprocessor, it results in a cache miss, forcing the cache subsystem to get the information from main memory where the valid data resides.
   b) Write-back policy caches cannot simply invalidate the line since modifications that may have been made by the microprocessor would be lost. Typically, the cache line is wider than the data bus, so the bus master is not updating the entire cache line. Since the cache cannot remember which bytes in the line are modified, it must push back the entire line to memory. Write-back policy caches must force the bus master to abort the bus cycle to prevent access to the stale data in main RAM. The cache then pushes back the modified line to memory and invalidates its copy of the line. The bus master then retries the bus cycle again and writes its changes into main RAM. When the microprocessor tries to access that location in the future, the cache has a miss and reads the latest information from main memory.
   c) Either write-through or write-back policy caches may snarf the data that the bus master is writing to memory. When the cache subsystem snoops the bus during another bus master's write operation and detects a snoop hit, it captures the data from the system bus while its being written to main memory by the bus master. In this way, both main memory and cache memory get updated at the same time, thus eliminating the potential consistency problem.

2. A bus master reads from a location in main memory. Only cache subsystems that use a write-back policy monitor the system buses for memory reads. In this case, the cache controller snoops memory reads by another bus master to determine if the location being read from is a location that has been updated in cache, but not in main memory (i.e. the cached location is modified). If so, this is a snoop hit on a modified line and the cache subsystem must force the bus master accessing the location to suspend its bus cycle (backoff) until the cache subsystem has updated main memory. Once the main memory location has been updated by the cache subsystem, the bus master is allowed to complete its access to main memory.

   Alternatively, when a snoop read hit is detected, the cache controller could cause the access to main memory to be aborted and supply the valid data directly to the bus master performing the read.

Coherency may be maintained in a variety of ways. The primary method is based on bus watching, or snooping, as described above, but there are at least two other, less commonly used (but perfectly viable), alternatives.

Coherency via Cache Flushing

All of the information in a write-through policy cache is an exact copy of the information currently in main RAM. Information would not be permanently damaged or lost if the write-through cache's contents were flushed (invalidated), because the information is also resident in main RAM. Therefore, one way to ensure that a write-through policy cache always gets the latest and greatest information that a bus master wrote to main RAM is to flush the cache whenever a bus master issues HOLD request to the cache. When the microprocessor subsequently tries to access any new information, the cache has a miss and gets the information from main RAM. Admittedly, this is a rather draconian method of maintaining coherency, but it works, and it's simple.

In fact, flushing is an economical method of maintaining coherency in systems where the cache rarely has to give up the busses to another bus master. A good example is a laptop system. It's very rare for a bus master (other than the refresh logic) to seize the busses away from the cache in a laptop. The cache doesn't need to maintain coherency with the RAM refresh logic, so the cache does not need to be flushed when the refresh logic asserts HOLD. In general, the only other bus master in the laptop is the DMA controller, and it is rarely used. So, the expense of bus snooping logic can be saved by merely flushing the write-through policy cache on those rare times that the DMA controller becomes the bus master.

It would be a major mistake to invalidate the contents of a write-back policy cache in this manner since the write-back policy cache may contain information that does not appear in main RAM. The information would be lost.

Software-Enforced Coherency

Most of the descriptions of how to maintain cache coherency are based on forcing the cache controller hardware to automatically enforce coherency. Hardware-enforced coherency isn't always necessary. If the operating system software can guarantee that no two microprocessors or bus masters will ever access the same memory locations, then there's no need for the cache controller to worry about another bus master's memory accesses.

Ultimately, the operating system software has control over the microprocessor and over every bus master in the system. The operating system's device drivers give commands to the bus masters, telling them which memory locations to access. The operating system can therefore issue commands that prevent bus masters from accessing (sharing) memory locations whose contents are currently contained in the microprocessor's cache.

Software-enforced coherency can provide better throughput than hardware-enforced coherency when transferring large blocks of data between main memory and peripherals by avoiding the time-consuming backoff's and pushbacks that are often used for hardware-enforced coherency. The pushbacks are avoided because the cache does not contain any of the locations that the bus master is accessing.

Of course, if software-enforced coherency is used, the operating system has to be smart enough to manage the cache itself. The caches used in the 486 and Pentium processors are designed to provide hardware-enforced coherency, software-enforced coherency, or both.

Cache Organization and Size

Cache memory can be organized in various ways to improve the likelihood of cache hits. In other words, cache memory of the same size can provide different hit rates depending on the way it is organized. The following sections discuss the common methods of organizing cache memory.

Fully-Associative Cache

The fully-associative method of organization provides the optimum structure for ensuring cache hits, and for a given cache size it provides the highest performance of all other organizations. Fully-associative caches view memory as a single page or block regardless of its size. See figure 14-8. Any location in main memory can be copied to any location within the cache. This provides maximum flexibility in storing information from anywhere in memory.

In order to keep track of any location in memory, the cache directory must store the entire address. Any given memory location requested might be found in any one of the cache locations. Thus, each access to memory requires that the address of the requested information be compared to each entry in the entire cache directory to determine if the requested information is in cache memory. The time needed to check the address against the directory entries (called the look-up penalty) can be unacceptably long in fully-associative caches. Since the look-up penalty can be substantial, fully-associative caches are usually quite small (4KB or less) reducing the number of cache entries that have to be checked with each memory request.

These types of smaller caches do not provide acceptable hit rates with most applications typically run on ISA machines. The challenge is to design a larger cache that provides higher hit rates without the associated look-up penalty. The concept of the direct-mapped cache provides this solution.

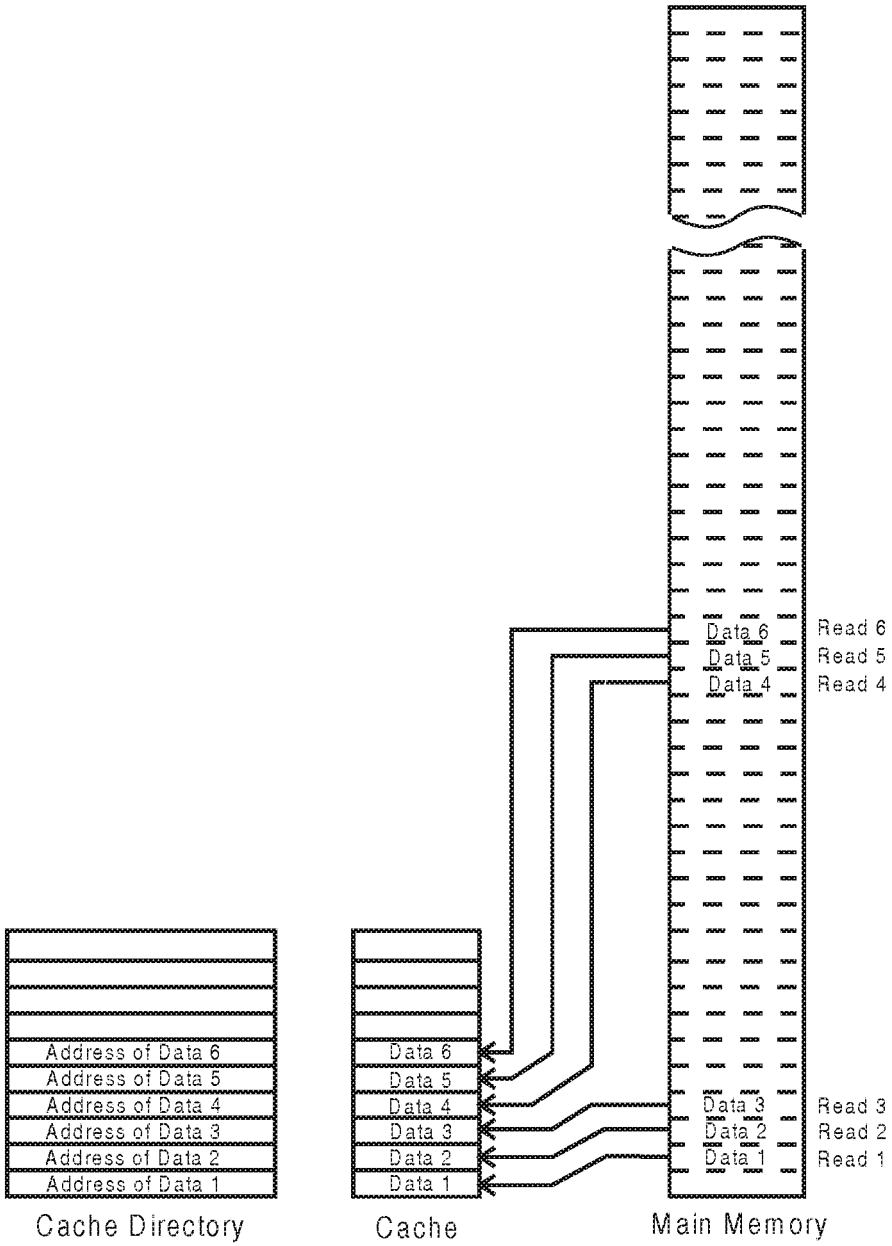


Figure 14-8. Fully-Associative Cache Organization

Direct-Mapped Cache (One-Way Set-Associative)

Direct-mapped caches require that only one directory entry be checked to determine whether the requested memory location is in cache. This means that cache hits and misses can be determined very quickly. Single look-ups are possible due to the way direct-mapped cache controllers organize cache and view memory.

Figure 14-9 is a simplified diagram of the memory structure in a direct-mapped cache system. Notice that the cache is subdivided into units called sets and each set has its own directory entry.

The cache controller views memory as logically divided into numerous pages each being the same size and having the same organization as the cache memory. Every location in memory occupies a given set position within a page. When a memory location is read, it is placed in cache at the same set location that it occupies in main memory. In other words, information from a given set in main memory is mapped into the same set within the cache. For a direct-mapped cache, each set has a single directory entry, therefore, the information stored in a set must come from one page in memory.

Direct-mapped cache structures allow the cache controller to identify the location of information in cache based on two criteria:

1. the set position that a location occupies in the cache (the set where the memory location comes from within a page)
2. the page number contained in the cache directory (the page number the location comes from in main memory)

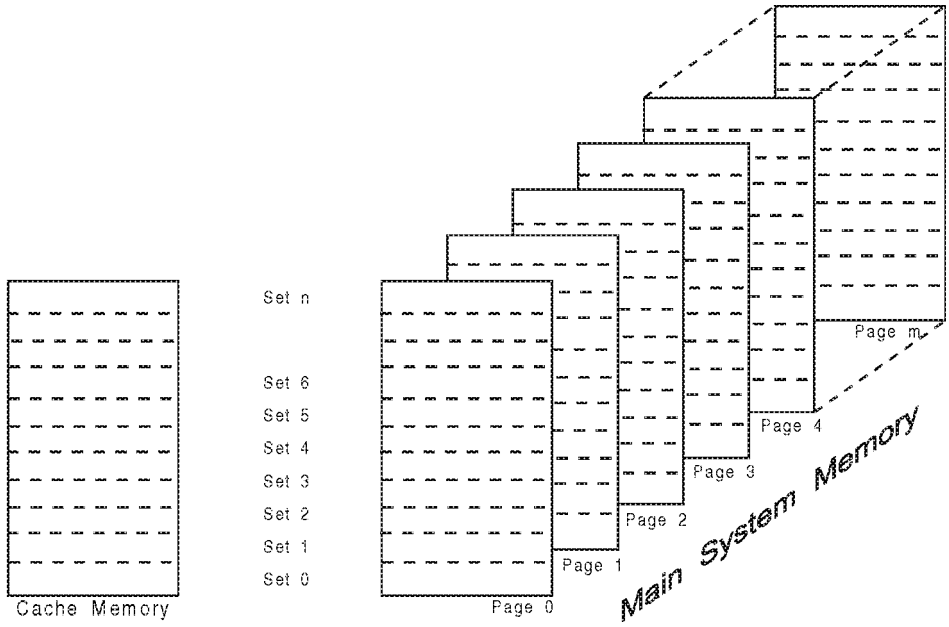


Figure 14-9. Set-Associative Organization—Direct-Mapped

Figure 14-10 is a simplified example of a direct-mapped cache organization and operation. This example illustrates how the cache would handle six read requests from locations not currently stored in cache. Each time the processor requests information from memory, the cache system compares the address to a directory entry. Each directory entry keeps track of which page in main memory information from each set comes from. Unlike the fully-associative cache, the processor need only compare the requested memory address with a single directory entry. The lower portion of the address identifies which set the location occupies in main memory, and thus identifies the directory entry that must be checked. The upper portion of the address is compared to the directory entry to determine if the memory page requested matches the entry. If they match, the request is a cache hit and if not it is a cache miss.

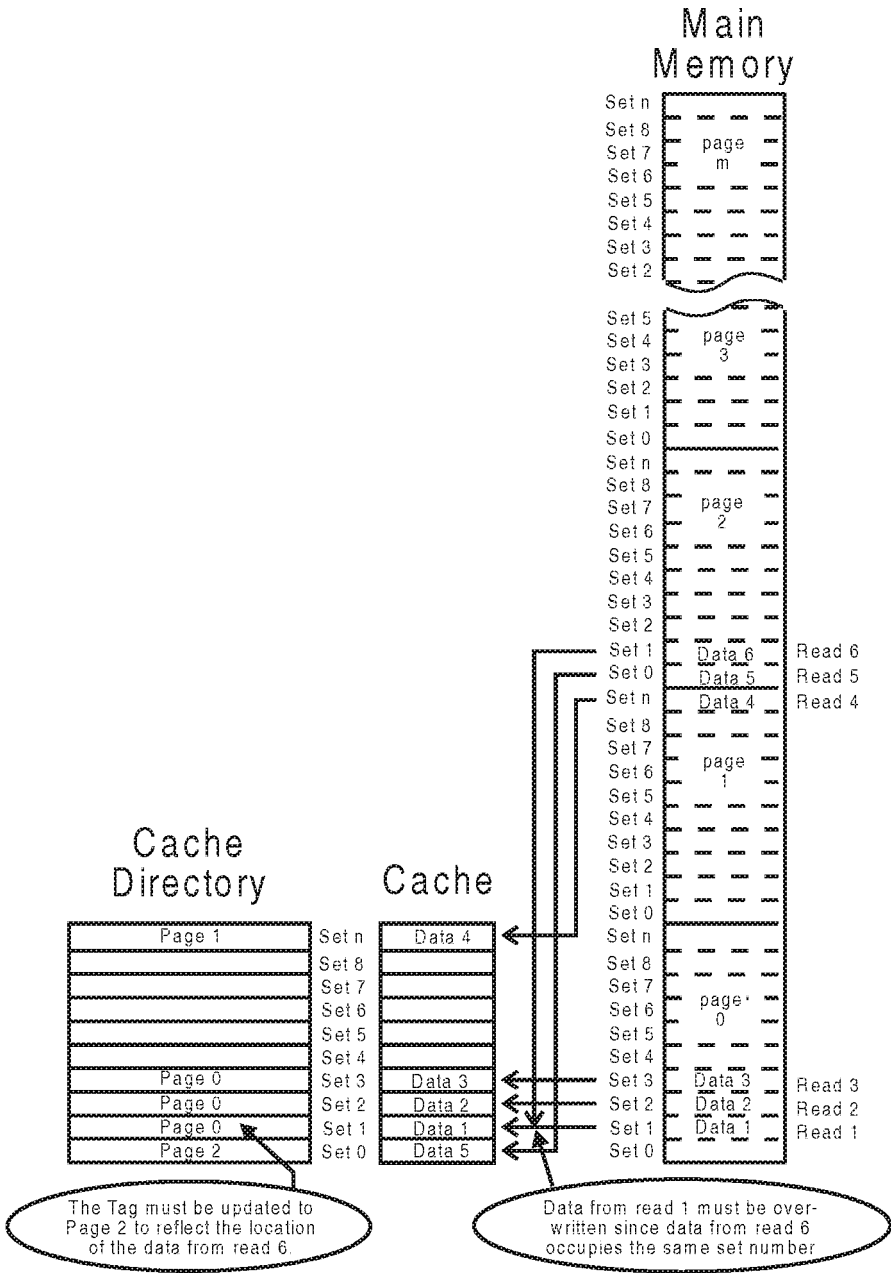


Figure 14-10. Direct-Mapped Cache Organization

Following is the sequence of events that occurs when each instruction is fetched from memory. Note that each of the following requests results in a cache miss and the information must be obtained from main memory.

1. The first request (read 1) is for information which resides in set 1 of page 0. Since the information read from memory is from set 1 of a given page, the information is stored in the set 1 position in cache memory. The corresponding directory entry for set 1 is updated to indicate that the information in set 1 of the cache is from page 0.
2. Similarly, read 2 is from a memory location that resides in set 2 of page 0 in main memory. This information is placed into the cache in the set 2 position and the corresponding directory entry is updated to indicate that the source page was 0.
3. Next, read 3 is accessed from set 3 of page 0 and is placed in the cache in the set 3 position and the corresponding directory entry is updated to page 0.
4. Read 4 jumps to a higher memory location that occupies set n (the last set within page 1. This information is stored in the set n position in cache and the directory for set n is updated to indicate page 1.
5. Read 5 comes from the first set (set 0) in page 2 and is placed in the set 0 position of cache. The corresponding directory entry is updated to page 2.
6. The last request from memory (read 6) is from set 1 in page 2. However, set 1 in the cache is already occupied by information from set 1 of page 0. Since no other set 1 position exists in the cache, the information currently in set 1 must be replaced. The new information from read 6 is placed in set position 1 and the directory entry is updated to page 2.

With a direct-mapped cache, two memory sets cannot be kept in cache at the same time if the sets come from different pages in memory. As you can see, a direct-mapped cache provides only one way to store information read from a given set.

Other cache organizations have more than one place to store information from a given set in cache. For example, the two-way set-associative cache provides two places, or ways, to store information that occupies the same set position in different memory pages.

Two-Way Set-Associative Cache

In a two-way set-associative organization, when the cache controller gets information from main memory, it has a choice of two cache memory banks to store the data in. Cache banks are frequently referred to as ways. In other

words, the controller must choose one of two ways to store the data in. Hence, the term two-way set-associative.

Figure 14-11 shows the same example used in the direct-mapped discussion to show the effect that two-way cache organizations can have on performance. Assume that the six memory reads access the same exact locations in main memory as before and that the total cache memory size is the same as the direct-mapped example in figure 14-10.

Notice that each cache way (A and B) has its own directory. Since there are two ways for information from a given set to be cached in, each memory request must be compared to two directory entries to determine whether or not the information is currently stored in cache.

When compared to the direct-mapped cache, the example two-way set-associative cache controller views memory as being organized in pages equal to one-half the size of the direct-mapped pages. The following steps illustrate the sequence of the six memory accesses. Note that the first three reads are identical to the direct-mapped example.

1. The first request (read 1) is for information which resides in set 1 of page 0. Since the information read from memory is from set 1 of a page, the information is stored in the set 1 position in cache way A. A subsequent section entitled "Least-Recently Used (LRU) Algorithm" describes howway A was selected to store the set. The corresponding directory entry for set 1 is updated to indicate information in set 1 of the cache is from page 0.
2. Similarly, read 2 is from a memory location that resides in set 2 of page 0 in main memory. This information is placed into way A in the set 2 position and the corresponding directory entry is updated to indicate page 0.
3. Next, read 3 is accessed from set 3 of page 0 and placed in way A in the set 3 position and the corresponding directory entry is updated to page 0.
4. Read 4 is from set n of page 3 and is placed in the set n position in way A.
5. The location for Read 5 occupies set 0 of page 4. This information is stored in cache way A in the set 0 position.
6. Read 6 comes from set position 1 within page 4 and cannot be stored in way A because it is already occupied with data from read 1. Data from read 6 instead is placed in set position 1 of way B.

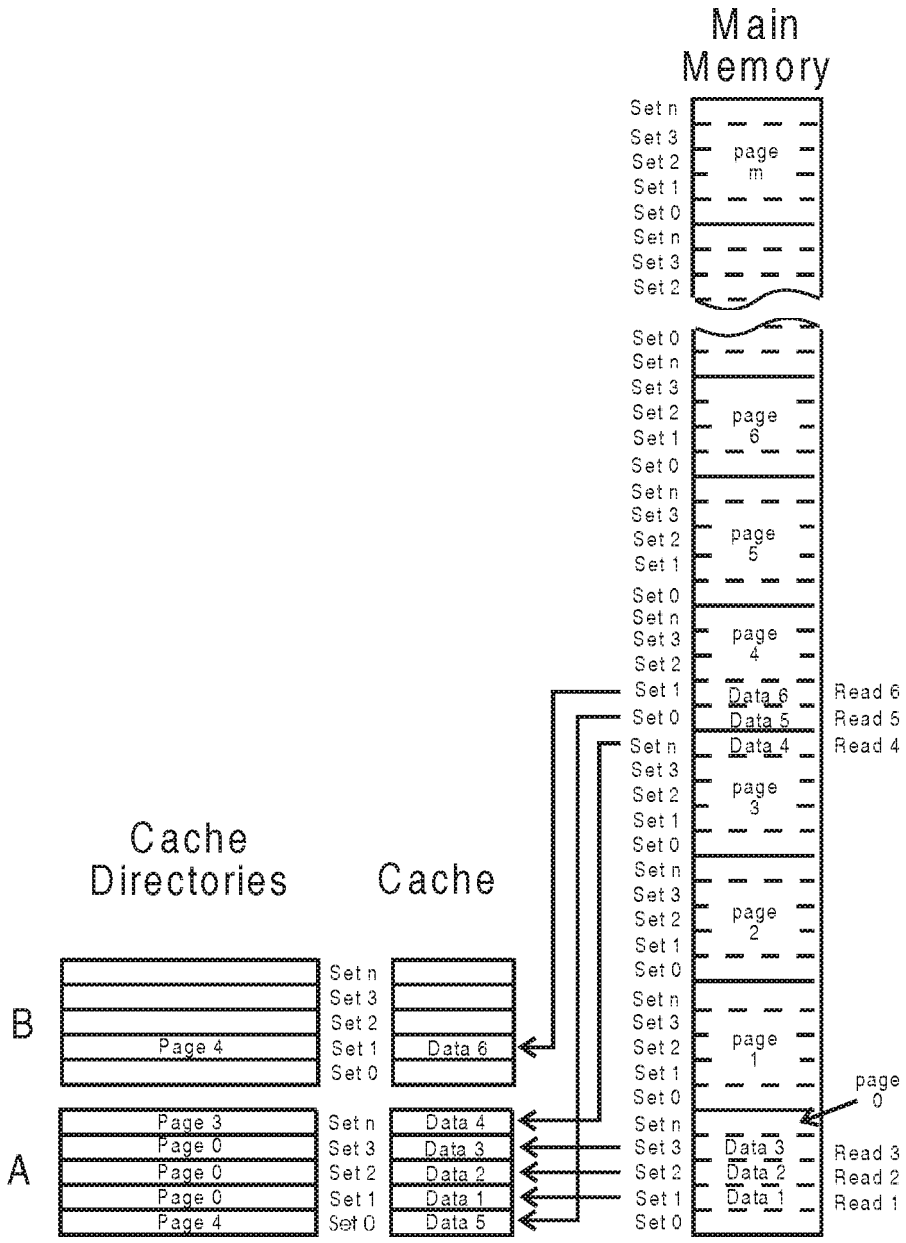


Figure 14-11. Two-Way Set-Associative Cache Organization

As you can see a two-way memory structure is more flexible than direct-mapped. The two-way set-associative cache was able to cache all six of the memory requests without overwriting an entry, while the direct-mapped cache could not. Generally speaking the two-way set-associative cache results in a higher percentage of cache hits.

Four-Way Set-Associative Cache

The four-way set-associative cache memory organization is simply an extension of the two-way structure. Cache memory is divided into four banks (ways) of equal size and main memory is viewed as divided into pages each equal in size to that of a way. The four-way structure provides even more flexibility than the two-way organization.

The four-way set-associative cache requires four entries to be checked to make sure the information is not in cache. Each time the number of cache ways increases, the number of look-ups increase proportionally. A cache with multiple ways usually implements a comparator that compares the target page address to all of the selected tags simultaneously (e.g., a four-way set associative cache compares the target page number to the four selected directory entries). Note that continuing to increase the number of ways would eventually lead to a fully-associative structure.

The internal cache of the 486 microprocessor is four-way set-associative. See the MindShare Architecture Series publication entitled *80486 System Architecture* from Addison-Wesley for details on the 486's internal cache.

Least-Recently Used (LRU) Algorithm

In two- or four-way caches, when all cache ways have valid information stored in a given set and new information needs to be stored in that set position, a decision must be made as to which information to replace. Most cache controllers use a least-recently used (LRU) algorithm to determine which cache entry to overwrite. The LRU logic keeps track of the cache locations within a set that have been least recently used. This is consistent with the principle of temporal locality, meaning that information that has not recently been used will not be likely to be used again soon.

Although the processor designer may implement an LRU algorithm that strictly keeps track of the least-recently used directory entry within the selected group, this can require a fairly substantial amount of logic. Rather, many designers

choose to implement a pseudo-LRU algorithm that makes a "best attempt" at keeping track of the least-recently used directory element. As an example, the 486 processor uses a pseudo-LRU algorithm.

Cache Line Size

When a cache miss occurs, the cache subsystem must go to main memory to get the requested information. The cache subsystem can be designed to fetch more information than was actually requested. The amount of information fetched from memory by the cache controller is defined as a line by Intel. Other manufacturers use the terms block or sector to define the amount of information the cache fetches in the event of a cache miss. Line size varies according to design, but typically ranges from 4 bytes to 32 bytes. The line is the smallest unit of information that a cache can map. This means that each time a cache miss occurs an entire line of information must be read from main memory. This is called a cache line fill.

For a given cache data RAM size, as the line size increases, the cache directory (tag RAM) size decreases. Since the cache directory is "overhead" and doesn't store data or code that the microprocessor can use, it makes sense to try to reduce the amount of tag RAM that the cache uses. The cache directory contains the numbers of the pages in main memory that lines come from. For example, a 4KB direct-mapped cache needs to be able to address 1,048,576 4KB-long pages in main memory (assume the cache is designed to handle addresses anywhere in a 4GB range, 4GB / 4KB = 1M.) Each of the cache directory entries is thus 20 bits wide to select 1 of 1M pages. Each of the 4KB-long pages consists of lines. Assume that the line size is 4 bytes. In this case, there are 1,024 sets in the cache (4KB cache data RAM / 4 bytes = 1,024 sets). Therefore, there must be 1,024 entries in the cache directory. Since each entry is 20 bits wide, the cache needs 1,024 x 20 bits or 2,560 bytes of SRAM for the directory. 2.5KB is a lot of overhead for a 4KB cache.

If the line size is increased to 16 bytes, the number of sets required drops to 256 (4KB / 16 bytes = 256). The width of the cache directory entries stays the same (20 bits) since the cache is still addressing 1M of 4KB-long pages in main memory. So, the cache needs 256 x 20 bits or 640 bytes of SRAM for the directory. That's 75% less.

Along with saving space and money, larger line sizes can increase cache performance, because a larger amount of information is stored as a result of each miss. Since instructions are normally requested from sequential memory locations (principle of spatial locality), a larger line size can result in a higher hit

rate. The cache line fill operation tends to prefetch instructions that the microprocessor will require in the near future, placing them in the high-speed cache before the microprocessor needs them.

Larger line size doesn't always mean better performance. There are limits to how large the line size can be before performance begins to suffer. A performance tradeoff of larger line sizes is that line sizes larger than the data path to main memory require several memory accesses to complete each cache line fill. This takes time and increases the system bus utilization. To counter this problem some cache subsystems are designed to do burst transfers when performing cache line fills. One example of such a cache is the internal cache in the 486.

Another performance tradeoff of larger line sizes is slack (wasted) space in the cache. If the line size is too large, then the cache may be storing many bytes that the microprocessor does not intend to access. Since the cache can only track complete lines, it must get all the bytes in a line, even if the microprocessor only wants one of the bytes in that line. The rest of the SRAM in the line is used up storing unwanted bytes. Statistically, the optimal line size for each cache design is based on the typical operating system/application mix and data structure requirements that the system is targeted for.

As discussed above, the larger the line size, the smaller the cache directory (but big line sizes can hurt performance). An approach that allows for small directories and good performance is the sectored cache. Intel defines a sector as a group of consecutively-addressed lines that share the same cache directory entry. Other manufacturers define sector and line differently, but the sectored cache concept is the same.

In a sectored cache, each sector consists of two or more lines. The lines must come from locations that are adjacent to one another in main memory because all of the lines in the sector have the same tag or page number in the cache directory. Although the cache directory only has one page number for the lines in a sector, it does have an individual set of state bits for each line. That means that the line is still the smallest unit of data that the cache can track.

Cache Size

The cache memory is relatively small compared to the main system memory. The size of the cache varies by design, but typically ranges from 4KB to 512KB and depends on whether the cache is a first- or second-level cache. The following are some general design considerations for first- and second-level cache subsystems.

First-Level Cache Size

To achieve the highest cache performance, information requested by the processor must reside in the first-level cache a high percentage of the time. The chances that information is in cache is related to cache size and organization. In general, a larger cache will give a higher hit rate and thus higher performance.

The rule of larger caches giving higher performance must be tempered in light of two factors: cost and locality of reference. Since SRAM, used in cache designs, is extremely expensive, tradeoffs must be made between cost of additional cache memory and increased performance. This tradeoff becomes particularly germane when considering the non-linear increase in performance as cache size increases. Refer to table 14-1. The information in this table is typical of results during design process simulations. Notice that additional cache memory provides only minimal increases in performance once cache size increases beyond 16KB. However, the optimum size of cache is highly dependent on how well the application fits the locality of reference model.

Table 14-1. Cache Size vs. Hit Rate

Size	Hit Rate
1KB	41%
8KB	73%
16KB	81%
32KB	86%
64KB	88%
128KB	89%

Second-Level Cache Size

The principle advantage of the second-level cache is that it can supply a line of information to the level one cache when it incurs a cache miss. The performance gains of second-level caches then relate to shortening the time needed for the first-level cache to perform a cache line fill. For example, if the first-level

cache has a hit rate of 85%, then 15% of all accesses to memory result in cache misses. A certain percentage of the misses from the first-level cache will result in cache hits at the second-level cache. Assuming the secondary cache also has a hit rate of 85%, then 85% of the 15% missed by the first-level cache result in cache hits. Overall then, 97.75% of all requests to memory are supplied by cache.

As is true for first-level caches, second-level caches enhance performance due to the locality of reference principles. Second-level caches must hold copies of information that the first-level cache misses. In other words, second-level caches hold information currently in the first-level cache along with information that was previously in the first-level cache (but was subsequently eliminated to make room for new information). By design then, second-level caches must be several times larger than the first-level cache so that information previously held in the first-level cache can be obtained from the second-level cache when it is needed again.

It should be apparent that the optimum size of any cache depends on the nature of the programs that access memory for code and data. Some manufacturers spend considerable time, money and effort to understand the nature of memory access in various operating environments so that the best cache can be designed and implemented.

Cache Addressing

Caches in x86-based systems are physically addressed. That means that the memory management unit (the segmentation unit and the paging unit) translates the logical address into the physical address before the physical address is submitted to the cache for a look-up.

Some other computer architectures submit the address to the cache before page translation is done. This speeds the access to the cache because it saves a step. Such caches are virtually-addressed instead of physically-addressed.

I/O Information Not Cached

Cache controllers do not cache information read from I/O addresses. It doesn't make sense to do so. Consider the scenario wherein a cache controller keeps a copy of the printer status that is returned in response to an I/O read from the printer status I/O port. If the microprocessor requests the printer status at some time in the future, the request would be honored from the cache, and the

program would be given the status of the printer five minutes ago. In fact, the printer may have run out of paper two minutes ago.

When the cache controller intercepts a processor-initiated I/O bus cycle, it immediately initiates the bus cycle on the system bus. When the I/O data is read from the I/O device, it is returned to the microprocessor and a copy is not kept in the cache.

Non-Cacheable Memory

Certain areas of memory space must be designated as non-cacheable address space. If the cache controller were to attempt to keep copies of information from these areas of memory space, it may end up with stale information that doesn't accurately reflect the actual contents of main memory.

A network controller card with dual-ported RAM is a good example. The network controller card can receive information over the network and write it into this memory without using the bus. The system board microprocessor can also access the memory. If the cache controller on the system board were to cache information from this area of memory, it would be unable to detect memory alterations performed by the network controller using its on-card buses. The information in the cache would no longer be coherent with main memory.

The system board must incorporate a memory address decoder that detects non-cacheable memory addresses and tells the cache controller not to keep a copy of information read from these memory locations. The requested information should be read from main memory and given to the microprocessor without making a copy in the cache. The 82385 cache controller has an input called NCA# (non-cacheable access) that is used to tell it when the microprocessor is accessing a non-cacheable memory location. The system board designer can accomplish this in one of two ways:

- The designer may set aside one or more pre-defined memory address ranges as non-cacheable space. Any expansion cards with addressable dual-ported memory should then be configured so as to map their on-board RAM into this space. The system board would include a memory address decoder designed to detect accesses within these fixed memory address ranges and assert NCA# to the cache controller.
- The designer may include a programmable memory address decoder on the system board. Each time the machine is powered up, the programmable address decoder is automatically configured to recognize the memory

address ranges used by expansion cards with non-cacheable memory on-board.

Testing Memory

When running main memory diagnostics, the cache controller can cause problems by intercepting memory reads and never letting them propagate out to main memory. The diagnostic would assume that data is being read from main memory when, in fact, it is being returned from the cache.

In order to ensure that this problem doesn't occur, cache controllers typically include an input called FLUSH#. When asserted, it causes the cache controller to clear the valid bit in every directory entry to 0, thereby invalidating all cache entries. All of the valid bits remain cleared as long as the FLUSH# input is kept asserted. Typically, the FLUSH# line is asserted or deasserted by writing the appropriate value to an I/O location that has been set aside for this purpose. The I/O location used and the data necessary to assert or deassert FLUSH# is system design-specific.

When FLUSH# is asserted, every memory read initiated by the microprocessor results in a memory read bus cycle, causing main memory to be accessed for every memory read.

Chapter 15

The Previous Chapter

Prior chapters covered the subject of RAM memory and memory cache subsystems.

This Chapter

This chapter covers the various types of ROM memory encountered in ISA systems. This includes PROM, MROM, EPROM, EEPROM and flash EEPROM. In addition, the subjects of ROM diagnostics, shadow RAM, double-mapping, and device ROMs are covered.

The Next Chapter

This chapter concludes Parts 1 and 2, which describe the background technology necessary to a complete understanding of any ISA machine. Part 3 follows this chapter and covers the ISA architecture. This includes ISA history, the ISA bus structure, ISA bus cycles, the interrupt subsystem, direct memory access (DMA), bus mastering, the real-time clock and configuration RAM, the keyboard/mouse interface, numeric coprocessors, and the ISA timers.

ROM Memory — Theory of Operation

Introduction

ROM stands for read-only memory. ROM memory can only be read once it is installed in a system and it is non-volatile — it retains its stored information indefinitely after power is removed from the system.

As illustrated in figure 15-1, the ROM memory chips used in ISA machines contain one byte of information per internal storage location. Each storage location consists of eight bit cells, each of which can store a 1 or a 0.

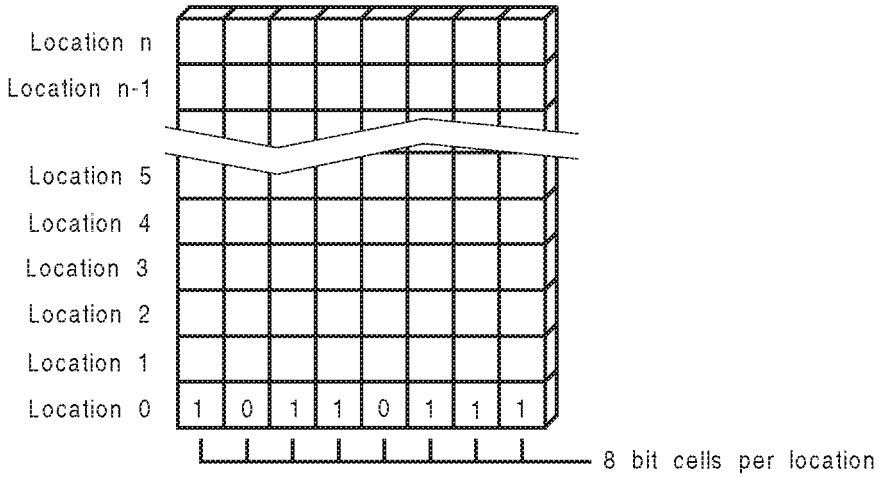


Figure 15-1. The Bit Cell

The information stored in a ROM typically consists of both programs and data to be used by the programs. The programs and data are developed by a programmer and are then assembled or compiled into machine language. Machine language is the series of hex numbers that the microprocessor treats as instructions and data.

The machine language representation of the programs and data is then loaded into a device called a ROM programmer. This is typically accomplished by transmitting the information to the ROM programmer using a serial or parallel communications link.

The ROM is inserted into a socket on the ROM programmer and the information is written into it. Once placed in the target unit, such as an ISA system, the ROM may only be read from.

The following sections discuss the types of ROM typically used in ISA machines:

- Fusible-Link PROM
- Masked ROM, or MROM
- EPROM
- EEPROM
- Flash EEPROM

Fusible-Link PROM

Figure 15-2 represents one storage location in fusible-link PROM (programmable read-only memory). Every bit cell contains a fuse, or fusible-link. In order to store the programs and data into the PROM's storage locations, it must be inserted into the socket on the ROM programmer. When the programming process begins, a byte is written into each location by blowing, or vaporizing, the fuses in some bit cells, while leaving others intact.

When the PROM is installed in an ISA system, the microprocessor may only read from it. When a PROM storage location is read from, the pattern of ones and zeros represented by the eight fuses is simultaneously output onto the data bus and sent back to the microprocessor.

Because some of the fusible links are physically vaporized during the programming process, a PROM may not be re-programmed with new information after its initial programming. For this reason, if a systems manufacturer intends to update the information stored in the ROMs periodically, PROMs should not be used.

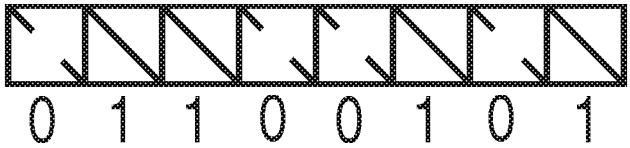


Figure 15-2. The PROM Bit Cells

Masked ROM (MROM)

The masked ROM is a variation of the fusible-link PROM. If the systems manufacturer is absolutely certain that the information in the ROMs will never change, it may choose to have the ROM chip manufacturer pre-program the information into the ROMs for them. The systems manufacturer sends the memory chip manufacturer a copy of the information to be stored in the ROM. The memory chip manufacturer then produces a photographic mask that represents the opens and shorts that would be produced by blowing some fuses and leaving others intact. This mask is then used to etch thousands of pre-programmed ROMs that are then shipped to the systems manufacturer.

Eraseable Programmable Read-Only Memory (EPROM)

If the systems manufacturer wants to reserve the option of re-programming the ROMs at some time in the future, EPROMs should be used instead of PROMs or Masked ROMs.

EPROM stands for erasable programmable read-only memory. When received from the memory chip manufacturer, the EPROM is blank (contains no information). As with the PROM, information is written into it using a ROM programmer.

Figure 15-3 illustrates the heart of the EPROM bit cell. At the bit cell level, the EPROM basically consists of a transistor. A transistor is nothing more than a switch. With a certain voltage, or charge, on its gate, the transistor provides a path for current flow through its source and drain. In other words, the switch is closed, allowing current to flow. With a different charge on its gate, however, the transistor ceases to conduct.

The gate of the transistor is floating, surrounded by a thin layer of silicon dioxide. Since silicon dioxide is an insulator (it doesn't conduct electricity), a charge placed on the floating gate doesn't have anywhere to go and tends to stay there.

When the EPROM is placed in the ROM programmer and the programming process is initiated, a bit is written into a bit cell by either charging the floating gate or discharging it. The charge on the floating gate may be changed by applying a large positive or negative voltage to either the control gate, source or drain. At the completion of the programming process, some of the bit cells have had charges placed on their floating gates and some have not.

The EPROM is removed from the ROM programmer and placed in the system. When the microprocessor attempts to read from one of the EPROM's storage locations, some of the eight transistors that make up the storage location conduct, while others do not as a result of the charge (or lack of a charge) placed on their floating gates. It acts just like a series of eight switches, producing a group of eight bits of information. The eight bits of information are then driven out onto the data bus and sent back to the microprocessor.

When the information in the EPROM must be altered, the EPROM is removed from the system and placed under a special ultraviolet light. An EPROM can be easily identified by the small, rectangular crystal window on its upper surface.

The EPROM is left under the light for approximately twenty-one minutes. During this period of time, every floating gate is bombarded with high energy ultraviolet light and a charge begins to build on every gate. Eventually, by the end of the twenty-one minute period, every floating gate has built up a charge sufficient to jump the thin layer of silicon dioxide and discharge. In this way, all of the data in the EPROM is cleared. The EPROM is then removed from the light source and may be re-programmed with new information.

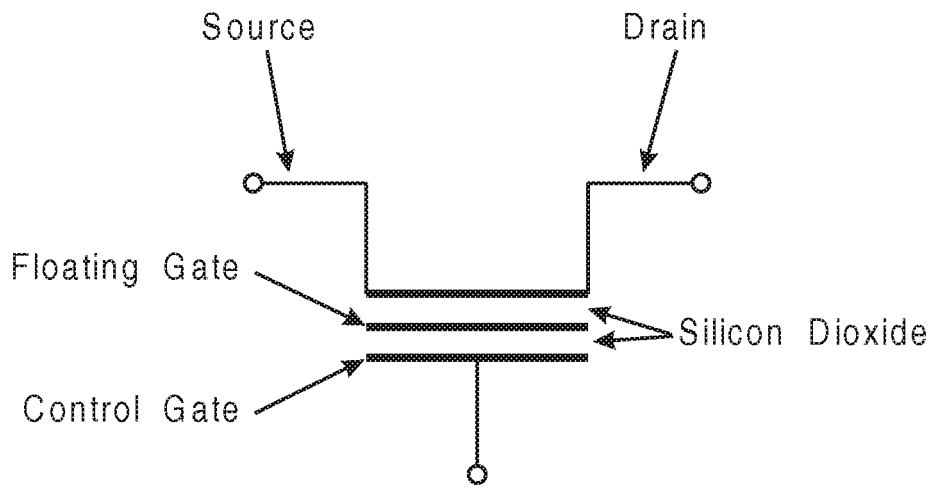


Figure 15-3. The EPROM Bit Cell

Electrically Eraseable Programmable Read-Only Memory (EEPROM)

In operation, EEPROM memory is very similar to EPROM. Each bit cell contains a transistor with a floating gate. There is no crystal window, however, so the EEPROM cannot be erased by ultraviolet light. Rather, the EEPROM is erased by placing a special programming voltage on its PROGRAM pin and addressing a specific location. The location is then erased and a new byte of information may be written into it. Each location to be altered must addressed, erased and re-written. A properly designed PC allows the EEPROM's contents to be altered while it is still installed in the machine.

The drawback arises in erasing each location, as it takes a considerable amount of time. Typically, the erase/rewrite time is in the millisecond range. The EEPROM is best used in applications where only a few bytes will be altered at a time (such as for storage of configuration information).

Flash EEPROM

The flash EEPROM is a variation on the EEPROM. Rather than erasing and rewriting a location at a time, the flash EEPROM is flash erased — the entire contents of the flash EEPROM (or a large block of it) is erased simultaneously. New data may then be written into the flash EEPROM. The erasure/rewrite time is considerably longer than the EEPROM due to the large bulk of information to be updated. Due to this operational characteristic, the flash EEPROM is better suited to applications wherein large amounts of information are altered at a time, while the EEPROM is better suited to applications where only a few bytes are altered at a time. A typical application of flash EEPROM would be as a substitute for a floppy drive for storing program and data files.

Flash EEPROMs cost approximately 75% less than an EEPROM of comparable bit density. As a result, EEPROM is considered to be too expensive for PC applications. As the bit density of flash EEPROMs has increased, they have become quite attractive to PC system designers. On the other hand, typical flash EEPROMs require both a 5Vdc and a 12Vdc power supply, while the typical EEPROM only requires a 5Vdc power supply. This means that flash EEPROM requires a more complex power supply, adding cost to the system. TI and several other chip vendors are developing flash EEPROMs that only require a single 5Vdc power supply, thus removing this disadvantage.

ROM's Interface to System

Figure 15-4 illustrates the ROM interface to the system. It consists of the signals described in table 15-1.

Table 15-1. The ROM's Interface Signals

Signal(s)	Description
CE#	**Chip-Enable or Chip-Select**. This signal is asserted by the ROM address decoder when it detects a memory address within the range of addresses assigned to the ROM. When CE# is asserted, the ROM uses its address inputs to select the target location to be read.
Address Lines	Once CE# is detected, the ROM uses the address present on its address inputs to select the target location to be read.
OE#	**Output Enable**. When OE# is asserted, the ROM outputs the byte of information from the target location onto the data bus lines to be sent back to the microprocessor.
Data Lines	The data lines are used by the ROM to send the byte read from the target location back to the microprocessor.

As an example, assume that the ROM in figure 15-4 is a 64KB ROM and the ROM's address decoder is designed to detect the memory address range from F0000h to FFFFFh. The ROM address decoder observes the upper address lines, A[19:16], to detect an Fh as the uppermost digit in the memory address. When an Fh is detected, the ROM address decoder chip-selects the ROM by asserting its CE# input. The ROM responds by using its address inputs, A[15:0], to select the exact location within the 64K locations of storage within the ROM. If the ROM's output enable (OE#) input is also asserted, the ROM reads the byte from the target location and drives it onto the data bus lines to be sent back to the microprocessor.

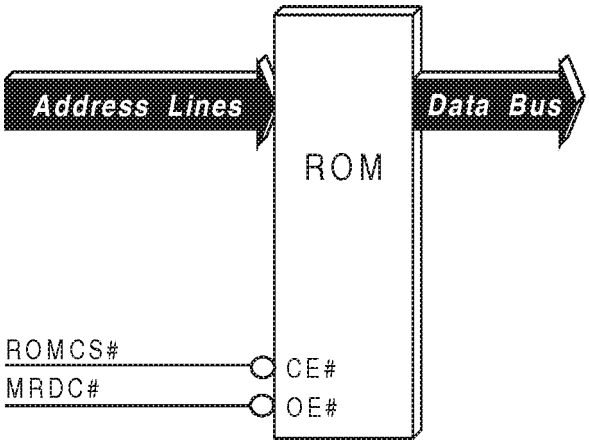


Figure 15-4. The ROM's Interface to the System

System Board ROM Memory

Many ISA system boards contain 128KB of ROM memory. This ROM contains the following:

- the power-on self-test, or POST.
- The BIOS routines for the standard I/O devices integrated onto the system board.
- The interrupt service routines for the standard I/O devices integrated onto the system board.
- The BASIC language interpreter that is executed if the boot program fails to find a valid operating system on either the floppy or hard drive (the BASIC language interpreter is only included in some ISA machines).
- The boot program used to read the operating system software, such as MS-DOS or OS/2, from the floppy or hard drive.

Testing

Checksumming is typically used to check ROMs for proper operation. When the programmer originally develops the programs and data to be written to the ROM, a simple utility program is used to add the contents of every ROM location together. As the utility program is running, it keeps the result of the additions (the running total) in an 8-bit register. An 8-bit register can't hold values larger than FFh or 255d, so an overflow occurs when the running total would exceed FFh. The utility program ignores the overflow. The following example illustrates the process:

```
Byte 1:          1011 1001b
Byte 2:        + 1001 1110b
Running Total:   0101 0111b
```

The programmer always stores a value in the last location in the ROM that will force the result to zero. As an example, assume that adding every ROM location but the last resulted in DAh (1101 1010b). The programmer would store the following value in the last ROM location to cause the overall result to be zero:

```
Running Total:        1101 1010b
Last Byte:          + 0010 0110b
Resulting Checksum:   0000 0000b
```

During the POST, the program reads the contents of every system board ROM location and adds them together. If the final result isn't zero, either the data in the ROM has changed, which it never should, or the system cannot properly read from the ROM. In either case, the ROM cannot be reliably read from and a ROM checksum error is generated.

In some ISA systems, a slightly different method is used. The POST adds the contents of every ROM location together, accumulating a 16-bit running total. The correct value is stored in the last two locations of the ROM to force the running total to 0000h when the last two bytes are added. If the final result isn't zero, a ROM checksum error is generated.

Checksumming only tests the contents of the ROM once, during the POST. This is usually sufficient because ROM is quite reliable and it is unlikely that a cell will ever fail in the useful lifetime of the machine. A primary reason that the POST even bothers to test the ROM is because there may be address conflicts in the system that make it impossible to reliably access the ROM. The conflicts occur because multiple cards may be decoding the same ROM addresses. There's no clearinghouse that says which addresses a manufacturer should use for the device ROMs on ISA expansion cards. Those conflicts show up as data bus contention during the checksum calculation, resulting in a garbled running total. There's only one chance in 256 (0.4%) that the checksum will fail to catch the conflict.

However, in systems where reliability is paramount, parity bits can be stored along with the data bytes in the ROM when the ROM is initially programmed. After that, every time any bytes are read from the ROM, a parity checker can check for failures in the ROM. If a failure occurs, it may not be the fault of the ROM. Instead, it may be noise or crosstalk on the system's data bus which garbles the data between the time it leaves the ROM and the time it reaches the bus master. This can occur in very high speed applications. For example, parity-protected ROM is often used to store the microcode (inside the microprocessor chip) that runs the microprocessor's execution unit.

Note that the DRAM parity generator/checker is part of the DRAM controller and it normally only checks for parity when reading from RAM. Therefore, device ROMs on ISA expansion cards do not need parity bits.

Shadow RAM

Many ISA systems based on the 80386DX or SX microprocessors copy the system board ROM contents into shadow RAM during the POST. This is done to compensate for the notoriously slow access time of ROMs.

If an applications program depends heavily on the BIOS routines in ROM, the instructions that make up the BIOS routines must be fetched from system ROM frequently. Since ROM memory has a notoriously slow access time relative to that of RAM memory, the instructions could be read more quickly and the BIOS routine therefore executed more quickly if the BIOS routines were located in RAM rather than ROM. In addition, the shadow RAM is a 32-bit device, while the ROM is typically implemented as a 16-bit device. Information stored in shadow RAM can therefore be accessed at double the rate of the ROM.

Although there may be plenty of extra RAM in the system to hold a copy of the ROM's contents, the applications programs and system software that make use of the ROM's contents must know where the ROM is. Those programs assume the ROM is at E0000h to FFFFFh. That means it would be a wasted effort to copy the ROM to RAM that had a different address after the copy was complete. There are several solutions to this problem and different systems use different solutions.

Shadow RAM and ROM Occupying Different Address Spaces

One solution is to use the 80386's virtual page translation feature to translate the ROM's address range into RAM addresses. A system software program would first copy the ROM into RAM and then set up the page directory and page table with entries that point to the RAM that contains a copy of the ROM's contents. Subsequently, when a program submits a ROM address to the memory management unit (MMU), the paging unit inside the MMU translates the ROM address to the address of the RAM that contains the ROM's contents. Programs are unaware that page translation is occurring. The address that the program submits to the MMU is called the linear address. The MMU translates the linear address to the physical address. Page translation is described in the chapter entitled "The 80386DX and SX Microprocessors."

Compared to the other solutions described below, page translation has the benefit that it does not require special address decoders or DRAM controllers. The limitations of this approach are that the microprocessor must be an 80386 or higher (the 80286 does not have a paging unit), it must be running in pro-

tected mode (page translation is not available in real mode) and it must be in virtual-8086 mode (to run code in ROM that was written for real mode). The following solutions do not have these limitations.

Shadow RAM and ROM Occupy Same Address Space

Two devices occupying the same address space is a unique situation. Any memory read within the ROM's assigned address range accesses the system ROM, while any memory write within the same range accesses the top 128KB of the RAM, known as the shadow RAM.

After POST has verified that RAM is working correctly, the complete contents of the 128KB system ROM is copied to RAM memory. This is accomplished by performing a read from each location in the 000E0000h to 000FFFFFh address range followed immediately by a write to the same location. In this manner, the data is read from the ROM location and written to the RAM location with the same address. Reading and writing every address in the range places a complete copy of the system ROM in the shadow RAM.

After the copy has been completed, the programmer uses an I/O control register to disable the ROM address decoder and make the shadow RAM read-only (as if it were ROM). Any subsequent calls to the BIOS or interrupt service routines now access the shadow RAM rather than the ROMs. The substantially faster RAM access time causes the BIOS routines to be executed more quickly from this point forward and can result in substantial speed increases when running applications programs that call the BIOS routines frequently.

Double Mapping ROM and Shadow RAM Address Space

This method of implementing shadow RAM designates an area at the very top of the microprocessor's address map (just below 16MB or 4GB) for the shadow RAM. After the ROM is copied into this RAM area, all addresses that are intended for ROM are redirected to the location in RAM where the ROM contents have been copied.

After the POST has verified that RAM is working correctly, the complete contents of the 128KB system ROM is copied to shadow RAM. This is accomplished on an 80286- or 80386SX-based system by performing a read from each location in the 000E0000h to 000FFFFFh address range followed immediately by a write to the new location in the 00FE0000h to 00FFFFFFh range. Similarly, a system based on the 80386DX or higher processor would do the writes to the new location in the FFFE0000h to FFFFFFFFh range. In this manner, the data is

read from the ROM location and written to the shadow RAM location. Reading and writing every address in the ranges places a complete copy of the system ROM in the shadow RAM.

After the copy has been completed, the programmer uses an I/O control register to modify the operation of the ROM address decoder so that the ROMs are never chip-selected again and all accesses within the ROM address range now accesses the shadow RAM instead. The shadow RAM is also write-protected so that it is read-only (as if it were ROM). Any subsequent calls to the BIOS routines now access the 32-bit shadow RAM rather than the ROMs. The substantially faster RAM access time causes the BIOS routines to be executed more quickly from this point forward and results in substantial speed increases when running applications programs that call the BIOS routines frequently.

Recovering Unused ROM Address Space

Although IBM reserved ranges of addresses in the first megabyte of memory for use by ROMs (boot ROM, option ROM and device ROMs), those ranges of addresses are not always fully occupied. For example, 128KB of space is reserved for device ROMs, but many systems have only a couple of small (2KB or so) device ROMs in them. That leaves a lot of unused address space. Often it is possible to map RAM into the addresses that were reserved for ROM so that MS-DOS and other real-mode applications can access more than the 640KB of RAM that IBM specified.

To determine if an address is actually assigned to a ROM, a system software program will do a hot memory search, writing and reading values to and from memory. If a location can be altered, it is assumed to be RAM. If the location cannot be altered, it is assumed to be either vacant or ROM. If the location and its neighboring locations contain FFh (all ones), it is assumed to be an address that is vacant because the ISA data bus floats high (to ones) when no device is driving it. The vacant addresses can be reassigned to RAM. The assumptions that the memory search makes are not foolproof and the memory search can cause the system to crash if a memory-mapped I/O location is accessed. If the system survives the memory search without crashing, RAM will be mapped to the vacant ROM addresses.

Real mode programs that were written to run on an 8086 or 8088 microprocessor only intend to produce addresses from 00000h to FFFFFh. In order for these programs to access more than 640KB of RAM, the RAM must have addresses below 100000h (1MB), from the program's perspective.

Using page translation and virtual-8086 mode, the system software can provide an exclusive 1MB linear address range for the real-mode program to use. The program believes it is accessing the first physical megabyte of RAM, but in fact the paging unit translates the linear address to a physical address that may be anywhere in the microprocessor's physical address range.

If the system uses one of the other ROM shadowing techniques that is compatible with the 80286 microprocessor and real mode, the unused ROM address space can still be recovered. The DRAM address decoder is programmed to map a block (usually 16KB at a time) of RAM into the unused ROM addresses. The mapped RAM is then tested at these new addresses to ensure that the memory search was not fooled when it found the vacant locations.

These blocks of ROM addresses that are freed up so that real-mode applications can access more RAM are called upper memory blocks (UMB).

Along with accessing upper memory blocks, real mode applications that were written for the 80286 microprocessor can access almost 64KB of extended memory in the high memory area (HMA). HMA addresses start at 1MB. See the section entitled "Accessing Extended Memory in Real Mode" in the chapter entitled "The 80286 Microprocessor" for information on accessing the HMA.

32KB System Board ROM Configuration

In figure 15-5, the ROMs are divided into two pairs, one of which responds to the address range from E0000h to EFFFFh and the other to the address range from F0000h to FFFFFh. Each of these pairs consists of two 32KB ROMs designated as the upper or odd ROM and the lower or even ROM.

The upper or odd ROM is called the upper ROM because it is connected to the upper data path, D[15:8]. It is also called the odd ROM because all of the odd-addressed ROM locations are within this ROM. This is evident because it is connected to the upper data path and the microprocessor always communicates with odd-addressed locations over the upper data path.
The lower or even ROM is called the lower ROM because it is connected to the lower data path, D[7:0]. It is also called the even ROM because all of the even-addressed ROM locations are within this ROM. This is evident because it is connected to the lower data path and the microprocessor always communicates with even-addressed locations over the lower data path.

When the programmer loads data into the ROMs using the ROM programmer, a utility program is first executed that divides the information in half. All of the

information to be stored in even addresses is programmed into the lower or even ROM, while all information to be stored in odd-addressed locations is programmed into the upper or odd ROM.

Whenever an address in the range from E0000h to EFFFFh is detected by the system board ROM address decoder, the EXXXXCS# line (chip-select for E0000h to EFFFFh range) is asserted by the decoder. This selects the EXXXX ROM pair. The two ROMs then use the lower part of the address from the microprocessor, A[15:1], to select one of 32K locations in each of the two ROMs. The system's bus control logic asserts the memory Read command line (MRDC#) which is connected to the Output Enable pin (OE#) on both ROMs. Each of the ROMs outputs the byte from the addressed location onto the data bus when OE# and CE# are both asserted. The lower or even ROM outputs the byte from the even address onto the lower data path, D[7:0], while the other ROM outputs the byte from the next sequential odd address onto the upper data path, D[15:8]. Even though the FXXXX ROMs also see their OE# pins asserted, they will not interfere with the transaction because their CE# pins are deasserted. The FXXXX ROM pair responds to the address range from F0000h to FFFFFh.

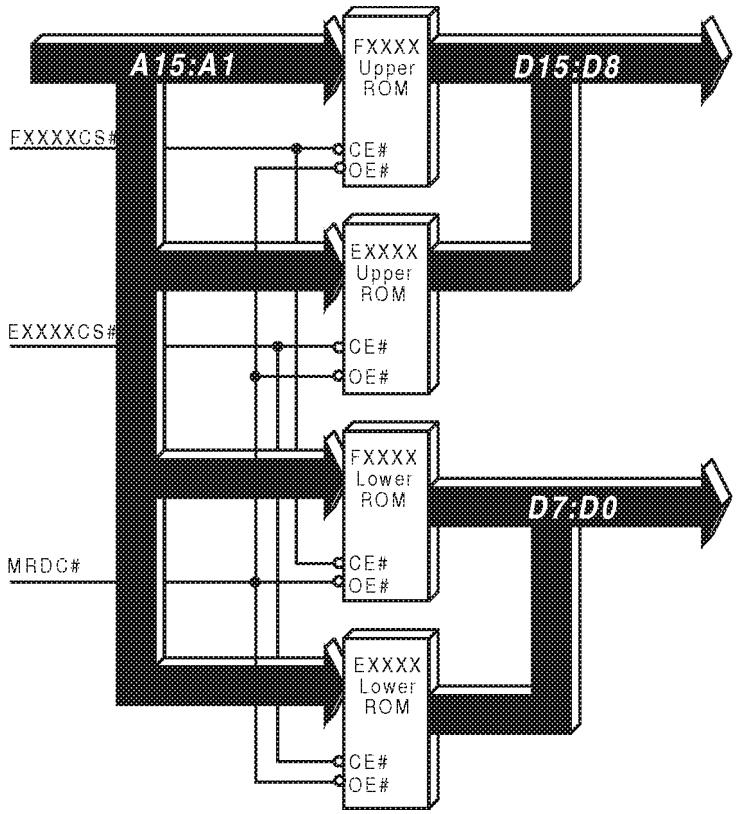


Figure 15-5. System Board ROM Configuration with 32KB ROMs

64KB System Board ROM Configuration

In figure 15-6, there are two 64KB ROMs that respond to the address range from E0000h to FFFFFh. They are designated as the upper or odd ROM and the lower or even ROM. This designation is explained in the previous section.

Whenever an address in the range from E0000h to FFFFFh is detected by the system board ROM address decoder, the ROMCS# line (ROM chip-select) is asserted by the decoder. This selects the ROM pair. The two ROMs then use the lower part of the address from the microprocessor, A[16:1], to select one of 64K locations in each of the two ROMs. The system's bus control logic asserts the memory read command line (MRDC#) which is connected to the output enable

pin (OE#) on both ROMs. Each of the ROMs outputs the byte from the addressed location onto the data bus when OE# and CE# are both asserted. The lower or even ROM outputs the byte from the even address onto the lower data path, D[7:0], while the other ROM outputs the byte from the next sequential odd address onto the upper data path, D[15:8].

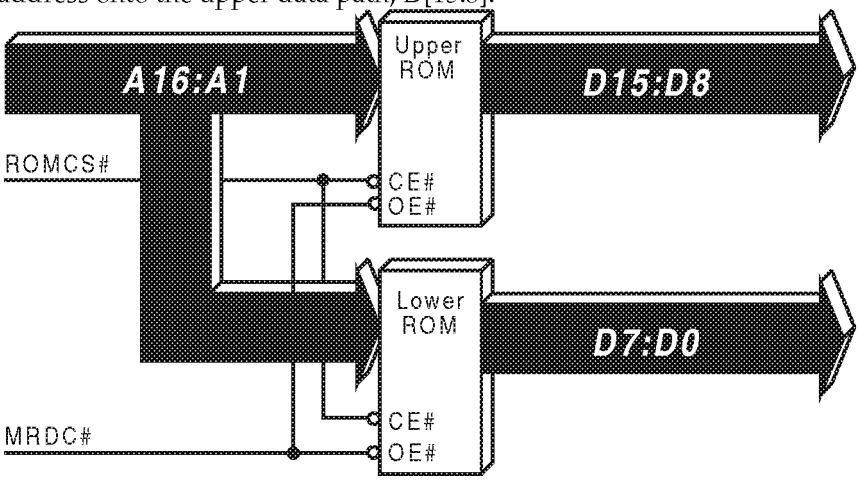


Figure 15-6. System Board ROM Configuration with 64KB ROMs

ROMs on ISA Cards (Device ROMs)

As already pointed out, the system board ROM contains:

- the POST for the standard devices.
- the BIOS routines for the standard devices.
- the interrupt service routines for the standard I/O devices.

A card designed to be installed in an ISA slot also needs to be tested during POST and may require a BIOS routine and an interrupt service routine. The card designer can include a ROM on the ISA card and place its self-test program, BIOS routine and interrupt service routine in it. This is called a device ROM.

ISA systems provide a way to automatically execute the ISA card's self-test during POST and to automatically update the interrupt table in RAM to point to the card's BIOS routine and interrupt service routine.

During the POST, a process called ROM scan is performed. Starting at memory address C0000h and going through DFFFFh, the POST program scans for device ROMs. The process is simple:

1. The POST reads two bytes starting at memory location C0000h. If the byte from C0000h is 55h and the byte from C0001h is AAh, then the POST has found the signature of a device ROM on an ISA card.
2. POST reads the next byte from location C0002h. This location contains the length of the device ROM in increments of 512 bytes. As an example, a 2KB device ROM would have a 04h in its third location (4 x 512 = 2048, or 2KB).
3. Knowing the start address and the length of the device ROM, the POST then checksums it to ensure it can be read from properly. Checksumming is described in the section entitled "Testing." The result should be zero. If it isn't, a checksum error is displayed on the screen along with the segment start address. As an example:

   CHECKSUM ERROR C000:0000 is interpreted as a device ROM checksum error for an ISA card whose device ROM starts at memory address C000:0000 (segment:offset format).
4. If the ROM checksum is OK, the POST program does a "call" to the fourth location in the device ROM, in this case, location C0003h. This is the start address of the initialization code, or program, in the ROM. A "call" is similar to a jump, except that by using a call, the system remembers where to return to when the device ROM program is finished.
5. The initialization code:

   - Runs a self-test on the card.
   - Writes the start address of its BIOS routine into the proper slot in the interrupt table.
   - Writes the start address of its interrupt service routine into the proper slot in the interrupt table.
   - Initializes any programmable I/O registers on the card to set it up for normal operation.
   - Returns to the POST program.

In this way, the device ROM is automatically detected and tested, and its BIOS and interrupt service routine pointers are installed in the interrupt table.

The POST then adds 2K (0800h) to the start address, C0000h, and checks for another ROM starting at C0800h. If another device ROM is detected, the process is repeated. This process is repeated at 2K address intervals up to location DFFFFh, the end of the ROM Scan address range.

When the ROM scan is completed, the POST has automatically detected, tested and executed the initialization programs for every installed ISA card that has a device ROM on it.

Part 3

The Industry Standard Architecture

Part 1, "The System Kernel," covered the mechanism utilized by microprocessors in the Intel X86 family when communicating with memory and I/O devices. This included the 80286 and 80386 microprocessors, as well as the kernel support logic necessary to interface these processors to 8-, 16-, and 32-bit devices. Part 2, "Memory Subsystems," provided detailed descriptions of RAM, ROM and cache memory. Together, the first two parts provide the background necessary to an understanding of the ISA architecture.

Part 3, "The Industry Standard Architecture," examines all ISA-specific aspects of system design. It includes the following chapters:

- Chapter 16: ISA Bus Structure
- Chapter 17: Types of ISA Bus Cycles
- Chapter 18: Interrupt Subsystem
- Chapter 19: Direct Memory Access (DMA)
- Chapter 20: ISA Bus Masters
- Chapter 21: RTC and Configuration RAM
- Chapter 22: Keyboard/Mouse Interface
- Chapter 23: Numeric Coprocessors
- Chapter 24: ISA Timers

Chapter 16

The Previous Chapter

The previous chapter covered the various types of ROM memory encountered in ISA systems. This included PROM, MROM, EPROM, EEPROM and flash EEPROM. In addition, the subjects of ROM diagnostics, shadow RAM, double-mapping, and device ROMs were covered.

This Chapter

This chapter provides a detailed description of the ISA bus. All signals on the expansion bus are fully defined, as well as the related system board logic.

The Next Chapter

The next chapter, "Types of ISA Bus Cycles," provides a detailed description of bus cycles as they appear on the ISA expansion bus.

Introduction

The expansion bus in ISA machines is frequently referred to as the I/O bus. This is incorrect. Both memory and I/O cards are interfaced to the system through the expansion bus. Figures 16-1, 16-2 and 16-3 illustrate the 8- and 16-bit ISA expansion bus connectors. The following sections describe the capabilities provided by the ISA interface. The characteristics of the 8-bit ISA slots were defined in the original IBM PC. The IBM PC/AT added additional functionality. The signals have been grouped by function and are described in the following sections.

Mounting Bracket

0Vdc	B01	A01	CHCHK#
RESET	B02	A02	SD7
+5Vdc	B03	A03	SD6
IRQ2/IRQ9	B04	A04	SD5
-5Vdc	B05	A05	SD4
DRQ2	B06	A06	SD3
-12Vdc	B07	A07	SD2
NOWS#	B08	A08	SD1
+12Vdc	B09	A09	SD0
0Vdc	B10	A10	CHRDY
SMWTC#	B11	A11	AEN
SMRDC#	B12	A12	SA19
IOWC#	B13	A13	SA18
IORC#	B14	A14	SA17
DAK3#	B15	A15	SA16
DRQ3	B16	A16	SA15
DAK1#	B17	A17	SA14
DRQ1	B18	A18	SA13
REFRESH#	B19	A19	SA12
BCLK	B20	A20	SA11
IRQ7	B21	A21	SA10
IRQ6	B22	A22	SA9
IRQ5	B23	A23	SA8
IRQ4	B24	A24	SA7
IRQ3	B25	A25	SA6
DAK2#	B26	A26	SA5
TC	B27	A27	SA4
BALE	B28	A28	SA3
+5Vdc	B29	A29	SA2
OSC	B30	A30	SA1
0Vdc	B31	A31	SA0

Figure 16-1. The 8-bit Portion of the ISA Bus

M16#	D01	C01	SBHE#
IO16#	D02	C02	LA23
IRQ10	D03	C03	LA22
IRQ11	D04	C04	LA21
IRQ12	D05	C05	LA20
IRQ15	D06	C06	LA19
IRQ14	D07	C07	LA18
DAK0#	D08	C08	LA17
DRQ0	D09	C09	MRDC#
DAK5#	D10	C10	MWTC#
DRQ5	D11	C11	SD8
DAK6#	D12	C12	SD9
DRQ6	D13	C13	SD10
DAK7#	D14	C14	SD11
DRQ7	D15	C15	SD12
+5Vdc	D16	C16	SD13
MASTER16#	D17	C17	SD14
0Vdc	D18	C18	SD15

Figure 16-2. The 16-bit Portion of the ISA Bus

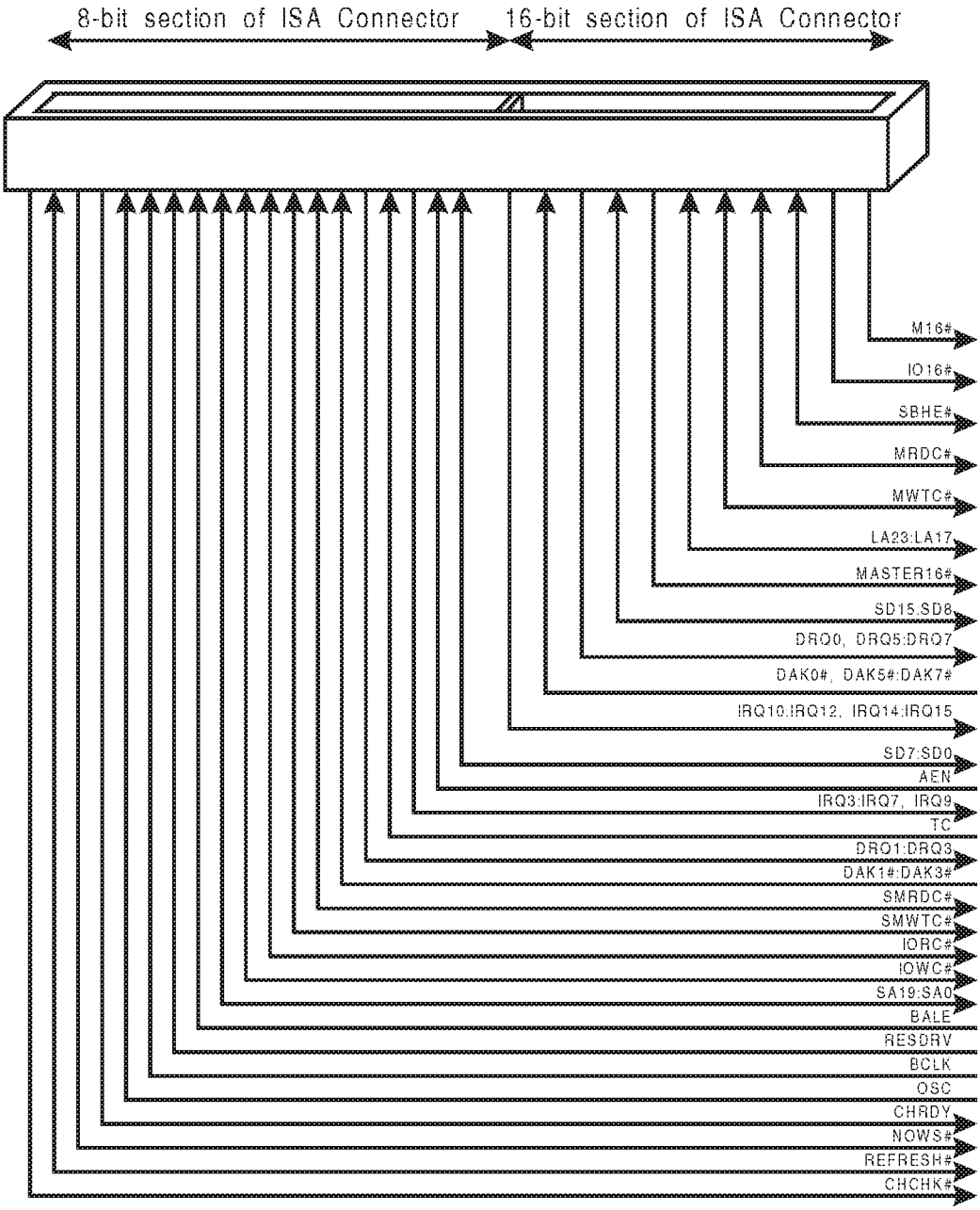


Figure 16-3. The ISA Connector

Address Bus

When the CPU initiates a bus cycle, it places the address on the address bus during address time (Ts).The address is visible to all installed ISA cards and to the logic on the system board. In other words, the current bus master can communicate with any memory or I/O slave in the system. Table 16-1 provides a description of the signals involved in addressing devices on the ISA expansion bus.

Table 16-1. The ISA Address-Related Signals

Signal	Function
SA[19:0]	**System address bus**. This is the microprocessor's latched address bus. The SA bus only latches the lower 20 bits of the address (A[19:0]) being emitted by the microprocessor. The address for a bus cycle begins to appear on the SA bus at the rising (leading) edge of BALE and is latched at the trailing edge of BALE (buffered address latch enable).
LA[23:17]	**Latchable address bus**. These signals are a buffered version of the upper bits on the microprocessor's address bus. During address pipelining, the address for the next bus cycle is present on this bus prior to the beginning of the actual bus cycle. When an ISA expansion board requires an early address decode, the LA bus should be decoded rather than the SA bus. The address for a bus cycle does not begin to appear on the SA bus until the midpoint of the ISA's address time. Decoding the LA bus allows the expansion board address decoder to start the address decode process early, ensuring that the addressed device will be ready to complete the bus cycle earlier than if its chip-select had been decoded from the SA bus. Designers use this method to allow boards employing relatively slow memory devices to respond to the microprocessor with fewer wait states.
SBHE#	**System bus high enable**. Also see the section on the data bus steering logic. The 80286 or 80386SX microprocessor asserts BHE# during an 8-bit transfer with an odd address and during a 16-bit transfer starting at an even address. When asserted, it signals that the upper half of the data bus, D[15:8], will be used to transfer a byte between the microprocessor and an odd address.

Data Bus

During a bus cycle's data time (Tc), data is transferred over the data bus. The ISA data bus consists of two 8-bit data paths: SD[7:0] and SD[15:8]. Table 16-2 describes these paths.

Table 16-2. The ISA Data Bus Paths

Signal	Function
SD[7:0]	**System data bus, lower path**. During transfers with 16-bit devices, the lower data path is used to transfer data between the current bus master and even-addressed locations. During transfers with 8-bit devices, the lower data path is used to transfer data between the current bus master and both even and odd-addressed locations. This data path is available on all 8- and 16-bit ISA expansion slots.
SD[15:8]	**System data bus, upper path**. During transfers with 16-bit devices, the upper data path is used to transfer data between the current bus master and odd-addressed locations. During transfers with 8-bit devices, the upper data path is not used. This data path is available only on 16-bit ISA expansion slots.

Bus Cycle Definition

When the system board processor initiates a bus cycle, it places the bus cycle definition on its bus cycle definition lines during address time (Ts). The state of these lines must be decoded by the system board's bus control logic to determine the type of bus cycle in progress. The bus control logic then asserts the appropriate command line to indicate the bus cycle type to all ISA expansion cards. Table 16-3 defines the setting of the microprocessor's bus cycle definition lines for the various types of bus cycles.

Table 16-3. Bus Cycle Definition Line Decoding

Microprocessor's Bus Cycle Definition Signals			Bus Cycle Type	ISA Command Lines Asserted
M/IO#	S0#	S1#		
0	0	0	Interrupt Acknowledge	none
0	0	1	I/O Write	IOWC#
0	1	0	I/O Read	IORC#
1	0	0	Halt or Shutdown	none
1	0	1	Memory Write	MWTC# and SMWTC#
1	1	0	Memory Read	MRDC# and SMRDC#

The ISA command lines are described in table 16-4.

Table 16-4. The ISA Command Lines

Signal	Function
SMRDC#	**System memory read command**. This signal is asserted by the system board logic during a memory read bus cycle that addresses a location in the lower 1MB memory address space (000000h to 0FFFFFh). This signal is connected to the 8-bit portion of all 8- and 16-bit ISA expansion connectors. 8-bit memory cards should use this command line as an enable to decode the memory address on the address bus. This prevents 8-bit memory cards from being selected when a memory address greater than 1MB is on the address bus. Also see MRDC#.
SMWTC#	**System memory write command**. This signal is asserted by the system board logic during a memory write bus cycle that addresses a location in the lower 1MB memory address space (000000h to 0FFFFFh). This signal is connected to the 8-bit portion of all 8- and 16-bit ISA expansion connectors. 8-bit memory cards should use this command line as an enable when decoding the memory address on the address bus. This prevents 8-bit memory cards from being selected when a memory address greater than 1MB is on the address bus. Also see MWTC#.
IOWC#	**I/O write command**. This signal is asserted by the system board logic during an I/O write bus cycle.
IORC#	**I/O read command**. This signal is asserted by the system board logic during an I/O read bus cycle.
MRDC#	The **memory read command** line is asserted by the system board logic during a memory read bus cycle. 16-bit memory cards should use this signal as an enable when decoding the memory address on the address bus. This prevents 8-bit memory cards from being selected when a memory address greater than 1MB is on the address bus. Also see SMRDC#.
MWTC#	The **memory write command** line is asserted by the system board logic during a memory write bus cycle. 16-bit memory cards should use this signal as an enable when decoding the memory address on the address bus. This prevents 8-bit memory cards from being selected when a memory address greater than 1MB is on the address bus. Also see SMWTC#.

Bus Cycle Timing

The timing for an ISA bus cycle is derived from BCLK. Table 16-5 describes the ISA bus signals related to bus cycle timing.

Table 16-5. Bus Cycle Timing Signals

Signal	Function
BCLK	**Bus clock**. The BCLK signal is derived from the input clock to the system board microprocessor. ISA bus is based on the version of the IBM PC/AT that was based on an 80286 microprocessor running at a speed of 8MHz (internal PCLK speed). The microprocessor's double-frequency clock input, CLK2, is divided by two to yield an 8MHz signal. In order to synchronize it to each bus cycle generated by the microprocessor, the BCLK generator on the system board monitors the S0# and S1# outputs of the microprocessor. The falling edge of either of these signals marks the start of a bus cycle and is used to synchronize the 8MHz BCLK supplied to the ISA expansion cards. Since BCLK is re-synchronized at the leading edge of every bus cycle, it will appear to jitter when viewed with an oscilloscope. One cycle of BCLK defines the duration of address time or data time on the ISA bus. In later ISA machines, the BCLK speed is usually 8.33MHz. This dates from the introduction of the first 25MHz 80386 machines. The designers of these machines simply divided the 50MHz CLK2 by six to yield a BCLK speed of 8.33MHz. Some problems were anticipated due to the slightly faster BCLK and the resulting shorter duration of address time and data time on the ISA bus, but few actual problems were encountered.

Table 16-5. Bus Cycle Timing Signals, continued

Signal	Function
BALE	**Buffered address latch enable**. On the leading, or rising, edge of BALE, the microprocessor's address bus (A[19:0]) is gated through the system board's Address Latch and through the bus control logic onto the SA bus (SA[19:0]). On the trailing edge of BALE, the address latch and the bus control logic hold the current address on the SA bus for the duration of the current bus cycle. The falling edge of BALE should be used to by 16-bit ISA memory boards to latch the result of the decoded LA bus address (LA[23:17]) and hold it for the duration of the current bus cycle. BALE is deasserted and reasserted (pulsed) for every bus cycle that the bus control logic runs. If address pipelining is enabled, A[19:0] is not gated through the address latch to SA[19:0] on the second and subsequent ISA cycles when the bus control logic is running multiple ISA bus cycles to satisfy one microprocessor bus cycle. BALE is asserted (high) constantly whenever the system's microprocessor is not the current bus master.
CHRDY	**Channel ready**. When an ISA expansion board requires more wait states than would normally be supplied by the default ready timer on the system board, it should deassert CHRDY when its address decoder recognizes an address and the appropriate command line (SMRDC#, SMWTC#, MRDC#, etc.). This prevents the default ready timer on the system board from timing out and issuing READY# to the microprocessor until CHRDY is asserted again by the currently-addressed ISA device. As long as the currently-addressed ISA card holds CHRDY deasserted, the microprocessor's READY# input is kept deasserted, causing wait states to be inserted in the bus cycle. When the currently-addressed device is ready to end the bus cycle, it allows CHRDY to become asserted again (CHRDY has a pull-up resistor on the system board), enabling the default ready timer to assert the microprocessor's READY# input. When the microprocessor samples its READY# input asserted at the end of the current data time, it ends the bus cycle.

Table 16-5. Bus Cycle Timing Signals, continued

NOWS#	**No wait state**. When an 8- or 16-bit ISA memory expansion board requires zero wait states rather than the number that would normally be supplied by the default ready timer on the system board, it should assert NOWS# when its address decoder recognizes an address and the appropriate command line (SMRDC#, SMWTC#, MRDC#, etc.). This causes the default ready timer to assert READY# immediately. This signals to the microprocessor that the device is ready to end the bus cycle. NOWS# can be used to shorten an access to an 8-bit card from the default of four wait states down to one wait state, or an access to a 16-bit memory card from one to zero wait states. Accesses to 16-bit I/O cards may not be shortened from the default of one wait state, but may be lengthened using the CHRDY signal.

Device Size

When a 16-bit slave's address decoder detects an address within its assigned range, it not only should assert the device's chip-select line, but should also assert the appropriate 16-bit size line. Refer to table 16-6. An asserted level on either of the 16-bit size lines informs the current bus master that it is communicating with a device that is connected to both data paths, SD[7:0] and SD[15:8].

If neither of these signals is asserted during a bus cycle, the bus master is communicating with an 8-bit device and all communications must take place over the lower data path, SD[7:0]. When necessary, the system board's data bus steering logic is automatically enabled by the bus control logic to route the data over the correct data path.

The device size lines (M16# and IO16#) control these aspects of the bus control logic:

1. ISA bus cycle timing, including:
    - whether the command lines are asserted early or are delayed, and
    - the number of default wait states

2. Bus sizing, including:
    - data bus steering, and
    - the number of ISA bus cycles that must be run

ISA bus cycle timing is described in detail in the next chapter. Bus sizing is described in detail in the chapters entitled "The 80286 System Kernel: The Engine" and "The 80386 System Kernel."

Table 16-6. The Size 16 Signal Lines

Signal	Function
M16#	**Memory size 16**. This signal is generated by a 16-bit memory expansion card when it recognizes it is being addressed. M16# tells the data bus steering logic that the addressed memory device is capable of communicating over both data paths. When accessing an 8-bit memory device, the M16# signal line remains deasserted, indicating to the data bus steering logic that the currently addressed device is an 8-bit memory device capable of communicating over the lower data path only. When asserted, M16# also pre-conditions the default ready timer on the system board to insert one wait state into the bus cycle unless NOWS# is asserted. If NOWS# is asserted, zero wait states are inserted in the bus cycle.
IO16#	**I/O size 16**. This signal is generated by a 16-bit ISA I/O expansion board when it recognizes that it is being addressed. IO16# provides the same function for 16-bit I/O expansion devices as the M16# signal provides for 16-bit memory devices. IO16# also pre-conditions the default ready timer on the system board to insert one wait state into the bus cycle.

Reset

When the unit is first powered up, the RESDRV (reset drive) signal on the ISA bus remains asserted until the power supply voltages have stabilized. RESDRV is also asserted when power has dropped below an acceptable level. When asserted, RESDRV has two effects on the installed ISA cards:

- The cards are prevented from doing anything until the power has stabilized and RESDRV has become deasserted.
- Forces all of the installed ISA cards to a known, initialized state to ensure that they always begin operating exactly the same way every time the unit is powered up.

DMA

Figure 16-4 illustrates the system board logic and ISA bus signals related to DMA. ISA systems employ two 8237 DMA controllers cascaded together to provide seven DMA channels for use by I/O cards. The signals associated with DMA operations are listed and defined in table 16-7. For a detailed description of DMA, see the chapters entitled "Direct Memory Access (DMA)" and "ISA Bus Masters."

Table 16-7. The ISA DMA Signal Lines and Associated System Board Logic

Signal	Function
DRQ[3:0] DRQ[7:5]	**DMA request** for channels 0 through 3 and 5 through 7. DRQ4 isn't present on the ISA expansion Bus because it's used by the Slave 8237 DMA controller to request the use of the buses from the master 8237. The other DMA request lines are used by I/O devices to request that a block data transfer between the I/O device and memory be performed by the DMAC. DRQ[3:1] are connected to all 8- and 16-bit ISA expansion slots, while DRQ[7:5] and DRQ0 are only connected to the 16-bit ISA expansion slots.
DAK#[3:0] DAK#[7:5]	**DMA acknowledge** for channels 0 through 3 and 5 through 7. DAK4# isn't present on the ISA expansion bus because it's used by the master 8237 DMAC to inform the slave 8237 DMAC that it has been granted the use of the buses. When a DMA channel has possession of the buses, it responds to a DMA request (DRQx) by asserting the corresponding DMA acknowledge line (DAKx#), along with either the I/O Read or I/O Write command line (IORC# or IOWC#). This combination of DAKx# and IORC# or IOWC# selects the I/O device that initiated the request, allowing it to transfer a byte (for the slave DMAC) or a word (for the master DMAC) between itself and memory.
TC	**Transfer complete**. Asserted by the master or slave DMAC when the word or byte transfer count for a DMA channel has been exhausted.
AEN	**Address enable**. Asserted when one of the system board's DMACs (master or slave) has been granted the buses. When asserted, AEN prevents the I/O devices from responding to the I/O command lines. The DMAC has placed a memory address on the address bus (not an I/O address) but it has asserted one of the I/O command lines.
MASTER16#	Asserted by a **16-bit ISA bus master** expansion board when it has been granted the buses. On the system board, an asserted level on MASTER16# causes the direction of the bus buffers to be reversed so the ISA bus master expansion card can drive addresses and bus cycle information onto the ISA bus and into the system board. The asserted level on MASTER16# also causes the system board to deassert AEN, thereby allowing the I/O devices to respond to the ISA command lines. This enables the bus master card to address any device on the system board and any device located on the ISA bus.

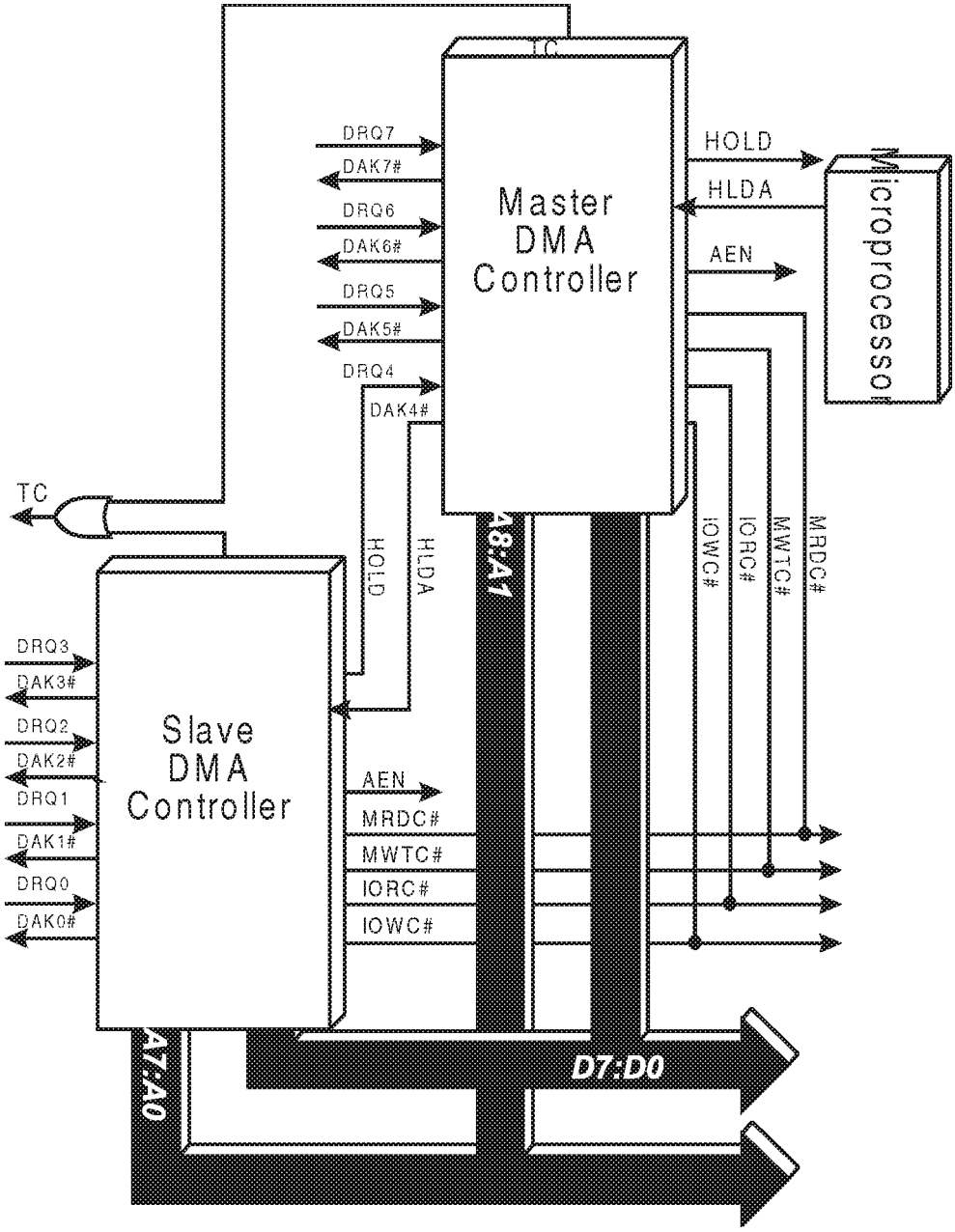


Figure 16-4. The ISA DMA Signals and Associated System Board Logic

Interrupts

Figure 16-5 illustrates the interrupt support logic on the system board and the interrupt-related signals on the ISA bus. Two Intel 8259 programmable interrupt controllers (PICs) are located on the system board. When a device on the ISA bus requires servicing by the system board microprocessor, it should assert its respective interrupt request (IRQ) line to inform the system board microprocessor. A detailed description of interrupts is located in the chapter entitled "Interrupt Subsystem." Table 16-8 lists the ISA interrupt request lines and the I/O devices that normally use them. The assignments of system interrupt request lines that are **not** on the ISA connector are shown in table 16-9.

Table 16-8. ISA Bus Interrupt Request Line Assignment

Interrupt Line	Device usually assigned to
IRQ3	Serial port 2
IRQ4	Serial port 1
IRQ5	Parallel port 2
IRQ6	Floppy disk interface
IRQ7	Parallel port 1
IRQ9	none (named IRQ2 on PC and PC/XT)
IRQ10	none
IRQ11	none
IRQ12	Mouse interface
IRQ14	Hard disk interface
IRQ15	none

*Table 16-9. Interrupt Request Line Assignment, Lines **Not** on ISA Bus*

Interrupt Line	Device assigned to
IRQ0	System timer
IRQ1	Keyboard interface
IRQ2	Cascade interrupt from slave PIC
IRQ8	Real time clock
IRQ13	Numeric coprocessor

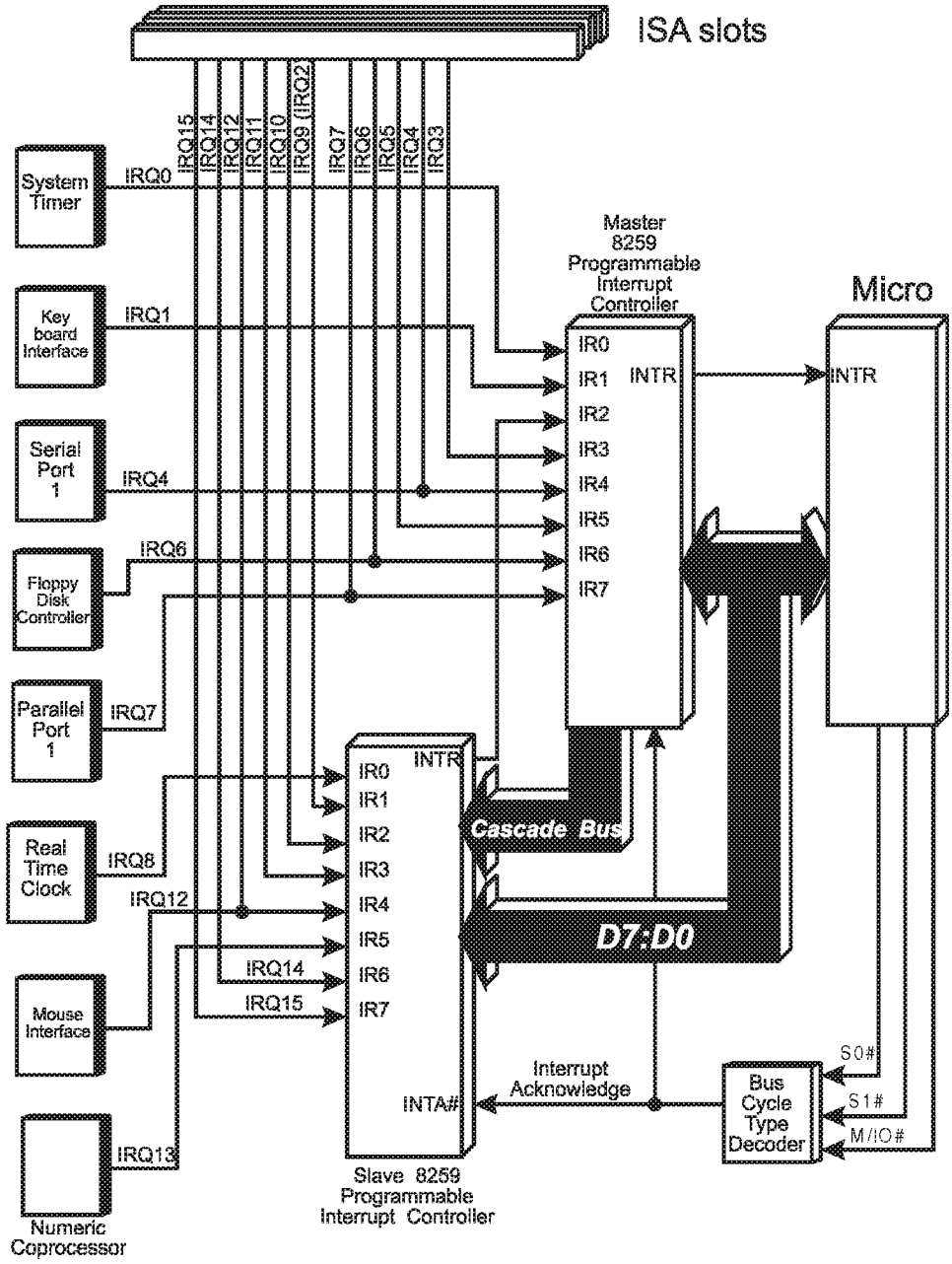


Figure 16-5. The ISA Interrupt Signals and Associated System Board Logic

Error Reporting Signal

The ISA expansion bus signal line associated with error reporting is described in table 16-10. Figure 16-6 illustrates the relationship between the ISA bus CHCHK# signal and the system board logic.

Table 16-10. The ISA Error Reporting Signal

Signal	Function
CHCHK#	**Channel or I/O check**. This signal is asserted by an ISA expansion board to signal an error condition to the microprocessor. An NMI interrupt is issued to the microprocessor if NMI interrupts have been enabled (bit 7 of port 70h is a 0) and if recognition of channel check has been enabled (bit 3 of port 61h is a 0).

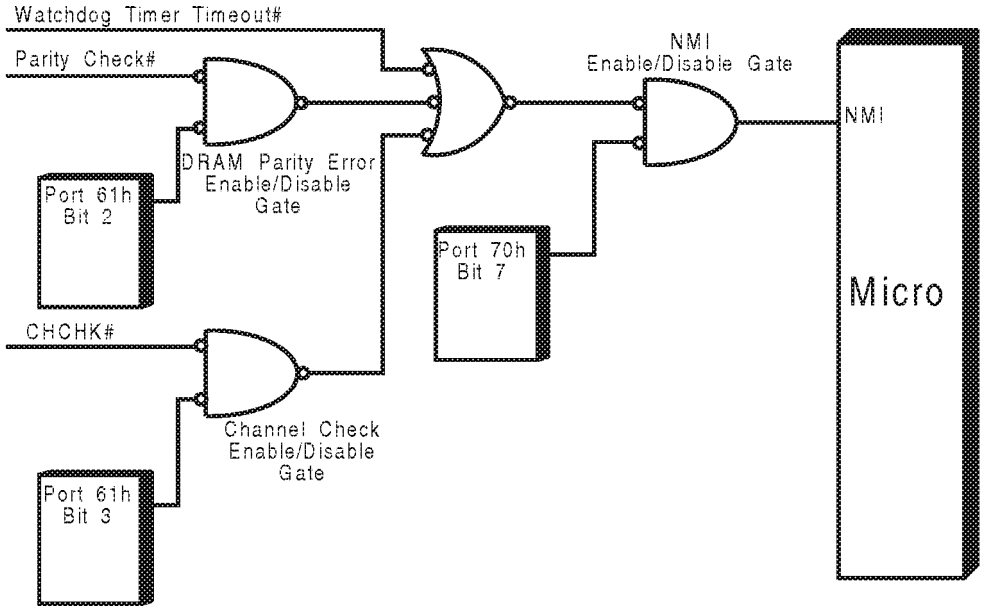


Figure 16-6. The Channel Check Logic

Miscellaneous Signals

The remaining signals on the ISA bus are listed and described in table 16-11.

Table 16-11. Miscellaneous ISA Signals

Signal	Function
REFRESH#	This signal is asserted whenever the system board **refresh** logic is executing a DRAM refresh bus cycle. See discussion of DRAM refresh in the chapter entitled "RAM Memory: Theory of Operation."
OSC	**Oscillator**. This is a free-running 14.31818MHz clock that may be used by ISA expansion boards. OSC runs at a multiple of the color burst frequency and was used by video display adapter cards. Additional information about OSC is in the chapter entitled "ISA Timers."

Chapter 17

The Previous Chapter

The previous chapter, "The ISA Bus Structure," provided a detailed description of all of the signals on the ISA expansion bus and their associated system board support logic.

This Chapter

This chapter provides a detailed description of the types of transfers performed over the ISA expansion bus.

The Next Chapter

The next chapter, "Interrupt Subsystem," provides a detailed description of the interrupt subsystem found in all ISA machines.

Introduction

The fastest bus cycle that can be run by an 80286 or higher microprocessor is a 0-wait-state bus cycle, with a duration of two PCLK cycles. For the clock-multiplied 486 and Pentium processors, a 0-wait-state bus cycle takes two ticks of the CLK input to the microprocessor. PCLK may be the same as or may be faster than CLK. The clock that is used to synchronize ISA bus transfers is called BCLK, or bus clock. The ISA bus is based on the 8MHz version of the 80286-based IBM PC/AT. In that machine, BCLK was a copy of PCLK, so BCLK ran at 8MHz.

In later ISA machines, the BCLK speed is up to 8.33MHz. This dates from the introduction of the first 25MHz 80386 machines. The designers of these machines simply divided the 50MHz CLK2 by six to yield a BCLK speed of 8.33MHz. BCLK can run at any frequency up to 8.33MHz, with typical frequencies of 8MHz, 8.25MHz and 8.33MHz. A single tick, or cycle, of BCLK is therefore 120ns to 125ns in duration. A 0-wait-state bus cycle on the ISA bus consists of two BCLK ticks, lasting 240ns to 250ns.

Note that for simplicity in the rest of this chapter, BCLK is assumed to run at 8MHz.

During a bus cycle, a maximum of two bytes may be transferred by using both data paths. At this rate, the machine can perform four million (1 second divided by 250ns per bus cycle) two-byte transfers per second. This yields a maximum transfer rate of 8MB/second.

The ISA bus allows both 8-bit transfers, for compatibility with the 8-bit expansion boards used in the original IBM PC, as well as 16-bit transfers. The following sections describe the basic types of ISA bus cycles.

Transfers with 8-bit Devices

The ISA bus cycle types used to communicate with 8-bit devices include:

- Standard 8-bit device ISA bus cycle — 4 wait states
- Shortened 8-bit device bus cycle — 1, 2, or 3 wait states
- Stretched 8-bit device bus cycle — more than 4 wait states

The steps below describe the sequence of events that take place during an 8-bit bus cycle using the default READY# timing and explains how the default timing can be either shortened or stretched. Figure 17-1 illustrates an example bus cycle. The step numbers in the text that follows corresponds to the numbered reference points in figure 17-1.

1. The address being presented by the microprocessor begins to appear on the LA bus at the start of address time. This corresponds to the leading edge of Ts. If the system board is based on an 80286, 80386 or Pentium microprocessor and address pipelining is enabled, the address may actually be present on the LA bus prior to the beginning of the bus cycle (as is the case in this example). 16-bit ISA memory expansion cards can use the portion of the address on the LA bus to perform an early address decode. 8-bit ISA expansion cards do not have access to the LA bus and therefore cannot perform an early address decode. I/O cards only use the lower 16 address bits and therefore cannot take advantage of address pipelining either.

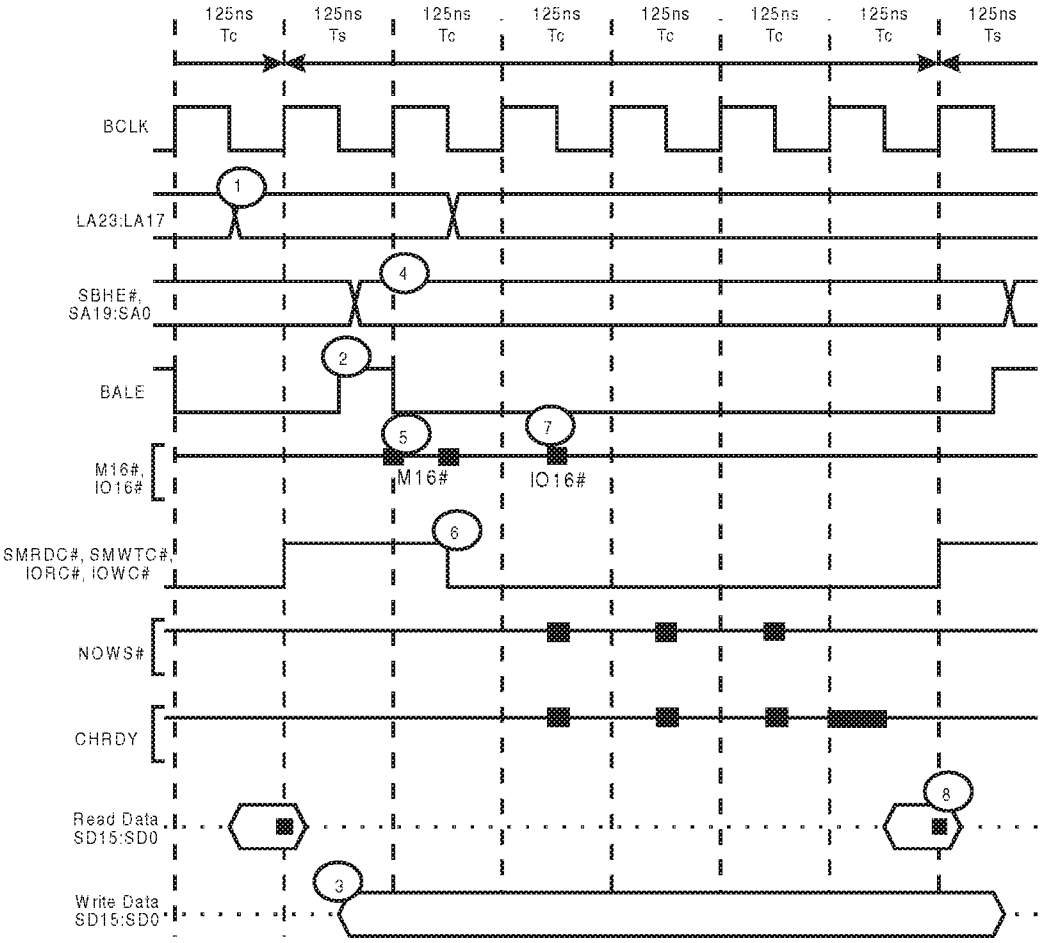


Figure 17-1. Standard Access to an 8-bit ISA Device

2. BALE is asserted halfway through address time, gating the address through the system board address latch and through the bus control logic onto the SA bus.
3. If this is a write bus cycle, the microprocessor's output data is gated onto the SD bus halfway through address time. It remains on the SD bus until half a BCLK cycle into the next bus cycle (halfway through address time of the next transfer).
4. The trailing edge of BALE (at the beginning of data time) causes the system board address latch and the bus control logic to latch the address being

output by the CPU so it will remain static on the SA bus for the remainder of the bus cycle. Also at this time, the addressed device can safely complete the decoding process.

5. If this is a memory transaction (M/IO# from microprocessor is high), then at the end of address time, the M16# signal is sampled by the system board bus control logic. Since this is a transaction to an 8-bit device, M16# is sampled deasserted, and the bus control logic does not assert the command line until halfway through the first data time. During a transfer with an 8-bit device, the assertion of the command line is delayed until the midpoint of data time to allow more time for address decode before command line assertion. (See the section entitled, "Transfers with 16-bit Devices" to see when the command lines are asserted early.)
6. The appropriate command line (SMRDC#, SMWTC#, IORC# or IOWC#) is asserted halfway through data time. The command line remains asserted until the end of the bus cycle (end of last Tc). If this is a memory transaction, M16# is also sampled by the bus control logic halfway through the first data time. Since this is a transaction to an 8-bit device, M16# is sampled deasserted. The second sample point on M16# gives additional time for 16-bit memory devices that failed to assert M16# early enough for the first sample point. These devices either did not see LA[23:17] early enough to assert M16# by the first sample point, or they are waiting for BALE to be asserted before decoding LA[23:17].
1. If this is an I/O transaction (M/IO# from microprocessor is low), then at the midpoint of the second data time (the first wait state), the IO16# signal is sampled by the system board bus control logic. Since this is a transaction to an 8-bit device, IO16# is sampled deasserted, indicating that the addressed expansion board is an 8-bit device.

Halfway through the second data time and halfway through each subsequent data time, the default ready timer on the system board samples the NOWS# and CHRDY lines. If both NOWS# and CHRDY are sampled asserted, the CPU READY# line is asserted and the bus cycle ends on the next rising edge of BCLK (the end of the current data time). In this way, an ISA board can terminate a bus cycle earlier than the default number of BCLK cycles (wait states) by asserting NOWS#. On the other hand, if CHRDY is deasserted, the default ready timer ignores NOWS# and inserts wait states. (Note that the EISA specification warns that devices should not assert NOWS# while deasserting CHRDY, implying that some bus controllers may not ignore NOWS# when CHRDY is deasserted.)

A bus cycle addressing an 8-bit ISA device defaults to six BCLK cycles (four wait states) if the following two conditions are met:

- the bus cycle isn't terminated earlier by the assertion of NOWS# and
- CHRDY is asserted when sampled during the first half of the last data time of the default cycle (first half of the 5th data time).

This means that the duration of an ISA bus cycle accessing an 8-bit device defaults to four wait states unless shortened by NOWS# (to 3, 4, or 5 BCLK cycles) or lengthened by CHRDY. The default bus cycle ends at the trailing edge of the fifth data time.

8. During a read bus cycle, the microprocessor reads the data on the data bus at the end of the last BCLK (Tc) of the bus cycle and the bus cycle is then terminated by deasserting the command line (SMRDC#, MRDC#, IORC#). When a write bus cycle terminates, the SMWTC# or IOWC# command line is deasserted but write data remains on the SD bus until halfway through address time of the next bus cycle. This accommodates the hold time of the device being written to and doesn't disturb the device being addressed in the next bus cycle because the command line for that bus cycle hasn't been asserted yet.

Transfers with 16-bit Devices

The ISA bus cycle types used to communicate with 16-bit devices include:

- Standard 16-bit device ISA bus cycle (Memory & I/O) — 1 wait state
- Shortened 16-bit device ISA bus cycle (Memory only) — 0 wait state
- Stretched 16-bit device ISA bus cycle — more than 1 wait state

Standard 16-bit Memory Device ISA Bus Cycle

Figure 17-2 illustrates the timing of a bus cycle on the ISA bus when the microprocessor is communicating with a one-wait-state 16-bit memory device. The timing diagram and the following numbered steps illustrate the sequence of events that occur during a default ISA 16-bit memory access. Each of the numbered steps corresponds to the numbered reference points in figure 17-2.

1. If the system board is based on an 80286, 80386 or Pentium microprocessor and address pipelining is enabled, the address is present on the LA bus prior to the beginning of the bus cycle. This allows the addressed memory device to start decoding the address early which may speed up access.
2. BALE is asserted halfway through address time. Some 16-bit memory devices wait until the rising edge of BALE before decoding the LA lines to determine if the address is for them. Also when BALE becomes asserted, the

lower portion of the address from the processor is transferred through the system board's address latch and through the bus control logic onto the SA bus.

3. The addressed memory board asserts M16# as a result of decoding the LA lines, indicating to the system board's bus control logic that it is capable of handling a 16-bit transfer without data bus steering being performed by the steering logic on the system board.
4. If this is a write bus cycle, the microprocessor's output data is gated onto the SD bus halfway through address time and remains on the SD bus until half a BCLK cycle into the next bus cycle (half-way through address time of the next bus cycle).
5. At the end of address time, the trailing edge of BALE causes two events to take place:
    a) 16-bit ISA memory devices latch the LA lines (or the chip select) so the addressed device will not be deselected when the LA lines are pipelined before the end of the current bus cycle.
    b) The address latch on the system board and the bus control logic latch the lower twenty bits of the address, SA[19:0], so they remain static on the SA bus for the remainder of the bus cycle. The addressed device can safely complete decoding the SA address on the bus on the falling edge of BALE.

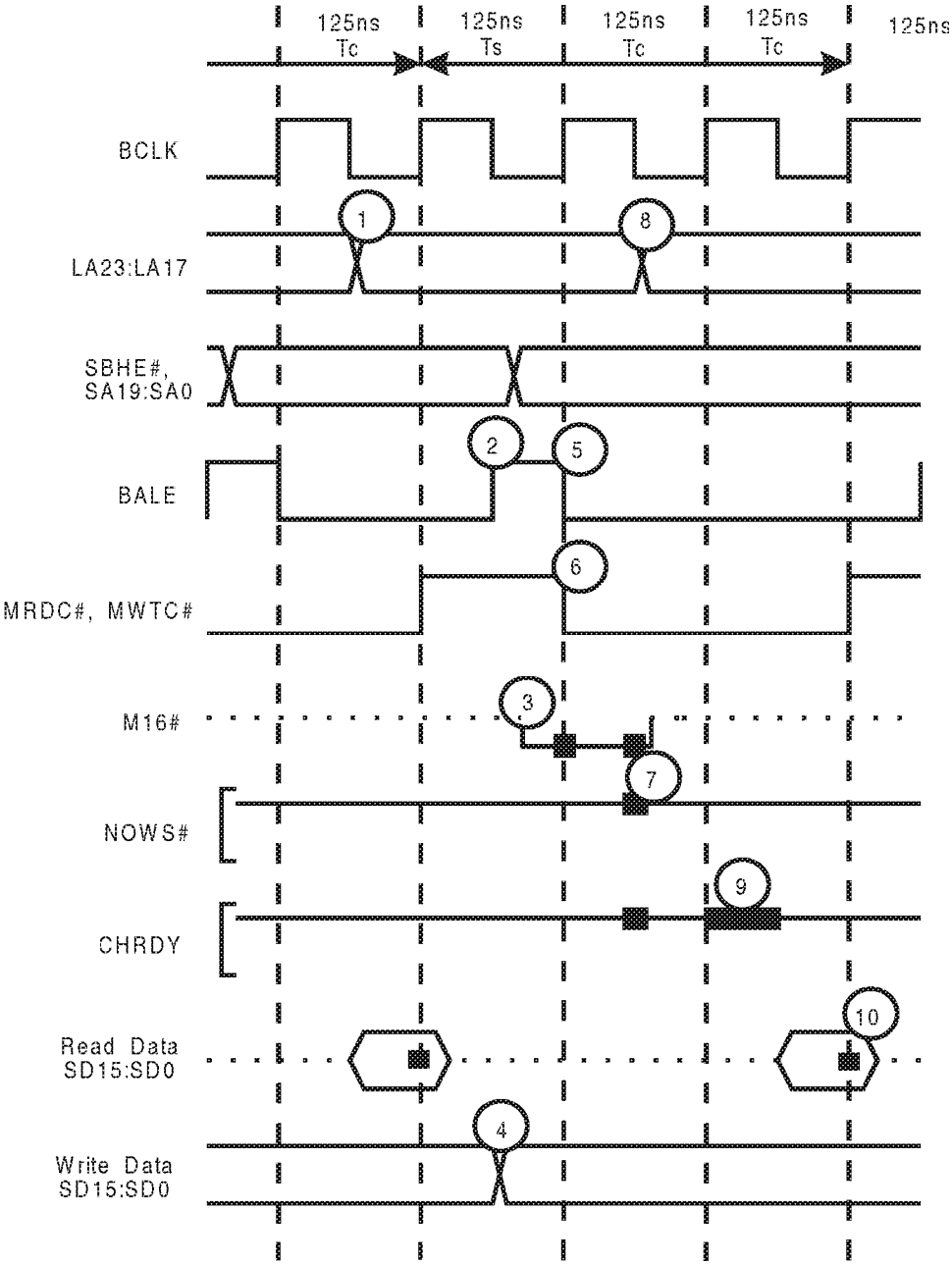


Figure 17-2. Standard Access to a 16-bit ISA Memory Device

6. The system board's bus control logic samples M16# at the end of address time to determine if the addressed device can take advantage of the MRDC# or MWTC# command lines being asserted immediately. 16-bit memory devices see the pipelined address long before 8-bit and I/O devices see the address on SA[19:0]. 16-bit memory devices can therefore assert M16# very early in the bus cycle. The appropriate command line (MRDC# or MWTC#) is asserted at the leading edge of data time if M16# was sampled asserted. This command line remains asserted until the end of the bus cycle (end of last Tc).If M16# is sampled deasserted, the command line (MRDC# or MWTC#) is asserted halfway through data time.
7. The system board's bus control logic samples M16# a second time at the midpoint of the first data time to determine if data bus steering is necessary. Since this is an access to a 16-bit device, no steering is necessary. The second sample point on M16# gives additional time for 16-bit memory devices that failed to assert M16# early enough for the first sample point. These devices either did not see LA[23:17] early enough to assert M16# by the first sample point, or they are waiting for BALE to be asserted before decoding LA[23:17]. Also at the midpoint of the first data time, the default ready timer on the system board samples the NOWS# and CHRDY lines. If NOWS# and CHRDY are sampled asserted, the microprocessor's READY# line is asserted and the bus cycle terminates at the end of the first data time. In this way, a 16-bit ISA memory board can complete a bus cycle in two BCLK cycles. On the other hand, if CHRDY is deasserted, the default ready timer ignores NOWS# and inserts wait states. (Note that the EISA specification warns that devices should not assert NOWS# while deasserting CHRDY, implying that some bus controllers may not ignore NOWS# when CHRDY is deasserted.)
8. During address pipelining, the microprocessor is free to output the address for the next bus cycle during the current bus cycle. Only the upper portion of the pipelined address appears at this time on the LA bus because these bits are buffered but not latched from the microprocessor's address bus. The remainder of the address will not appear on the SA bus until the midpoint of address time in the next bus cycle.
9. CHRDY is sampled by the default ready timer during the second data time to determine if the device will be ready to complete the bus cycle at the end of this BCLK cycle. If the device cannot complete the bus cycle by the end of this BCLK cycle, it should deassert the CHRDY line. If CHRDY were sampled deasserted by the default ready timer, it would respond by extending the bus cycle by adding another data time. CHRDY is then checked at the beginning of each additional data time until the device releases CHRDY to indicate that the bus cycle can be completed.

10. An ISA 16-bit memory bus cycle defaults to three BCLK cycles (one wait state) if the bus cycle isn't terminated earlier by the assertion of NOWS# and if CHRDY stays asserted throughout the bus cycle. This means that the length of an ISA bus cycle when accessing a 16-bit memory card defaults to one wait state unless shortened by NOWS# or lengthened by CHRDY. READY# is then asserted to the microprocessor, telling it to read the data from the data bus. The bus control logic then deasserts the command line (MRDC#, MWTC#, SMRDC#, SMWTC#). When a memory write bus cycle terminates, the MWTC# command line is deasserted, but the data remains on the SD bus for the first half of address time in the next bus cycle. This provides hold time for the device being written to and doesn't affect the device being addressed in the next bus cycle because the command line hasn't been asserted yet.

Standard 16-bit I/O Device ISA Bus Cycle

Figure 17-3 illustrates the timing of a bus cycle on the ISA bus when the microprocessor is communicating with a 16-bit I/O device. The timing diagram and the following numbered steps illustrate the sequence of events that occur during a default ISA 16-bit I/O access. Each of the numbered steps corresponds to the numbered reference points in figure 17-3.

1. If the system board is based on an 80286, 80386 or Pentium microprocessor and address pipelining is enabled, the address is present on the LA bus prior to the beginning of the bus cycle. The LA bus has no impact on I/O bus cycles since the LA bus will always be zeros during I/O operations.
2. BALE is asserted halfway through address time, gating the address through the system board's address latch and through the bus control logic onto the SA bus.
3. If this is a write bus cycle, the microprocessor's output data is gated onto the SD bus halfway through address time and remains on the SD bus until half a BCLK cycle into the next bus cycle (half-way through address time of the next bus cycle).
4. At the start of data time, the trailing edge of BALE causes the address latch on the system board and the bus control logic to latch the lower twenty bits of the address, SA[19:0], so it remains static on the SA bus for the remainder of the bus cycle. The addressed device can safely complete decoding the SA address on the falling edge of BALE.

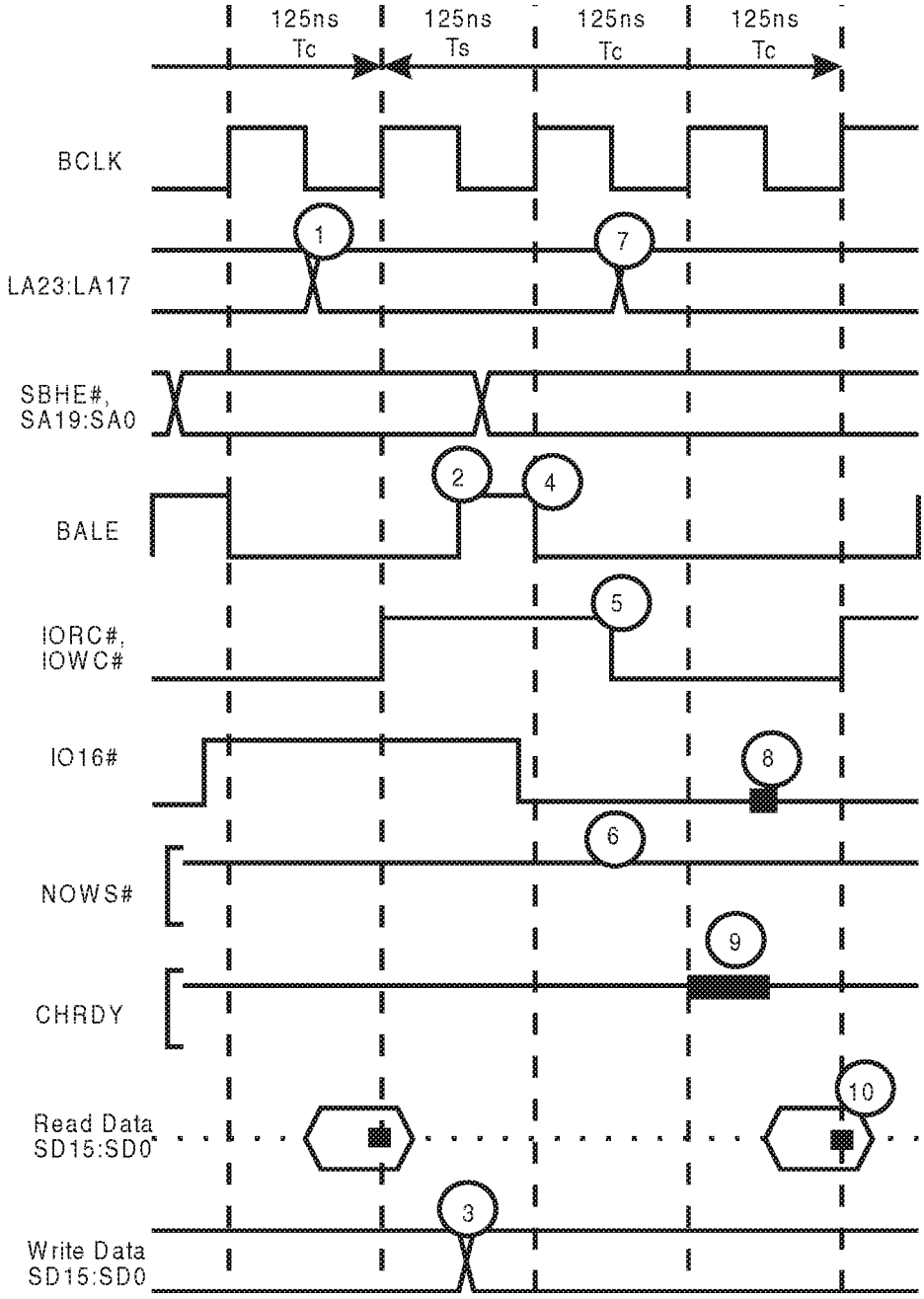


Figure 17-3. Standard Access to 16-bit I/O device

5. The appropriate command line (IORC# or IOWC#) is also asserted at the midpoint of data time. This command line remains asserted until the end of the bus cycle (end of last Tc).
6. At the midpoint of the first data time, the default ready timer on the system board ignores the NOWS# and CHRDY lines since an I/O device is being accessed. The only devices that can have zero-wait-state ISA bus cycles are 16-bit memory devices.
7. During address pipelining, the microprocessor is free to output the address for the next bus cycle during the current bus cycle. Only the upper portion of the pipelined address appears at this time on the LA bus because these bits are buffered but not latched from the microprocessor's address bus. The remainder of the address will not appear on the SA bus until the midpoint of address time in the next bus cycle.
8. IO16# is sampled at the midpoint of the second data time to determine if the I/O device is an 8- or 16-bit device. If sampled asserted, no data bus steering will be performed and the bus cycle is terminated on the next rising edge of BCLK, which is the end of the second data time. The bus cycle will not be terminated if the CHRDY line is sampled deasserted.
9. CHRDY is sampled by the default ready timer during the second data time to determine if the device is ready to complete the bus cycle. If the device cannot complete the bus cycle by the end of the second BCLK cycle, it should have deasserted the CHRDY line. If CHRDY were sampled deasserted by the default ready timer, it would respond by extending the bus cycle by one data time. CHRDY is then checked during each additional data time until the device releases CHRDY to indicate that the bus cycle can be completed.
10. An ISA 16-bit I/O bus cycle defaults to three BCLK cycles (one wait state) if CHRDY stays asserted throughout the bus cycle. The bus cycle cannot be terminated earlier by the assertion of NOWS#. This means that the length of an ISA bus cycle when accessing a 16-bit I/O card defaults to one wait state unless lengthened by CHRDY. READY# is then asserted to the microprocessor, telling it to read the data from the data bus and the bus control logic deasserts the command line. When an I/O write bus cycle terminates, the IOWC# command line is deasserted, but the data remains on the SD bus for the first half of address time in the next bus cycle. This accommodates the hold time of the device being written to and doesn't disturb the device being addressed in the next bus cycle because the command line for that bus cycle hasn't been asserted yet.

0-Wait State Access to 16-bit Memory Device

Figure 17-4 illustrates the timing of a bus cycle on the ISA bus when the CPU is communicating with a zero-wait-state 16-bit memory device. Note that only 16-bit memory devices can complete bus cycles at zero wait states. The timing diagram and the following numbered steps illustrate the sequence of events during this bus cycle. Each of the numbered steps corresponds to the numbered reference points in figure 17-4.

1. In this example, address pipelining is enabled and the LA address is present on the ISA bus prior to the beginning of the bus cycle. This allows the addressed device to start decoding the address early. In some cases, this allows a device to operate at 0 wait states.
2. BALE is asserted halfway through address time. When BALE becomes asserted, the lower portion of the address from the processor is transferred through the system board's address latch and through the bus control logic onto the SA bus.
3. The addressed memory board asserts M16# as a result of decoding the LA lines, indicating to the system board's bus control logic that it is capable of handling a 16-bit transfer without data bus steering being performed by the steering logic on the system board.
4. If this is a write bus cycle, the microprocessor's output data is gated onto the SD bus halfway through address time and remains on the SD bus until half a BCLK cycle into the next bus cycle (half-way through address time of the next bus cycle).
5. At the end of address time, the trailing edge of BALE causes two events to take place:
   a) 16-bit ISA memory devices latch the LA lines (or latch the chip select) so the addressed device will not be deselected when the LA lines are pipelined before the end of the current bus cycle.
   b) the address latch on the system board and the bus control logic latch the lower twenty bits of the address, SA[19:0], so they remain static on the SA bus for the remainder of the bus cycle. The addressed device can safely complete decoding the SA address on the bus on the falling edge of BALE.

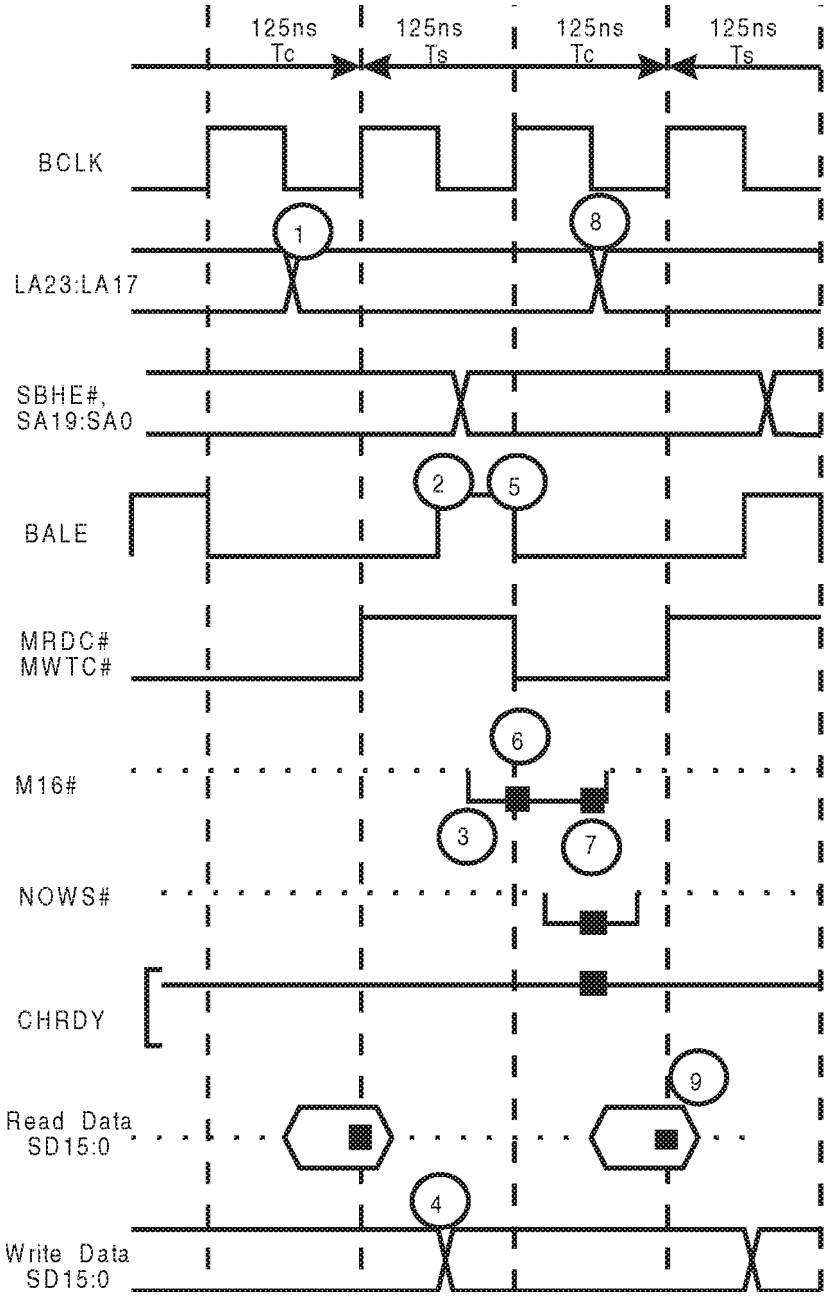


Figure 17-4. Zero-Wait-State Access to a 16-bit ISA Memory Device

6. The system board's bus control logic samples M16# at the end of address time to determine if the addressed device can take advantage of the MRDC# or MWTC# command lines being asserted immediately. M16# is sampled asserted and the appropriate command line (MRDC# or MWTC#) is asserted at the leading edge of data time. This command line remains asserted until the end of the bus cycle (end of Tc).
7. The system board's bus control logic samples M16# a second time at the midpoint of the first data time to determine if data bus steering is necessary. Since this is an access to a 16-bit device, no steering is necessary. The second sample point on M16# gives additional time for 16-bit memory devices that failed to assert M16# early enough for the first sample point. These devices either did not see LA[23:17] early enough to assert M16# by the first sample point, or they are waiting for BALE to be asserted before decoding LA[23:17]. Since the M16# line is asserted, the default ready timer samples the NOWS# and CHRDY lines to see if the bus cycle can end in zero ISA wait states. In this example, M16#, NOWS# and CHRDY are sampled asserted, forcing the default ready timer to assert the microprocessor's READY# line before the end of the current data time. In this way, faster ISA memory boards can complete a bus cycle in two rather than three BCLK cycles. (Note that the EISA Specification warns that devices should not assert NOWS# while deasserting CHRDY, implying that some bus controllers may not ignore NOWS# when CHRDY is deasserted.)
8. Since the LA lines have already done their job, (the addressed device has already decoded the LA lines and latched the chip select), the microprocessor is free to output the address for the next bus cycle.
9. During a read, the microprocessor latches the contents of the data bus, and the bus control logic deasserts the command line (SMRDC#, SMWTC#, MRDC#, MWTC#) thereby ending the bus cycle. During a write, the command line is deasserted, but the data remains on the SD bus for the first half of address time in the next bus cycle. This accommodates the hold time of the device being written to and doesn't disturb the device being addressed in the next bus cycle because the command line for that bus cycle hasn't been asserted yet.

Chapter 18

The Previous Chapter

The two chapters prior to this one described the ISA expansion bus and the bus cycle types that are used to transfer data over it.

This Chapter

This chapter provides a detailed discussion of the interrupt subsystem found in every ISA machine. This includes in-depth hardware and software descriptions of its operation and programming.

The Next Chapter

The next chapter, "Direct Memory Access (DMA)" provides a detailed description of the DMA subsystem found in all ISA machines.

What Is an Interrupt?

At various times, an I/O device within a PC system may require help, or service, from the microprocessor. The I/O device uses a signal line known as an interrupt request line to request the microprocessor's help.

The exact type of servicing a device requires is defined by the type of device and its current condition. For example, a parallel port connected to a printer may generate an interrupt request to ask the microprocessor for the next character to be sent to the printer. In this case, the microprocessor would respond by performing an I/O write bus cycle to send the next character to the parallel port.

A second example would be the interrupt request generated by a hard disk controller to tell the microprocessor that a data transfer to or from memory has been completed. If this were a disk read operation, for example, the microprocessor would respond to the interrupt request by using the data that has been transferred from the disk controller into memory.

As a third and final example, the keyboard interface generates an interrupt request to inform the microprocessor that a key has been pressed. The microprocessor should respond by performing an I/O read from the keyboard interface to get the keyboard character.

In each of these cases, the microprocessor responds to an interrupt request by executing a special program known as an interrupt service routine, or ISR. The interrupt service routine's task is to service the request. Different types of interrupt requests require different types of servicing on the part of the microprocessor. This means that there must be a separate interrupt service routine for each type of interrupt request that can be generated by the various hardware devices found in an ISA system.

An interrupt causes the microprocessor to temporarily suspend execution of the current program and forces it to jump to another program, the ISR. At the conclusion of the ISR, the microprocessor must then return to the original program flow at the point it was interrupted.

Microprocessor Response to Interrupt Request

This section provides a detailed explanation of how an Intel x86 microprocessor services an interrupt request. Figure 18-1 flowcharts the series of actions that must take place.

The Intel x86 microprocessors have one interrupt request input that is used to detect service requests from I/O devices. The interrupt request lines from the various I/O devices are not connected directly to the microprocessor's single interrupt request input, but rather to a device known as an interrupt controller. The actual device used in PC-compatible machines is the Intel 8259 programmable interrupt controller, or PIC. Up to eight interrupt request lines can be tied to one 8259 interrupt controller.

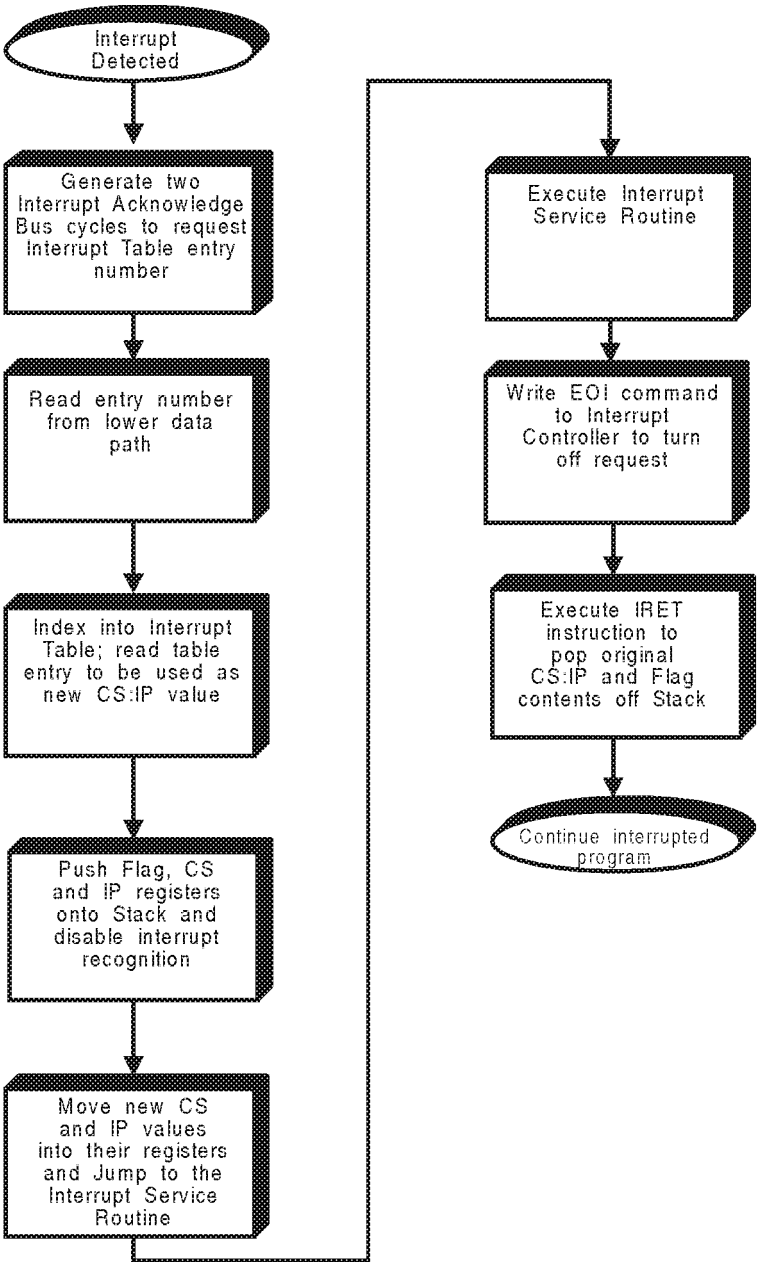


Figure 18-1. Flowchart of Interrupt Servicing

Figure 18-2 illustrates the 8259 PIC. At any given moment in time, one or more of the PIC's interrupt request inputs may be asserted. The PIC uses a priority scheme to determine which of the pending interrupt requests is the most important. It then passes that interrupt request along to the microprocessor by asserting the microprocessor's interrupt request, or INTR, input.

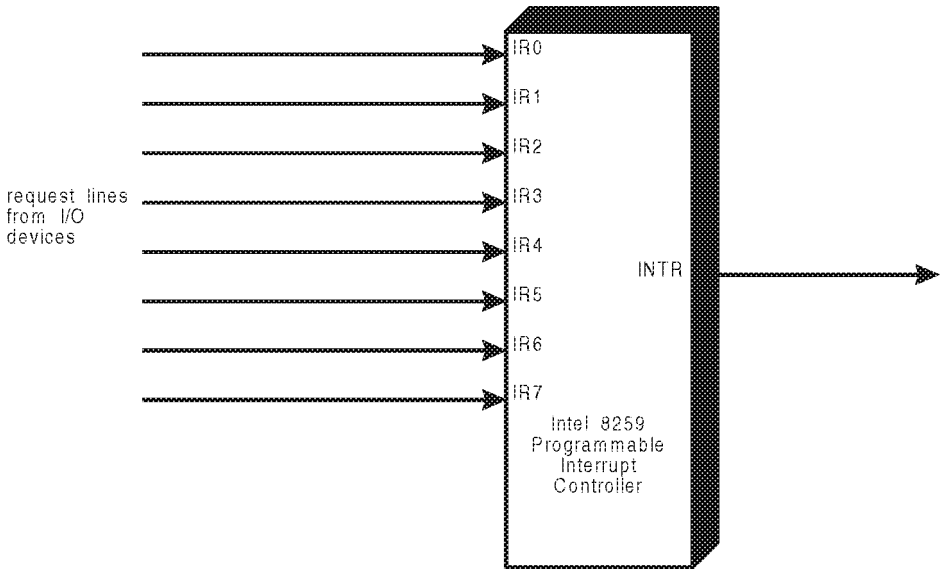


Figure 18-2. The Intel 8259 Programmable Interrupt Controller (PIC)

The INTR input to the microprocessor is referred to as the maskable interrupt request line. While executing a series of instructions, the programmer has the ability to turn off recognition of external interrupt requests by executing a CLI, or clear interrupt enable, instruction. The programmer is then ensured that subsequent instructions will not be interrupted. As soon as the critical series of instructions has been executed, the programmer re-enables recognition of external interrupt requests by executing an STI, or set interrupt enable, instruction. Any interrupt request pending on the INTR line will now be recognized by the microprocessor. The programmer should take care, however, to ensure that interrupt recognition isn't disabled for an appreciable amount of time. To do so could possibly cause problems.

As an example, the serial port may be generating an interrupt request because it has a character to be placed in memory. If the microprocessor doesn't read

the character from it on a timely basis, it may be written over by the next character received over the communications line. This is referred to as a data overrun condition.

Interrupt Acknowledge Bus Cycles

The microprocessor's immediate response to an interrupt request is to find out what device initiated the request. The microprocessor performs this inquiry by executing two, back-to-back interrupt acknowledge bus cycles. Logic external to the microprocessor converts the bus cycles into two pulses on the PIC's interrupt acknowledge (INTA) input. When the PIC senses these two interrupt acknowledges, it recognizes that the microprocessor is requesting the ID of the highest-priority interrupting device.

The microprocessor must determine the highest-priority requesting device so it can jump to the proper interrupt service routine. As with any other program, each of the interrupt service routines, or ISRs, reside in memory. In order to execute an ISR, the microprocessor must know the start memory address of each one of the ISRs. Intel sets aside the first 1,024 memory locations, from 00000h through 003FFh, as the default locations for the interrupt table. On the 80286 and higher microprocessors, the LIDT (Load interrupt descriptor table) instruction can be used to change the start address and the size of the interrupt table. LIDT may be executed in either real or protected mode, but if the interrupt table is relocated in real mode, then programs that initialize the interrupt table (such as those in expansion ROMs) may initialize the wrong locations in memory. The interrupt table may be located anywhere in memory in either real or protected mode. This discussion focuses on real mode operation and assumes that the interrupt table starts at address 00000h. Refer to figure 18-3. See the section entitled, "Protected Mode Interrupts" for differences between real mode and protected mode interrupts.

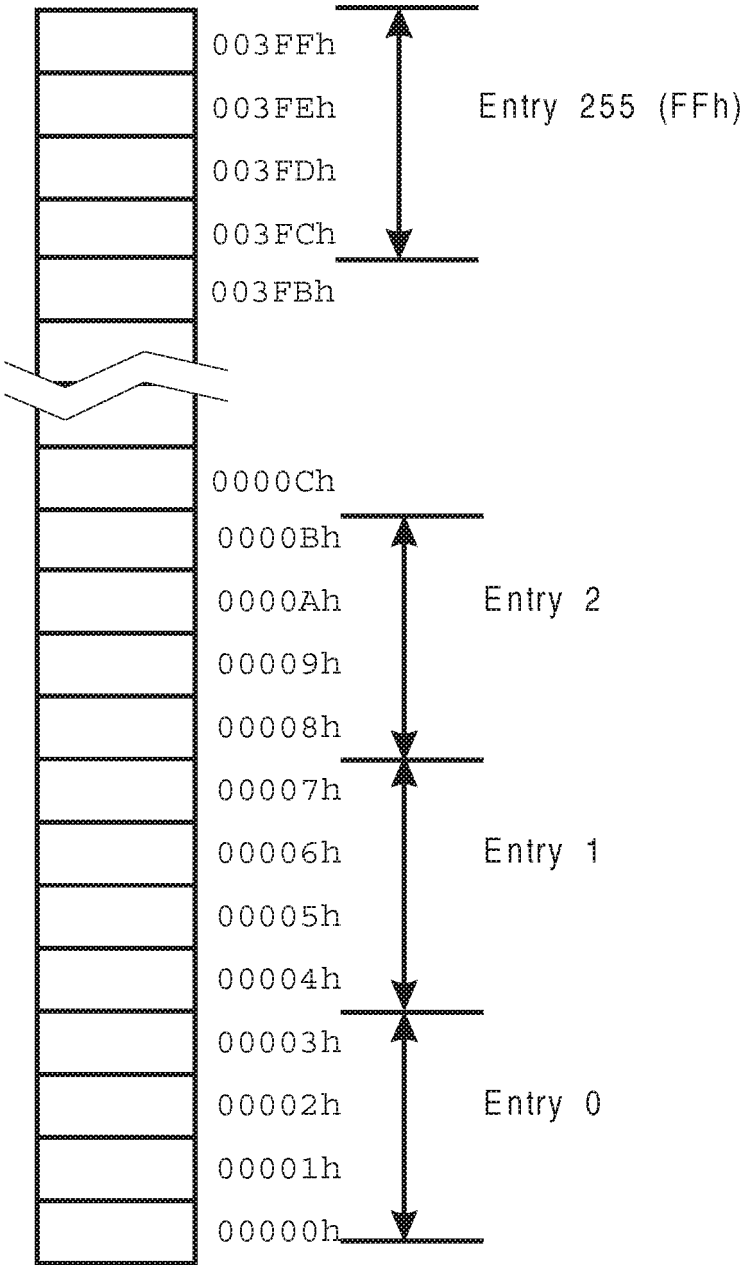


Figure 18-3. The Interrupt Table (in Real Mode)

The interrupt table is subdivided into 256 individual entries, each containing 4 bytes of information (256 divided into 1024 equals 4 locations per entry). The system ROMs that are shipped with ISA systems contain (among other things):

- The Power-On Self-Test, or POST.
- The interrupt service routines (ISRs) for the standard I/O devices that are shipped with the system.
- The BIOS routines for the standard I/O devices that are shipped with the system (discussed later).

In other words, the programmers who write the system ROM code (instructions) determine the start memory address of each of the interrupt service routines for the standard I/O devices that are shipped with the system. During the execution of the POST, the programmer copies the start memory address of each of these interrupt service routines into the proper entry in the interrupt table. At the completion of the POST, the interrupt table entries associated with each of the standard I/O devices have been filled with the start memory address of their respective interrupt service routines located in ROM memory.

When the microprocessor generates the two interrupt acknowledge bus cycles to request the ID of the interrupting device from the PIC, the PIC responds as follows:

- If the interrupt request line is high during the first acknowledge bus cycle (which it normally is), then the PIC resets the highest-priority bit that is currently set in the interrupt request register, or IRR, and sets the respective bit in the in-service register. If the interrupt request line is low, a phantom interrupt has occurred. These are described later.
- The second acknowledge causes the PIC to place an 8-bit number on the lower data path, D[7:0]. This number is commonly referred to by the following names: interrupt type code; interrupt ID; interrupt vector; interrupt table entry number. It is formed by adding the three-bit binary value of the highest-priority bit set in the in-service register to the base address stored in register ICW2 (covered later in this chapter).

Note: Although interrupt vector is the most commonly used term, interrupt table entry number is used in this chapter because it is the most correct of the four. Also note that many documents use the same abbreviation "ISR" to stand for two different items: interrupt service routine and in-service register. For clarity in this chapter, the abbreviation ISR always refers to interrupt service routine.

At the end of the second interrupt acknowledge bus cycle, the microprocessor reads the interrupt table entry number from the lower data path. The entry number must then be converted into the actual start memory address of the respective entry in the interrupt table. Since each entry in the interrupt table consists of four locations, the microprocessor indexes into the interrupt table by multiplying the entry number from the PIC by the number four. This will produce the actual start memory address of the desired table entry. This entry contains the start memory address of the target interrupt service routine in the form of a new CS and IP value.

The microprocessor then automatically performs one or two memory read bus cycles (the 80386DX, 80486 and Pentium processors perform one bus cycle, while the 80286 and the 80386SX must perform two) to read the four bytes of information from the interrupt table entry. These four bytes of information are temporarily stored inside the processor.

As an example, the keyboard interface uses the IRQ1 input to the master interrupt controller to generate an interrupt request when a key is pressed. the interrupt table entry number assigned to this input is 09h. This means that entry nine in the interrupt table contains the four byte start address of the keyboard interrupt service routine in memory.

The microprocessor calculates the actual start memory address of entry nine in the interrupt table by multiplying the interrupt table entry number, 09h in this case, by a factor of four. Nine times four equals 36 in decimal, or 24h, so the start memory address of entry nine in the interrupt table is 00024h.

The start address of the keyboard interrupt service routine is contained in memory addresses 00024h through 00027h. The microprocessor then:

1. Reads the contents of locations 00024h and 00025h.
2. Reads the contents of locations 00026h and 00027h.

The microprocessor will use these stored values later to fetch the first instruction of the interrupt service routine for the keyboard. First, the microprocessor must save its current place in the program that was interrupted.

Saving Pointer to Interrupted Program

The microprocessor is engaged in fetching, or reading, instructions from memory until the interrupt request is recognized. The only purpose of the microprocessor's code segment (CS) and instruction pointer (IP) registers is to point to the location in memory where the next instruction should be fetched from. The microprocessor had just completed the execution of an instruction and was about to fetch the next instruction from memory when the interrupt request was sensed. The CS and IP registers now point to where the microprocessor would have fetched its next instruction from had it not been interrupted. If the microprocessor is to resume execution successfully where it left off after servicing the request, it must save the contents of these two registers. In other words, the microprocessor must save a bookmark.

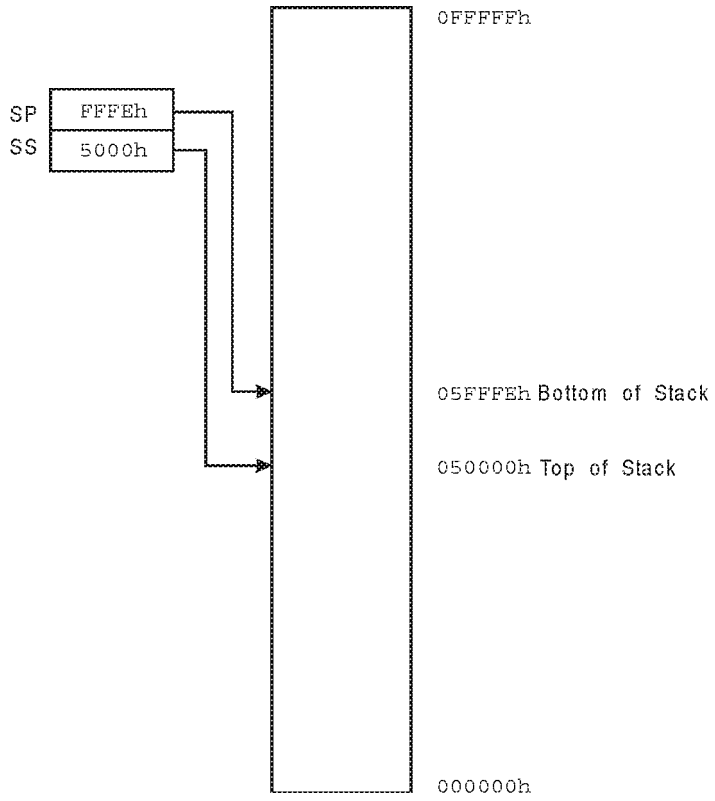


Figure 18-4. The Stack

The microprocessor's immediate response to an interrupt request is to run two interrupt acknowledge bus cycles to obtain the interrupt table entry number. The microprocessor then reads the new CS and IP values from the entry and saves them. Before these new CS and IP values are actually placed in their respective registers, the contents of the current CS and IP registers must be saved in stack memory. Refer to figure 18-4.

Assume that the stack is currently empty (contains no data). In the figure, the stack segment register, SS, contains 5000h, indicating that the stack segment starts at memory location 50000h. The stack pointer register, SP, contains FFFEh. In response to an interrupt request, the microprocessor automatically performs three push operations. Note that the programmer does not need to write code to cause these automatic push operations to occur.

Before jumping to the interrupt service routine, the microprocessor must first save its place in the current program flow by performing the following operations:

1. The microprocessor first pushes the contents of the flag register onto the stack. To do this, the microprocessor decrements SP by 2 (FFFEh - 2 = FFFCh), and then performs a 16-bit memory write to the address specified by SS:SP (5000:FFFCh). Figure 18-5 illustrates the stack's contents after the contents of the flag register has been pushed into stack locations 5FFFCh and 5FFFDh.

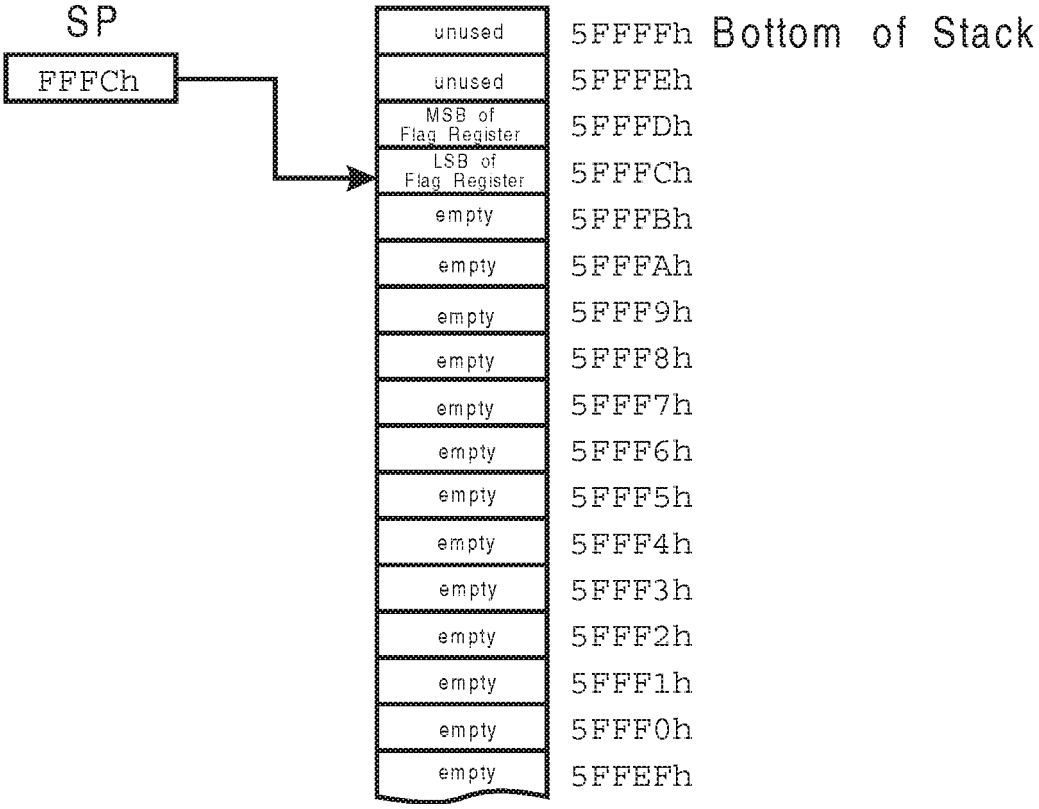


Figure 18-5. Stack After Flag Register Push Operation

2. The microprocessor then pushes the contents of the CS register onto the stack. To do this, the microprocessor decrements SP by 2, and then performs a 16-bit memory write to the address specified by SS:SP (5000:FFFAh). Figure 18-6 illustrates the stack's contents after the contents of the CS register has been pushed into stack locations 5FFFAh and 5FFFBh.

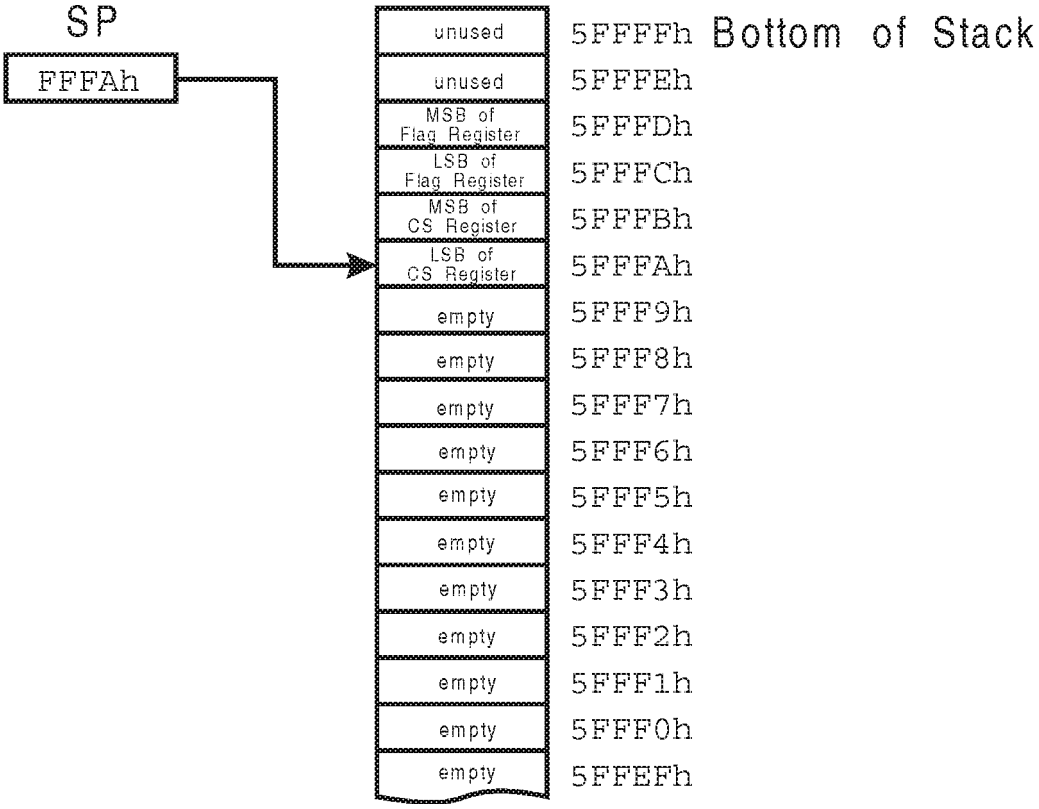


Figure 18-6. Stack After CS Register Push Operation

3. The microprocessor then pushes the contents of the IP register onto the stack. To do this, the microprocessor decrements SP by 2, and then performs a 16-bit memory write to the address specified by SS:SP (5000:FFF8h). Figure 18-7 illustrates the stack's contents after the contents of the IP register has been pushed into stack locations 5FFF8h and 5FFF9h.

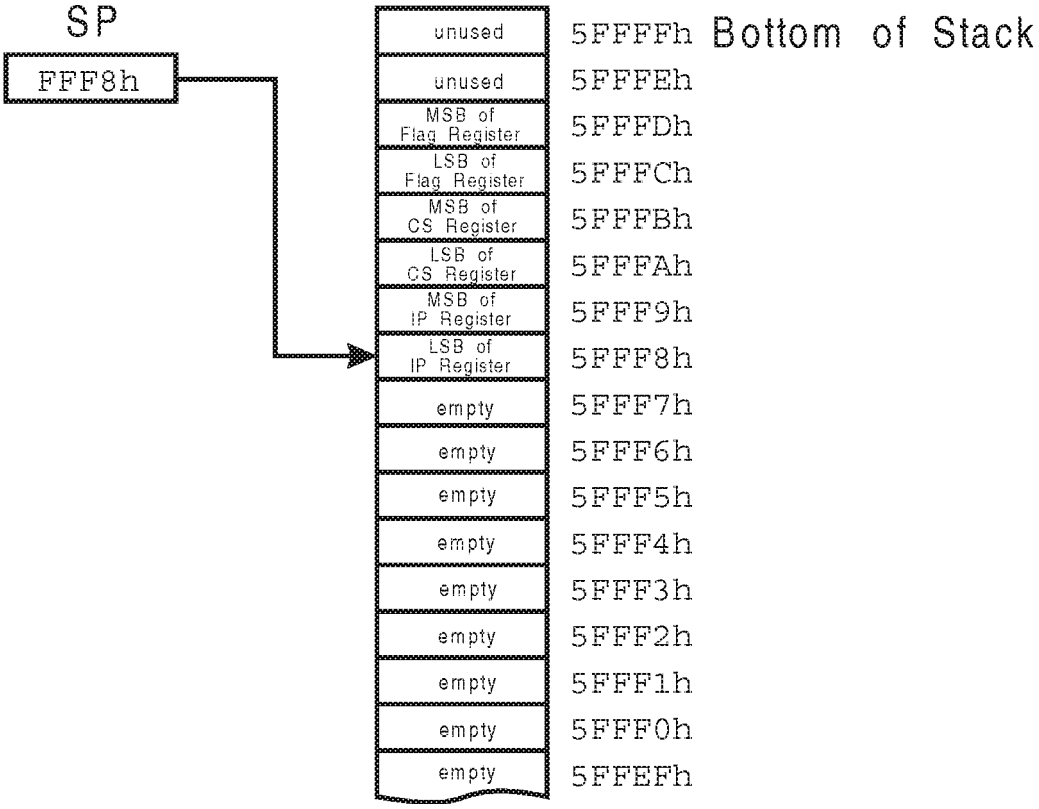


Figure 18-7. Stack After IP Register Push Operation

The microprocessor saves the contents of the Flag register because it is quite possible that the instruction that will be executed when normal program execution resumes will check the Flag bits to determine the result of the instruction executed just prior to the interruption.

Summarizing, the microprocessor's immediate response to an interrupt request is to get the starting address of the interrupt service routine (ISR) from the interrupt table. The starting address in the form of new CS and IP values are stored inside the microprocessor. Next the microprocessor pushes the contents of the Flag, CS and IP registers into stack memory. This is done so the microprocessor can resume its original program flow after the interrupt request servicing has been completed.

Clearing the Interrupt Enable Flag

Once the Flag, CS and IP registers have been saved to the stack, the microprocessor clears the interrupt enable flag to prevent another interrupt request from interrupting the interrupt service routine (ISR) currently running. If the programmer of an ISR chooses to permit another interrupt request to interrupt his interrupt service routine (called nesting of interrupts), the programmer must enable interrupts by writing an STI instruction. At the conclusion of an interrupt service routine, the contents of the Flag register are popped off the stack, restoring the original value of the interrupt enable flag.

Jumping to the ISR

The CS and IP values that were read from the interrupt table and stored in the un-named registers are moved into the CS and IP registers. The microprocessor then uses the contents of the CS and IP registers to determine the memory location to read the next instruction from. Since these two registers have just been loaded from the interrupt table entry, they now point to the first instruction of the appropriate interrupt service routine. In this way, the microprocessor begins to fetch and execute the instructions that make up the interrupt service routine. The interrupt table entry number assignments are detailed later in this chapter.

Does the microprocessor "jump to" the ISR or does it "call" the ISR? At this point in the sequence of events, the microprocessor is simply reloading CS and IP, which is exactly the same as a jump, or more specifically, a far jump. By previously saving the old CS and IP on the stack before jumping to the ISR, the microprocessor is doing some of the same things that it would do when executing a CALL instruction, but there are two significant differences. When executing a CALL instruction, the microprocessor does not save the Flag register and the new CS and IP come from the CALL instruction instead of from the interrupt table. As a result, it may be misleading to say that the microprocessor "calls" an interrupt service routine.

In the interrupt service routine, the programmer must first save the contents of any registers that will be altered within the service routine. This is accomplished by performing a series of one or more PUSH instructions to save the register contents on the stack. The programmer then performs the actions necessary to service the request that was issued by the I/O device. As an example, if this were the keyboard interrupt service routine, the programmer causes the microprocessor to perform an I/O read from the keyboard data port at I/O address 0060h to get the keyboard scan code (the character pressed on the keyboard).

After servicing the request, but before the end of the interrupt service routine, the programmer must write a non-specific EOI (End-of-Interrupt) command to the interrupt controller. This forces the interrupt controller to reset the highest-priority bit that is currently set in its in-service register. The interrupt controller uses the in-service register to maintain a fixed priority scheme so that higher-priority interrupt service routines are never interrupted by lower-priority I/O devices. The interrupt controller will assert a new interrupt request output (INTR) only when the interrupt request (IR pin) has a higher priority than the highest-priority bit that is set in the in-service register. By issuing the EOI command, the programmer is letting the interrupt controller know that the currently-highest-priority interrupt service routine has finished servicing the I/O device and that a lower-priority interrupt may now be passed on to the microprocessor.

At the end of the interrupt service routine, the programmer must perform a series of one or more POP instructions to restore the original contents of any registers PUSHed onto the stack at the start of the service routine.

Resuming the Interrupted Program

Each interrupt service routine must end with an IRET (interrupt Return) instruction. When the microprocessor executes the IRET instruction, it automatically performs three pop operations to read the original IP, CS and Flag register contents from the stack.

1. Refer to figure 18-8. The microprocessor first pops the old contents of the IP register off the stack. To do this, the microprocessor performs a 16-bit read from the memory locations specified by SS:SP (5000:FFF8h), and then increments SP by 2. The two bytes that were read are placed in the IP register.

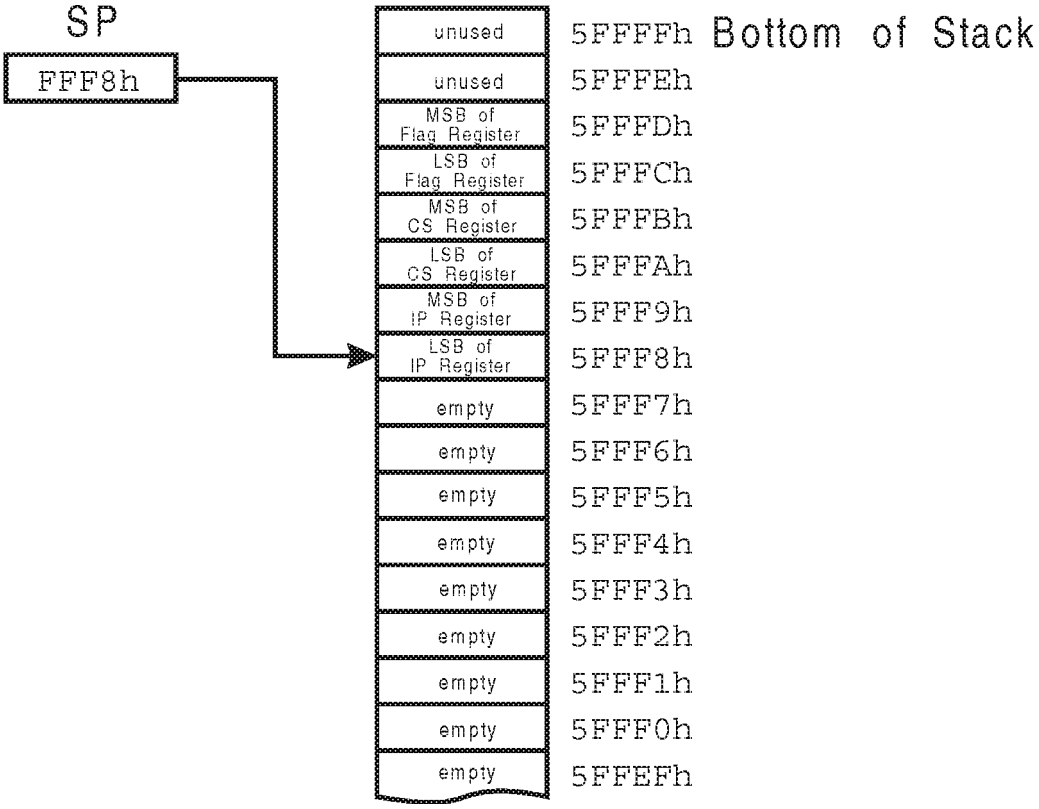


Figure 18-8. Stack Before Pop Operation to Read IP Register

2. Refer to figure 18-9. Next, the microprocessor pops the old contents of the CS register off the stack. To do this, the microprocessor performs a 16-bit read from the memory locations specified by SS:SP (5000:FFFAh), and then increments SP by 2. The two bytes that were read are placed in the CS register.

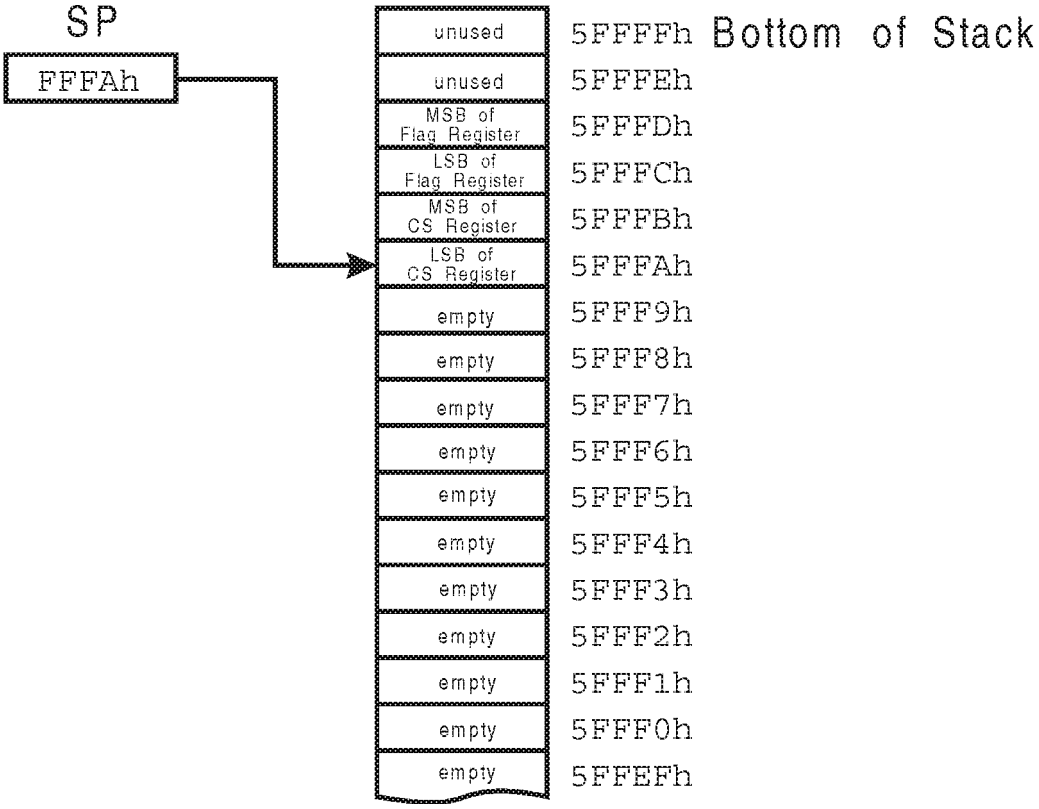


Figure 18-9. Stack Before Pop Operation to Read CS Register

3. Refer to figure 18-10. Next, the microprocessor pops the old contents of the Flag register off the stack. To do this, the microprocessor performs a 16-bit read from the memory locations specified by SS:SP (5000:FFFCh), and then increments SP by 2. The two bytes that were read are placed in the Flag register. The stack is now empty again.

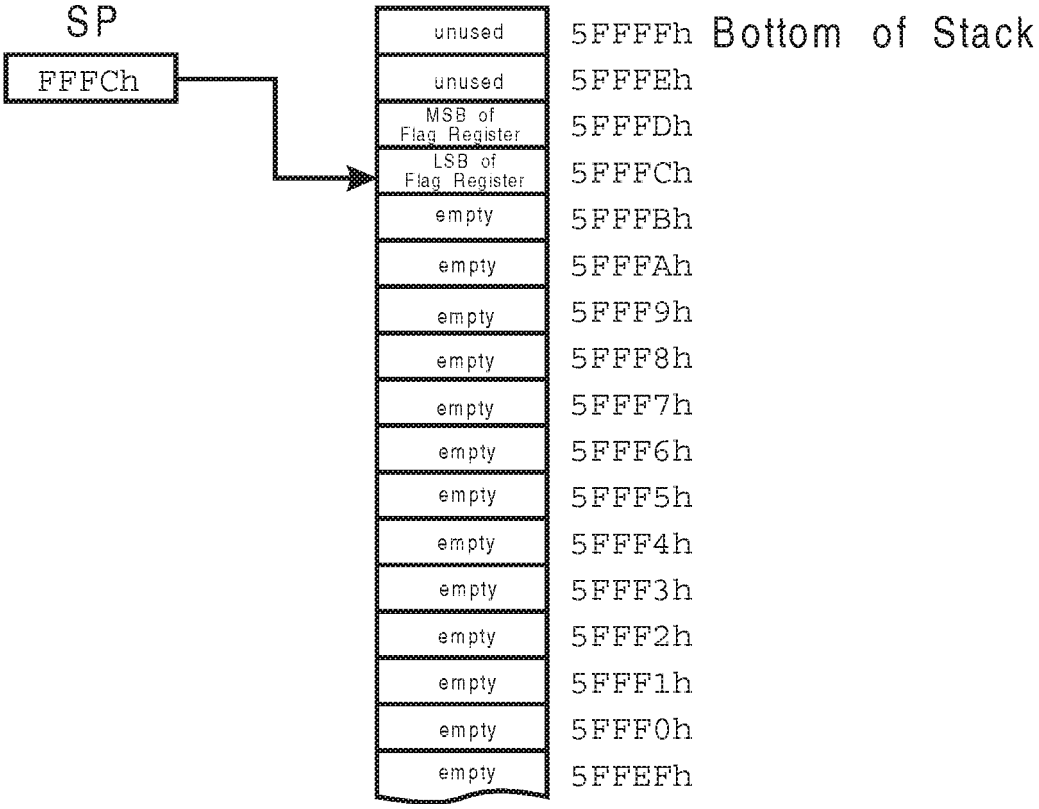


Figure 18-10. Stack Before Pop Operation to Read Flag Register

At this point, the CS, IP and Flag registers have been reloaded with the values that were present when the interrupt occurred. Specifically, the Flag register has been restored to reflect the status of the last instruction that was executed prior to the interrupt. Note that restoring the flag value also causes interrupts to be enabled once again. Finally, the microprocessor uses the restored contents of the CS and IP registers to fetch the next instruction from memory. In this way, the microprocessor has automatically returned to its original program flow after completing the interrupt service routine.

When One 8259 Interrupt Controller Isn't Enough

The PC and PC/XT only had one 8259 Programmable interrupt controller. This imposed a limit of eight interrupt request lines available for usage by I/O devices. In order to provide an expanded number of usable I/O interrupt request lines, the designers of the PC/AT added a second 8259 interrupt controller.

Refer to figure 18-11. As shown in the illustration, the two interrupt controller chips are in a master/slave configuration. The 8259 PIC connected directly to the microprocessor is the master interrupt controller, while the additional PIC is designated as a slave interrupt controller. The interrupt request output of the slave interrupt controller is connected to one of the interrupt request inputs of the master interrupt controller (IR2). This is referred to as cascading one interrupt controller through another.

In the original PC and PC/XT, the master interrupt controller's IR2 input pin is tied to the system's IRQ2 line on the PC's 8-bit connector. But in ISA machines, the slave interrupt controller's IR1 input pin is tied to the 8-bit connector pin that used to be IRQ2 in the PC. In ISA systems (at least in most of the various and competing ISA specifications), the pin is named IRQ9. Beware that IRQ2 and IRQ9 are the same pin. More information is in the section entitled, "IRQ2 Redirect."

Each of these highly programmable PIC chips contains a series of registers. By programming (writing values to) these registers during the POST, the programmer can specify the operational characteristics of each of the chips.

During the POST, a special 8-bit register in the master interrupt controller is programmed with a value indicating that the interrupt request Level Two (IR2) input of the master interrupt controller is connected to the slave interrupt controller rather than an I/O device. All of the other interrupt request inputs of the master interrupt controller are designated as I/O interrupt request inputs to be handled in the fashion described earlier.

The slave interrupt controller contains a register known as the slave ID register. During the POST, this register is programmed with the value 02h. This is its slave ID number and corresponds to the input on the master it is connected to.

In addition, during the POST, both of the interrupt controllers are programmed to use a fixed priority scheme. This means that, on each of the interrupt controllers, the interrupt request Level 0 (IR0) input has the highest-priority, while the interrupt request Level 7 (IR7) input has the lowest priority. If all of the inter-

rupt request lines were to be asserted at the same time, the interrupt request Level 0 input would be serviced first, followed by the others in decreasing order of priority.

Up until now, the terms interrupt request levels 0 through 7 have been used to identify the individual interrupt request inputs to the interrupt controller chip. The actual signal lines connected to the these inputs are designated as shown in figure 18-11. The signal names given to these lines are IRQ0 (interrupt request Level 0) through IRQ15. IR0 through IR7 are chip pin names. IRQ0 through IRQ15 are ISA system signal names.

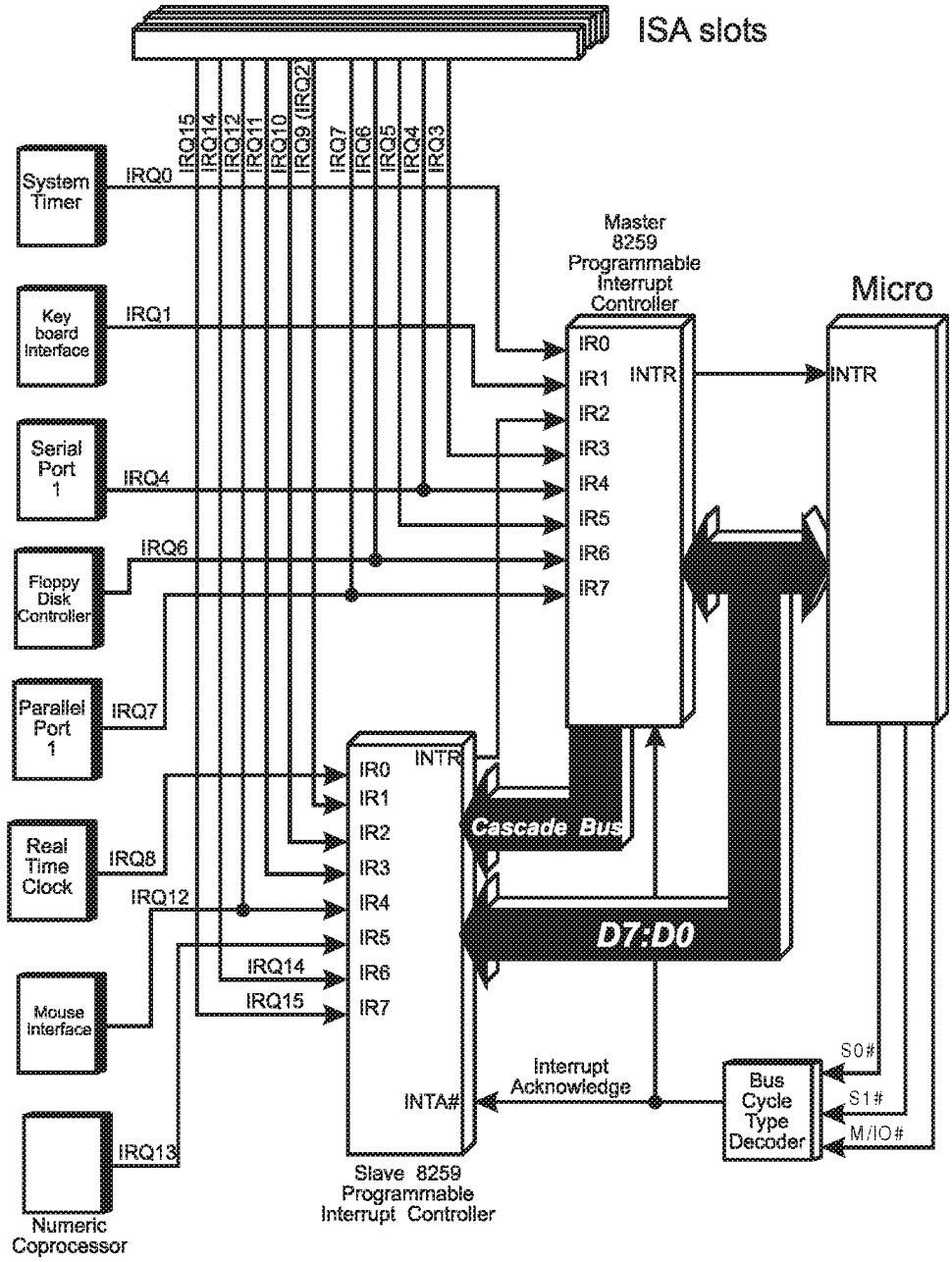


Figure 18-11. Cascaded Interrupt Controllers

IRQ0 through IRQ7 are connected to the master interrupt controller, while IRQ8 through IRQ15 are connected to the slave interrupt controller. Note that IRQ2 and IRQ9 are on the same ISA bus pin so there is not a unique IRQ2 interrupt pin in an ISA machine. More information is in the section entitled, "IRQ2 Redirect." With the exception of IRQ2, each of the interrupt request lines is available to be connected to the interrupt request output of an I/O device.

The fact that both controllers are programmed for fixed priority yields the following overall priority. IRQ0 is the highest-priority, followed immediately by IRQ1. Since IR2 on the master interrupt controller is the interrupt request output of the slave interrupt controller, the slave's eight inputs, IRQ8 through IRQ15, come next in the priority sequence. Finally, IRQ3 through IRQ7 are the lowest five in the priority sequence.

Refer to figure 18-11. Some of the interrupt request lines are dedicated to special I/O devices that reside on the system board.

- IRQ0 is connected to the output of Timer 0, frequently called the System Timer. This timer and other timers are discussed in detail in the chapter entitled "ISA Timers."
- IRQ1 is connected to the keyboard interrupt request line coming from the Keyboard Interface. The keyboard interface is described in the chapter entitled "Keyboard/Mouse Interface."
- IRQ8 is connected to the Alarm output of the Real Time Clock and Configuration RAM chip. This subject is covered in more detail in the chapter entitled "RTC and Configuration RAM."
- IRQ13 is connected to the ERROR# output of the Numeric Coprocessor. This is described in the chapter entitled "Numeric Coprocessors."

The remainder of the interrupt request lines are attached to the ISA connectors and are available for use by ISA cards. Although available for general use, the following ISA interrupt request lines are typically used by the indicated devices.

- IRQ3 is used as the interrupt request line for serial port 2. It is connected to all 8- and 16-bit ISA expansion slots.
- IRQ4 is used by the interrupt request output of serial port 1. It is connected to all 8- and 16-bit ISA expansion slots.
- IRQ5 is used by parallel port 2. It is connected to all 8- and 16-bit ISA expansion slots.
- IRQ6 is used by the Floppy Disk Controller on the system board. It is connected to all 8- and 16-bit ISA expansion slots.

- IRQ7, is typically used by parallel port 1. It is connected to all 8- and 16-bit ISA expansion slots.
- IRQ9 is available for use by ISA cards. It is connected to all 8- and 16-bit ISA expansion slots. In some systems, the pin is named IRQ2. See the discussion under the heading, "IRQ2 Redirect."
- IRQ10 is available for use by ISA cards. It is connected to all 16-bit ISA expansion slots.
- IRQ11 is available for use by ISA cards. It is connected to all 16-bit ISA expansion slots.
- IRQ12 is available for use by ISA cards, but is typically used by the mouse interface. It is connected to all 16-bit ISA expansion slots.
- IRQ14 is available for use by ISA cards. It is typically used by Hard Disk Controllers. It is connected to all 16-bit ISA expansion slots.
- IRQ15 is available for use by ISA cards. It is connected to all 16-bit ISA expansion slots.

Servicing Requests to Slave Interrupt Controller

When the slave interrupt controller detects an interrupt request on one or more of its inputs, it generates an interrupt request to the master interrupt controller. If no higher priority interrupts are currently being serviced, the master interrupt controller senses its IR2 input asserted and generates an interrupt request to the microprocessor on the INTR line. If higher-priority interrupts are being serviced (system timer or keyboard interrupts), then the master interrupt controller waits for the higher-priority interrupts to finish before interrupting the microprocessor. This enforces the fixed priority scheme.

The microprocessor requests the interrupt type by generating two back-to-back interrupt acknowledge bus cycles. When the master interrupt controller detects the double interrupt acknowledge, it interprets this as the microprocessor's request for the interrupt type. If IR2, the output of the slave interrupt controller, is the highest-priority request pending on the master's inputs, the master commands the slave to supply the interrupt type to the microprocessor.

To do this, the master interrupt controller drives the slave ID, a 2 (010b), onto the three-bit cascade bus. The slave interrupt controller compares the slave ID present on the cascade bus to its slave ID (written into its slave ID register during the POST) and responds by placing the proper interrupt table entry number onto the lower data path, D[7:0].

At the end of the second interrupt acknowledge bus cycle, the microprocessor reads the interrupt table entry number from the lower data path. It multiplies the entry number by four to produce the start memory address of the entry in the interrupt table, and reads the four bytes of information from the entry. After saving the current CS, IP and Flag register on the stack, the microprocessor places the four bytes from the interrupt table in the CS and IP registers.

The microprocessor then uses the contents of the CS and IP registers to identify the memory location to read the next instruction from. This is the address of the first instruction in the target interrupt service routine. At the end of the interrupt service routine, the programmer must issue two non-specific EOI (End-of-Interrupt) commands:

- One to the master interrupt controller to reset bit 2 in its in-service register. Note that bit 2 was the highest-priority bit that was set before the EOI command was issued, due to the fixed priority scheme. Therefore, the non-specific EOI command works correctly.
- One to the slave to reset the highest-priority bit that is currently set in its in-service register.

Interrupt Table Entry Assignments

During the POST, the two interrupt controllers are each programmed with the interrupt table entry numbers for each of their interrupt request inputs. Table 18-1 defines the entry number assignments.

Table 18-1. Standard ISA Interrupt Table entry number Assignments

IRQ Line	Assigned Interrupt entry number
0	08h
1	09h
3	0Bh
4	0Ch
5	0Dh
6	0Eh
7	0Fh
8	70h
9 (or 2)	71h
10	72h
11	73h
12	74h
13	75h
14	76h
15	77h

IRQ2 Redirect

The original IBM PC, PC/XT and compatibles have IRQ2 available on the 8-bit expansion slots and that pin is connected to IR2 of the master interrupt controller. An expansion device installed in the PC or PC/XT may use IRQ2 to request service. The software device driver that controls the expansion device will initialize entry 0Ah in the interrupt table during POST or when the device driver is loaded from disk and executed so that entry 0Ah points to the start address of the ISR in the device driver. Afterwards, when the expansion device issues an IRQ2 interrupt in the PC or PC/XT, the master interrupt controller gives interrupt entry number 0Ah (IR2 plus base of 8 = 10d = Ah) to the microprocessor. The microprocessor reads the CS and IP from entry 0Ah in the interrupt table and runs the ISR for the expansion device.

In an ISA machine, IRQ2 is IRQ9 and it is connected to IR1 of the slave interrupt controller. When the old expansion device generates what it thinks is IRQ2, the slave interrupt controller reports entry number 71h (IR1 plus base of 70h = 71h) to the microprocessor. The microprocessor reads the CS and IP from entry number 71h in the interrupt table, but it does not run the expansion device's ISR because entry number 71h was not initialized by the expansion device's driver. This is a problem.

The solution is to redirect IRQ9 interrupts (entry 71h) to what used to be IRQ2 (entry 0Ah). During POST or during initialization of the operating system, entry 71h in the interrupt table is initialized to point to the IRQ2 redirect routine in the boot ROM or in the operating system. The IRQ2 redirect routine uses the software interrupt instruction, INT 0A, to cause the microprocessor to run the ISR pointed to by entry 0Ah in the interrupt table, thus causing the microprocessor to run the expansion device's ISR. The INT instruction is described in the section entitled, "The Software Interrupt Instruction."

Recall that each interrupt service routine (ISR) must perform an End Of Interrupt (EOI) command to clear its bit in the in-service register. ISRs using IRQ lines that connect to the master interrupt controller must perform a single EOI command, whereas, ISRs using IRQ lines that connect to the slave interrupt controller must perform two EOI commands (one for the master and one for the slave). ISRs designed for IRQ2 expect it to be connected to the master interrupt controller (as it was in the PC and PC/XT) and therefore perform a single EOI command; however, since the IRQ2 line connects to the slave interrupt controller (IRQ9), two EOIs are needed. The following code sample shows how a typical IRQ2 Redirect routine transfers control from IRQ9 to IRQ2 and handles the second EOI command:

```
PUSH AX         ;save contents of AX register
MOV  AL,20      ;move 20h into AL register
OUT  A0,AL      ;write 20h to slave interrupt
                ;controller (EOI command)
POP  AX         ;restore AX
INT  0A         ;use software interrupt to
                ;redirect interrupt to interrupt
                ;table entry 0Ah (old IRQ2)
IRET            ;interrupt return from IRQ9
```

Note that the redirect code must perform the EOI for the slave interrupt controller since the ISR for IRQ2 performs only the master controller EOI.

Shareable Interrupts in ISA Machines

When expansion devices (add-in cards) that use interrupts are installed in an ISA system, they typically use one of the available interrupt request lines that is not used by the standard I/O devices (such as the printer port or the floppy controller). In some environments however, the demand for IRQ lines exceeds the number of available lines. In such cases, the temptation is great to share a single interrupt line between two devices.

The conditions required to share interrupts in an ISA environment include the following:

- The devices should generate interrupts by driving the IRQ line low and then not driving it, allowing the pull-up resistor to generate the rising edge (as opposed to driving the IRQ line high).
- Initialization of expansion devices must be done properly to ensure the starting addresses for the interrupt service routines of shared devices are linked.
- Expansion devices must implement an interrupt pending status bit.
- Interrupt service routines must check the interrupt pending bit for their devices and pass control to the next ISR in the chain.
- If a standard device is sharing the interrupt, then either the standard device must also meet all these requirements, or the interrupts must be guaranteed to never occur simultaneously.

These requirements are described in the next sections.

Generating the Interrupt Request

On ISA systems, the 8259 interrupt controllers are programmed to recognize a positive-going edge as a valid interrupt request on an input. There are a couple of ways to generate these low-to-high transitions. The ISA IRQ lines are pulled high by resistors on the system board. When an ISA expansion device must generate an interrupt request, the card may drive the line low and then stop driving the line (set its driver to a high-impedance state so it is neither driving high nor low), thus allowing the line to float high again due to the pull-up resistor. Alternatively, the expansion card can be designed to actively drive the IRQ line low and then actively drive it high. In either case, the low-to-high transition is registered as an interrupt request by the 8259 interrupt controller on the system board.

The 8259 specification requires that the 8259's IR input must remain high until after the leading edge of the first interrupt acknowledge bus cycle in order for the interrupt request to be valid. To meet this requirement, the expansion device must either stop driving the IRQ line and allow the pull-up resistor to keep the line high or the expansion device must actively drive the IRQ line high at least until the interrupt service routine has begun servicing the device.

If there is only one expansion device using a particular IRQ line, the interrupt is not being shared. When more than one expansion device is connected to the same IRQ line, that line is being shared.

For non-shared IRQ lines, only one expansion device is driving the IRQ line, so it doesn't matter whether the expansion device drives the IRQ line low, drives it high or doesn't drive it at all after the first interrupt acknowledge bus cycle. Contention will not occur on a non-shared IRQ line. In order to share the IRQ line, however, the expansion device should use the first method that was described above for generating the interrupt. That is, it should drive the line low, then stop driving it, allowing the pull-up resistor to cause the low-to-high transition. This can be accomplished with an open-collector or open-drain driver. With open-collector or open-drain drivers, the line is never actively driven high, so it is safe for one driver to drive the line low while the other drivers are not driving, and it is safe for several drivers to drive the same line low at the same time.

Each of the expansion devices that are sharing the IRQ line would have to generate interrupts this way. Interestingly enough, there is no requirement for manufacturers to use this technique, so interrupts are often not shareable.

Interrupt Table Initialization – Add-in Devices

Standard I/O devices that use interrupts normally have interrupt service routines that reside in the manufacturer's system ROM. When an expansion card using an interrupt is installed in an ISA system, the interrupt service routine (ISR) must be provided by the manufacturer. The ISR may be contained either in a ROM located on the card or on a diskette shipped with the card. In each case, the starting address of the ISR must be placed in the interrupt table entry assigned to the IRQ line used.

Assuming the IRQ line to be shared is already in use by a standard device, the starting address of the standard device will already occupy the entry. The expansion device then will overwrite the starting address of the standard device when it initializes. However, during initialization of the expansion card, the

contents of the interrupt entry to be used (which currently contains the starting address of the standard device's ISR) must be saved before overwriting the entry with its own ISR address. In this way, a link is created between the ISR address of the expansion device (which occupies the entry after initialization) and the ISR address of the standard device (which has been saved).

Designers of expansion cards who use ROM to store their ISR typically implement I/O locations on their cards to store the contents of the interrupt table entry they are about to overwrite. Designers who implement their ISR via installable device drivers typically leave space in the header of their ISR for saving the current value of the interrupt table entry.

Shared Interrupt Procedure

When two devices share an single interrupt request line, the device installed last will have its starting address in the interrupt table entry corresponding to the IRQ in use. Regardless of which device issued the interrupt request, the ISR of the device last installed will be executed first. In the previous example, this means that even though the standard device requests servicing, the expansion device's ISR occupies the interrupt table entry and its ISR will be executed.

The expansion device's ISR must first check the interrupt pending bit to determine if its device needs to be serviced. If so, the expansion device is serviced in normal fashion. If the interrupt pending bit indicates that no interrupt is pending for its device, the ISR will pass control to the ISR belonging to the standard device by using the value it previously saved. The standard device then would be serviced as usual.

If both of the devices generate interrupts simultaneously (or at least before the microprocessor runs the interrupt acknowledge bus cycle), then only one interrupt request will be registered in the 8259. Since the expansion device generated an interrupt, the interrupt pending bit that is on the expansion device would be set. The expansion device's ISR is run first since its CS and IP are in the interrupt table. The expansion device's ISR detects that the expansion device's interrupt pending bit is set, so it services the expansion device and returns to the interrupted program. Note that the standard device's interrupt never gets serviced.

There are two solutions to this problem. If the standard device has an interrupt pending bit, then the expansion device's ISR can do a software interrupt (INT) to run the standard device's ISR. Since the standard device's interrupt pending bit is set, the standard device's ISR will service the standard device correctly.

The expansion device's ISR must be written to always do the software interrupt, regardless of whether the expansion device generated an interrupt. In fact, in order to share an interrupt request line among two or more expansion devices, the expansion device's ISR must always do the software interrupt to link back to the other ISR's in the list.

If the standard device lacks an interrupt pending bit or if the standard device's ISR fails to test that bit, then it could be a mistake for the standard device's ISR to service the device if no interrupt had occurred.

The other solution is to guarantee that the two devices never generate interrupts simultaneously. (Easier said than done. There are many potholes in the road to sharing interrupts in ISA.)

Phantom Interrupts

On ISA systems, the 8259 interrupt controllers are programmed to recognize a positive-going edge as a valid interrupt request on an input. A transitory noise spike on an interrupt request line could be registered as a valid interrupt request in the 8259's interrupt request register (IRR). What happens next depends on how the expansion card is designed to drive the IRQ line. The methods for driving the IRQ line are described in the section above entitled, "Shareable Interrupts in ISA Machines."

After an interrupt is serviced (indeed, any time after the first interrupt acknowledge), it doesn't matter whether the expansion device drives the IRQ line low, drives it high or doesn't drive it at all, so long as no low-to-high transitions occur (unless, of course, the expansion device was requesting more service). This is true because the 8259 is programmed to latch low-to-high transitions only.

If the expansion device uses the shareable method of generating interrupts and a transitory noise spike occurs on the IRQ line, then the IRQ line will be high when the first interrupt acknowledge bus cycle occurs. In response to the interrupt acknowledge bus cycle, the 8259 will clear the bit in its IRR and will set the corresponding bit in its in-service register. The 8259 will then give the entry number of the expansion device to the microprocessor during the second interrupt acknowledge bus cycle. The microprocessor will then run the interrupt service routine for the expansion device. This is a problem because the expansion device did not request service. Some expansion devices will cease to function correctly if they are serviced prematurely. The solution to this problem is for the expansion device to internally have an interrupt request pending bit

which it sets along with generating any interrupt. The ISR should first check this bit before actually servicing the expansion device. If the interrupt were caused by a noise spike, the interrupt pending bit on the expansion card would not be set and the ISR would exit without servicing the expansion device.

If the expansion device uses a non-shareable method to generate the interrupt, then the expansion device either continues to drive the IRQ line high or it drives the line low after the interrupt is serviced instead of ceasing to drive the line. If the expansion device drives the IRQ line high after an interrupt has been serviced, then a noise spike on the IRQ line will have to be handled the same way that it was for the shareable technique (that is, the ISR should check the interrupt pending bit).

Things are more interesting when the expansion device drives the IRQ line low after an interrupt has been serviced. In this case, if a noise spike is registered by the 8259, the IRQ line is low when the first interrupt acknowledge bus cycle occurs. The 8259 requires that its IR input be high at least until the first interrupt acknowledge in order for the interrupt to be valid. In this case, the IR input is low, so the 8259 knows that a ghost or phantom interrupt has occurred.

When the second interrupt acknowledge bus cycle occurs, the 8259 must report an entry number to the microprocessor, but no valid interrupts have occurred. Intel designed the 8259 to automatically return the entry number for its number seven input in this case. On the master 8259, this will be 0Fh, the entry number for IRQ7. On the slave, it will be 77h, the entry number for IRQ15. The microprocessor will therefore run either the IRQ7 or the IRQ15 interrupt service routine.

As stated earlier, some expansion devices will cease to operate correctly if they are serviced when they did not request service. In these two ISRs, therefore, the programmer must perform a check to see if the IRQ7 or the IRQ15 was real. This is accomplished by reading the contents of the respective 8259's in-service register and checking to see if bit seven is really set. If it is, then the request is real and the programmer should execute the remainder of the interrupt service routine to service the request. If, on the other hand, the bit is clear, it was a phantom or ghost interrupt and the programmer should just return to the interrupted program flow without servicing the expansion device.

Programming the 8259

Introduction

This section is intended as an introduction to programming the 8259 interrupt controller. For a more detailed description, refer to the appropriate Intel manual.

Programming the Registers

All of the 8259 registers are accessed through two I/O locations. The master 8259's registers are accessed through I/O ports 20h and 21h, while the slave's are accessed through A0h and A1h. The 8259 accepts two types of commands: initialization command words, or ICWs; and operation command words, or OCWs.

Before the 8259 can be used to prioritize interrupt requests, it must be initialized to a known state. This is accomplished by writing a series of bytes to the ICWs and OCWs. The contents of these initialization bytes define the basic operational characteristics of the 8259. Tables 18-2 and 18-3 illustrate the sequence of ICWs and OCWs written to the 8259s during the ISA POST initialization. The ICW registers must be accessed in a predefined sequence. The programmer begins the sequence by writing to ICW1. This is accomplished by writing the desired data to port 20h or A0h (master or slave 8259) with data bit 4 set to a 1. The 8259 then expects three subsequent writes to port 21h or A1h with the values for ICW2 through ICW4. After all of the ICW registers have been loaded, any subsequent data written to port 21h or A1h is loaded into OCW1, the interrupt mask register.

Table 18-2. Initialization Sequence for the Master Interrupt Controller

Data Written to Master	Description
1) 15h written to I/O port 20h	An I/O write to port 20h with bit 4 = 1 accesses ICW1. Bit 0 set to 1 instructs the 8259 to expect ICW4. Bit 1 set to 0 indicates that one or more slaves are cascaded through the master's IRQ inputs. Bit 2 has no effect when the 8259s are programmed for 8086/8088 operation. Bit 3 set to 0 places the 8259 in edge-triggered mode. The remainder of the bits have no meaning when the 8259s are programmed for 8086/8088 operation.
2) 08h written to I/O port 21h	First access to port 21h after ICW1 is written accesses ICW2. Upper five bits of the data written to port 21h are the upper five bits of the master's interrupt ID. The lower three bits of the interrupt ID represents the level of interrupt currently being serviced. This means that the master 8259's eight inputs have interrupt ID's ranging from 08h through 0Fh.
3) 04h written to I/O port 21h	After writing ICW2, the next write to port 21h accesses ICW3. The data written to this register indicates which of the master's IRQ inputs are connected to slave 8259s. Since bit 2 is set to 1 and the remaining bits are 0, IRQ2 is connected to a slave while the remaining inputs are connected to I/O devices.
4) 01h written to I/O port 21h	After writing ICW3, the next write to port 21h accesses ICW4. The data written to this register indicates the following: bit 0 = 1: The system is based on an 8086/8088, not an 8080/8085. This means that the 8259 must send an interrupt ID to the processor in response to interrupt acknowledge, not a call instruction. bit 1 = 0: The programmer must write a non-specific End-of-Interrupt, or EOI, command to the 8259 to reset the highest-priority bit that was set in the in-service register at the end of each interrupt service routine. bits 3:2 = 00b — non-buffered mode. Has no effect. bit 4 = 0 — special fully-nested mode off. bits 7:5 = 000b — not used.
5) FBh written to I/O port 21h	After the four ICW registers have been loaded, any subsequent write to port 21h accesses OCW1, the interrupt mask register(IMR). Writing a FBh to this register disables all interrupt request inputs other than IRQ2 (requests from the slave 8259).

Table 18-3. Initialization Sequence for the Slave Interrupt Controller

Data Written to Slave	Description
1) 15h written to I/O port A0h	See description of write to master's port 20h.
2) 70h written to I/O port A1h	First access to port A1h after ICW1 is written accesses ICW2. Upper five bits of the data written to port A1h are the upper five bits of the interrupt ID. The lower three bits of the interrupt ID represents the level of interrupt currently being serviced. This means that the slave 8259's eight inputs have interrupt ID's ranging from 70h through 77h.
3) 02h written to I/O port A1h	After writing ICW2, the next write to port A1h accesses ICW3. The data written to this register loads the slave's ID into ICW3. Since the slave is connected to IRQ2 on the master, it's slave ID is two.
4) 01h written to I/O port A1h	After writing ICW3, the next write to port A1h accesses ICW4. The data written to this register indicates the following: bit 0 = 1: The system is based on an 8086/8088, not an 8080/8085. This means that the 8259 must send an interrupt ID to the processor in response to interrupt acknowledge, not a call instruction. bit 1 = 0: The programmer must write a non-specific End-of-Interrupt, or EOI, command to the 8259 to reset the highest-priority bit that was set in the in-service register at the end of each interrupt service routine. bits 3:2 = 00b — non-buffered mode. Has no effect. bit 4 = 0 — special fully-nested mode off. bits 7:5 = 000b — not used.
5) FFh written to I/O port A1h	After the four ICW registers have been loaded, any subsequent writes to port A1h access OCW1, the interrupt mask register. Writing a FFh to this register disables all of the slave's interrupt request inputs.

After the interrupt controllers have been initialized, any subsequent read from or write to I/O port 21h or A1h accesses the interrupt mask register, OCW1, in either the master or slave interrupt controller, respectively.

In an interrupt service routine, the programmer must issue a non-specific End-of-Interrupt, or EOI, command to the interrupt controller to clear the highest-priority bit that was set in the in-service register. This is accomplished by writ-

ing 20h to port 20h or A0h, OCW2. In an interrupt service routine for a request originating on the slave, a non-specific EOI command must be issued to both the slave and the master to clear the respective bits in both of their in-service registers.

The programmer writes to OCW3 under two circumstances:

- To set or reset the special mask mode. For further information, refer to the Intel hardware reference manual.
- To identify a register to be read on a subsequent read.

To read from the in-service register or the interrupt request register (IRR), the programmer first writes either 03h (in-service register) or 02h (IRR) to OCW3 at port 20h or A0h. The next time port 20h is read from, the contents of the identified register is placed on the data bus. There is no need to write to OCW3 every time a read is to be performed from the IRR or in-service register (as long as the same register is to be read from again). After initialization, the 8259 will return the status of the IRR when port 20h or A0h is read from.

Non-Maskable Interrupt Requests (NMI)

In addition to the maskable interrupt request input, INTR, Intel x86 microprocessors have an additional interrupt request input known as the NMI input. Unlike the maskable interrupt request input, the non-maskable interrupt request input, or NMI, can not be masked out by the programmer. In other words, if the NMI input is asserted, the microprocessor must immediately service that interrupt request. The NMI line is typically used to report serious or fatal hardware failures to the microprocessor.

A classic example would be a memory parity error. When the microprocessor senses an asserted signal on the NMI input, it responds by first pushing the Flag, CS and IP registers onto the stack.

Rather than request the interrupt table entry number from the interrupt controller, however, the microprocessor automatically accesses entry two in the interrupt table. This entry is dedicated to the NMI interrupt. During the POST, the programmer writes the start address of the NMI interrupt service routine located in ROM memory into entry two of the interrupt table.

After saving the three registers on the stack, the microprocessor then jumps to the NMI interrupt service routine by loading CS and IP from entry two. In the

NMI interrupt service routine, the programmer reads the contents of the following I/O ports to determine the cause of the NMI:

- system control port B at I/O address 0061h.
- system control port A at I/O address 0092h.

The error is then reported on the screen.

Tables 18-4 and 18-5 define the bit assignment of these two ports and identify the bits related to NMI with an asterisk. The bit assignments shown are fairly standard, but they are different on some systems. The NMI interrupt service routine that examines these bits is normally in the system's boot ROM. Since the boot ROM is written specifically for a particular style of system board, the ROM will know what the bit assignments are. The meaning of the bits in system control port B at I/O address 0061h are different for a read versus a write. For more information, see the "ISA I/O Addresses" section in the Appendix.

Table 18-4. System Control Port B at I/O Address 0061h (Read)

Bit	Function
7*	System Board RAM Parity Check
6*	Channel Check
5	Timer 2 (Speaker Timer) Output
4	Refresh Request
3	Channel Check Enabled
2	Parity Check Enabled
1	Speaker Data Enable
0	Timer 2 Gate to speaker

Table 18-5. System Control Port A at I/O Address 0092h

Bit	Function
7	Fixed Disk Activity LED
6	Fixed Disk Activity LED
5	Reserved
4*	Watchdog Timer Status
3	Security Lock Latch
2	Reserved
1	Alternate Gate A20
0	Alternate Hot Reset

As illustrated in the preceding tables and in figure 18-12, there are three possible causes for an NMI:

- System board RAM parity check. The chapter entitled "RAM Memory: Theory of Operation," contains a detailed description of the system board RAM parity check logic.
- Channel check. The chapter entitled "ISA Bus Structure," provides a detailed description of the channel check logic.
- Watchdog timer status. The chapter entitled "ISA Timers," provides a detailed description of the watchdog timer.

The programmer can selectively enable or disable the ability of external logic to generate an NMI. This is accomplished by writing the desired value to I/O port 0070h. A 1 written to bit 7 disables the NMI logic's ability to generate an NMI, while a 0 enables it. For additional information, see the chapter entitled "ISA Bus Structure," under the heading "Error Reporting Signal."

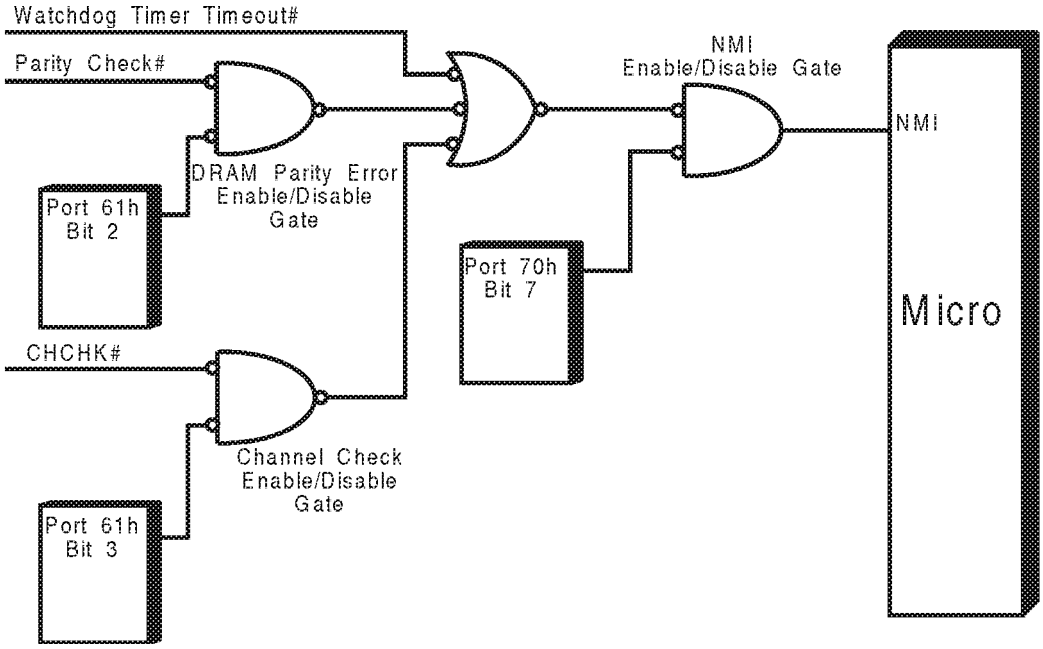


Figure 18-12. Causes of a NMI

Software Interrupts

There are two types of software conditions that can cause an interrupt to an Intel x86 microprocessor:

- Exception interrupts.
- Software interrupts generated by the INT (interrupt) instruction.

Both types of software conditions are discussed in the following paragraphs.

Software Exceptions

Generally speaking, an exception interrupt results when the microprocessor attempts to execute an instruction and incurs an error while during so. A classic example would be an attempt to divide a number by zero. When an Intel x86 microprocessor attempts execute of an instruction that would cause a divide-by-zero condition, the microprocessor immediately generates a divide-by-zero exception interrupt. Entry 0 in the interrupt table is dedicated to this condition and points to the start address of the divide-by-zero interrupt service routine supplied by the programmer.

Various other types of illegal software conditions can cause exception interrupts as well. When Intel introduced the 8086, Intel reserved thirty two entries in the lower-most part of the interrupt table to point to the interrupt service routines for various software error conditions. Table 18-6 defines the conditions and their respective entries in the interrupt table for an 80386 microprocessor. Due to the hardware orientation of this book, a detailed description of each of these conditions is not included. This information may be found in the Intel programmer's reference manuals.

Notice that in table 18-6 exception interrupts use entry numbers 8 through 10h which coincide with entry numbers used by hardware devices. To eliminate possible conflicts, the exception handlers (ISRs) share the table entries with the hardware devices just as two hardware devices share the same entry.

Table 18-6. Interrupt Table Entry Assignment

entry number	Assigned To
0	Divide-By-Zero Exception
1	Single Step, or Trap, Interrupt
2	NMI
3	Breakpoint
4	Interrupt on Overflow
5	BOUND Range Exceeded Exception
6	Invalid Opcode Exception
7	Processor Extension Not Available Exception
8	Double Exception Detected
9	Processor Extension Segment Overrun Exception
Ah	Invalid Task State Segment
Bh	Segment Not Present
Ch	Stack Segment Overrun or Not Present Exception
Dh	General Protection Exception
Eh	Page Fault
Fh	Reserved by Intel
10h	Coprocessor Error
11h to 1Fh	Reserved by Intel

Software Interrupt Instruction

The interrupt instruction allows the program to simulate a hardware interrupt request. This instruction take the following format:

```
INT  13
```

When the microprocessor executes the example instruction shown above, it reacts very much as it would to a hardware interrupt request received on the maskable interrupt request line, INTR.

It first saves the contents of the Flag, CS and IP registers on the stack. At this time, CS and IP point to the instruction immediately after the INT instruction. Since the interrupt was not actually caused by a hardware interrupt request on the INTR line, the microprocessor doesn't perform two interrupt acknowledge bus cycles to request the interrupt table entry number from the 8259 interrupt controller. Instead, the interrupt table entry number is supplied by the hexadecimal number to the right of the INT instruction.

In the example above, the programmer has designated entry number 13h (a decimal 19). The microprocessor multiplies this number by four (13h x 4 = 4Ch) to yield the start memory address of entry 13h in the interrupt table.

Just as it would for a hardware request, the microprocessor then reads the four bytes of information starting at this memory address and places them into the CS and IP registers. The CS and IP registers now point to the start address of the INT 13h interrupt service routine in memory. The microprocessor begins to fetch and execute the instructions that make up this routine. As defined by IBM, entry 13h in the interrupt table always points to the disk BIOS routine. At the conclusion of the routine, execution of the IRET (interrupt Return) instruction pops the original CS and IP off the stack, causing the microprocessor to resume executing the original program at the instruction immediately after the INT instruction.

The true value of the INT instruction lies in the fact that a programmer can call (jump to) a routine (e.g., a BIOS routine) without knowing where it actually resides in memory. All the programmer needs to know is the entry number that contains its start address.

When Microsoft first came out with the disk operating system (DOS) for the IBM PC, it was identified as version 1.0. An operating system such as MS-DOS is really nothing more than a collection of useful routines that the programmer can call (jump to) to perform various commonly used functions. The MS-DOS routines allow the programmer to control the various devices in the system without having an intimate knowledge of the operations required to control them.

One direction that Microsoft could have taken would have been to publish a list of the start memory addresses of the various routines that make up MS-DOS. A programmer could then jump to the start memory address of the appropriate routine to accomplish a particular task. This method would have caused tremendous compatibility problems.

Consider this. Not too long after version 1.0 of MS-DOS was released, IBM released version 1.1. This was followed rather closely by various other versions leading up to the version currently on the market. Now consider the fact that as the various versions of MS-DOS were issued, the size, functionality, and placement of the various routines that make up MS-DOS were altered. This means that the list of start memory addresses for the various routines specified in version 1.0 is no longer valid for any of the other versions. In fact, it would

require a separate list of start memory addresses for the routines in each and every version of MS-DOS on the market.

If the designers of an applications program had accessed the MS-DOS routines by jumping to the start memory addresses of the various routines embedded in version of 1.0 of MS-DOS, that version of the applications program would only run correctly with version 1.0 of MS-DOS. They would have had to issue a separate version of the applications program for every version of MS-DOS.

It would also mean that the end-user would have to know the version of MS-DOS that they are running in order to buy the correct version of the application program they want to run. The end-user could never upgrade MS-DOS without relegating virtually all of his or her application's programs to the scrap heap. This obviously would have been an untenable solution to the problem.

As MS-DOS is loaded into memory when the machine is powered up, the MS-DOS load procedure includes the automatic placement of the start memory addresses for the various MS-DOS routines into pre-defined entries in the interrupt table. IBM guaranteed applications program developers that the pointers to the various MS-DOS routines would always reside in the same entries in the interrupt table, irrespective of the MS-DOS version. This means that an application program developer doesn't need to know the actual start memory address of a MS-DOS routine that he or she wants to use. To call the routine, all the programmer has to do is specify the correct entry number in an interrupt instruction. The Microsoft and IBM technical reference guides contain information detailing the routines called by each interrupt entry.

MS-DOS routines aren't the only ones that can be called using the interrupt instruction. The BIOS (basic input/output system) routines are a collection of useful routines written by the PC manufacturer and embedded in the system ROMs. There is one routine for each standard I/O device shipped with the system. Some examples would be the keyboard, floppy disk, and hard disk BIOS routines.

Rather than expect each application's programmer to understand the proper sequence of I/O writes and reads required in order to control and sense the status of various I/O devices, the PC manufacturer supplies the BIOS routines. All the applications programmer has to know is how to call the appropriate routine to access the target I/O device. During the POST, the start memory address of each of the BIOS routines is stored in predefined entries in the interrupt table. The PC manufacturer supplies a list of the interrupt entries assigned

to each BIOS routine. This list is typically available in the technical reference guide for the PC system.

Protected Mode Interrupts

The discussions thus far in this chapter have assumed that the microprocessor is in real mode. Due to the hardware orientation of this book, a detailed discussion of protected mode is not included, but some of the basic similarities and differences between real mode and protected mode interrupts are described here.

In protected mode, the interrupt table is more properly named the interrupt descriptor table, or IDT, because it contains 8-byte interrupt descriptors. The IDT can be located anywhere in memory. The location (base address) and size (limit) of the IDT is specified by the interrupt descriptor table register, IDTR, which is the same register that specifies the location of the interrupt table in real mode.

The 8259 interrupt controllers handle interrupt requests the same way in protected mode as they did in real mode. They report interrupt table entry numbers to the microprocessor on the second interrupt acknowledge bus cycle.

When the microprocessor receives an entry number from the 8259, the microprocessor multiplies the entry number by 8 if it is in protected mode (It multiplies by 4 if it is in real mode.). It then adds the result to the IDT base address from the IDTR. The final result is the address of the interrupt descriptor in memory. The microprocessor multiplies the entry number by 8 because the interrupt descriptors are each 8 bytes long.

The microprocessor then runs memory read bus cycles to the calculated address to read in all eight bytes of the interrupt descriptor. The interrupt descriptor includes a new CS register value, a new IP register value and other bytes which control how protected mode will handle the interrupt. For an 80386 or higher microprocessor, the interrupt descriptor may contain a new EIP instead of just IP.

Similarly to real mode, the microprocessor pushes the old Flags, CS and IP registers onto the stack before putting the new CS and IP values that it read in from the interrupt descriptor into its CS and IP registers. Depending on the setting of the control bytes in the interrupt descriptor, the microprocessor may also push the old SS and SP registers before pushing the CS and IP registers.

Whenever a segment register is reloaded in protected mode, the microprocessor automatically reads a descriptor from either the global descriptor table (GDT) or the local descriptor table (LDT). There are several types of interrupt descriptors, so to just get the basic idea across, let's use a segment descriptor. Segment descriptors are described in the chapter entitled "The 80286 Microprocessor" in the section entitled "Segment Register Usage in Protected Mode."

Since the interrupt caused the microprocessor to reload its CS segment register with the value from the interrupt descriptor, the microprocessor now reads the segment descriptor that the CS register points to, from the GDT or the LDT. The segment descriptor defines the base address and the limit of the new code segment. The microprocessor adds the new code segment base address to the new IP (that it already read from the interrupt descriptor) and forms the address of the first instruction of the interrupt service routine (ISR). The microprocessor is then fetches the first instruction of the ISR.

Since the microprocessor is in protected mode, the ISR must either be written for protected mode, or the microprocessor must be placed in virtual-8086 mode or back in real mode before running a real mode ISR.

Chapter 19

The Previous Chapter

Definitions of the ISA bus signal lines and functions were introduced along with details about ISA bus cycles. Details of the implementation and operation of interrupts within the ISA environment were provided.

This Chapter

Transfers between memory and I/O devices can be handled either by the microprocessor or by the direct memory access (DMA) controller. DMA allows the processor to continue to execute instructions from its prefetch queue or cache while transfers are being made between I/O and memory. This chapter explains the process, logic and timing of the ISA DMA subsystem.

The Next Chapter

ISA bus mastering is explained, including its relationship to DMA and its capabilities and limitations in the ISA structure.

DMA Concept

When using direct memory access (DMA) to transfer a block of information between an I/O device and memory, the microprocessor must still initiate the block data transfer between the I/O device and memory, but the actual data transfer and its termination are handled solely by the DMA controller (DMAC). This frees up the microprocessor to continue with other productive activities while the data transfer is taking place. Once a DMA block data transfer has been initiated, the DMA controller and the I/O device do not disturb the microprocessor again until the entire block of data has been transferred. The DMAC used in ISA machines performs a "fly-by" transfer; that is, it is able to transfer data between memory and an I/O device (read and write) in a single DMA cycle without latching the data internally, a job that would normally require two separate bus cycles (a read followed by a write) by the microprocessor.

Upon completion of the overall data transfer, the I/O device interrupts the microprocessor to indicate completion. In response, the microprocessor temporarily suspends its current task and performs an I/O read from the I/O device to check the completion status of the transfer. If the I/O device indicates no errors were encountered, the microprocessor may continue processing.

DMA Example

The DMAC used in ISA machines is the Intel 8237, providing four separate DMA channels. Two 8237s are used in a master/slave configuration, providing a total of seven DMA channels. Each DMA channel is used by a separate I/O device to handle block data transfers with memory.

Four distinct steps must occur to initiate and complete a DMA transfer.

- Set up the DMA channel for the transfer.
- Command the I/O device to initiate the block data transfer.
- Grant the system buses to the DMA controller so it can run the bus cycle.
- Notify the microprocessor that the transfer is complete.

Each DMA channel within the DMA controller has its own set of I/O registers that the programmer uses to set up the data transfer. The set of I/O registers associated with each DMA channel allows the programmer to specify:

- The Transfer Count (number of DMA bus cycles to run).
- The start memory address (start address in memory where data will be read from or written to).
- The direction of transfer with reference to memory (type of transfer).

After the DMA channel has been set up by the programmer, the I/O device must be programmed to initiate the overall block data transfer. As an example, the programmer would issue the proper series of I/O write commands to a floppy disk controller to initiate a disk read operation.

Having set up the respective DMA channel and issued the proper commands to the I/O device (in this case, the floppy disk controller and DMA channel two), the microprocessor can then go on to another task. The entire data transfer and its termination will be handled by the I/O device and its respective DMA channel. The following paragraphs describe an example data transfer between system memory and a floppy disk controller.

When performing a floppy disk transfer, the programmer may only perform data transfers in multiples of the sector size. When running MS-DOS, a sector on a floppy disk contains 512 bytes of information. This would be the smallest data transfer possible when transferring information between a disk drive and memory.

Refer to figure 19-1 for a block diagram of the components involved.

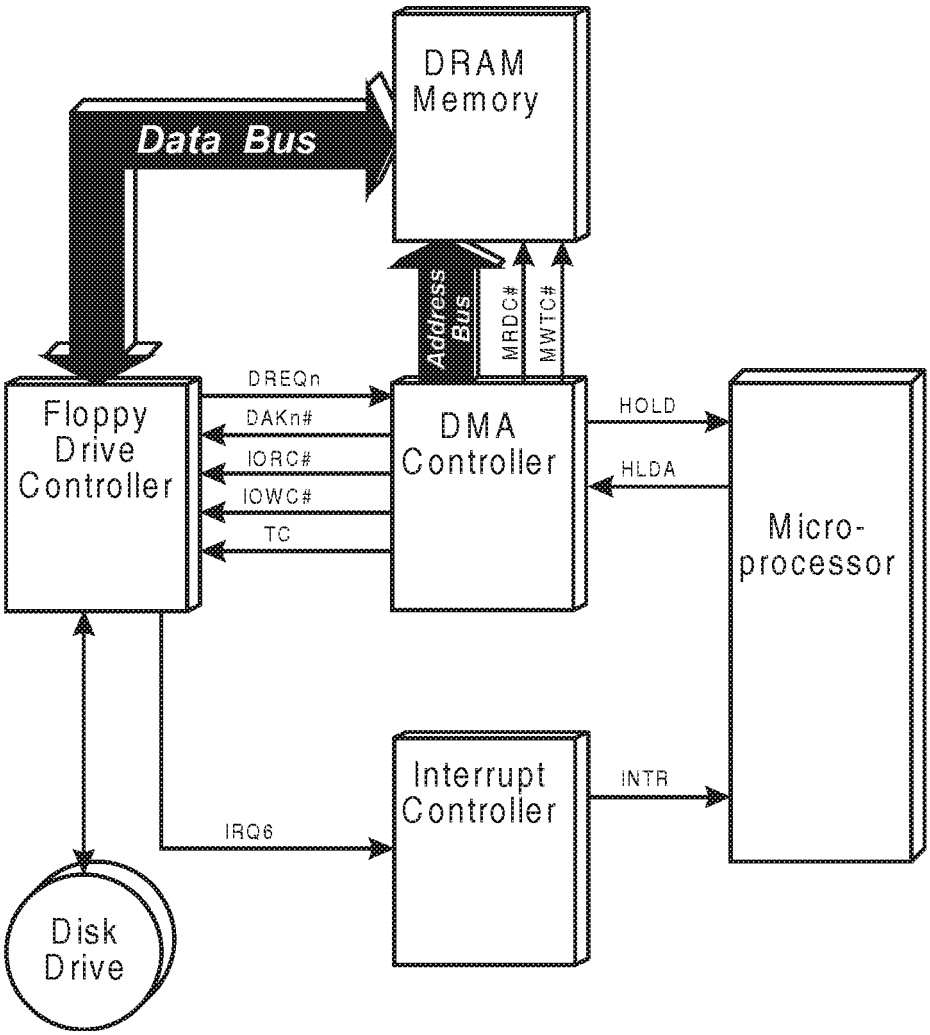


Figure 19-1. Example Disk/DMA Hardware System

1. The programmer issues a series of I/O writes to DMA channel two's registers to set up the start memory address, transfer count and the direction of the transfer with reference to memory. In the disk read example, assume that the start memory address is 1000h, the transfer count is 512 bytes (one sector) and that this is a write transfer (with reference to memory).
2. The floppy controller must be programmed with the disk read command, the cylinder number, the head (or surface) number, the start sector number and the number of sectors to be read. In this example, we will assume a one sector (512 bytes) read from cylinder three, head one, sector five.
3. Upon receiving the parameters and the disk read command, the disk controller initiates a seek operation to position the read/write head mechanism over cylinder three. It must then wait for the disk to spin until the start of sector five is detected under the read head on surface one. As the disk is a mechanical device, this will take some time.
4. In the example of a read operation from a floppy disk controller, some time will elapse before the disk controller has read the first byte from disk.
5. When the first byte has been transferred from the disk to the disk controller, the floppy disk controller must then request that its associated DMA channel transfer the data. The floppy disk controller asserts DMA request line two (DRQ2) that goes to the DMAC.
6. The DMAC responds by asserting its HRQ hold request output to seize the buses (address, data, and control buses) from the microprocessor. Since the refresh logic must also be able to seize the buses, the refresh logic also has a hold request output. These two hold request lines go to a tiny first-come-first-served arbiter which in turn drives the microprocessor's HOLD request input.
7. Depending on the X86 microprocessor that is being used, the microprocessor will either complete the current bus cycle, complete the current instruction, or perform all the writes in its posted write buffer. Then the microprocessor will tri-state all of its bus output drivers, thereby floating the buses. The microprocessor also asserts HLDA to tell the requesting device (the DMAC in this case) that it is now the bus master. Logic on the system board asserts the ISA bus AEN signal when it sees HLDA asserted. Note that the ISA bus AEN signal is not the same as the DMAC AEN pin.
8. The DMAC then responds to DRQ2 from the disk controller by asserting DMA Acknowledge (DAK2#) and the I/O read command line (IORC#). These two lines go to the disk controller via the ISA bus.
9. The disk controller deasserts DRQ2 and begins the access to the data register to place data onto the data bus.
10. The DMAC asserts the memory write command line (MWTC#) and places the address from channel two's start address register onto the address bus. Normally, the command lines indicate whether the address bus carries a

memory or I/O address. At this point, however, two command lines (IORC# and MRDC#) are asserted. The asserted ISA bus AEN signal prevents all I/O devices, except for the one that is connected to DAK2#, from responding during this bus cycle.

11. The data on the data bus is written into memory at the address currently on the address bus.
12. The DMAC then increments channel two's memory address register by one to point to the address in the RAM where it will store the next byte it receives from the disk controller.
13. The DMAC also decrements the byte transfer count. If the transfer count is not exhausted, the data transfer is not complete and the DMAC waits for another DMA Request (DRQ2) from the floppy drive controller, indicating that it has another byte to transfer (step eight).
14. When the transfer count is exhausted, the data transfer is complete. The DMAC will then deassert the hold request line (HRQ) to tell the microprocessor that it no longer needs the buses. The microprocessor will reattach itself to the buses (exit the tri-state condition) and deassert the Hold Acknowledge line (HLDA) to tell the DMAC it has resumed control of the buses. The microprocessor is now bus master again. Logic on the system board deasserts the ISA bus AEN signal so that I/O devices can resume decoding their addresses.
15. The DMAC also generates EOP (end-of-process). This supplies the signal TC (terminal count reached) to the disk controller. The disk controller will then generate a device-specific interrupt request to the 8259 interrupt controller which in turn generates INTR (interrupt request) to the microprocessor to inform it that the transfer operation is complete.

This was an example of a DMA channel operating in block transfer mode. Block transfer mode is best suited for I/O devices that can store or retrieve all of the requested data in memory that is on the I/O device before requesting a DMA transfer. Single transfer mode is preferable for devices such as the floppy disk controller which can hold only one or two bytes of data in their buffers. Demand transfer mode is preferable for I/O devices, such as later versions of the floppy disk controller, which have FIFO (first-in-first-out) buffers. All three modes are explained later in this chapter.

In the rest of this chapter, for the sake of simplicity, the tiny first-come-first-serve "arbiter" that routes the hold requests from the DMAC and the refresh logic to the microprocessor is not shown in the figures. Also, the descriptions assume that the DMAC's HRQ output is tied directly to the microprocessor's HOLD input.

DMA Controller (DMAC)

Each 8237 DMA controller has four DMA channels. In the original IBM PC, a single 8237 controller was used, limiting the number of channels to four. The IBM PC/AT added another DMAC, increasing the total number of channels available to seven (one channel is used to cascade the controllers together). Refer to figure 19-2.

DMA Transfer Types

Each of the seven DMA channels may be programmed to perform one of three types of transfers. This selection defines the method used to transfer a byte or word during one DMA bus cycle. The three types are:

- **Read transfer**. When the read transfer type is selected for a DMA channel, the DMAC will read data from memory and write it to the I/O device associated with the DMA channel.
- **Write transfer**. When the write transfer type is selected for a DMA channel, the DMAC will read data from the I/O device associated with the DMA channel and write to the memory.
- **Verify transfer**. The verify transfer type causes the DMAC channel to execute DMA transfer bus cycles, including generation of memory addresses, but neither the read nor the write command liness is asserted. It was used by DMA channel zero to implement DRAM refresh in the original IBM PC and PC/XT.

DMA Transfer Modes

In the preceding example, the DMAC controlled the buses while the entire block of data was transferred from the disk controller to memory. This is called block transfer mode. An 8237 DMAC channel however, may be programmed to operate in one of four modes. The mode selected defines the DMAC treatment of the overall information transfer. The four modes are:

- Single Transfer Mode
- Block Transfer Mode
- Demand Transfer Mode
- Cascade Mode

Single Transfer Mode

Single transfer mode is the one most commonly used because it allows the DMAC to timeshare the buses with the microprocessor. The DMAC gives up the buses after each byte (or word) is transferred so the microprocessor can have access to the buses on a regular basis. This is imperative, especially in situations where relatively large amounts of data will be transferred, because, otherwise, the microprocessor and the refresh logic get locked out for too long.

When a DMA channel is programmed to operate in single transfer mode, it operates as follows:

1. The DRQ(n) is received from the requesting I/O device (e.g., the floppy disk controller) when it is ready to transfer data.
2. The DMAC asserts HOLD to request the use of the buses from the microprocessor.
3. The microprocessor asserts HLDA, granting the buses to the DMAC. Logic on the system board asserts the ISA bus AEN signal, disabling the I/O address decoders.
4. The DMAC issues DAK(n)# to the requesting I/O device in conjunction with either IORC# or IOWC#. The I/O device deasserts its DRQ(n) because the DMAC is servicing its data transfer request.
5. The DMAC executes a DMA bus cycle to move the byte (or word) between the I/O device and memory by driving the memory address and appropriate memory command line as described earlier.
6. The word/byte transfer count is decremented and the memory address is incremented.
7. After the byte has been transferred, DAK(n)# is deasserted.
8. The HOLD line is deasserted to surrender the buses back to the microprocessor. ISA bus AEN signal is deasserted also. HOLD is reasserted to request the use of the buses again when the I/O device has another byte or word to transfer. Upon receipt of HLDA from the CPU, the next bus cycle will occur. This process continues until the transfer is completed.
9. When the transfer count is exhausted, TC is generated to signal the end of the transfer and the I/O device interrupts the CPU.

Block Transfer Mode

Block transfer mode can be used when the DMAC needs to transfer a block of data, but the designer and programmer must keep in mind that if the DMAC retains ownership of the buses for too long, the DRAM refresh interval may be exceeded and there may also be harmful effects from locking the microprocessor out for too long. An example of operation under block transfer mode was already covered under the heading, "A DMA Example."

Demand Transfer Mode

When demand mode is used, the DMAC runs DMA bus cycles back-to-back as long as the I/O device keeps it DRQ(n) line asserted, meaning it can continue to supply data. If the I/O device falls behind the DMAC's ability to transfer data, the DMA transfer is temporarily suspended. This is done when the I/O device deasserts its DRQ(n) line. When the I/O device is ready to resume demand transfers, it again asserts its DRQ(n) line.

As an example, if a streaming tape drive uses DMA to transfer information, it could transfer information on a continuing basis until it reached the end of a track. While the tape drive mechanism reverses the direction of the tape, data flow would be interrupted and the transfer would temporarily be suspended.

When a DMA channel is programmed to operate in demand transfer mode, it operates as follows:

1. The DRQ(n) is received from the requesting I/O device (e.g., the streaming tape controller).
2. The DMAC asserts HOLD to request the use of the buses from the microprocessor.
3. The microprocessor asserts HLDA, granting the buses to the DMAC. ISA bus AEN signal is asserted.
4. The DMAC asserts DAK(n)# and either the IORC# or IOWC# to the requesting I/O device. Note that the I/O device keeps DRQ(n) asserted.
5. As described earlier, the DMAC executes a DMA bus cycle to move a byte (or word) between the I/O device and memory.
6. After the transfer, the word/byte transfer count is decremented and the memory address is incremented.
7. The DMAC continues to execute transfer cycles until the I/O device deasserts DRQ(n) indicating its inability to continue delivering data. The DMAC deasserts HOLD to give the buses back to the microprocessor and the DMAC deasserts DAK(n)#. Logic on the system board deasserts ISA bus AEN signal. The I/O device can re-initiate the demand transfer by re-

asserting DRQ(n). Each time the I/O device deasserts DRQ(n) the buses will be released to the microprocessor.

8. Transfer continues in this fashion until the transfer count has been exhausted.

Cascade Mode

When a DMA channel is programmed to operate in cascade mode, this means that it is connected to another DMAC or to an ISA bus master, rather than an I/O device. See the discussion of ISA bus mastering in this chapter.

Refer to figure 19-2. To get more than four DMA channels, two DMACs are cascaded together. This is accomplished by designating one of the 8237s as the master and the other as the slave. The HRQ (hold request) output of the slave is tied to the DRQ0 input of the master (channel 4) and the master's DAK0# output is tied to the slave's HLDA (hold acknowledge) input. In this way, the slave uses the master to obtain control of the buses.

During POST, channel zero of the master DMAC must be programmed for cascade mode. When the slave requires the use of the buses, it asserts its Hold Request output. The master DMAC detects the request for the buses and asserts its HRQ output to seize the buses from the microprocessor. When the microprocessor surrenders the buses and asserts HLDA, the master DMAC asserts its DAK0# output to grant the buses to the slave DMAC. The master's DAK0# output is connected to the slave's HLDA input. It should be noted that the signals interconnecting the master and slave are commonly referred to as DRQ4 and DAK4#, even though they are connected to channel zero on the master DMAC.

DMAC Priority Logic

The 8237 DMAC offers two types of software-selectable priority schemes:

- **Fixed priority**. The channels are fixed in priority order based upon the descending value of their number. Channel 0 has the highest priority and channel 3 has the lowest priority. After the recognition of any channel for service, the other channels are prevented from interfering with that service until it is completed.
- **Rotating priority**. The last channel to get service becomes the lowest priority channel with the priority of the others rotating accordingly. This prevents any one channel from monopolizing the system.

Both DMACs in an ISA system are default-programmed by the POST for fixed priority. As a result, the highest priority channel on both the slave and the master is channel zero, followed by channels one through three. This results in master channel zero having the highest priority. Since master channel zero is really connected to the slave DMAC, this means that the slave's four DMA channels have the highest priority, with channel zero as highest and channel three as the lowest. Immediately following slave channel three, master channel one, also known as channel five, is the next highest, followed by channels six and seven (master channels two and three).

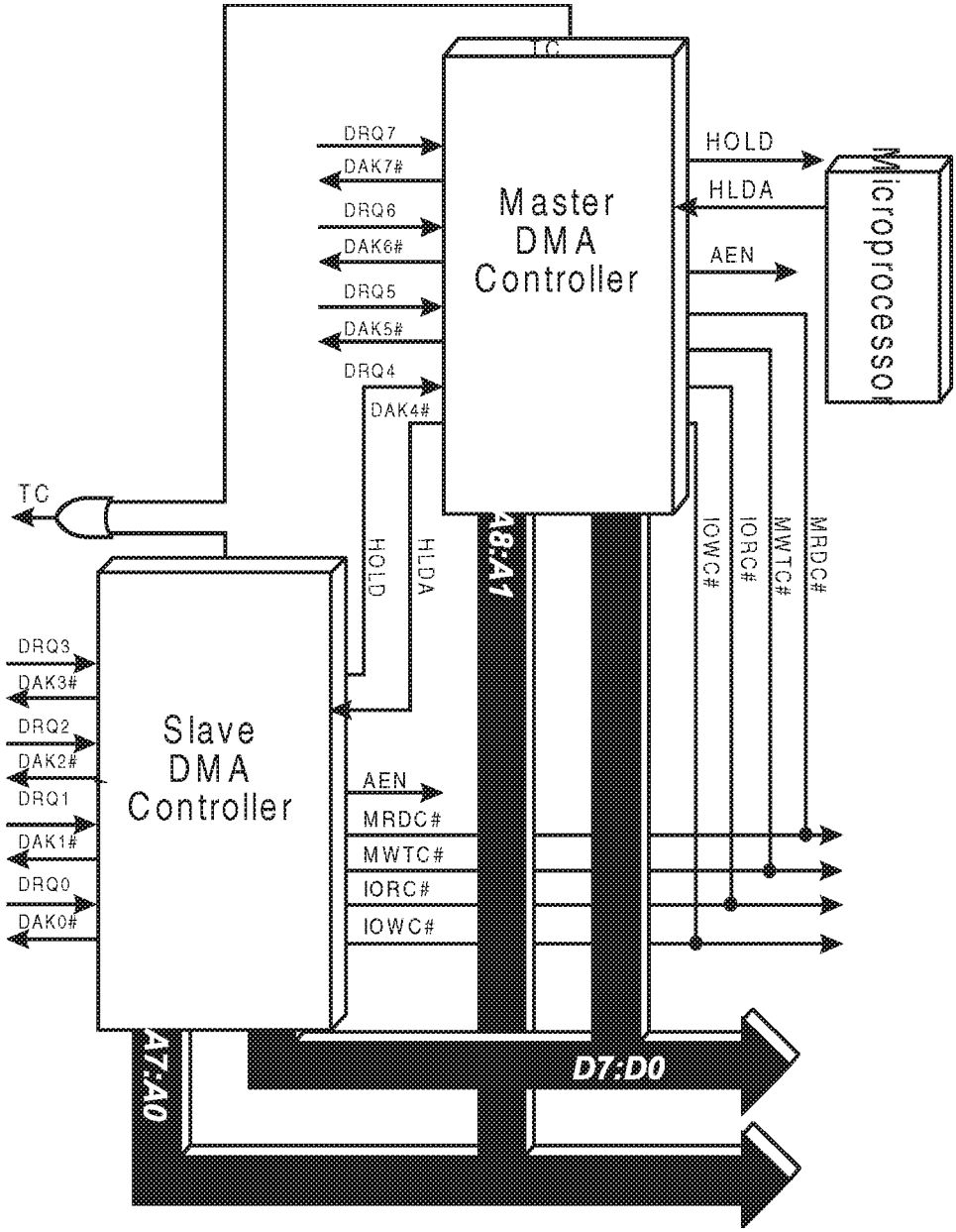


Figure 19-2. Cascaded DMA Controllers

DMA Bus Cycle

The 8237 is built around a state machine with seven possible states, each one DMA clock period wide. When a DMAC is bus master, it uses its own clock when executing bus cycles. This clock is referred to as the DMA clock and is 1/2 the BCLK frequency. Depending on the unit and the selected processor speed, this will yield a DMA clock of either 3MHz (6MHz PC/AT), or 4MHz (8MHz ISA-compatible machine) to 4.165MHz (8.33MHz ISA-compatible machine).

Table 19-1 lists the clock period for three possible BCLK speed settings:

Table 19-1. DMA Clock Frequency

BCLK Speed Setting	DMA Clock Frequency	DMA Clock Period
6MHz	3MHz	333.3ns
8MHz	4MHz	250ns
8.33MHz	4.166MHz	240ns

Prior to receiving a DMA Request, the DMAC is in the idle state, Si. When a DRQ is sensed, the DMAC enters a state where it asserts HOLD (hold request) to the microprocessor and awaits the HLDA (hold acknowledge). This state is called S0 (S zero). The DMAC remains in the S0 state until HLDA is sensed asserted.

The DMAC can then proceed with the DMA transfer. S1, S2, S3 and S4 are the states used to execute a transfer (of a byte or word) between the requesting I/O device and system memory. In addition, when accessing a device that is slow to respond, a DMA transfer cycle can be stretched by deasserting the DMAC's READY input until the device is ready to complete the transfer. This will cause the DMAC to insert wait states, Sw, in the bus cycle until READY is asserted again.

Table 19-2 describes the actions that take place during states S1 through S4. See figure 19-3 for the actual timing of a single transfer.

Table 19-2. DMAC States

State	Actions Taken
S1	During single transfer mode, S1 is used to output the middle byte of the memory address, A[15:8] during each transfer. The middle byte of the memory address is output onto data bus pins D[7:0]. The DMAC also pulses its address strobe (ADSTB) output during S1 and on the falling edge of ADSTB the new middle byte of the address to be latched into the external DMA address latch. Address lines A[15:8] receive the middle portion of the address during S2. During block and demand transfers, the middle byte of the memory address only changes once every 256th transfer. For this reason, when in these modes the DMAC only enters the S1 state every 256th transfer to update the middle byte of the address.
S2	During S2, the lower byte (A[7:0]) of the memory address is output directly onto the address bus, A[7:0]. The DMAC's AEN output is asserted causing the external DMA address latch to output the middle portion of the address and to act as an enable for the DMA page register addressing. Note that DMAC's AEN pin is *not* the ISA bus AEN signal. In addition, DAKn# is asserted to tell the I/O device that the transfer is in progress.
S3	S3 will only occur in a bus cycle if compressed timing hasn't been selected for this DMA channel. See text below for a discussion of compressed timing. During S3, the MRDC# or the IORC# line is asserted. If the DMA channel is programmed for extended writes, the MWTC# or IOWC# line is also asserted during S3.
S4	If the DMA channel was not programmed for extended write, the MWTC# or IOWC# is asserted at the start of S4. If extended write had been selected, the write command line was already asserted at the start of S3. The actual read/write takes place at the trailing edge of S4 when both the read and write command lines are deasserted by the DMAC. This completes the transfer of a byte or word between memory and the requesting I/O device.

When compressed timing is selected, S3 is eliminated from the DMA transfer cycle. The only real purpose of S3 is to allow the read command line to be asserted for twice the duration it is when compressed timing is used. Not all memory and I/O devices will tolerate this abbreviated read command line, so it must be used cautiously.

When extended write is selected, it causes the write command line to be asserted during S3 rather than S4, effectively doubling the duration of the write command line's asserted period.

It should be obvious that extended write and compressed timing are mutually exclusive because S3 is essential for extended write and is eliminated when compressed timing is selected.

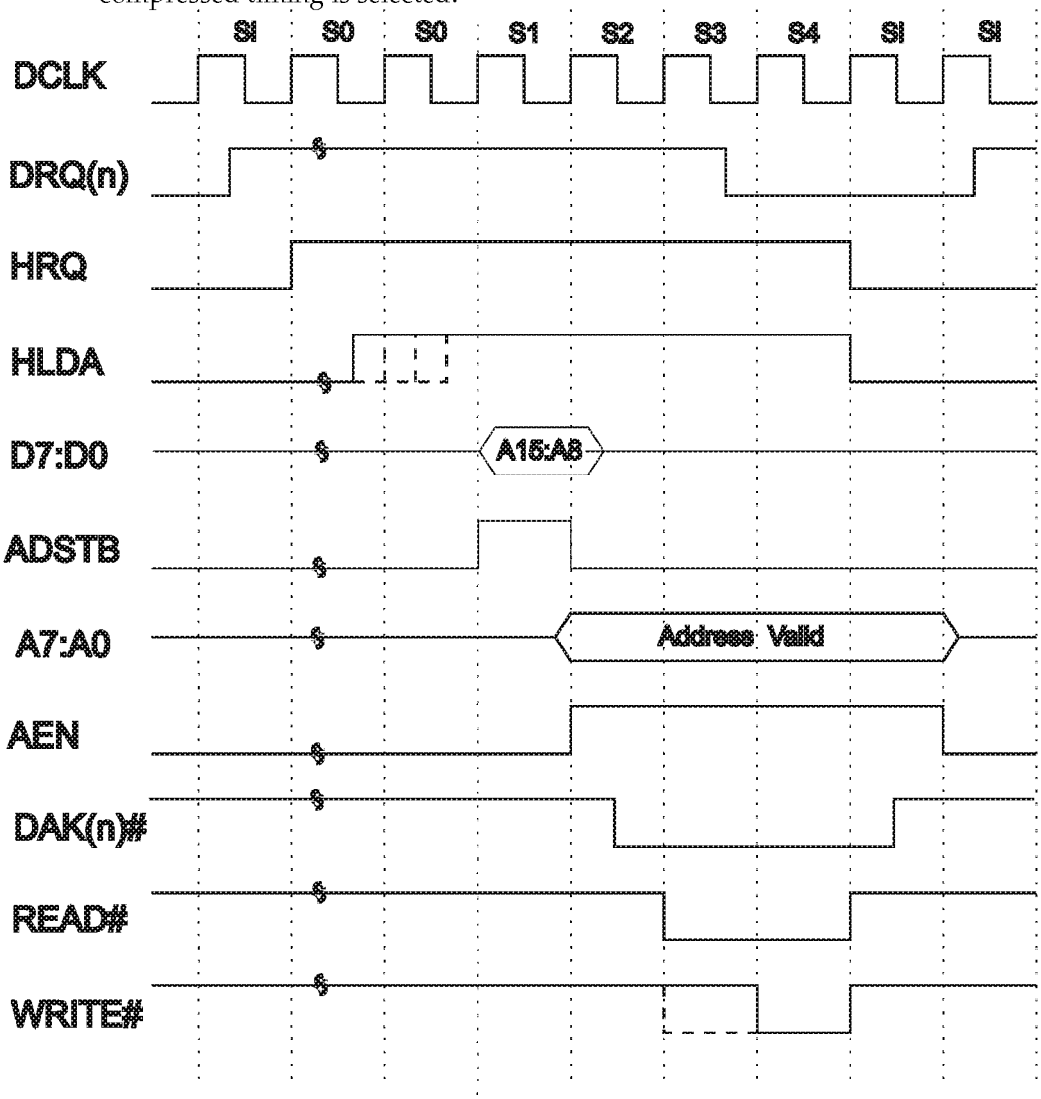


Figure 19-3. Single Transfer Mode Timing

Byte or Word Transfers

ISA systems can perform byte transfers via slave channels zero through three and word transfers via master channels five through seven. The DMACs were designed to handle byte transfers, but the master DMAC is connected to the address bus in a way that allows 16-bit transfers.

The address outputs of the master DMAC and its associated address latch and page registers are connected to SA[23:1]. SA0 is not connected to it. This means that the programmer must program the master DMA memory address registers with the correct start address for a word transfer. For example, to access location 1000h in memory, the master DMAC's memory address register would need to be programmed to 0800h. Whenever the master DMAC becomes bus master, logic external to the DMAC asserts SBHE# and forces SA0 to a 0 (even address) to indicate a word transfer. Note that DMA channels five through seven can only perform 16-bit transfers because SA0 is always set to zero and SBHE# is always asserted when the master DMAC outputs a memory address. The slave DMAC and its associated memory address and page registers, on the other hand, are connected to SA[23:0] and can therefore generate both even and odd addresses one at a time. This gives DMA channels zero through three the ability to perform byte-oriented transfers.

DMAC Addressing Capability

Each DMA channel can address any memory location within the 16MB range from 000000h to FFFFFFh. This is accomplished by using the DMAC's 16-bit Memory address registers associated with each DMA channel in conjunction with an 8-bit external DMA page register. This pair of registers associated with each channel provides the 24-bit memory address capability.

The starting address in memory where data is to be written to or read from must be programmed into the DMAC and the channel's respective page register prior to beginning the DMA transfer. The lower sixteen bits of the address are written into the channel's 16-bit memory address register inside the DMAC using two successive write operations over its 8-bit data path. Table 19-3 identifies the I/O location associated with each channel's memory address register. The most significant byte of the address must be written to the DMA channel's respective page register. Table 19-3 also gives the page register locations.

Table 19-3. DMAC Memory Address and Page Registers—I/O Locations

Slave DMAC Channel	Master DMAC Channel

	0	1	2	3	4	5	6	7
Memory Address Register I/O Address	00h	02h	04h	06h	C0h	C4h	C8h	CCh
Page Register I/O Address	87h	83h	81h	82h	none	8Bh	89h	8Ah

Since the DMAC has only eight address pins, it must output the lower sixteen bits of the address using both A[7:0] and D[7:0]. The lower byte of the memory address is output onto A[7:0], while the middle byte of the memory address is output onto D[7:0].

Refer to figure 19-4. The slave DMAC first places the middle byte of the address (bits 15:8) on data bus lines D[7:0]. Once the data has stabilized, it is latched into the external address latch by the address strobe signal, ADSTB. The slave DMAC outputs the lower byte of the memory address onto A[7:0] and asserts AEN, enabling both the address latch and the channel's DMA page register to transfer their contents to address lines A[15:8] and A[23:16] respectively. (Note that the DMAC's AEN pin is not the same as the ISA bus AEN signal.) The appropriate DMA page register is selected by the DAKn# line associated with the channel in use.

Refer to figure 19-5. The master DMAC first places the middle byte of the address (bits 16:9) on data bus lines D[7:0]. Once the data has stabilized, it is latched into the external address latch by the address strobe signal, ADSTB. The master DMAC outputs the lower byte of the memory address (bits 8:1) onto A[7:0] and asserts AEN, enabling both the address latch and the channel's DMA page register to transfer their contents to address lines A[16:9] and A[23:17] respectively. Note that the DMAC's AEN pin is not the same as the ISA bus AEN signal. The appropriate DMA page register is selected by the DAKn# line associated with the channel in use.

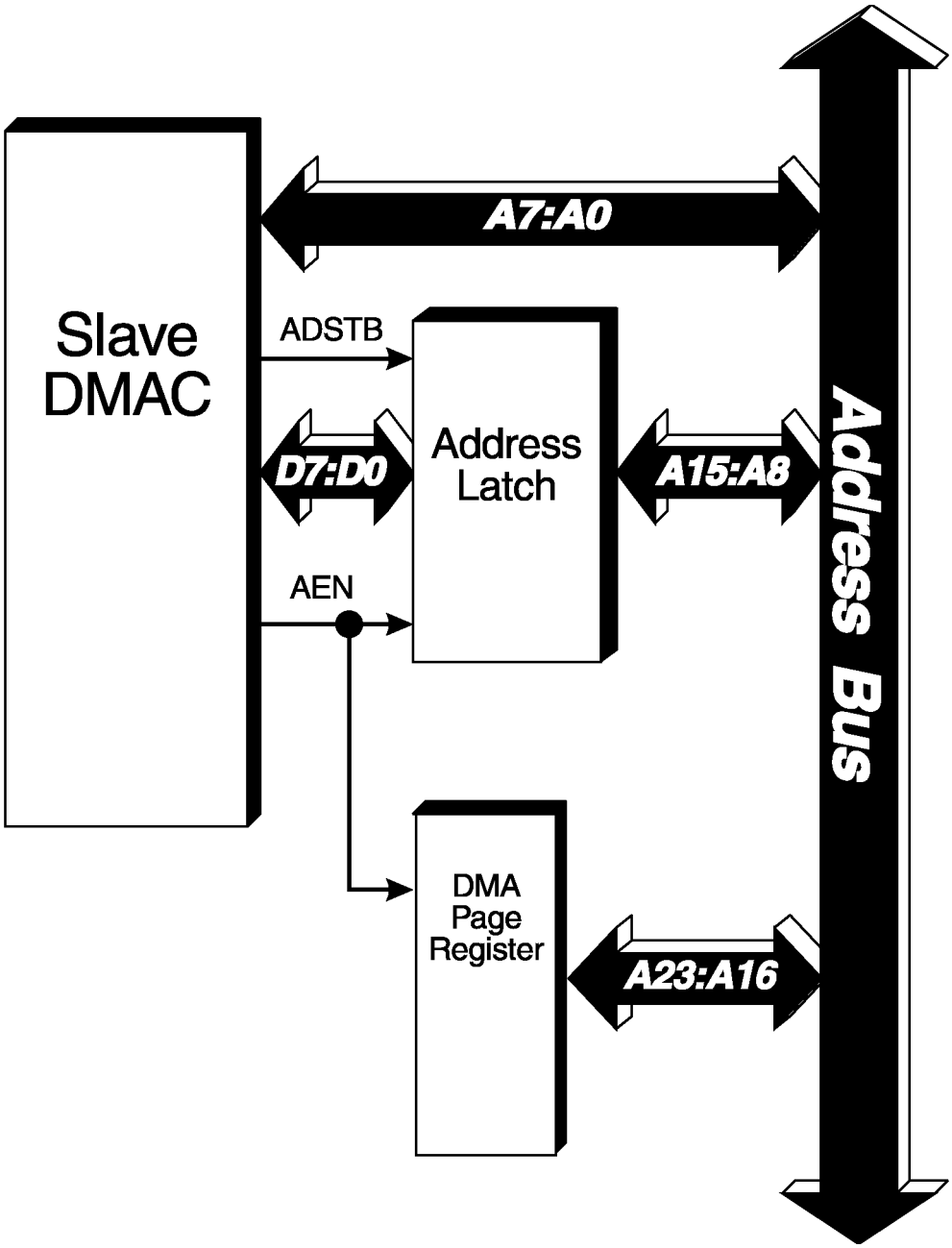


Figure 19-4. The Slave DMAC Memory Address Logic

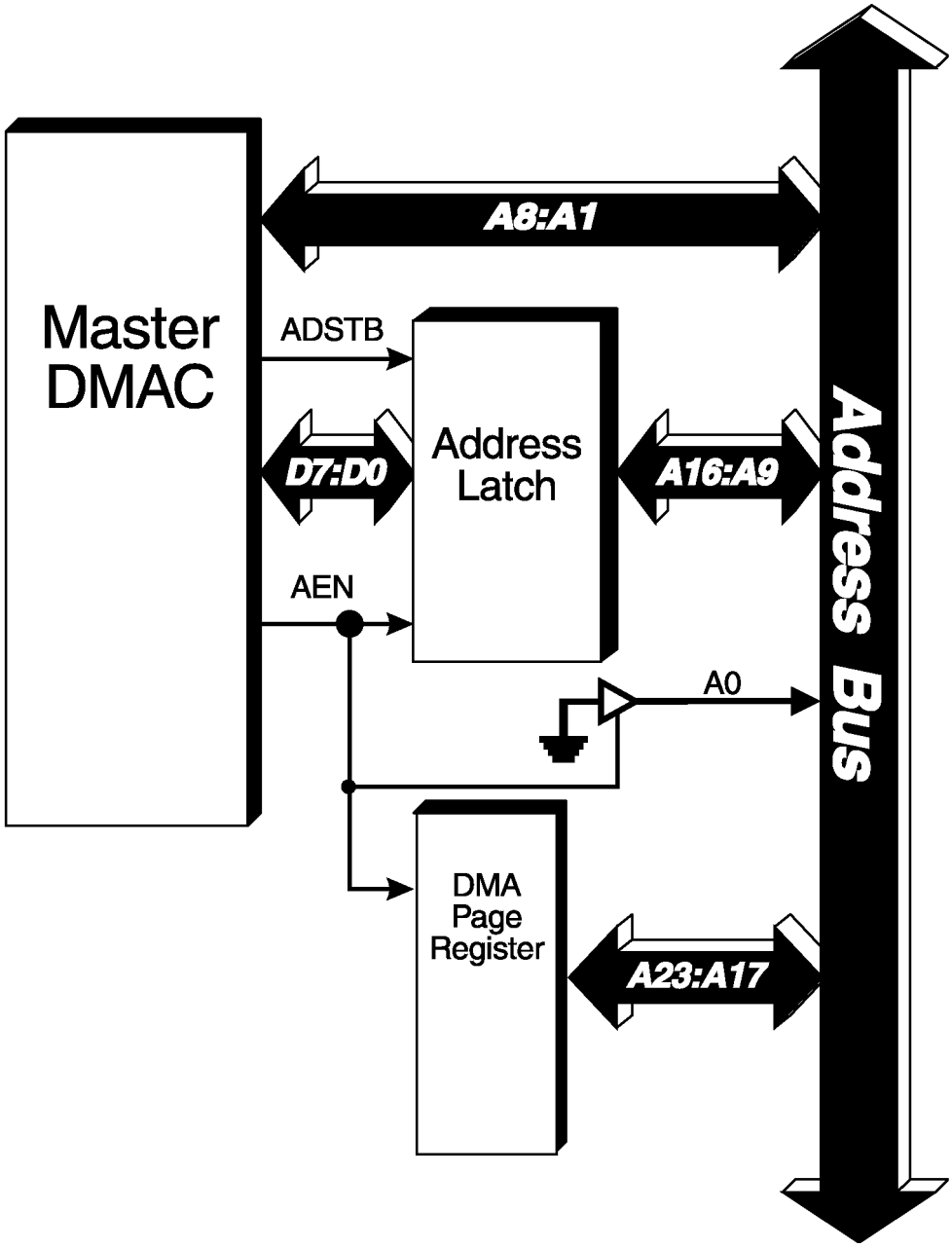


Figure 19-5. The Master DMAC Memory Address Logic

Addressing ISA Memory

Refer to figure 19-6. The DMA controller (DMAC) resides on the X-bus, and must be capable of addressing memory that resides on the ISA bus. To do this the address must be passed from the X-bus to the SA bus. Normally, when the main CPU is running bus cycles the buffer passes the address from the SA bus on to the X-bus to address the I/O devices that reside on the X-bus; however, when the DMAC is the bus master, addresses produced by the DMAC must go in the opposite direction. During DMA transfers, the DMAC's AEN output goes to the X-bus address buffer to change the direction of the address.

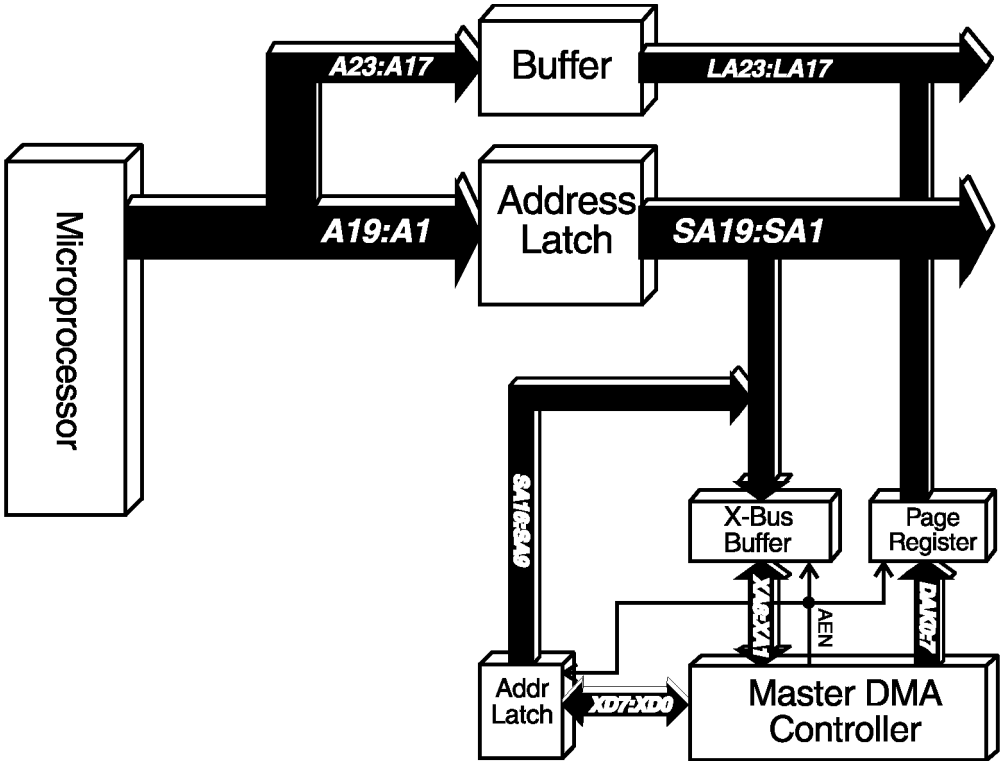


Figure 19-6. Master DMAC Addressing 16-bit ISA Memory

Addressing Local Bus Memory

ISA systems based on 80386 or higher microprocessors typically have main system memory on the processor's local bus. This design allows the processor to access memory at the full speed and data path width of the microprocessor. However, this design also creates the need to perform address translation and data bus steering in order for the DMAC to access 32-bit memory. Refer to figure 19-7.

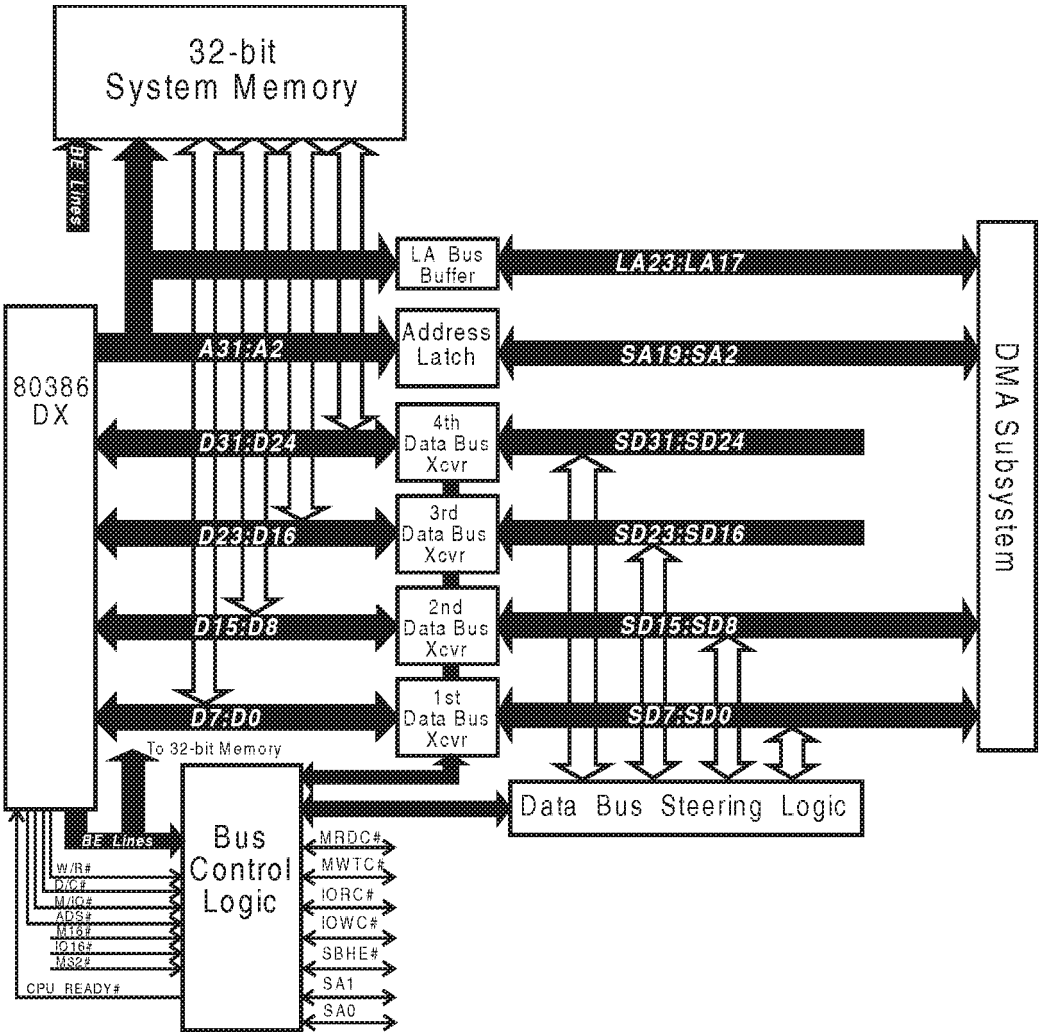


Figure 19-7. DMA (16-bit channel) Addressing 32-bit System Memory

Refer to figure 19-8. Notice also that the address buffers in addition to the X-bus buffer must be reversed so that the address can be passed from the X-bus bus back toward main memory. This reversal of the buffer occurs whenever the main CPU gives up control of the buses (asserts the HLDA signal). The HLDA signal causes the buffers to change direction.

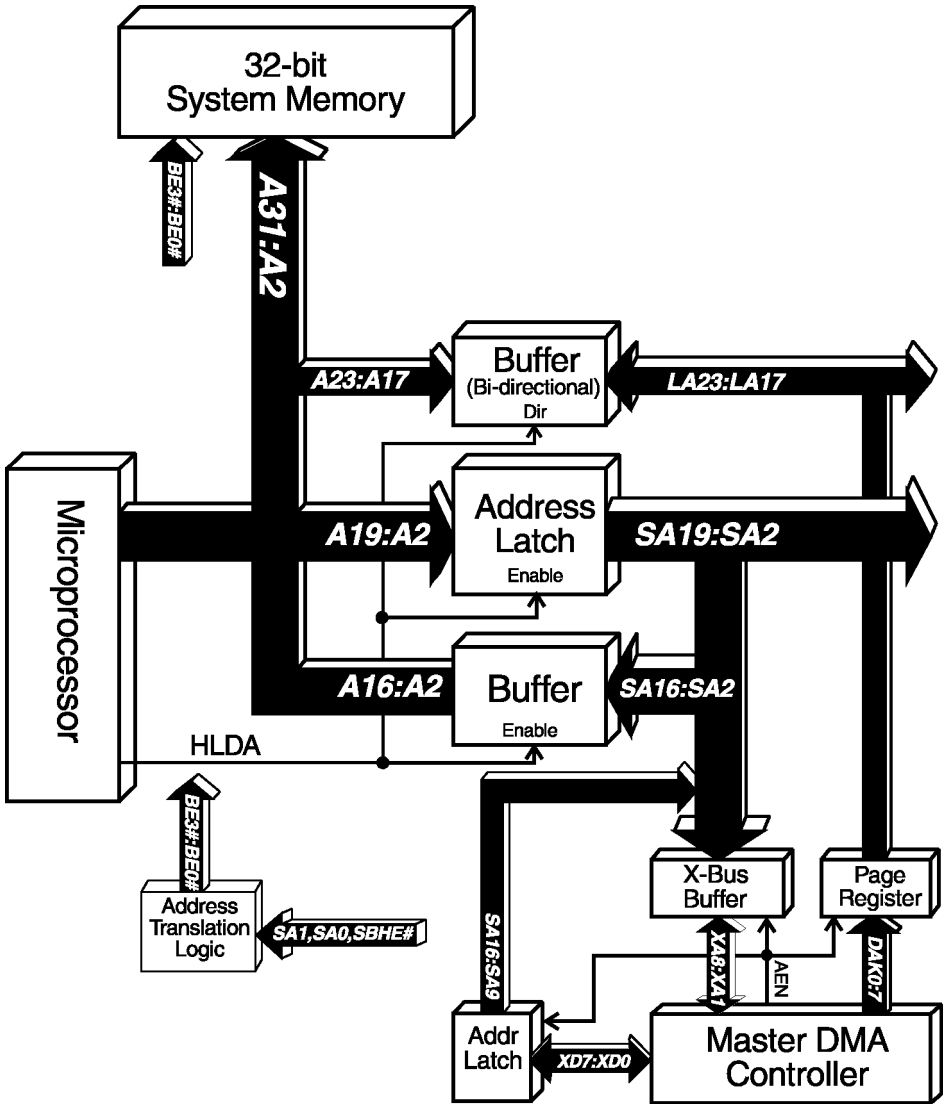


Figure 19-8. 16-bit Access to 32-bit Memory with DMA

The system address used by the DMACs consists only of SA[16:0]. Note that SA[16:2] pass through the buffer back to system memory, while SA0, SA1 and SBHE# must be converted to BE#[3:0] by the bus controller. The LA lines pass through a bi-directional buffer whose direction is controlled by HLDA. When the CPU has control of the buses, the buffer passes the address from the CPU to the ISA bus. When the CPU gives up ownership of the buses, the direction is reversed to pass the address from the ISA bus back toward the CPU and system memory.

Address Translation

Since the DMA subsystem resides on the X-bus (a subset of the ISA bus) it has access only to the ISA bus and cannot supply the 32-bit address needed by the 32-bit memory. Therefore, the bus controller must be designed to translate the 8- or 16-bit address from the DMA subsystem (SA[16:0], LA[23:17] and SBHE#) to the 32-bit address (A[31:2] and BE#[3:0]) required by memory.

Address lines SA0, SA1 and SBHE#, normally outputs from the bus controller when the CPU is running bus cycles, are inputs when DMA or bus masters other than the CPU are running bus cycles. When the HLDA signal from the CPU becomes asserted, the bus controller begins translating the ISA type addresses to 32-bit memory addresses.

Data Bus Steering

The bus controller in ISA machines with 32-bit memory subsystems must also perform data bus steering to ensure proper operation with DMA. The MRDC# and MWTC# lines are used by the DMA subsystem to indicate the type of bus cycle being run. These command lines, along with SA0, SA1 and SBHE#, and the M32# size line are used by the bus controller to ensure information transferred between the DMA subsystem and the 32-bit memory subsystem is transferred over the correct path(s). In essence, the data bus steering required by DMA is the exact inverse of the data bus steering required when an 80386 or higher microprocessor communicates with an 8- or 16-bit device. In this instance, the DMA controller is either an 8- or 16-bit device communicating with 32-bit memory, necessitating the steering.

DMA Transfer Rate

Table 19-4 illustrates the transfer speeds possible at the three clock speeds under the following conditions:

- Compressed timing turned off
- Compressed timing turned on

The table assumes that the transfer is no more than 256 DMA bus cycles in length. This was done for simplicity's sake. Once for every 256 bus cycles, the DMAC must insert an S1 state in the bus cycle to update the middle byte of the memory address (A[16:9] for master DMAC or A[15:8] for slave DMAC), which must be latched into the external DMA address latch. This adds one DMA clock period to the length of every 256th bus cycle. In short, table 19-4 gives rates based on DMA bus cycles consisting of states S2, S3 and S4 (3 DMA clock cycles with compressed timing off) and states S2 and S4 (2 DMA clock cycles with compressed timing on). Using the master DMAC, the maximum DMA transfer rate in an ISA system is 4.166MB/sec (16 bits times 2.083M bus cycles per second).

Table 19-4. ISA DMA Transfer Rates

	Compressed Off			**Compressed On**		
DMA Clock Frequency	3MHz	4MHz	4.166MHz	3MHz	4MHz	4.166MHz
Transfers per Second	1M/s	1.33M/s	1.39M/s	1.5M/s	2M/s	2.083M/s

DMAC Initialization During POST

During POST, both the slave and master DMA controllers are initialized for ISA operation. Recall that the DMA controllers are programmed during POST to operate in single mode, using fixed priority, normal timing, and are cascaded together via master channel zero. Two registers are programmed to provide these ISA operational characteristics:

- DMAC control register
- DMAC mode register

The programming sequence typically used is shown in table 19-5. For a complete definition of each bit see the I/O address map in the appendix.

Table 19-5. Initialization of DMACs

Register	Slave DMAC		Master DMAC	
	I/O Addr.	Functions Controlled	I/O Addr.	Functions Controlled
Control	08h	Normal timing, fixed priority, DMAC enabled, no memory-to-memory transfers	D0h	Normal timing, fixed priority, DMAC enabled, no memory-to-memory transfers
Mode	0Bh	Channel 0 - single transfer mode	D6h	Channel 4 - cascade mode
Mode	0Bh	Channel 1 - single transfer mode	D6h	Channel 5 - single transfer mode
Mode	0Bh	Channel 2 - single transfer mode	D6h	Channel 6 - single transfer mode
Mode	0Bh	Channel 3 - single transfer mode	D6h	Channel 7 - single transfer mode

Chapter 20

The Previous Chapter

The previous chapter explains the process, logic and timing of the ISA DMA subsystem.

This Chapter

This chapter covers ISA bus mastering. The previous chapter on DMA provides a foundation for understanding ISA bus masters, since bus masters must use the DMA subsystem to gain control of the system buses.

The Next Chapter

The next chapter explains the ISA implementation of the real time clock and configuration memory.

ISA Bus Master Capability

The microprocessor on the system board is the bus master most of the time in ISA systems. It is also possible for a card inserted into an ISA expansion slot to become bus master, but there is a major drawback. When a card becomes bus master in a ISA system, it can remain bus master as long as it keeps the microprocessor's HOLD line asserted. There are no safety mechanisms built into a PC/AT or compatible to prevent a bus master card from monopolizing the use of the buses to the exclusion of the microprocessor on the system board, the RAM refresh logic and potential bus master cards inserted into other expansion slots.

Refer to figure 20-1. In order to gain ownership of the buses, the requesting device must be interfaced to the system board via DMA channel 5, 6 or 7. These are the channels connected to the master 8237 DMAC on the system board. Prior to requesting the use of the buses, the respective channel of the master DMAC must be programmed for cascaded operation (to indicate that a slave DMAC is attached to that channel). In this case the ISA bus master will be at-

tached to the cascade channel rather than a slave DMAC. When the DMA acknowledge is sent to the bus master it has been granted control of the buses and will maintain ownership until the DMA request is released.

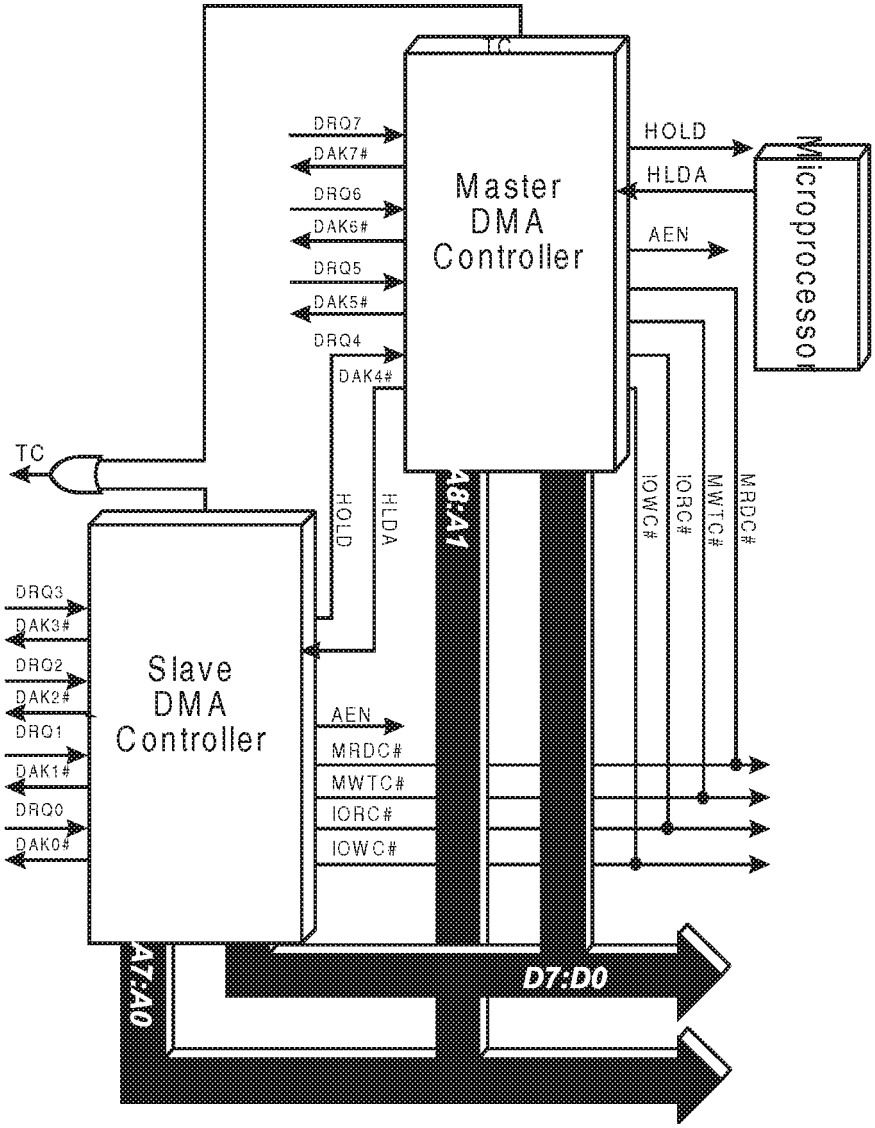


Figure 20-1. DMA and Bus Mastering

When the ISA expansion board wishes to request the use of the buses, the following sequence of actions takes place:

1. The requesting device asserts its respective DRQ line.
2. The Master 8237 DMAC asserts HOLD (hold request) to the microprocessor to request the use of the buses.
3. When the microprocessor's bus unit completes the current bus cycle, the microprocessor floats the buses and asserts HLDA (hold acknowledge) to inform the master DMAC that it is now the bus master.
4. Upon sensing the HLDA, the master DMAC asserts the DAK line (DMA acknowledge) associated with the DRQ line from the requesting bus master card.
5. The requesting ISA expansion board is now the bus master and can run bus cycles to communicate with other system devices.
6. The ISA bus master card must assert the MASTER16# signal if it wishes to access I/O devices. When the CPU asserts the HLDA signal, the AEN signal is asserted, disabling all I/O address decoders. This prevents I/O devices from mistaking the memory address as theirs during DMA transfers (see the chapter entitled "Direct Memory Access (DMA)" for details). Asserting MASTER16# causes the AEN signal to become deasserted, thereby enabling all I/O address decoders.
7. When the expansion device has completed its use of the buses, it must deassert its DRQ line.
8. Upon sensing that DRQ is deasserted, the master DMAC on the system board will deassert DAK and HOLD.
9. The microprocessor will deassert HLDA and re-attach itself to the buses.

ISA Bus Masters in 80386DX and Higher Systems

ISA systems based on 80386DX and higher microprocessors typically have main system memory on the processor's local bus. This design allows the processor to access memory at the full speed and data path width of the microprocessor. However, this design also creates the need to perform address translation and data bus steering in order for the ISA bus master to access 32-bit memory. Refer to figure 20-2.

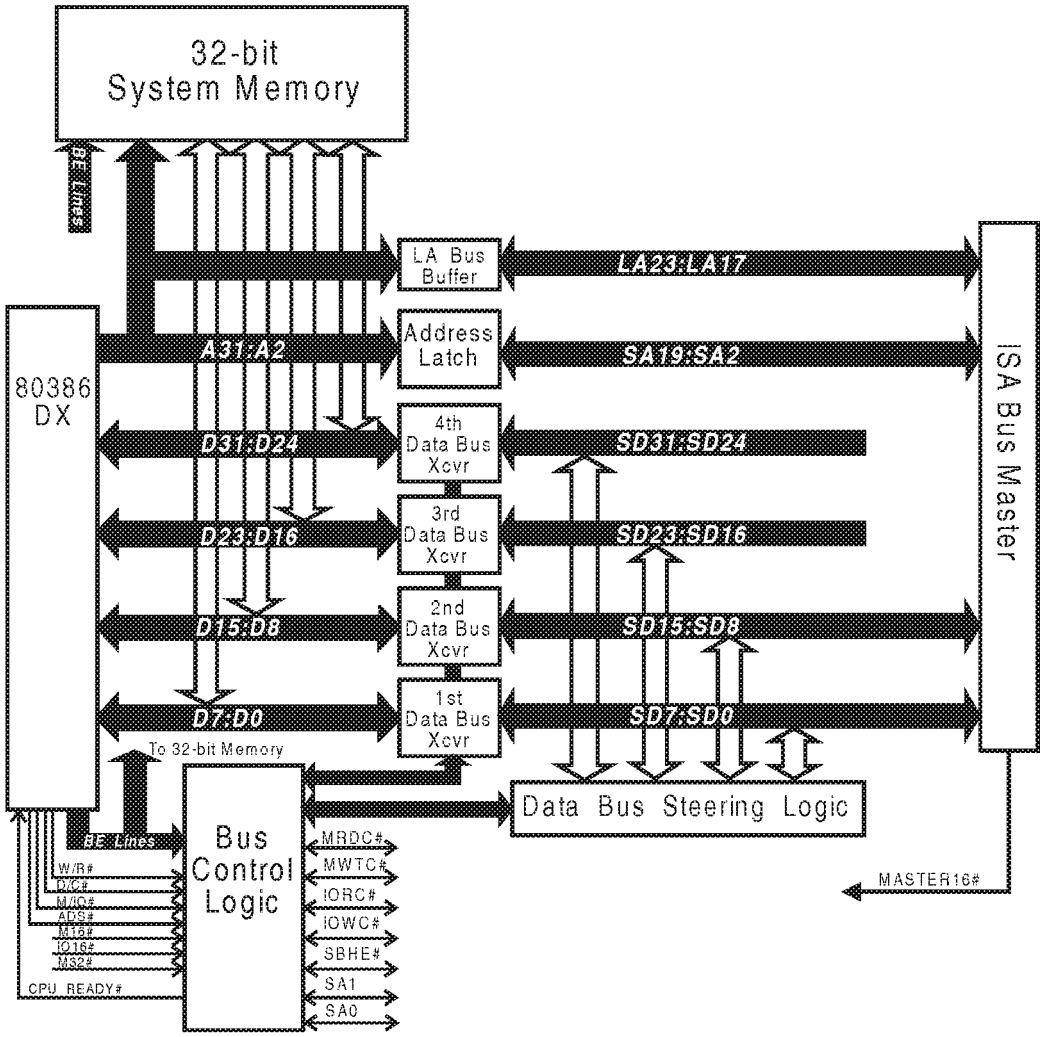


Figure 20-2. ISA Bus Master Addressing 32-bit System Memory

Refer to figure 20-3. Notice also that the address buffers must be reversed so that the address can be passed from the ISA bus back toward main memory. This reversal of the buffer occurs whenever the main CPU gives up control of the buses (asserts the HLDA signal). The HLDA signal causes the buffers to change direction. ISA bus masters generate addresses using the following ISA address lines:

- SA[16:0]
- SBHE#
- LA[23:17]

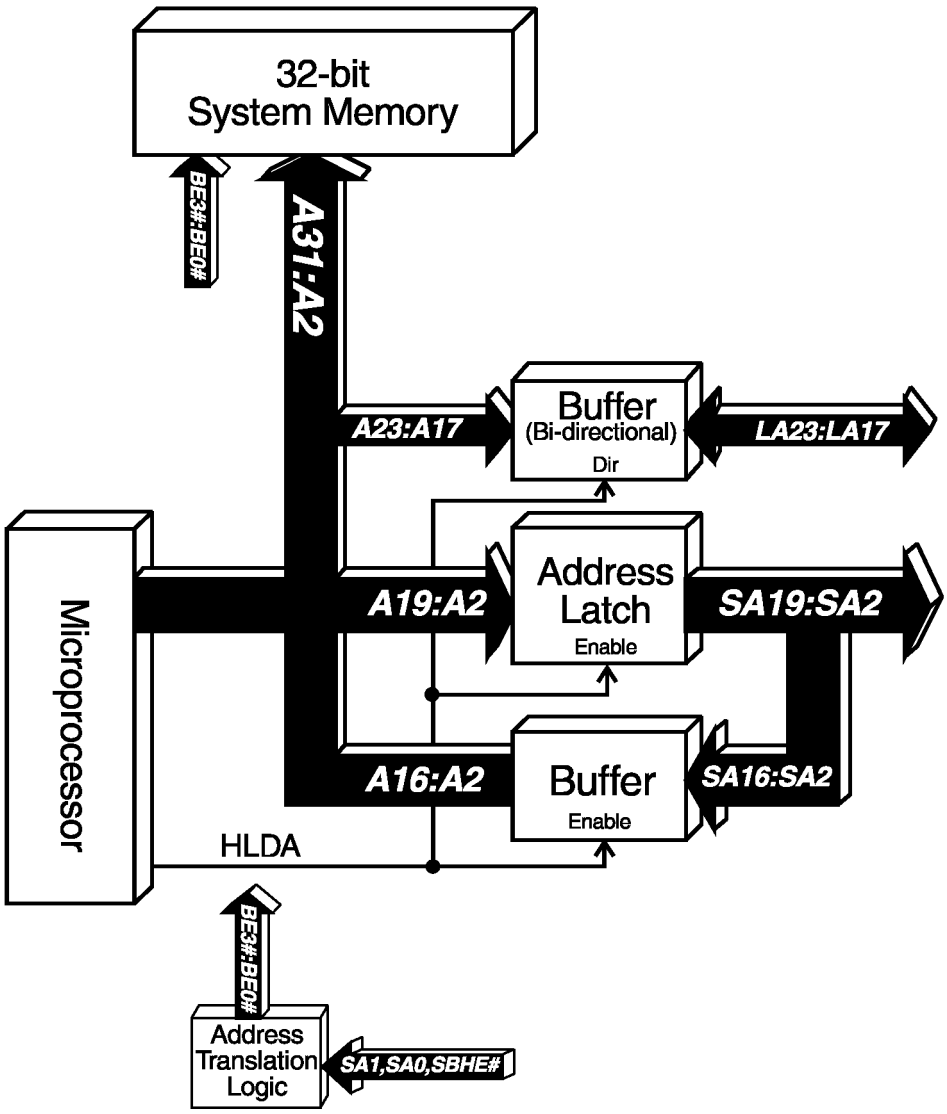


Figure 20-3. Addressing 32-bit Memory with ISA Bus Masters

The system address used by ISA bus masters consists only of SA[16:0]. Recall that SA0, SA1 and SBHE# are converted to BE#[3:0] by the bus controller. Only SA[16:2] pass through the buffer back to system memory.

The LA lines pass through a bi-directional buffer whose direction is controlled by HLDA. When the CPU has control of the buses, the buffer passes the address from the CPU to the ISA bus. When the CPU gives up ownership of the buses, the direction is reversed to pass the address from the ISA bus back toward the CPU and system memory.

Address Translation

Since ISA bus master reside on the ISA bus they have access only to the ISA bus and cannot supply the 32-bit address needed by the 32-bit memory. The bus controller must be designed to translate the 16-bit address from the bus master (SA[16:0], LA[23:17] and SBHE#) to the 32-bit address (A[31:2] and BE#[3:0]) required by memory.
Address lines SA0, SA1 and SBHE#, normally outputs from the bus controller when the CPU is running bus cycles, are inputs when a bus master other than the CPU is running bus cycles. When the HLDA signal from the CPU becomes asserted, the bus controller begins translating the ISA type addresses to 32-bit memory addresses.

Data Bus Steering

The bus controller in ISA machines with 32-bit memory subsystems must also perform data bus steering to ensure proper operation with ISA bus masters. The MRDC# and MWTC# lines are used by ISA bus masters to indicate the type of bus cycle being performed. These command lines, the status of address lines SA0, SA1 and SBHE#, and the M32# size line are used by the bus controller to ensure that information transferred between the bus master and 32-bit memory subsystem is transferred over the correct path(s). In essence, the data bus steering required by ISA bus masters is the exact inverse of the data bus steering required when a 80386DX or higher microprocessor communicates with a 16-bit device. In this instance, the bus master is a 16-bit device communicating with 32-bit memory.

Bus Masters and DRAM Refresh

Since DRAM refresh is usually handled by the refresh logic on the system board, DRAM will start to lose information if it is not refreshed within the refresh interval specified by the DRAM chip manufacturer. The system board refresh logic refreshes one row every fifteen microseconds. The refresh logic requires the use of the buses to accomplish this task. It should be noted that an ISA bus master that controls the buses for any period of time should be prepared to handle DRAM refresh. The bus master should either give up ownership of the buses a least every 15 microseconds or should request that a refresh cycle be run every 15 microseconds.

ISA bus masters can trigger the refresh logic to run a refresh bus cycle by asserting the REFRESH# signal. The refresh logic normally asserts the REFRESH# signal when running a bus cycle to inform DRAM address decoders that a refresh cycle is in progress; however, the REFRESH# signal also acts as an input to the refresh logic to provide ISA bus masters with a means to trigger a refresh bus cycle. Note that ISA bus masters do not give up ownership of the buses during the refresh cycle, but must wait until the refresh cycle completes before continuing use of the buses. The steps involved in triggering a refresh cycle via bus mastering is as follows:

1. REFRESH# is asserted by the bus master. The bus master will not use the buses again until the refresh cycle has been completed.
2. The refresh logic sees the REFRESH# signal become asserted and starts a refresh bus cycle.
3. The refresh logic outputs the lower portion of the address to identify a row of memory to be refreshed, along with the MRDC# line.
4. The ISA bus master watches for the MRDC# line to become deasserted indicating that the refresh cycle has ended.
5. The bus master continues to run bus cycles.

Chapter 21

The Previous Chapter

The previous chapter provided a detailed description of system support required for ISA bus masters to function in various ISA machines.

This Chapter

This chapter describes the operation and usage of the real-time clock and configuration RAM found in all ISA machines.

The Next Chapter

The next chapter, "Keyboard/Mouse Interface," describes the interface between the keyboard and mouse and the system.

Introduction

Most ISA machines contain either a Motorola MC146818 real-time clock and configuration RAM chip or a Dallas Semiconductor equivalent. These chips consist of two functional areas:

1. 64 locations of CMOS low power RAM memory that retains stored information after power-down via a battery backup. In the case of the Motorola version, the battery is external, whereas the Dallas Semiconductor chip has the battery built in. This means that the Dallas Semiconductor chip must be replaced approximately every ten years.
2. The real-time clock function automatically keeps track of the date and time.

An external, 32.768KHz crystal oscillator supplies the RTC chip's timebase. The first ten RAM locations are used by the real-time clock function to keep track of the date (day and month) and time. The next four locations are used to control the real-time clock function. The remaining fifty locations are used to store system configuration information and the century portion of the date.

Accessing the RAM Locations

Via a two-step process, RAM locations inside the chip are accessed using I/O operations.

1. The programmer first identifies which of the 64 RAM locations is to be accessed by writing the RAM address to I/O port 70h, the chip's address port. The address must be in the range from 00h to 3Fh. Caution should be exercised when writing the address to port 70h because port 70h has a second function — bit 7 controls the enabling or disabling of NMI interrupts to the microprocessor (0 = enable NMI).
2. Next, the programmer either reads the contents of the previously identified CMOS RAM location from the chip's data port, I/O port 71h, or writes the desired data byte to the data port.

As an example, the proper sequence to read the contents of CMOS RAM location five would be:

1. Write the RAM address, 05, to the address port at I/O port 70h.
2. Read RAM location 5's contents by performing an I/O read from the data port, I/O port 71h.

Caution must be exercised to always follow a write to the address port with either a read from or a write to the data port. Failure to do so may result in unreliable operation of the real-time clock.

Address Decoder and RTC's Hardware Interface

Figure 21-1 provides a basic view of the RTC chip's interface to the system. The RTC's address decoder is designed to generate one of three possible chip-selects to the RTC chip:

- **ADSWR**. Address write is generated whenever the address decoder detects an I/O write to port 70h, the RTC's address port. When ADSWR goes high, the RTC chip accepts the data from the lower data path, D[7:0], and latches it into its address port. This identifies the CMOS RAM location to be communicated with during a subsequent read from or write to the data port (I/O port 71h).
- **DATARD#.** Data read is generated whenever the address decoder detects an I/O read from port 71h, the RTC's data port. When DATARD# goes low, the RTC chip reads the data from the CMOS RAM location identified

by the address currently in its address port. The data is placed onto the lower data path, D[7:0], to be read by the microprocessor.

- **DATAWR#.** Data write is generated whenever the address decoder detects an I/O write to port 71h, the RTC's data port. When DATAWR# goes low, the RTC chip accepts the data from the lower data path, D[7:0], and stores it into the CMOS RAM location identified by its address port.

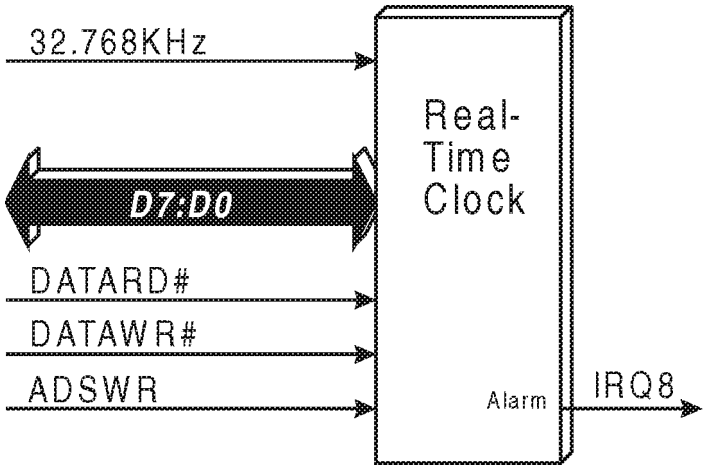


Figure 21-1. The RTC/Configuration RAM Chip

Real-Time Clock Function

The first ten RAM locations are used by the real-time clock function and are defined in table 21-1.

Table 21-1. The Real-Time Clock Bytes

Address	Function
0	Seconds
1	Seconds Alarm
2	Minutes
3	Minutes Alarm
4	Hours
5	Hours Alarm
6	Day of Week
7	Date of Month
8	Month
9	Year

Locations 1, 3 and 5 can be used to specify an alarm event to occur once per day (IRQ8 will be generated).

Locations Ah through Dh are referred to as status registers A through D and are used to control the operational characteristics of the real-time clock function. Tables 21-2 through 21-5 define the function of the bits in the four real-time clock status registers.

Table 21-2. RTC Status Register A

Bit(s)	Description
7	**Update in progress**. When set to 1, indicates that the chip is updating the time and date.
6:4	**22-Stage divider**. Indicate timebase in use. Always set to 010b to indicate a 32.768KHz timebase.
3:0	**Rate selection bits**. Allow selection of an output frequency. Set to 0110b, selects a 1.024KHz square-wave output frequency and a 976.562 microsecond Periodic Interrupt rate.

Table 21-3. RTC Status Register B

Bit(s)	Description
7	**Set bit**. When 0, this bit allows the date/time to be updated normally at the rate of once per second. When set to a 1, updates are disallowed and the programmer may change the date and time without interference from updates.
6	**Periodic interrupt enable**. When set to 1, allows IRQ8 to occur at the 1.024KHz rate specified in status register A. This bit is normally 0.
5	**Alarm interrupt enable.** When set to 1, this bit allows an IRQ8 to be generated at the selected alarm date/time. This bit is normally 0.
4	**Update-ended interrupt enable.** When set to 1, the chip will generate an IRQ8 each time a date/time update is completed in the device's RAM. This bit is normally 0.
3	**Square-wave enabled.** When set to 1, this bit enables the generation of the output frequency selected in status register A. This bit is normally 0.
2	**Date mode**. This bit defines whether the date and time will be represented in Binary or BCD formats. A 1 selects binary mode. This bit is normally 0.
1	**24-hour mode.** This bit defines whether the hours byte is presented in 24 or 12 hour format. A 1 selects 24-hour mode. This bit is normally 1.
0	**Daylight-savings enabled.** A 1 enables daylight-savings time mode. This bit is normally 0.

Table 21-4. RTC Status Register C

Bit(s)	Description
7	**Interrupt request flag**. This bit is set to a 1 when the chip has generated an interrupt on IRQ8. Bits 4, 5 and 6 of this register may then be checked to ascertain the cause of the interrupt.
6	**Periodic interrupt flag.** Set to 1 when an periodic interrupt is generated.
5	**Alarm interrupt flag.** Set to 1 when an alarm interrupt is generated.
4	**Update-ended interrupt flag.** Set to 1 when an update-ended interrupt is generated.
3:0	Not Used.

Table 21-5. RTC Status Register D

Bit(s)	Description
7	**RAM valid**. When set to 0, indicates that the battery is dead or disconnected and RAM data has therefore been lost.
6:0	Not Used.

Using BIOS to Control Real-Time Clock

The recommended method for controlling the real-time clock chip is through BIOS function calls. INT 1Ah can be used to call the time-of-day BIOS routine. A complete description of this BIOS routine's usage can be found in either of the following publications:

- *The IBM PS/2 and Personal Computer BIOS Interface Technical Reference* published by IBM.
- *System BIOS for IBM PC's, Compatibles, and EISA Computers* by Phoenix Technologies Ltd., published by Addison-Wesley.

Configuration RAM Usage

Table 21-6 defines the ISA-standard usage of the fifty configuration RAM locations in the MC146818 chip.

Table 21-6. Configuration RAM Usage

Location (Hex)	Description of Use
0E	Diagnostic Status Byte
0F	Reset Code Byte
10	Diskette Drive Type Byte
11	Reserved
12	Fixed Disk Drive Type Byte
13	Reserved
14	Equipment Installed Byte
15 to 16	Low and High Base Memory Bytes
17 to 18	Low and High Extended Memory Bytes
19 to 2C	Reserved
2D	Additional Flags
2E to 2F	Checksum Value
30 to 31	Memory above 1MB
32	Century part of time and date function
33	System information
34 to 3F	Reserved

Table 21-7 identifies the valid values that may be stored in the reset code byte at CMOS RAM location 0Fh. Whenever the microprocessor is reset, it begins executing the POST again. At the beginning of the POST, the program reads the reset code byte from CMOS RAM to determine the course of action to be taken.

Table 21-7. The Reset Code Byte

Value	Meaning
00h	Normal Power-Up reset or Control-Alternate-Delete reset, depending on the value of the reset flag word in memory at location 0040:0072. When the reset flag contains the value 1234h, the POST memory test is skipped.
04h	POST is skipped. Causes DOS to be loaded from disk.
05h	POST is skipped and memory preserved. The 80286 microprocessor cannot be switched from protected mode into real mode without resetting the processor. This presents problems when the programmer wishes to temporarily switch to protected mode to access extended memory and then switch back into real mode to continue execution. When a program must temporarily leave real mode and enter protected mode, the programmer first stores a four byte memory address (pointer) into memory location 0040:0067. This pointer specifies the address at which real mode execution will resume. Having done this, the program may switch the processor into protected mode and access extended memory. Upon completing the protected mode function, the programmer must place the value 05h or 0Ah into the reset code byte in CMOS RAM and then reset the 80286 microprocessor. When reset goes away, the processor is in real mode and begins executing the POST. The POST code checks for a 05h or a 0Ah in the reset code byte, and, if present, jumps to the memory address specified in memory starting at location 0040:0067 to resume execution at the point where it left off in real mode. In addition, if the reset code byte value is 05h, an EOI command is issued to the interrupt controller.
09h	Block-move return. This value is placed in the reset code byte by a BIOS call to INT 15h to move a block of information between conventional and extended memory. In order to accomplish this move, the processor must switch into protected mode. On an 80286-based system, the INT 15h routine sets the reset code byte to 09h prior to resetting the processor to return to real mode after the block move. When reset, the processor starts executing the POST, checks the reset code byte value, and then performs an interrupt return, or IRET, instruction to return to the original program after completion of the call to the INT 15h BIOS routine.
0Ah	Jump to reset vector 0040:0067 without issuing EOI. See the explanation for reset code byte = 05h.

It should be noted that systems based on the 80386 and higher microprocessors can switch from protected mode to real mode without resetting the processor (by turning off the PE bit in the MSW register).

Chapter 22

The Previous Chapter

The methods of communication over the ISA bus have been covered, including read and write operations, interrupt driven communication with I/O devices, DMA communication between memory and I/O, and bus mastering.

This Chapter

The interface for the keyboard controller is implemented using an Intel 8042 series microcontroller known as a universal peripheral interface. This device also serves other miscellaneous system functions. This chapter discusses the 8042 implementation of the combination keyboard/mouse interface and the standard ISA functions handled by this interface.

The Next Chapter

The next chapter explains the interface between the microprocessor and numeric coprocessor. The Intel 80287, 80387 and the Weitek coprocessors are covered.

Keyboard/Mouse Interface

Every ISA system includes a keyboard interface on the system board and some systems implement a combined keyboard/mouse interface. Both implementations consist of an Intel 8042 microcomputer with its own RAM, ROM and I/O. The 8042 is a self-contained computer system that the system board microprocessor communicates with using two I/O registers.

The 8042 communicates with the keyboard and the mouse using two independent serial interfaces. The keyboard and the mouse both receive their power from the system board.

The 8042 microprocessor has additional input and output pins that can be used for other functions. Four of the output functions are ISA specific: IRQ1 (the

keyboard interrupt request), IRQ12 (the mouse interrupt request), Hot Reset and A20 Gate. These signals are covered in detail later in this chapter. Other input and output functions may also be found in ISA systems but are manufacturer specific.

Keyboard

The key switches are laid out in rows and columns within the keyboard. Another 8042 (or a dedicated part such as an 8052) controller resides in the keyboard and is constantly scanning the key matrix to see if a key has been pressed. When a key is pressed, the 8042 creates an 8-bit scan code that represents the key pressed. This code is commonly referred to as the "make" code. The scan code is then converted to a serial bit stream and sent to the keyboard/mouse interface on the system board. When the key is released, another scan code is generated and is called the "break" code.

Once an entire 8-bit scan code has been received, the keyboard/mouse interface generates an IRQ1 interrupt request to alert the microprocessor. The microprocessor jumps to the keyboard interrupt service routine and an I/O read from data port 60h is performed to read the scan code from the 8042 on the system board. IRQ1 remains asserted until the scan code is read and is then automatically deasserted.

Mouse

Like the keyboard, the mouse generates an 8-bit code that is handled in the same manner by the keyboard/mouse interface. When the mouse code is received by the interface, it generates an IRQ12 to alert the microprocessor that a byte is ready to be transferred.

8042 Local I/O Ports

Refer to figure 22-1. The keyboard controller's input and output ports have additional lines that can be used for other functions besides the keyboard and mouse interface. Many older ISA systems use the local input ports to read system configuration information from switches or jumpers. These inputs are defined in various ways and are manufacturer dependent. However, the keyboard controller output signals HOT RESET and A20 GATE are ISA specific and virtually all manufacturers implement these outputs in order to remain compatible. Figure 22-1 shows the standard uses of the local ports.

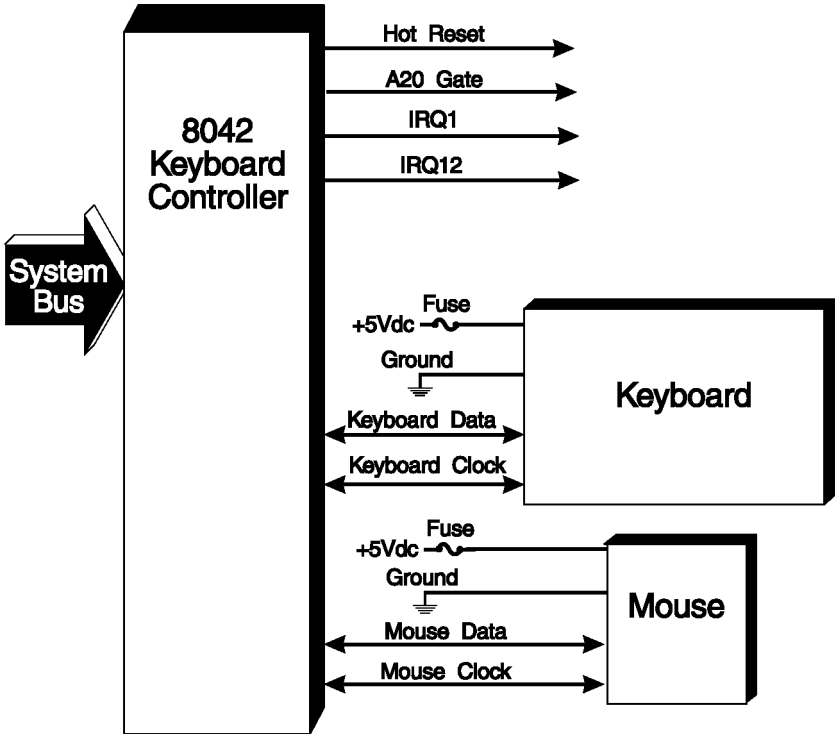


Figure 22-1. The Keyboard Controller

Hot Reset

The 80286 microprocessor must be reset to change the processor from protected back to real mode. Hot Reset causes the "CPU only" reset needed to change the microprocessor from protected to real mode. Writing a FEh to I/O port 64h, the keyboard command port, causes the HOT RESET line to be pulsed. Refer to the chapter entitled "The 80286 Microprocessor" for additional information on changing from protected back to real mode.

A20 Gate

Depending on the schematic, A20 GATE may be called A20MASK#, PASS A20, FORCE A20, etc., all of which apply to the same function and signal. When deasserted, A20 GATE inhibits the generation of address line A20 during real

mode operation, making the 80286 and higher microprocessors compatible with the segment wrap-around that occurs on 8086 and 8088 microprocessors. Note that the polarity of the signal affects its meaning. Whereas A20 GATE would be asserted to pass A20 from the microprocessor to memory, A20MASK# would be deasserted to do the same thing.

Local Port Definition

The 8042 microcontroller has three local ports that are used in most configurations as shown in figure 22-2. These ports include:

- An 8-bit input port used for keyboard data, mouse data and miscellaneous inputs.
- An 8-bit output port used for keyboard data, mouse data, Hot Reset, A20 Gate and miscellaneous outputs.
- A 2-bit test port used to read the keyboard and mouse clock signals.

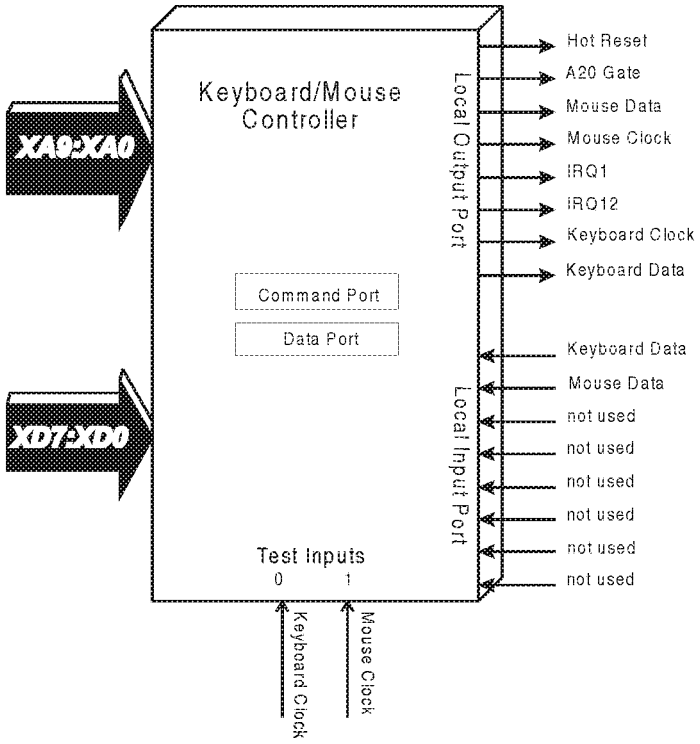


Figure 22-2. Sample 8042 Configuration

Note that these definitions are typical of the keyboard/mouse interface. Use of these ports vary depending on the specific implementation. Tables 22-1 through 22-3 provide sample bit assignments for the 8042's local I/O ports.

Table 22-1. Sample Keyboard/Mouse Interface—Local Input Port

Bit	Description
7:2	Not used.
1	Mouse Data input line.
0	Keyboard Data input line.

Table 22-2. Sample Keyboard/Mouse Interface—Local Output Port

Bit	Description
7	Keyboard Data output line
6	Keyboard Clock output line
5	IRQ12
4	IRQ1
3	Mouse Clock output line
2	Mouse Data output line
1	Gate Address Bit 20
0	Reset microprocessor (Generates Hot Reset)

Table 22-3. Keyboard/Mouse Interface—Test Port

Test Port	Description
0	Keyboard Clock input
1	Mouse Clock input

System Interface

The microprocessor uses two I/O registers to communicate with the 8042 keyboard/mouse interface:

- Command/Status Port 64h
- Data In/Out Port 60h

Command/Status Port

To command the keyboard/mouse interface (the 8042) to perform a task, the microprocessor uses an I/O write to send the command to the 8042. The 8042 microcomputer's ROM program then interprets the command and jumps to the appropriate subroutine (series of instructions) to accomplish the requested task.

The microprocessor issues commands to the keyboard/mouse command port at I/O address 0064h. Table 22-4 provides a list of typical commands that can be sent to the 8042 keyboard/mouse interface's command port. The precise commands used by a particular system can vary depending on the features built into the system. For example, not all systems support the password feature that is found in the list. In short, the commands listed in table 22-4 will not necessarily match the commands found in all ISA systems. Some systems will not support all of the commands found in this list, while others may include additional commands.

Note that the command port allows the microprocessor to indirectly read from and write to the keyboard/mouse interface's local ports. For example, to read the contents of the local input port requires that a C0h be first written to the command port (64h). This command causes the values at the local input port to be sent to the data port (60h) where they can then be read by the microprocessor.

This command port can also be read to obtain status information from the keyboard/mouse interface.

Table 22-4. 8042 Sample Keyboard/Mouse Interface Command List

Cmd (hex)	Description
20 to 3F	Bits 5 through 0 of this command designate one of 64 possible 8042 internal RAM locations to be read. As an example, 23h causes internal RAM location 03h to be read. Its contents are placed in the data port at I/O address 0060h so the microprocessor can read it.
60 to 7F	Bits 5 through 0 of this command designate one of 64 possible 8042 internal RAM locations to be written. The next byte written to the data port at I/O address 0060h is written to the designated internal RAM location. As an example, 63h causes internal RAM location 03h to be written. The next byte written to the data port at I/O address 0060h is written to the designated location.
A4	Test password installed. This command checks for a password installed in the 8042's internal RAM. FAh is placed in the data port at I/O address 0060h if the password is installed. A F1h indicates it isn't installed.
A5	Load password. Initiates the password load procedure. The 8042 then accepts input from the data port at I/O address 0060h until a null (00h) is received.
A6	Enable password feature.
A7	Disable mouse Interface.
A8	Enable mouse Interface.
A9	Mouse interface test. Tests the mouse clock and data lines. Result is placed in the 8042's data port at I/O address 0060h.
AA	Self-test. Runs an 8042 self-test. 55h placed in the data port at I/O address 0060h if no errors are detected.
AB	Keyboard interface test. Tests the keyboard clock and data lines. Result is placed in the 8042's data port at I/O address 0060h.
AD	Disable keyboard interface.
AE	Enable keyboard interface.
C0	Read input port. Allows the programmer to read the contents of the 8042's local input I/O port. The result is placed in the data port at I/O address 0060h.
C1	Poll input port low. Bits 3:0 of the 8042's local input I/O port are read and placed in bits 7:4 of the status register at I/O address 0064h.
C2	Poll input port high. Bits 7:4 of the 8042's local input I/O port are read and placed in bits 7:4 of the status register at I/O address 0064h.
D0	Read output port. Causes the 8042 to read the current state of its local output I/O port and place the result in the data port at I/O address 0060h.
D1	Write output port. The next byte written to the data port at I/O address 0060h is written to the 8042's local output I/O port.

Table 22-4. 8042 Sample Keyboard/Mouse Interface Command List, continued

D2	Write keyboard output buffer. The next byte written to the data port at I/O address 0060h causes a keyboard Interrupt on IRQ1 (if the keyboard Interface is enabled). The byte is supplied as the keyboard scan code when the data port is read.
D3	Write mouse output buffer. The next byte written to the data port at I/O address 0060h causes a mouse Interrupt on IRQ12 (if the mouse Interface is enabled). The byte supplies the mouse scan code when the data port is read.
D4	Write to mouse. The next byte written to the data port at I/O address 0060h is transmitted to the mouse.
E0	Read test inputs. Causes the 8042 to read the current state of the keyboard and mouse clock lines. The results are stored in bits 0 and 1, respectively, of the data port at I/O address 0060h.
F0 to FF	I/O port are pulsed low for approximately 6 microseconds. Bits 3:0 of this command indicate which bits are to be pulsed. 0 indicates the respective bit should be pulsed, while 1 indicates the respective bit shouldn't be altered.

Data Port

The keyboard/mouse data port at I/O address 0060h is used to:

- Read the scan code sent from the keyboard or mouse.
- Send data/commands to the keyboard.

Scan codes from the keyboard and mouse go directly to the output buffer (port 60h) where they can be read from the microprocessor. The local output port and test port also can be read at port 60h. By writing the proper command to port 64h, the contents of the local ports are sent to port 60h where they also can be read by the microprocessor.

Through port 64h, the keyboard Interface can also be commanded to write the contents of port 60h to the local ports. In this way, the Hot Reset and Gate A20 commands can be issued to the system.
Most keyboards have LED indicators for scroll lock, num lock, caps lock, etc., that can be turned on and off by issuing commands to the keyboard.

BIOS Routine

The recommended method for controlling the keyboard/mouse interface is through BIOS function calls. INT 09h and INT 16h can be used to call the keyboard/mouse BIOS routine.

Chapter 23

The Previous Chapter

The previous chapter provided a detailed description of the keyboard and mouse interfaces to the system.

This Chapter

This chapter provides a description of the numeric coprocessors commonly found in ISA systems. This includes both Intel and Weitek coprocessors.

The Next Chapter

The next chapter describes the timers found in all ISA systems.

Introduction

When Intel was designing the 8086 and 8088 microprocessors, they had a decision to make — should the 8086 and 8088 be given the ability to perform floating-point mathematics? Floating-point numbers are numbers with decimal places. If this ability were included within the microprocessor itself, it would substantially increase the price of the microprocessor. Marketing information indicated that most users didn't need this capability, so it was not included in the final processor design. The microprocessor can perform integer, or whole number, calculations, but not floating-point.

Rather than completely eliminate the floating-point capability, however, they included hooks on the microprocessor that would enable users to include a companion floating-point processor in designs that required this capability. The companion processor, or coprocessor, was the 8087 numeric coprocessor. It is also referred to as a processor extension because it extends the microprocessor's instruction set. When the 80286 and 80386 microprocessors were introduced, the same philosophy was adhered to — the 80286 can be coupled with the 80287 and the 80386 with either the 80287 or the 80387 numeric coprocessor.
The microprocessor treats the numeric coprocessor as an I/O device. When the microprocessor fetches an instruction from memory and senses that it is an in-

struction for the coprocessor, it automatically performs a series of one or more I/O writes to forward the instruction to the coprocessor, which then executes the instruction. The I/O address range from 00F8h through 00FFh are assigned to the coprocessor.

Setup

At the beginning of the POST, the programmer checks the configuration RAM to see if a numeric coprocessor is installed. Bits 1 and 2 of the microprocessor's machine status word (MSW) register are then set according to table 23-1.

Table 23-1. The Numeric Coprocessor Control Bits in the MSW Register

Bit 2 (EM)	Bit 1 (MP)	Description
0	0	Coprocessor is installed. Microprocessor forwards all floating point instructions to coprocessor. Microprocessor does not check to see if the it has switched tasks when executing floating point instructions. Microprocessor operates as an 8086 or 8088, single-tasking.
0	1	Coprocessor is installed. Microprocessor forwards all floating point instructions to coprocessor. Microprocessor checks to see if it has switched tasks when executing floating point instructions. Used for multi-tasking.
1	0	Coprocessor not installed. Emulate coprocessor.
1	1	Invalid.

Note: In the 80386 and higher microprocessors, the MSW register is found in the lower half of CR0 (control register 0).

With Numeric Coprocessor Installed

Refer to figure 23-1. The microprocessor is aware of the I/O addresses of the coprocessor's two ports. The coprocessor's opcode, or instruction, register resides at port 00F8h in an 80286-based system, 8000F8h in an 80386SX-based system, and 800000F8h in an 80386DX-based system. The coprocessor's data register resides at port 00FCh in an 80286-based system, 8000FCh in an 80386SX-based system, and 800000FCh in an 80386DX-based system. When the coprocessor is installed and the MP bit is set to one, the following actions take place when the microprocessor fetches a coprocessor instruction from memory:

1. If the execution of the coprocessor instruction requires a memory read or a memory write operation, the microprocessor saves the memory address, the length of the data and notes whether a memory read or write is required.
2. If the coprocessor's BUSY# output is deasserted, indicating that it is available to accept the next instruction, the microprocessor performs one or more I/O writes to the coprocessor's instruction register at I/O port 00F8h (800000F8h for the 80386DX or 8000F8h for the 80386SX) in order to forward the instruction to the coprocessor. The 80286 and 80386SX microprocessors have a 16-bit data bus, allowing one or two bytes to be transferred to the coprocessor in a single bus cycle. The 80386DX microprocessor, on the other hand, has a 32-bit data bus, allowing from one to four bytes to be transferred to the coprocessor in a single bus cycle. Simple instructions that are only one or two bytes in length can be transferred in a single I/O write bus cycle, while more complex instructions consisting of three or more bytes may require additional I/O write bus cycles.
3. When the coprocessor begins execution of the instruction, it asserts its BUSY# output, thereby signaling that it cannot accept another instruction from the microprocessor until the currently executing instruction is completed. The microprocessor will not attempt to forward another instruction to the coprocessor until BUSY# is deasserted.
4. If execution of the instruction requires that a memory read or memory write bus cycle take place, the coprocessor asserts PEREQ (processor extension request) to request that the microprocessor run the memory read or write bus cycle.
5. This step only applies to the 80286. The 80386 microprocessor does not have a PEACK# output. The 80286 microprocessor responds with PEACK# (processor extension acknowledge) to inform the coprocessor that the transfer request has been acknowledged.

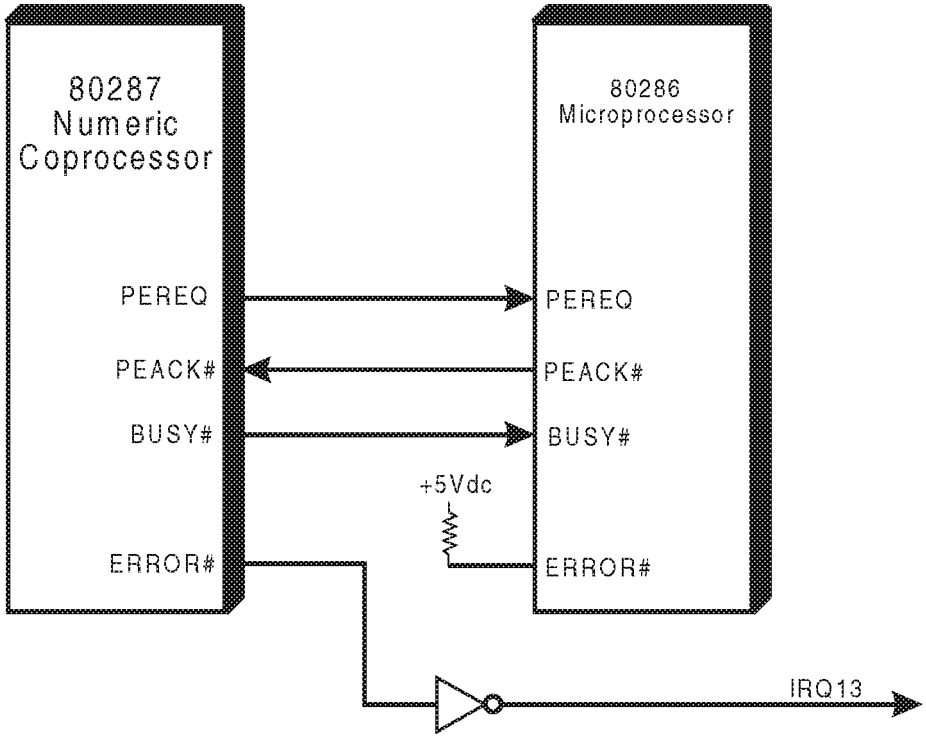


Figure 23-1. The 80287 Numeric Coprocessor

6. If the requested transfer is a memory write, the microprocessor performs two (or more) bus cycles to transfer the data into memory. During the first bus cycle, the microprocessor performs an I/O read from the coprocessor's data register at I/O address 00FCh (800000FCh for the 80386DX or 8000FCh for the 80386SX) to get the data. The processor then performs a memory write bus cycle to write the data into the target memory locations.

   If the requested transfer is a memory read, the microprocessor performs two (or more) bus cycles to transfer the data from memory to the coprocessor. During the first bus cycle, the microprocessor performs a memory read from the target memory locations to get the data. The processor then performs an I/O write to I/O address 00FCh (800000FCh for the 80386DX or 8000FCh for the 80386SX) to pass the data to the coprocessor's data register.

7. If the coprocessor incurs an error while executing the instruction, its ERROR# and BUSY# outputs are asserted. This generates an IRQ13 to the microprocessor and prevents the microprocessor from forwarding another coprocessor instruction. In the IRQ13 interrupt service routine, the programmer reads the coprocessor's status register to ascertain the cause of the error. Any coprocessor error may be reset by writing 00h to I/O address 00F0h. This causes the coprocessor's RESET input to be asserted, clearing any error conditions.
8. When the coprocessor completes execution of the instruction, it deasserts its BUSY# output to inform the microprocessor that it is ready to accept another coprocessor instruction.

Numeric Coprocessor Reset

In systems incorporating the 80287 numeric coprocessor, an output to I/O port F1h is necessary to reset the 80287 and return it to real mode operation after execution of a FSETPM (floating-point set processor mode) instruction. Writing any data to this port causes the 80287's reset line to be asserted momentarily. The 80387 numeric coprocessor doesn't require a reset when changing processor modes.

Writing a 00h to I/O port F0h causes the numeric coprocessor to clear its BUSY# and ERROR# outputs.

Table 23-2 provides a description of the Intel coprocessor signals involved in communication with the microprocessor.

Table 23-2. Intel Coprocessor/Processor Communication Signals

Signal Name	Coprocessor Type	Description
PEREQ	287/387	Processor extension request. Asserted by the coprocessor when a transfer to or from memory is required. In response, the microprocessor performs two or more bus cycles to transfer the data between memory and the coprocessor.
PEACK#	287	Processor extension acknowledge. Asserted by the 80286 coprocessor in response to PEREQ.
BUSY#	287/387	Asserted by the coprocessor when it begins executing an instruction. The processor may not send another instruction to the coprocessor while this signal is asserted.
ERROR#	287/387	Generated by the coprocessor when an error condition is encountered while executing an instruction. In an ISA machine, ERROR# is connected to the IRQ13 line.
D[31:0]	287/387	Data bus is used to transfer instructions or data to the coprocessor from the processor. Also used to transfer data or status from the coprocessor to the processor. The 80286 and 80386SX processors and the 80287 and 80387SX coprocessors have a 16-bit data bus, while the 80386DX processor and the 80387DX coprocessor have a 32-bit data bus.
W/R#	387	This is the 80386 processor's write or read output. During an access to the coprocessor, W/R# indicates whether a write or read operation is in progress.
ADS#	387	This is the 80386 processor's address status output. When asserted at the start of a bus cycle, indicates to the coprocessor that it should examine its chip-select inputs to determine if the processor is addressing the coprocessor.
READY#	387	This is the same signal that is connected to the microprocessor's READY# input. It indicates when a bus cycle is being completed by the processor.
READYO#	387	This is the coprocessor's ready output to the processor. It indicates to the processor when the coprocessor is ready to end a bus cycle initiated by the microprocessor.
STEN	387	Status Enable. This pin serves as a chip-select to the coprocessor. When sensed asserted, the coprocessor examines its CMD and NPSn inputs to determine which of its internal registers is currently being addressed.
NPS1#	287/387	Numeric Processor Select 1. When asserted (low), indicates that an I/O operation is in progress. Should be connected to the microprocessor's M/IO# output. In conjunction with NPS2, the CMD input(s) and STEN (in the case of an 80387), it is used as a chip-select for the coprocessor.

Table 23-2 Intel Coprocessor/Processor Communication Signals, continued

Signal Name	Coprocessor Type	Description
NPS2	287/387	Numeric Processor Select 2. In conjunction with NPS1#, the CMD input(s) and STEN (in the case of an 80387), it is used as a chip-select for the coprocessor. In a system incorporating the 80287, NPS2 is usually just tied high. In an 80386SX/80387SX system, it is tied to A23, while in an 80386DX/80387DX system, it is tied to A31.
CMD0 or CMD0#	287 (CMD0) 387 (CMD0#)	Used to select the target I/O register in the coprocessor. In an 80386-based system, the processor's A2 output is connected to CMD0# on the 80387. A2 goes high when the coprocessor's data register is addressed, and low when its instruction register is addressed. In an 80286-based system, CMD0 and CMD1 are used to select the target I/O register in the coprocessor. CMD0 is connected to the processor's A1 output.
NPRD#	287	Numeric Processor Read. Connected to the bus control logic's IORC# line. Indicates to the coprocessor when an I/O read bus cycle is in progress.
NPWR#	287	Numeric Processor Write. Connected to the bus control logic's IOWC# line. Indicates to the coprocessor when an I/O write bus cycle is in progress.
CMD1	287	In an 80286-based system, CMD0 and CMD1 are used to select the target I/O register in the coprocessor. CMD1 is connected to the processor's A2 output.

Detailed information about coprocessor programming may be found in the following Intel manuals:

- *iAPX 286 Hardware Reference Manual*
- *iAPX 286 Programmer's Reference Manual* including the iAPX 286 Numeric Supplement
- *80386 Hardware Reference Manual*

Without Numeric Coprocessor Installed (Emulation)

When a system running Lotus 1-2-3 recalculates a spreadsheet, the decimal portion of the numbers are calculated correctly, even if the coprocessor isn't installed. Since the microprocessor can only perform integer mathematics, this doesn't seem possible. In this case, the floating-point calculations are performed by special routines known as floating-point emulation routines. These routines are supplied as part of the applications program (such as Lotus 1-2-3).

Using the standard instruction set, floating-point calculations can be performed through laborious manipulation of floating-point numbers stored in memory.

This process is much less efficient than the floating-point instruction capability embedded within the coprocessor. Where the coprocessor can directly execute a single floating-point instruction with its hardware, an emulation routine would have to execute hundreds of instructions to accomplish the same end. When running applications programs that are floating-point intensive, the performance improvement provided by the coprocessor can be quite dramatic. Improvements of from 100% to 500% aren't uncommon.

Floating-point emulation requires the following:

- Inclusion of a set of emulation routines in the applications program (such as 1-2-3).
- In the MSW register, the EM bit must be set to 1 and the MP bit set to 0.

This section describes the actions taken when a coprocessor isn't installed and the EM bit is set to 1 in the MSW register.

1. The microprocessor fetches a coprocessor instruction from memory. When the microprocessor decodes the instruction and senses that it belongs to the coprocessor, it checks the MP bit in the MSW register to see if a coprocessor is installed. If it isn't (MP = 0), it checks the EM bit to see if the coprocessor instruction is to be emulated. In this case, it is (EM = 1).
2. The microprocessor internally generates an exception 7 interrupt request. As with any interrupt, the microprocessor responds by pushing the contents of the CS, IP and Flag registers into stack memory. The stored CS and IP values point to the coprocessor instruction that caused the exception.
3. The microprocessor jumps to the exception 7 interrupt service routine. This is the emulation program supplied as part of the applications program. In the routine, the programmer reads the coprocessor instruction using the memory address stored in the stack and determines the type of instruction (such as a floating-point divide). The programmer then jumps to the routine (in the package of emulation routines) that emulates that particular type of instruction.
4. Upon conclusion of the instruction emulation, the programmer alters the IP value stored on the stack to point to the instruction immediately following the coprocessor instruction that caused the exception.
5. The programmer executes an IRET (interrupt return) instruction. The microprocessor responds by reloading the CS, IP and Flag registers from the stack and resuming normal operation — in other words, the microprocessor fetches the next instruction from the memory location pointed to by CS and IP. This is the instruction after the coprocessor instruction that caused the exception.

Weitek Numeric Coprocessor

Introduction

ISA systems may incorporate either an Intel or a Weitek coprocessor, or both. The Weitek coprocessor does not reside in the same address space as the Intel coprocessor, so addressing conflicts will not arise if both are implemented. Rather than using I/O addresses as does the Intel coprocessor, the Weitek device is memory-mapped. It responds to memory addresses C0000000h through C1FFFFFFh.

Systems Incorporating Just the Weitek

General

In a system using the Weitek coprocessor to perform floating-point calculations, two possible scenarios may coexist:

- By re-compiling a program that uses Intel coprocessor-specific instructions, the Weitek can handle the execution of all Intel coprocessor-specific instructions, or alternatively a driver routine can be installed to automatically handle the instructions.
- Applications programs can incorporate Weitek-specific instructions as in-line code within the applications itself.

The second case is clear-cut, but the first one merits some explanation.

Weitek Handling of Intel Coprocessor Instructions

Most developers that use the Weitek coprocessor embed move instructions into their floating-point code to pass the instruction directly to the Weitek Coprocessor. Alternatively, a Weitek driver routine could perform a memory write to pass the instruction and any data operands to the Weitek coprocessor.

In order for a driver to direct all instances of Intel coprocessor-specific instructions to the Weitek, the EM bit in the MSW register should be set to a one. An Intel-specific floating-point instruction fetched from memory then causes the processor to experience an exception 7 interrupt. As a result, the processor jumps to the exception 7 handler routine. This is the Weitek driver routine.

As illustrated in table 23-3, the memory address that is written to identifies both the coprocessor's internal register as well as the instruction type. In addition, up to four bytes of data are simultaneously written to the Weitek over the four data paths. Upon receipt of the instruction, the coprocessor asserts READY#, permitting the processor to end the bus cycle. Having passed the instruction to the coprocessor, the Weitek driver routine modifies the return address pointer on the stack to point to the instruction after the one that caused the exception. It then executes an interrupt return, or IRET, instruction to return to the original program flow.

Table 23-3. Weitek Interpretation of the Memory Address

Address Bits	Identify
A[31:16]	The register within the Weitek coprocessor.
A[15:10]	Instruction opcode, or type.
A[9:7]	Source 1 register in the Weitek.
A[6:2]	Source 2 or destination register in the Weitek.
BE#[2:0]	Additional information regarding the source 1 register in the Weitek.

Systems Incorporating Both Coprocessor Types

In a system incorporating both coprocessor types, the Intel coprocessor handles execution of all Intel coprocessor-specific instructions, while the Weitek handles execution of all Weitek-specific code embedded within applications programs. In the MSW register, EM is set to zero and MP is set to one to indicate the presence of an Intel coprocessor.

Chapter 24

The Previous Chapter

The previous chapter provided a description of the numeric coprocessor interface and floating-point emulation.

This Chapter

This chapter describes the system timers incorporated in all ISA-compatible machines.

ISA Oscillator, OSC

ISA systems include a free-running 14.31818MHz clock named OSC that may be used by ISA add-in boards. OSC runs at four times the television color burst frequency and may be used by video display adapter cards or any other device needing a free-running clock.

The 14.31818MHz OSC oscillator signal is divided by 12 by logic on the system board to provide the 1.19318MHz signal that drives timer 0, timer 1 and timer 2. The OSC signal was divided by 3 by the 8284 clock chip in the original PC to provide the 4.77MHz clock that drove the 8088 microprocessor.

System Timer (Timer 0)

The system timer, timer 0, is a programmable frequency source. The programmer specifies a divisor to be divided into the input clock to yield the desired output frequency. During the POST, the divisor FFFFh, or a decimal 65535, is programmed into timer 0 at I/O address 0040h. Based on the 1.19318MHz input clock, the output frequency of timer 0 after POST is 18.21Hz.

Refer to figure 24-1. Timer 0's output is connected to the IRQ0 latch. When timer 0's output becomes asserted, the latch is set, sending IRQ0 to the master 8259 interrupt controller. In the IRQ0 interrupt service routine, the programmer

can reset (deassert) IRQ0 by setting bit 7 of system control port B at I/O address 0061h to a one.

At a frequency of 18.21Hz, IRQ0 is generated every 54.9ms, causing the IRQ interrupt service routine to be executed once every 54.9ms.

In the original IBM PC and XT, this interrupt service routine was used to maintain the date and time in RAM memory. In the IBM PC/AT, a real-time clock chip was added to perform this function, but timer 0 is still available for use as a general purpose timer by programmers.

There's a funny story about this timer. In 1989, a number of industry publications reported the astounding news that a designer in Taiwan had altered a PC/AT so that it could perform at the positively astounding rate of 100 MIPS (million instructions per second). A number of benchmark (performance verification) programs had been run on the system to verify his claims. Quite a few people in the industry were reeled in; hook, line and sinker. All this guy had done was re-program timer 0 to interrupt at a different rate. Since the benchmark programs used this interrupt as their timebase, it altered the results. Red faces all around.

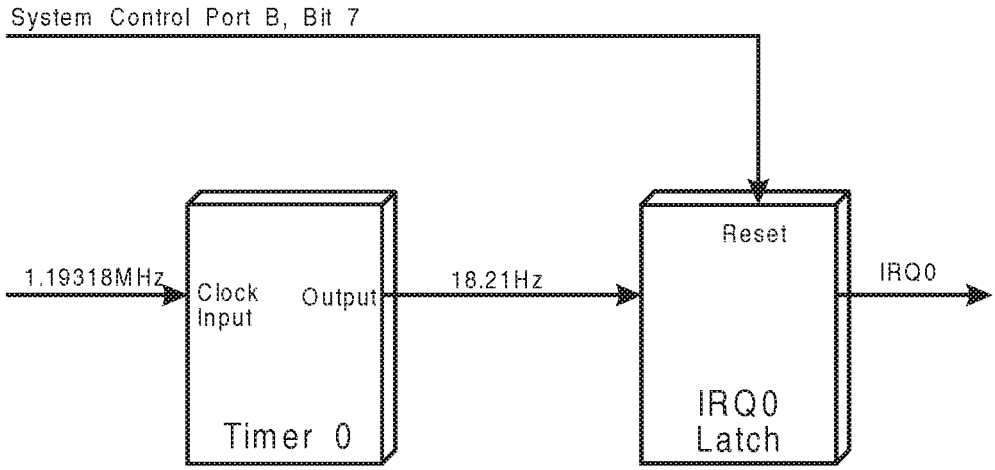


Figure 24-1. The System Timer, Timer 0

Refresh Timer (Timer 1)

The refresh timer, or timer 1, is a programmable frequency source. The same 1.19318MHz signal (used by timer 0) provides the refresh timer's clock input. The programmer specifies a divisor to be divided into the input clock to yield the desired output frequency. During the POST, a divisor of 0012h, or a decimal 18, is written to the refresh timer at I/O address 0041h. The input clock frequency of 1.19318MHz is therefore divided by 18 to yield an output frequency of 66287.77Hz, or a pulse every 15.09 microseconds.

This is the refresh request signal that triggers the DRAM refresh logic to become bus master once every 15.09 microseconds so it can refresh another row in DRAM memory throughout the system. For more information on DRAM refresh, refer to the chapter entitled "RAM Memory: Theory of Operation."

In 1988, a number of industry publications published a brief but dangerous article that addressed the issue of the refresh timer. It was revealed that this demon timer "steals" the buses from your computer once every 15 microseconds to refresh DRAM memory. A small program was provided that would reprogram this timer so it couldn't ask for the buses quite so frequently, thereby giving more bus time to your computer.

This was an extremely dangerous factor to be messing around with. It's entirely possible that users who implemented this change would start to experience mysterious RAM parity errors from time to time and lose all data they currently had in RAM because the unit locked up after outputting the message.

Speaker Timer (Timer 2)

Timer 2, the speaker timer, resides at I/O address 0042h and, as with timer 0, is a programmable frequency source derived from the 1.19318MHz timebase.

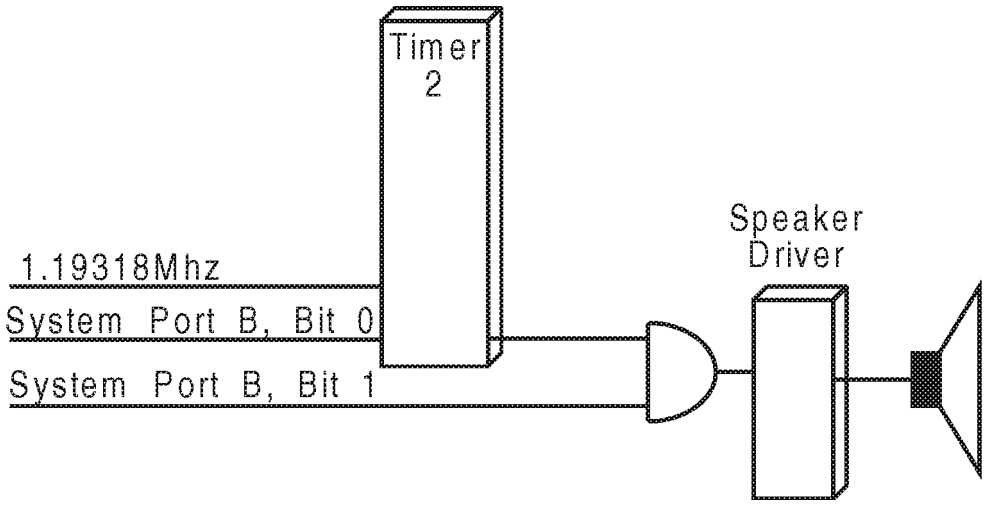
The programmer specifies a divisor to be divided into the input clock to yield the desired output frequency. The divisor is written to timer 2 at I/O address 0042h. Figure 24-2 illustrates the speaker timer and its associated logic.

In order for timer 2 to actually use the input clock, bit 0 of system control port B at I/O address 0061h must be a 1. The timer then divides the timebase by the programmer-specified divisor, yielding an output frequency that is applied to one input of a two input AND gate. If the AND gate's other input is a 1 (from bit 1 of system control port B at I/O address 0061h), the timer 2 output is gated

through the AND gate and applied to the speaker driver chip, where it is amplified and applied to the speaker.

The output frequency of timer 2 is applications-specific — it can be programmed to make happy or sad sounds, depending on the occasion.

Figure 24-2. The Speaker Timer and Associated Logic



Watchdog Timer

The watchdog timer is used to detect ill-behaved programs. If a multitasking operating system, such as OS/2, were to give control to a program that disables interrupt recognition for an extended period of time, problems could result. The various devices generating interrupt requests to obtain microprocessor servicing could experience problems due to the resulting delay in servicing.

One of the interrupt requests that wouldn't be serviced is the system timer interrupt, IRQ0. The system timer, or timer 0, requires servicing every 54.9ms. The watchdog timer is designed to detect a cessation of IRQ0 servicing.

Many systems do not have watchdog timers. For those that do have timers, the address and function of the timers varies from system to system. The following discussion applies to the IBM PS/2.

Figure 24-3 illustrates the watchdog timer and its associated logic. The watchdog timer at I/O address 0044h is programmed with a count (a number). The output of timer 0 is connected to the IRQ0 latch and to the clock input of the watchdog timer. When timer 0 generates a pulse, it sets the IRQ0 latch and generates IRQ0 to the interrupt controller. When IRQ0 becomes asserted, it enables the watchdog timer. Starting with the next timer 0 output pulse, the watchdog timer begins to count down once for each pulse received on its clock input.

In the IRQ0 interrupt service routine, the programmer should set bit 7 of system control port B to a 1 to reset the IRQ0 latch, thereby deasserting IRQ0. When IRQ0 is deasserted, it disables the watchdog timer so it can't count down any more. The programmer should also reload the start count into the watchdog timer. (Note that the meaning of the bits in system control port B at I/O address 0061h are different for an I/O read versus an I/O write.)

If interrupt recognition has been disabled for an extended period of time by an ill-behaved program, IRQ0 will not be reset. The watchdog timer will therefore continue to count down towards zero because of the repeated timer 0 output pulses. If the count is exhausted, the watchdog timer's output is asserted, thereby asserting NMI to the microprocessor.

In the NMI interrupt service routine, the programmer can check bit 4 of system control port A at I/O address 0092h to see if the watchdog timer was the source of the NMI. Bit 4 will be set to 1 if the watchdog timer generated the NMI. Writing a new count to the watchdog timer will deassert NMI.

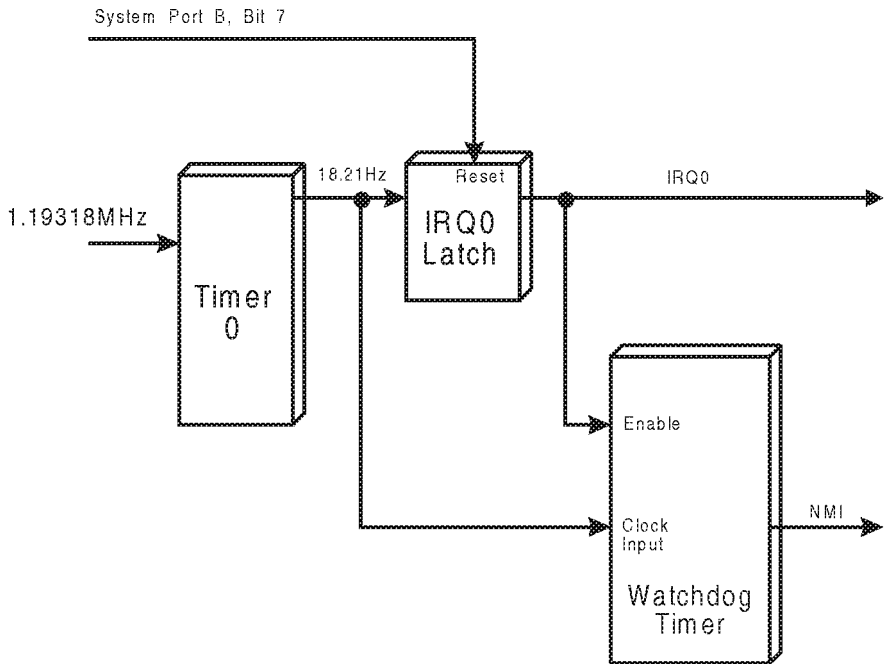


Figure 24-3. The Watchdog Timer and Associated Logic

Slowdown Timer

When faster microprocessors were introduced, several problems began to occur with programs that were timing sensitive. For example, some games ran too fast for reasonable user interaction and some copy protection schemes failed to operate properly at the faster processor speeds.

The slowdown timer is implemented in systems using microprocessors running at speeds of 16MHz and faster, to provide a method for artificially slowing them down. The term artificial is used because the microprocessors are not slowed at all, rather the refresh hold time is extended, causing the microprocessor to remain in the hold state longer than normal. Since the processor is unable to fetch and execute programs it appears to slow down.

Figure 24-4 illustrates the logic associated with the slowdown timer. The slowdown timer is triggered by the refresh request signal that goes to the refresh logic. When the refresh timer generates refresh request, two things occur:

- The refresh logic generates HOLD to seize the buses from the microprocessor. This HOLD causes the OR gate to generate HOLD to the microprocessor. The microprocessor is forced off the buses.
- The slowdown timer is triggered. When triggered, its output goes high for a pre-determined amount of time. The high on this output causes the OR gate to output a high.

The refresh logic only keeps its HOLD output asserted long enough to run one refresh bus cycle. It then deasserts its HOLD output to release the buses back to the processor so it can continue to fetch and execute instructions. The slowdown timer, however, may not have timed out yet and its HOLD output may still be high. This would delay the release of the buses back to the processor. The programmer defines the duration of the slowdown timer's output pulse by writing a count into its register at I/O address 0046h. The PC vendor's MODE command is usually used to program this timer.

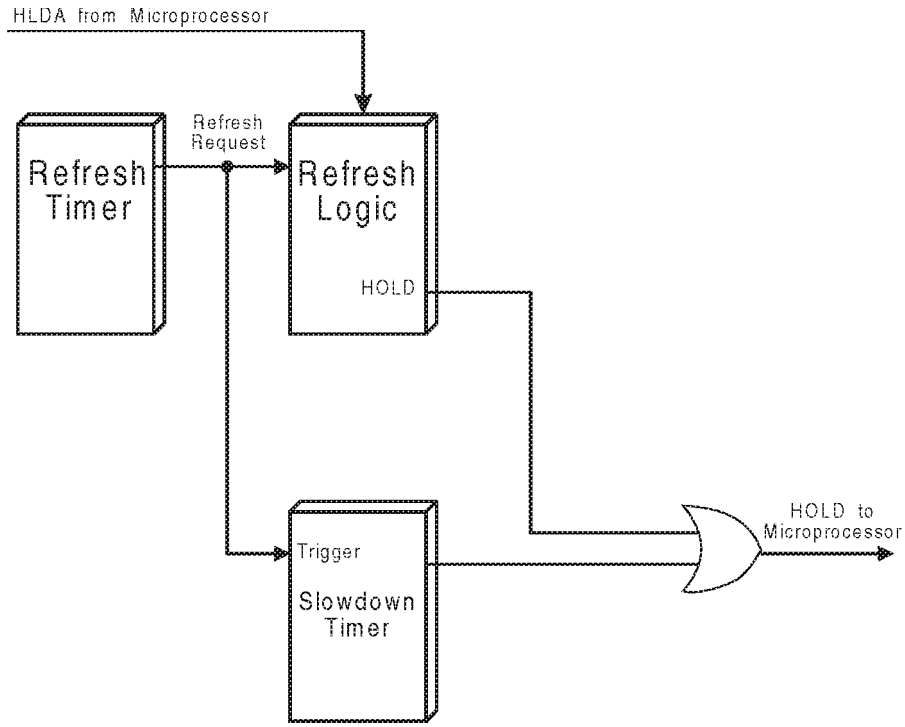


Figure 24-4. The Slowdown Timer

Appendix A

I/O Bit Assignment

ISA I/O Addresses

I/O Port	R/W	Note	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0000h	R/W	Slave DMA Channel 0 Memory Address Register	lower 16 bits of memory address written as 2 successive bytes							
0001h	R/W	Slave DMA Channel 0 Transfer Count Register	lower 16 bits of transfer count written as 2 successive bytes							
0002h	R/W	Slave DMA Channel 1 Memory Address Register	lower 16 bits of memory address written as 2 successive bytes							
0003h	R/W	Slave DMA Channel 1 Transfer Count Register	lower 16 bits of transfer count written as 2 successive bytes							
0004h	R/W	Slave DMA Channel 2 Memory Address Register	lower 16 bits of memory address written as 2 successive bytes							
0005h	R/W	Slave DMA Channel 2 Transfer Count Register	lower 16 bits of transfer count written as 2 successive bytes							
0006h	R/W	Slave DMA Channel 3 Memory Address Register	lower 16 bits of memory address written as 2 successive bytes							
0007h	R/W	Slave DMA Channel 3 Transfer Count Register	lower 16 bits of transfer count written as 2 successive bytes							
0008h	R	Slave DMA Status Register (Channels 0 to 3)	1 = Channel 3 request	1 = Channel 2 request	1 = Channel 1 request	1 = Channel 0 request	1 = Channel 3 Terminal Count	1 = Channel 2 Terminal Count	1 = Channel 1 Terminal Count	1 = Channel 0 Terminal Count
0008h	W	Slave DMA Control Register (Channels 0 to 3)	0 = DAKn sense asserted low 1 = DAKn sense asserted hi	0 = DRQn sense asserted hi 1 = DRQn sense asserted low	0 = late write selection 1 = extended write selection	0 = fixed priority 1 = rotating priority	0 = normal timing 1 = compressed timing	0 = Controller enabled 1 = Controller disabled	0 = ch 0 addr hold disabled 1 = ch 0 addr hold enabled	0 = mem-to-mem disabled 1 = mem-to-mem enabled
0009h	W	Software DRQn Request	not used					0 = Reset request bit 1 = Set request bit	00b = select Channel 0 mask bit 01b = select Channel 1 mask bit 10b = select Channel 2 mask bit 11b = select Channel 3 mask bit	
000Ah	W	Slave DMA Mask Register (Channels 0 to 3)	reserved					0 = clear mask bit 1 = set mask bit	00b = select Channel 0 01b = select Channel 1 10b = select Channel 2 11b = select Channel 3	
000Bh	W	Slave DMA Mode Register (Channels 0 to 3)	00b = demand transfer mode 01b = single transfer mode 10b = block transfer mode 11b = cascade mode		0 = Address increment 1 = Address decrement	0 = Auto init disabled 1 = Auto init enabled	00b = verify transfer 01b = write transfer 10b = read transfer 11b = reserved		00b = select Channel 0 01b = select Channel 1 10b = select Channel 2 11b = select Channel 3	

I/O Port	R/W	Note	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
000Ch	W	Slave DMA Clear Byte Pointer	Writing to this address causes the DMAC to reset the pointer that keeps track of 16-bit data transfers to and from the DMAC for hi/lo byte sequencing.							
000Dh	W	Slave DMA Master Clear	Writing to this address resets the DMAC to the same state as a hardware reset.							
000Eh	W	Slave DMA Reset Mask Register	Writing to this address resets the mask register thus activating the four DMA channels.							
000Fh	W	Slave DMA General Mask Register	not used				0 = reset ch 3 mask bit 1 = set ch 3 mask bit	0 = reset ch 2 mask bit 1 = set ch 2 mask bit	0 = reset ch 1 mask bit 1 = set ch 1 mask bit	0 = reset ch 0 mask bit 1 = set ch 0 mask bit
0020h	R	Master 8259 Interrupt Request Register	0 = IR7 not asserted 1 = IR7 asserted	0 = IR6 not asserted 1 = IR6 asserted	0 = IR5 not asserted 1 = IR5 asserted	0 = IR4 not asserted 1 = IR4 asserted	0 = IR3 not asserted 1 = IR3 asserted	0 = IR2 not asserted 1 = IR2 asserted	0 = IR1 not asserted 1 = IR1 asserted	0 = IR0 not asserted 1 = IR0 asserted
0020h	R	Master 8259 In-Service Register	0 = IR7 not currently being serviced 1 = IR7 being serviced	0 = IR6 not currently being serviced 1 = IR6 being serviced	0 = IR5 not currently being serviced 1 = IR5 being serviced	0 = IR4 not currently being serviced 1 = IR4 being serviced	0 = IR3 not currently being serviced 1 = IR3 being serviced	0 = IR2 not currently being serviced 1 = IR2 being serviced	0 = IR1 not currently being serviced 1 = IR1 being serviced	0 = IR0 not currently being serviced 1 = IR0 being serviced
0020h	W	Master 8259 ICW1	must = 000b (only used in 8080/8085)			must = 1	0 = edge sensitive mode	1 = interrupt vectors are separated by 4 locations	0 = cascade mode	1 = ICW4 is needed
0020h	W	Master 8259 OCW2	000b = rotate in Auto EOI Mode (clear) 001b = non-specific EOI 010b = no operation 011b = specific EOI 100b = rotate in Auto EOI Mode (set) 101b = rotate on non-specific EOI command 110b = set priority command 111b = rotate on specific EOI command			reserved		Interrupt request to which the command applies: 000b = IR0 001b = IR1 010b = IR2 011b = IR3 100b = IR4 101b = IR5 110b = IR6 111b = IR7		
0020h	W	Master 8259 OCW3	reserved	00b = nop 01b = nop 10b = reset special mask 11b = set special mask		reserved		0 = not poll command 1 = poll command	00b = nop 01b = nop 10b = read Interrupt Request Register on next read fm port 20h 11b = read In-Service Register on next read fm port 20h	
0021h	W	Master 8259 ICW2	upper 5 bits of Interrupt Vector (master = 00001b, slave = 01110b)					always = 000b000		

I/O Port	R/W	Note	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0021h	W	Master 8259 ICW3	0 = I/O device attached to IRQ7	0 = I/O device attached to IRQ6	0 = I/O device attached to IRQ5	0 = I/O device attached to IRQ4	0 = I/O device attached to IRQ3	1 = slave attached to IRQ2	0 = I/O device attached to IRQ1	0 = I/O device attached to IRQ0
0021h	W	Master 8259 ICW4	reserved (should be 000b000)			0 = fully-nested mode off 1 = fully-nested mode on	00b00 = non-buffered mode 01b01 = non-buffered mode 10b10 = buffered mode/slave 11b11 = buffered mode/master		0 = norm. EOI 1 = auto EOI	0 = 8080/8085 mode 1 = 8086 mode
0021h	R/W	Master 8259 Interrupt Mask Register (OCW1)	0 = enable IR7 1 = mask IR7	0 = enable IR6 1 = mask IR6	0 = enable IR5 1 = mask IR5	0 = enable IR4 1 = mask IR4	0 = enable IR3 1 = mask IR3	0 = enable IR2 1 = mask IR2	0 = enable IR1 1 = mask IR1	0 = enable IR0 1 = mask IR0
0040h	R/W	PIT #1/System Timer 0	read or write counter 0							
0041h	R/W	PIT #1/Refresh Timer	read or write counter 1 (Note: This timer should not be changed)							
0042h	R/W	PIT #1/Speaker Timer	read or write counter 2							
0043h	W	PIT #1/Mode Control Register	00b = select counter 0 01b = select counter 1 10b = select counter 2		00b = counter latch command 01b = read/write counter bits 7:0 10b = read/write counter bits 15:8 11b = read write counter bits 7:0 first followed by 15:8		000b = mode 0 (interrupt on terminal count) 001b = mode 1 (hardware retriggerable one-shot) 010 or 110b = mode 2 (rate generator) 011 or 111b = mode 3 (square wave generator) (system timer and speaker timer mode) 100b = mode 4 (software retriggerable strobe) 101b = mode 5 (hardware retriggerable strobe)			0 = 16-bit binary counter 1 = BCD counter
0048h	R/W	PIT #2/Watchdog Timer	counter 0							
0049h	R/W	PIT #2/Not used	counter 1							
004Ah	R/W	PIT #2/Slowdown Timer	counter 2							
004Bh	W	PIT #2 /Mode Control Register	00b = select counter 0 01b = select counter 1 10b = select counter 2		00b = counter latch command 01b = read/write counter bits 7:0 10b = read/write counter bits 15:8 11b = read write counter bits 7:0 first followed by 15:8		000b = mode 0 (interrupt on terminal count) 001b = mode 1 (hardware retriggerable one-shot) 010b = mode 2 (rate generator) 011b = mode 3 (square wave generator) 100b = mode 4 (software retriggerable strobe) 101b = mode 5 (hardware retriggerable strobe)			0 = 16-bit binary counter 1 = BCD counter
0060h	R/W	Keyboard/Mouse Interface Data Port	See Manufacturer's Technical Reference Guide							
0061h	R	System Control Port B/NMI Status	1 = Parity Error	1 = Channel Check	1 = Timer 2 (speaker) output is high	Toggles for each refresh	0 = Channel Check enabled	0 = Watchdog timer enabled	1 = speaker data enabled	1 = Timer 2 speaker gate enabled
0061h	W	System Control Port B	reserved				1 = Clear Channel Check	0 = Enable Watchdog timer	1 = Enable speaker data	1 = Enable Timer 2 speaker gate

I/O Port	R/W	Note	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0064h	R	Keyboard/Mouse Interface Status Port	1 = Parity Error	1 = General timeout	1 = Mouse output buffer full	0 = Keyboard password protected	0 = Keyboard data port accessed 1 = Keyboard command port accessed	System flag bit	1 = Keyboard Interface input buffer full	1 = Keyboard Interface output buffer full
0064h	W	Keyboard/Mouse Interface Command Port	See Manufacturer's Technical Reference Guide							
0070h	W	RTC/Config. RAM Address Port	0 = enable NMI	Configuration RAM address						
0071h	R/W	RTC/Config. RAM Data Port	Data to be read or written							
0081h	R/W	Channel 2 DMA Page Register	Upper 8 bits of memory address (A23:A16)							
0082h	R/W	Channel 3 DMA Page Register	Upper 8 bits of memory address (A23:A16)							
0083h	R/W	Channel 1 DMA Page Register	Upper 8 bits of memory address (A23:A16)							
0087h	R/W	Channel 0 DMA Page Register	Upper 8 bits of memory address (A23:A16)							
0089h	R/W	Channel 6 DMA Page Register	Upper 8 bits of memory address (A23:A17)							
008Ah	R/W	Channel 7 DMA Page Register	Upper 8 bits of memory address (A23:A17)							
008Bh	R/W	Channel 5 DMA Page Register	Upper 8 bits of memory address (A23:A17)							
008Fh	R/W	DMA Refresh Page Register	Must be 00h in some systems for proper operation							
0092h	R/W	System Control Port A (PS/2 Compatibility Port)	1 = First fixed disk LED on (write only bit)	1 = Second fixed disk LED on (write only bit)	reserved	1 = Watchdog Timer timeout (read-only bit)	1 = Security lock (password) engaged	reserved	1 = set Alternate Gate A20 asserted	1 = set Alternate Hot Reset Asserted
00A0h	R/W	Slave 8259 Interrupt Controller	See bit definition for Port 0020h							
00A1h	R/W	Slave 8259 Interrupt Controller	See bit definition for Port 0021h							
00C0h	R/W	Master DMA Channel 4 Memory Address Register	lower 16 bits of memory address written as 2 successive bytes							
00C2h	R/W	Master DMA Channel 4 Transfer Count Register	lower 16 bits of transfer count written as 2 successive bytes							
00C4h	R/W	Master DMA Channel 5 Memory Address Register	lower 16 bits of memory address written as 2 successive bytes							
00C6h	R/W	Master DMA Channel 5 Transfer Count Register	lower 16 bits of transfer count written as 2 successive bytes							
00C8h	R/W	Master DMA Channel 6 Memory Address Register	lower 16 bits of memory address written as 2 successive bytes							

I/O Port	R/W	Note	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
00CAh	R/W	Master DMA Channel 6 Transfer Count Register	lower 16 bits of transfer count written as 2 successive bytes							
00CCh	R/W	Master DMA Channel 7 Memory Address Register	lower 16 bits of memory address written as 2 successive bytes							
00CEh	R/W	Master DMA Channel 7 Transfer Count Register	lower 16 bits of transfer count written as 2 successive bytes							
00D0h	R	Master DMA Status Register (Channels 4 to 7)	1 = Channel 7 request	1 = Channel 6 request	1 = Channel 5 request	1 = Channel 4 request	1 = Channel 7 Terminal Count	1 = Channel 6 Terminal Count	1 = Channel 5 Terminal Count	1 = Channel 4 Terminal Count
00D0h	W	Master DMA Control Register (Channels 4 to 7)	0 = DAKn sense asserted low 1 = DAKn sense asserted hi	0 = DRQn sense asserted hi 1 = DRQn sense asserted low	0 = late write selection 1 = extended write selection	0 = fixed priority 1 = rotating priority	0 = normal timing 1 = compressed timing	0 = Controller enabled 1 = Controller disabled	0 = ch 0 addr hold disabled 1 = ch 0 addr hold enabled	0 = mem-to-mem disabled 1 = mem-to-mem enabled
00D2h	W	Software DRQn Request	reserved					0 = Reset request bit 1 = Set request bit	00b = select Channel 4 mask bit 01b = select Channel 5 mask bit 10b = select Channel 6 mask bit 11b = select Channel 7 mask bit	
00D4h	W	Master DMA Mask Register (Channels 4 to 7)	reserved					0 = clear mask bit 1 = set mask bit	00b = select Channel 4 01b = select Channel 5 10b = select Channel 6 11b = select Channel 7	
00D6h	W	Master DMA Mode Register (Channels 4 to 7)	00b = demand transfer mode 01b = single transfer mode 10b = block transfer mode 11b = cascade mode		0 = Address increment 1 = Address decrement	0 = Auto init disabled 1 = Auto init enabled	00b = verify transfer 01b = write transfer 10b = read transfer 11b = reserved		00b = select Channel 4 01b = select Channel 5 10b = select Channel 6 11b = select Channel 7	
00D8h	W	Master DMA Clear Byte Pointer	Writing to this address causes the DMAC to reset the pointer that keeps track of 16-bit data transfers to and from the DMAC for high/low byte sequencing.							
00DAh	W	Master DMA Master Clear	Writing to this address resets the DMAC to the same state as a hardware reset.							
00DCh	W	Master DMA Reset Mask Register	Writing to this address resets the mask register thus activating the four DMA channels.							
00DEh	W	Master DMA General Mask Register	not used				0 = reset ch 7 mask bit 1 = set ch 7 mask bit	0 = reset ch 6 mask bit 1 = set ch 6 mask bit	0 = reset ch 5 mask bit 1 = set ch 5 mask bit	0 = reset ch 4 mask bit 1 = set ch 4 mask bit
00F1h	W	Numeric Coprocessor Reset	All bits = 0							

I/O Port	R/W	Note	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
00F8 to 00FFh	R/W	80287 Numeric Coprocessor Ports	See Manufacturer Technical Reference Guide							
0278h	R/W	Parallel Port 3 Data Port	Data read from or written to the data port							
0279h	R	Parallel Port 3 Status Port	0 = Printer Busy	0 = Printer Acknowledge	1 = Out of paper	1 = Printer Selected	0 = printer error	0 = Printer interrupt request pending	reserved	
027Ah	R/W	Parallel Port 3 Command Port	reserved		0 = Output port (default) 1 = Input port (extended mode)	1 = enable Interrupt Request	1 = select printer	0 = initialize printer	1 = auto line feed	1 = Printer Strobe asserted
02F8h	W	(when DLAB bit = 0) Serial Port 2 Transmit Holding Register	Byte to transmit							
02F8h	R	(when DLAB bit = 0) Serial Port 2 Receive Buffer Register	Receive character							
02F8h	R/W	(when DLAB bit = 1) LSB of Serial Port 2 Divisor Latch	LSB of baud rate divisor							
02F9h	R/W	(when DLAB bit = 1) MSB of Serial Port 2 Divisor Latch	MSB of baud rate divisor							
02F9h	R/W	(when DLAB bit = 0) Serial Port 2 Interrupt Enable Register	reserved				1 = enable Modem Status interrupt	1 = enable receiver line status interrupt	1 = enable transmit holding register empty interrupt	1 = enable receive data available interrupt. In FIFO mode also enables time-out interrupt
02FAh	R	Serial Port 2 Interrupt ID Register	11b = FIFO feature present 10b = FIFO feature not present		reserved		000b = Modem Status Interrupt 001b = transmit Holding Register Empty interrupt 010b = receive Data Available interrupt 011b = receive Line Status interrupt 110b = FIFO timeout interrupt			0 = Interrupt Pending

I/O Port	R/W	Note	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
02FAh	W	Serial Port 2 FIFO Control Register	Receiver FIFO Register Trigger 00b = 1 byte 01b = 4 bytes 10b = 8 bytes 11b = 14 bytes		reserved			1 = transmit FIFO Register cleared; counter cleared bit is self-clearing	1 = Receive FIFO Register cleared; bit is self-clearing	0 = clears Receive and Transmit FIFO Registers and enters Character Mode 1 = Receive and Transmit FIFOs enabled
02FBh	R/W	Serial Port 2 Line Control Register	DLAB (Divisor Latch Access Bit) 0 = Receive Buffer; Transmit Holding; Interrupt Enable Register access enabled 1 = Divisor Latch access enabled	1 = Set Break enabled	Stick Parity	1 = Even Parity enable	1 = Parity enable	Number of Stop Bits 0 = 1 stop bit 1 = 1.5 stop bits with char length of 5 bits 1 = 2 stop bit with char length of 6/7/8 bits	00b = Character length is 5 bits 01b = Character length is 6 bits 10b = Character length is 7 bits 11b = Character length is 8 bits	
02FCh	R/W	Serial Port 2 Modem Control Register	reserved			1 = loopback mode	1 = enable OUT2 interrupt	1 = force OUT1 output asserted	1 = set Request-to-Send (RTS) asserted	1 = set Data Terminal Ready (DTR) asserted
02FDh	R	Serial Port 2 Line Status Register	reserved	1 = Transmit Shift and Holding Registers empty	1 = Transmit Holding Register empty	1 = Break interrupt	1 = Framing error	1 = Parity error	1 = Overrun error	1 = Data Ready
02FEh	R	Serial Port 2 Modem Status Register	1 = Data Carrier Detect (DCD) asserted	1 = Ring Indicator (RI) asserted	1 = Data Set Ready (DSR) asserted	1 = Clear-to-Send (CTS) asserted	1 = change detected on DCD line	1 = trailing edge of RI signal detected	1 = change detected on DSR line	1 = change detected on CTS line
02FFh	R/W	Serial Port 2 Scratch Register	can be used by the microprocessor to hold a byte. It's not used by the Serial Port.							
0378h	R/W	Parallel Port 2 Data Port	see address 0278h							
0379h	R	Parallel Port 2 Status Port	see address 0279h							
037Ah	R/W	Parallel Port 2 Command Port	see address 027Ah							
03B4h	R/W	VGA CRT Controller Address Register (mono)	see Technical Reference Guide							

I/O Port	R/W	Note	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
03B5h	R/W	VGA CRT Controller Data Register (mono)	see Technical Reference Guide							
03BAh	R	VGA Input Status Register 1	see Technical Reference Guide							
03BAh	W	VGA Feature Control Register (mono)	see Technical Reference Guide							
03BCh	R/W	Parallel Port 1 Data Port	see address 0278h							
03BDh	R	Parallel Port 1 Status Port	see address 0279h							
03BEh	R/W	Parallel Port 1 Command Port	see address 027Ah							
03C0h	R/W	VGA Attribute Controller Address Register	see Technical Reference Guide							
03C0h	W	VGA Attribute Controller Data Register (write address)	see Technical Reference Guide							
03C1h	R	VGA Attribute Controller Data Register (read address)	see Technical Reference Guide							
03C2h	W	VGA Miscellaneous Output Register	see Technical Reference Guide							
03C2h	R	VGA Input Status Register 0	see Technical Reference Guide							
03C3h	R/W	VGA Subsystem Enable Register	see Technical Reference Guide							
03C4h	R/W	VGA Sequencer Address Register	see Technical Reference Guide							
03C5h	R/W	other VGA Sequencer data register	see Technical Reference Guide							
03C6h	R/W	Video DAC Pixel Mask	see Technical Reference Guide							
03C7h	W	Video DAC Palette Address/Read Mode	see Technical Reference Guide							
03C7h	R	Video DAC State Register	see Technical Reference Guide							
03C8h	R/W	Video DAC Palette Address/Write Mode	see Technical Reference Guide							
03C9h	R/W	Video DAC Palette Data Register	see Technical Reference Guide							
03CAh	R	VGA Feature Control Register	see Technical Reference Guide							
03CCh	R	VGA Miscellaneous Output Register	see Technical Reference Guide							

I/O Port	R/W	Note	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
03CEh	R/W	VGA Graphics Controller Address Register	see Technical Reference Guide							
03CFh	R/W	VGA Graphics Controller Address Register	see Technical Reference Guide							
03D4h	R/W	VGA CRT Controller Address Register (color)	see Technical Reference Guide							
03D5h	R/W	VGA CRT Controller Data Register (color)	see Technical Reference Guide							
03DAh	R	VGA Color Input Status Register	see Technical Reference Guide							
03DAh	W	VGA Color Feature Control Register	see Technical Reference Guide							
03F0h	R	Floppy Status Register A	1 = Interrupt Pending	0 = Second drive installed	1 = Step signal asserted	0 = Track 0 sensor asserted	0 = Head 0 selected 1 = Head 1 selected	0 = Index sensor asserted	0 = Write Protect sensor asserted	state of Direction Select signal
03F1h	R	Floppy Status Register B	reserved		0 = Drive 0 Select signal asserted 1 = Drive 1 Select signal asserted	1 = Write Data signal asserted	1 = Read Data signal asserted	1 = Write Enable signal asserted	1 = Motor 1 On signal asserted	1 = Motor 0 On signal asserted
03F2h	W	Floppy Digital Output Register (DOR)	reserved		1 = set Motor 1 On signal asserted	1 = set Motor 0 On signal asserted	0 = enable interrupts	0 = reset Floppy Disk Controller	reserved	0 = set Drive 0 Select signal asserted 1 = set Drive 1 Select signal asserted
03F4h	R	Floppy Disk Controller (FDC) Status Register	1 = FDC Data Register is ready	0 = transfer is from microprocessor to FDC	1 = non-DMA mode	1 = FDC busy	reserved		1 = drive 1 is in Seek mode	1 = drive 0 is in Seek mode
03F5h	R/W	Floppy Disk Controller Data Register	see Technical Reference Guide							
03F7h	R	Floppy Digital Input Register	1 = Diskette has been changed since last access							0 = High Density selected 1 = Low Density selected
03F7h	W	Floppy Configuration Control Register	reserved						00b = 500kbs mode (high density) 01b = reserved 10b = 250kbs (low density) 11b = reserved	

I/O Port	R/W	Note	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
03F8h	W	Serial Port 1 Transmit Holding Register	see address 02F8h							
03F8h	R	Serial Port 1 Receive Buffer Register	see address 02F8h							
03F8h	R/W	LSB of Serial Port 1 Divisor Latch	see address 02F8h							
03F9h	R/W	MSB of Serial Port 1 Divisor Latch	see address 02F9h							
03F9h	R/W	Serial Port 1 Interrupt Enable Register	see address 02F9h							
03FAh	R	Serial Port 1 Interrupt ID Register	see address 02FAh							
03FAh	W	Serial Port 1 FIFO Control Register	see address 02FAh							
03FBh	R/W	Serial Port 1 Line Control Register	see address 02FBh							
03FCh	R/W	Serial Port 1 Modem Control Register	see address 02FCh							
03FDh	R	Serial Port 1 Line Status Register	see address 02FDh							
03FEh	R	Serial Port 1 Modem Status Register	see address 02FEh							
03FFh	R/W	Serial Port 1 Scratch Register	see address 02FFh							
0680h	W	Manufacturing Checkpoint Port	Checkpoint character							
3220-3227h		Serial Port 3	see 02F8-02FFh							
3228-322Fh		Serial Port 4	see 02F8-02FFh							
4220-4227h		Serial Port 5	see 02F8-02FFh							
4228-422Fh		Serial Port 6	see 02F8-02FFh							
5220-5227h		Serial Port 7	see 02F8-02FFh							
5228-522Fh		Serial Port 8	see 02F8-02FFh							
8000F8 to 8000FFh	R/W	80387SX Numeric Coprocessor Ports	See Manufacturer Technical Reference Guide							

I/O Port	R/W	Note	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
800000F8 to 800000FFh	R/W	80387DX Numeric Coprocessor Ports	See Manufacturer Technical Reference Guide							

Appendix B

Glossary

access time - The time span between a device being addressed and when it delivers or accepts valid data.

active - see asserted

address bus - The group of signal lines that carry an address from a bus master.

address decoder - Every device that can be read from or written to has an associated piece of logic called an address decoder. The address decoder selects its associated device when an address within its defined range is detected.

address latch - A device connected to the microprocessor's local address bus. This device drives the address to all devices throughout the system and latches (holds) the address for the duration of the bus cycle. Once latched, the Address Latch outputs the address to the system on the system address (SA) bus, SA19:SA1.

address pipelining - A process in which the microprocessor places the address for the next cycle on its local address bus during the current bus cycle. This can reduce access time to system memory if designed to take advantage of address pipelining.

address translation - The process of converting one type of address to another. For example: translating the address from an ISA bus master (SA16:SA0, LA23:LA17 and SBHE#) to a 32-bit memory address (A31:A2 and BE3#:BE0#) required by 32-bit memory.

AEN - An ISA bus signal or a pin on the 8237 DMA controllers. Note that the same name is used for two differently-acting signals. When the CPU asserts the HLDA signal, the ISA bus AEN signal is asserted, disabling all I/O address decoders to prevent I/O devices from mistaking the address as theirs during DMA transfers. When the DMA controller sees its HLDA input asserted, it asserts its AEN pin to enable the outputs of its address latch and page register.

ALE - The Address Latch Enable signal that controls the operation of the Address Latch.

arbiter - An arbiter is a device that evaluates the pending requests for access to the bus and grants the bus to a bus master based on a system-specific algorithm.

asserted - A signal is asserted when it is at its logic true state. A signal such as HOLD is asserted when it is high (positive logic). A signal such as MRDC# is asserted when it is low (negative logic). A signal such as W/R# is always asserted since both its high and low states represent true logic states. The opposite of asserted is deasserted. The term "asserted" is used in this book instead of "active" because "active" is ambiguous. The term "active" can mean either that a high impedance buffer's output is on (driven), or that a signal is at a logic true state. For example, a tri-state buffer may be actively driving CHRDY low (false), in which case the signal is active (by one of the ambiguous definitions) but inactive (by the other definition). In this case, CHRDY is deasserted.

BALE - An ISA bus signal that is used by expansion devices to notify them that a valid address is on the ISA bus. BALE is a buffered version of ALE on 8088- and

80286-based PC's, but it is asserted at different times than ALE on 80386 and higher PC's.

BCLK - An ISA bus signal (bus clock) that provides the timing reference for all bus transactions.

big-endian byte-ordering - A bus master using big-endian byte-ordering stores the MSB from the specified source register into the start memory address and the less-significant bytes from the register into the ascending memory locations immediately following the start address.

BIOS - Basic Input/Output System routines usually contained in system ROM and device ROMs that provide a low level interface to devices.

bit cell - A one bit storage location in a typical DRAM consisting mainly of a capacitor.

block - Same as cache line in X86 processors. The cache stores information in its cache in blocks of a fixed length. The cache block length is processor design-dependent. As an example, an 80486 cache block is 16 bytes long.

block transfer mode - A DMA transfer mode in which an entire block of data is transferred prior to the DMA Controller giving up ownership of the buses.

bridge - The device that provides the interface between two independent buses. An example is the Bus Control Logic and associated transceivers between the 80386 host processor's bus and the ISA bus.

buffered write-through - A variation of the write-through policy where a look-through cache controller stores an entire write operation in a buffer so that it can complete the memory write to main memory later.

bus concurrency - Separate transfers occurring simultaneously on two or more separate buses. An example would be an ISA bus master transferring data to or from another ISA device while the host processor is transferring data to or from system memory. Bus concurrency allows a processor to gain access to memory from its memory cache, while another bus master accesses main memory over the system bus at the same time.

bus control logic - The logic that assists the microprocessor in running bus cycles over the ISA bus.

bus master, ISA - An ISA card that can gain ownership of the ISA buses and run bus cycles.

bus snooping - Bus snooping is the method used by cache subsystems to monitor main memory accesses made by another bus master.

byte enables - 80386 and higher X86 processor address bus signals that indicate which data paths will be used during a transfer. The byte enables indicate which bytes within an addressed doubleword are being accessed.

cache - A relatively small amount of high-speed static RAM (SRAM) that is used to keep copies of information recently read from system DRAM memory. The cache controller maintains a directory that tracks the information currently resident within the cache. If the host processor should request any of the in-

formation currently resident in the cache, it will be returned to the processor quickly (due to the fast access time of the SRAM).

cache coherency - If the information resident in a cache accurately reflects the information in DRAM memory, the cache is said to be coherent or consistent.

cache consistency - see **cache coherency**

cache controller - A cache memory controller manages cache memory which stores copies of frequently accessed information read from DRAM memory.

cache directory - Memory inside of a cache controller that keeps track of information stored in cache memory. Sometimes called the tag RAM.

cache FLUSH# - A cache controller input that causes the cache controller to clear the valid bit in every directory entry to 0, thereby invalidating all cache entries.

cache line - A line is the smallest unit of data that a cache can keep track of.

cache line fill - When a processor's internal cache, or its external second level cache has a miss on a read attempt by the processor, it will read a fixed amount (referred to as a line) of information from the external cache or system DRAM memory and record it in the cache. This is referred to as a cache line fill. The size of a line of information is cache controller design dependent.

cache memory - Cache memory is a relatively small amount of high cost, fast access SRAM designed to improve access to system memory.

cache page - The size of the cache data RAM in direct-mapped caches and the size of each way in a set-associative cache. The cache views main memory as divided into pages that are the same size as the cache pages.

cache read hit - The process in which the cache memory controller sees the microprocessor initiate a memory read bus cycle, checks to determine if it has a copy of the requested information in cache memory, and if a copy is present, immediately reads the information from the cache and sends it back to the microprocessor at zero wait-states.

cache read miss - The process in which the cache memory controller sees the microprocessor initiate a memory read bus cycle, checks to determine if it has a copy of the requested information in cache memory, and find no copy is present. The cache memory controller then passes the read bus cycle on to slow main memory.

CAS# - A memory enable called Column Address Strobe used to latch the column portion of the address inside a DRAM chip.

CAS before RAS refresh - Some DRAMs incorporate their own row counters to be used for DRAM refresh. The external DRAM refresh logic has only to assert the DRAM's CAS line and then assert its RAS line. The DRAM will automatically increment its internal row counter and refresh (recharge) the next row of storage.

CBR refresh - see **CAS before RAS refresh**

channel check - This signal is asserted by an ISA expansion board to signal a catastrophic error condition to the microprocessor.

channel ready - By using the CHRDY signal (Channel Ready) on the ISA bus, designers can override the default timer and stretch out the bus cycle by the required number of wait states to match the access time of the device.

CHCHK# - see **channel check**

CHRDY - see **channel ready**

CMOS RAM - see **configuration RAM**

coherency - see **cache coherency**

concurrent bus operation - see **bus concurrency**

configuration RAM - A small amount of low power battery-backed memory used to store configuration information used during the boot-up process.

consistency - see **cache coherency**

control bus - A group of signal lines used by the microprocessor to control a variety of miscellaneous functions.

cycle time - The amount of time needed between accesses to DRAM memory, consisting of access time plus precharge delay.

DAKn# - A group of signals on the ISA bus (named DACKn# or DAKn#) used to notify DMA I/O devices that a DMA cycle is about to be performed to transfer data for them.

data bus - The group of signal lines used to transfer data between devices.

data bus contention - An address decoding problem that arises if the address decoders for two devices are both designed to detect the same address range. In such cases both devices would deliver data to the data bus simultaneously during a read operation, causing contention.

data bus steering logic - Logic used to match transfers between devices with different sized data busses. Consists of transceivers connected between the data paths.

deasserted - A signal is deasserted when it is at its logic false state. See **asserted**.

debug registers - Six programmer-accessible debug registers provide on-chip support for debugging and are present in the 80386. Debug registers DR0 through DR3 specify the four linear breakpoint addresses. The Breakpoint Control Register, DR7, is used to set the breakpoints, define them as data or code breakpoints, and whether to break on an attempted read or write. The Breakpoint Status Register, DR6, reflects the current state of the breakpoints.

demand transfer mode - The transfer mode where the DMAC runs DMA bus cycles back-to-back as long as the I/O device can continue to supply data.

device ROM - ROM on an ISA expansion device. Also known as expansion ROM. This ROM typically contains the initialization code and possibly the BIOS and interrupt service routines for its associated device. Device ROMs should reside in memory address space between C0000h and DFFFFh.

direct-mapped cache - A cache organization in which only one directory entry needs to be checked to determine whether the requested memory location is in the cache. A direct-mapped cache is the same size as a (cache) page in memory. When a line of information is fetched from a position within a page of DRAM

memory, it is stored in the same relative position within the cache. If the processor should subsequently request a line of information from the same relative position within a different page of memory, the cache controller will fetch the new line from memory and must overwrite the line currently in the cache.

dirty line - A write-back policy cache controller has cached a line of information from memory. The processor subsequently performs a memory write to a location within the cache line. There is a hit on the cache and the cache line is updated with the new information, but the cache controller will not perform a memory write bus cycle to update the line in memory. The line in the cache no longer reflects the line in memory and now has the latest information. The cache line is said to be "dirty" and the memory line is "stale." Dirty is another way of saying "modified."

dirty bit - A bit in a write-back cache directory that when set, indicates that the cache controller should ensure that the line is written back to memory to maintain coherency between cache and main memory. Also, a bit in the page table entries of 80386 and higher microprocessors used to maintain coherency between disk and main memory when page translation is enabled.

DMA acknowledge - see **DAKn#**

DMA channels - ISA has seven DMA channels. Each channel consists of a pair of ISA signal lines (DRQn and DAKn#) used by ISA DMA devices and the DMA controller to establish communication channels between the I/O device and memory.

DMA clock - The clock used by the DMA Controller to control its data transfer timing. DMA clock also called DCLK is typically one-half the speed of BCLK.

DMA controller - DMAC. The devices used to perform the DMA transfers in an ISA system. Two 8237 DMA controllers are cascaded together to provide support for seven DMA channels.

DMA page register - Each DMA channel has an external Page Register used to provide additional address capability. The slave DMAC and its associated DMA address latch produce the least-significant 16 bits of the address. The DMA page register provides the most-significant 8 bits of the address.

DMA request - The signal (DRQn) used by a DMA device to request the DMAC to move data for it.

DMAC - see **DMA controller**

fly-by transfer - The name given to ISA DMA bus cycles due to their ability to send information between an I/O device and memory in a single bus cycle.

DRAM - see **dynamic RAM memory**

DRAM bank - An organization of main system memory that supplies information to fill all data paths.

DRAM controller - The DRAM controller converts memory read or write bus cycles on the host bus to the proper sequence of actions on the memory bus to access the target DRAM location.

DRAM parity - A reliability measure used to notify the system that information read from memory is not the same as the information initially written.

DRQ0:DRQ3 - DMA request lines used by 8-bit ISA DMA devices to request that data be transferred.

DRQ5:DRQ7 - DMA request lines used by 16-bit ISA DMA devices to request that data be transferred.

dynamic RAM memory - Dynamic Random Access Memory. This type of RAM is commonly used as main system memory due to its low cost, high density, low power consumption and relatively good performance.

end-of-interrupt command - see **EOI command**

EOI command - End-of-interrupt command issued to the interrupt controller to notify it that the interrupt has been serviced. The 8259 responds by clearing a bit in its in-service register (ISR). If the EOI command is non-specific, the 8259 clears the highest-priority bit that is currently set in its ISR.

EOP (End-of-Process) - The name of the pin on the DMACs which is routed to ISA TC signal, terminal count.

EPROM - Erasable Programmable Read Only Memory. A type of ROM that can be erased when exposed to ultraviolet light and reused. Commonly used in PCs as the System or Boot ROM.

exception interrupts - A type of internally generated interrupt that generally results when the microprocessor attempts to execute an instruction and incurs an error while doing so.

exclusive access - A series of accesses to a target by one bus master while other masters are prevented from accessing the device. Also known as exclusive ownership.

exclusive state - One of the states in the MESI protocol. A cache line is said to be exclusively owned by a cache if it is in the exclusive state. This indicates two things: no other cache has a copy of the line; and the cache line is an exact copy of main memory. Having a copy of a cache line in the exclusive state does not mean that no other cache may obtain a copy from memory. It just means that, at this instant in time, no other cache has a copy of it.

expansion ROM - see **device ROM**

first-level cache - see **L1 cache**

flag register - A status and control register inside the microprocessor that performs a variety of functions. Generally used to check the status of the previously executed instruction.

floating-point processor - Also known as the numeric coprocessor used to process floating-point math operations.

FLUSH# - A input to cache memory controllers to cause all entries in the cache directory to be ignored. The effect is that the cache memory is disabled because all requests made to cache will be cache misses.

four-way set-associative cache - A cache organization that provides four cache memory banks (ways) in which information can be stored.

fully-associative cache - A cache organization that allows information from anywhere in memory to be stored in any location within cache memory.

ghost interrupt - see **phantom interrupt**

halt - A type of microprocessor bus cycle initiated by instructions that cause the microprocessor to stop.

hidden refresh - If a DRAM controller uses the memory bus to perform DRAM refresh when the system is currently accessing a device other than DRAM memory, this is referred to as hidden refresh. Refreshing DRAM in this fashion doesn't impact system performance.

hi/lo byte copier - The transceiver that connects SD15:SD8 to SD7:SD0. Part of the data bus steering logic.

hit - Refers to a hit on the cache. The processor or a bus master initiates a memory read or write and the cache controller has a copy of the target line in its cache.

HLDA - Hold acknowledge. A microprocessor output that notifies the requesting bus master that the microprocessor has given up ownership of the buses.

HOLD - Hold request. A microprocessor input that is used by bus masters to gain ownership of the buses.

I/O ports - I/O address locations

I/O Size 16 - see **IO16#**

IN instruction - An I/O read instruction

in-line code fetch - Sequential fetching of instructions from memory by the instruction prefetcher.

in-service register - A register in the 8259 interrupt controller which tracks which interrupt service routines are currently running. Also known as ISR (one of two meanings). The respective bit in the ISR is set when a valid interrupt is acknowledged by the microprocessor. The highest-priority bit that is currently set in the ISR is cleared when the 8259 receives a non-specific end-of-interrupt command.

inactive - Either means that a high impedance buffer is off or that a signal is at its logic false state. See **asserted.**

INT instruction - see **interrupt instruction**

INTA - see **interrupt acknowledge**

interleaved memory - A DRAM memory architecture in which multiple memory banks are structured such that each can supply the maximum amount of data the processor can request in one access. In other words, the width of each DRAM bank matches the width of the processor's data bus. As an example, the 80386's data bus is four bytes (one doubleword) wide. Two-way interleaved memory is structured so that each successive doubleword is in the opposite bank of memory. During a cache line fill operation, the processor reads a doubleword from one bank, the second doubleword from the opposite bank, the third from the original bank and the fourth from the other bank. When a DRAM bank is accessed for a read, the DRAM cells are discharged during the read and need to charge back up before the bank can be successfully accessed again. If an interleaved memory architecture is used, the charge up (referred to as the precharge delay) of each DRAM bank is hidden behind the processor's access to the opposite bank. This results in better performance.

interrupt acknowledge - The host X86 processor responds to an interrupt request on its INTR input by generating two, back-to-back interrupt acknowledge bus cycles. The Bus Control Logic translates the two cycles into pulses on the INTA# pin which goes to the 8259 interrupt controllers. In response to the second interrupt acknowledge pulse, the interrupt controller sends the interrupt entry number corresponding to the highest priority device generating the request back to the processor.

interrupt cascade bus - A three signal bus used to notify the slave interrupt controller that this interrupt acknowledge cycles belong to it.

interrupt controller - Intel 8259 Programmable Interrupt controller (PIC) used in ISA systems.

interrupt instruction - An instruction that calls an interrupt service routine referenced by an interrupt table entry number.

interrupt request lines - ISA signal lines named IRQ. The ISA bus has eleven IRQ lines available and are used by devices to request servicing by programs designed for them, called interrupt service routines.

interrupt service routine - A program written to service hardware devices that generate interrupts. Each interrupting device typically has its own unique interrupt service routine.

interrupt table - A data structure in the low range of memory (real mode) that contains pointers to the start address of interrupt service routines.

interrupt type code - The code sent by the interrupt controller to the microprocessor to identify which entry in the interrupt controller to call.

interrupt vector - Another name for an interrupt table entry which points to the starting address of an interrupt service routine.

INTR - Microprocessor Interrupt Request. The microprocessor's input that notifies it that an interrupt-driven device needs servicing.

invalid state - One of the states in the MESI protocol. State of a cache line that does not currently contain a valid copy of a line from main memory.

IO16# - An ISA signal used to indicate to the system board when a 16-bit I/O device is being addressed.

IORC# - I/O Read Command. The ISA signal that specifies that an I/O read bus cycle is being run.

IOWC# - I/O Write Command. The ISA signal that specifies that a I/O write bus cycle is being run.

IRET instruction - Interrupt Return instruction. The last instruction in an interrupt service routine that tells the microprocessor that the interrupt service routine has completed and it is time to return to normal program flow.

ISA address bus - The buses used to transfer the address over the ISA bus. SA19:SA0, LA23:LA17 and SBHE#.

ISR - see **interrupt service routine** and **in-service register**

jump instruction - An instruction that causes the microprocessor to change the address location from which the processor in performing in-line code fetching. The new location is specified in the jump instruction.

kerchunk - Sound the microprocessor makes when it latches data from the data bus.

keyboard command port - The I/O location (64h) that commands are sent to for the keyboard.

keyboard data port - The I/O location (60h) that data is transferred to and from for the keyboard.

keyboard IRQ - The keyboard's interrupt request line, IRQ1.

keyboard scan code - The code sent from the keyboard to the keyboard controller that represents keystrokes that have been entered.

keyboard interface - Usually an 8042 series microcontroller used as an interface between the system and keyboard.

L1 cache - Also known as **level-one cache** or **first-level cache**. The first layer of cache memory that the microprocessor's prefetcher and/or execution unit is connected to. The internal cache of the 80486 or the external cache that connects directly to an 80386 microprocessor are examples of first-level caches.

L2 cache - Also known as **level-two cache** or **second-level cache**. A secondary cache that is several times larger than the L1 cache. Used primarily in systems that have a small L1 cache, such as the 8KB internal cache in the 80486.

LA bus - see **latchable address bus**

latchable address bus - Pipelined address bus consisting of LA23:LA17 on the ISA bus that connects to 16-bit memory devices.

level-one cache - see **L1 cache**

level-two cache - see **L2 cache**

line - see **cache line**

linear address - The 32-bit address produced by the X86 processor's segmentation unit. The address submitted to the paging unit if page translation is enabled, or

submitted to the bus unit if page translation is disabled. The linear address is the same as the physical address on an 80286 since it lacks a paging unit.

little-endian byte-ordering - A bus master using little-endian byte-ordering stores the LSB from the specified source register into the start memory address and the more-significant bytes from the register into the ascending memory locations immediately following the start address.

local data bus - Used typically to reference the processor's data bus.

logical address - The 16- or 32-bit address submitted by the X86 processor's execution unit to the segmentation unit. The offset portion of SEGMENT:OFFSET addresses.

look-aside cache - A cache memory design in which the cache controller sets in parallel with main system memory. When memory is addressed both the cache controller and system memory are addressed. Compare Look-through Cache.

look-through cache - A cache memory design in which memory request go first to the cache controller which looks through its cache memory to determine if this request is a hit or miss. If the information is not in cache then the bus cycle is broadcast on to main memory.

LRU algorithm - Least Recently Used Algorithm. Used in cache memory designs to determine which cache entry is to be overwritten by newer data. Conforms to the principle of temporal locality.

M/IO# - Memory or I/O. The signal output from the microprocessor that informs external logic whether the address on the bus is for memory or I/O devices.

M16# - Memory size 16 line. This ISA signal is asserted by 16-bit memory devices to inform the system they are being addressed.

machine status word - A control and Status register inside the microprocessor to control a variety of functions including the processor's mode of operation (real or protected).

MASTER16# - An ISA signal asserted by ISA bus masters to enable access to I/O devices. The ISA AEN signal prevents I/O devices from responding to the memory addresses which are on the address bus during DMA transfers. MASTER16# overrides AEN and allows I/O address decoders to resume decoding the I/O addresses which ISA bus masters can generate.

memory-based semaphore - see **semaphore**

memory-mapped I/O port - An I/O register that responds to memory rather than I/O accesses.

MESI protocol - A protocol for maintaining cache coherency. At a given instant in time, each line in a write-back policy cache may be one of the following states: modified (M), exclusive (E), shared, (S), or invalid (I). Each of these states is defined in this glossary.

miss - Refers to a miss on the cache. The processor or a bus master initiates a memory read or write and the cache controller does not have a copy of the target line in its cache.

modified state - One of the states in the MESI protocol. When a cache line is in the modified state, it has been modified by the processor since it was copied from memory.

MOV instruction - An assembly instruction used to transfer information to and from memory.

MRDC# - Memory Read Command. The ISA signal that specifies that a memory read bus cycle is in process.

MSW register - see **machine status word**

MWTC# - Memory Write Command. The ISA signal that specifies that a memory write.

NCA# - Non-Cacheable Address. Specifies which address locations should not be cached in main memory.

NMI - Non-Maskable Interrupt. Typically used to report serious system problems.

non-cacheable memory - Memory whose contents can be altered by a local processor without using the bus should be designated as non-cacheable. A cache somewhere else in the system would have no visibility to the change and could end up with stale data and not realize that it no longer accurately reflects the contents of memory.

no wait state - The fastest bus cycle that a microprocessor can run consisting of two complete cycles of the processor clock.

NOWS# - No Wait State. An ISA signal that permits designers to end the bus cycle earlier than the default timing. Asserting NOWS# before the default timing expires causes the bus cycle to be shortened.

numeric coprocessor - A processor that handles floating point instructions.

OUT instruction - An I/O write instruction.

page - The most over-worked term in the computer industry. It is used to represent at least four independent, unrelated items: virtual memory page, page mode DRAM, cache page and DMA page register, each of which is defined in this glossary.

page granular segment - Also known as a large grain segment. A memory segment for which the segment descriptor has the G bit set to 1, indicating that the size of the segment is limited to multiples of 4KB, from 4KB to 4GB. As opposed to byte granular segments (small grain segments) for which the size is multiples of one byte, from 1 byte to 1MB. 4KB is also the size of the virtual memory pages, thus the reference to "page." This has no relationship to cache pages, page mode DRAM or DMA page registers.

page mode DRAM - A variation of DRAM that provides improved performance when accesses occur along the same row (sequentially). All accesses occurring on the same row can be done in approximately half the time because the RAS timing can be eliminated, since the row does not change.

page register - The upper portion of the start memory address for a DMA transfer is written to the respective DMA channel's page register. The contents of this reg-

ister are then driven onto the upper part of the address bus during a DMA transfer.

parity - see **DRAM parity**

phantom interrupt - An erroneous interrupt triggered at the input of the interrupt controller, usually caused by a noise spike.

physical address - The address output onto the address bus by the processor, as contrasted to logical, linear or virtual addresses which may not be equal to the physical address.

PIC - see **interrupt controller**

policy of inclusion - When a processor is connected (via its local bus) to a look-through L2 cache, all information that is read into the L1 cache has to pass through the L2 cache. The L2 cache therefore may have a complete copy of the information that is contained in the L1 cache at a given instant in time. If the L2 cache always maintains a copy of information that is in the L1 cache, then it is enforcing the policy of inclusion. The L2 cache can then act as a snoop filter for the L1 cache. Whenever another bus master is accessing main memory, the L2 cache snoops the address to determine if it has a copy of the cache line. If it doesn't, then it needn't force the L1 cache to snoop the access. The L2 cache only has to alert the L1 cache about another bus master's memory access when it has a hit on the L2 cache.

POST - see **power-on self-test**

posted-write capability - The ability of a device to "memorize" or post a write transaction and signal immediate completion to the bus master. As long as there is room in the posted-write buffer, the bus master can complete writes with no wait states. After posting the write and signaling completion to the bus master, the device will then perform the actual memory write.

power-on restart address - The first address that the microprocessor places on the address bus after being powered up.

power-on self-test - POST. The instructions performed when the system is first turned on. Usually refers to the entire power-up sequence until the operating system is loaded.

POWERGOOD - The signal that is asserted by the power supply after all voltages have stabilized. When POWERGOOD is asserted, RESET is deasserted to the microprocessor and the microprocessor begins to run.

principles of locality - Principles used to explain the nature of code and data access from memory, which provides cache memory systems their performance gains.

PROM - Programmable Read Only Memory

RAS# - Row Address Strobe. Used to latch the row portion of the address into DRAM chips.

RAS/CAS delay - The time interval required by DRAM between the start of RAS# and CAS# signals.

real mode - The microprocessor mode of operation when it is first powered up. Real mode is the mode of operation that is available with the 8086 and 8088 microprocessors. It is designed for single tasking and has address capability of 1MB

real-time clock chip - The battery-backed device used to keep track of the date, time and configuration information.

refresh - The process of keeping dynamic memory from loosing information from the bit cell due to capacitor discharge. All DRAM throughout the system is refreshed approximately every fifteen microseconds.

refresh logic - The logic that runs refresh bus cycles. The refresh logic is a bus master capable of gaining ownership of the buses on a regular basis.

refresh request - The output of the refresh timer used to trigger the refresh logic.

refresh timer - The timer used to trigger the refresh logic to run bus cycles every 15 microseconds.

ROM scan - The process of scanning device ROMs during the power-up sequence to identify and initialize expansion devices.

RTC chip - see **real-time clock chip**

SA bus - see **system address bus**

SBHE# - System Bus High Enable. An ISA address signal that is asserted when an odd-addressed location is accessed.

SCRAM - Static Column RAM. A variety of DRAM that works on the same principles as page mode RAM. The difference between page mode and static column RAM lies only in the way SCRAM recognized that a new column address has been presented to it.

second-level cache - see **L2 cache**

segment wrap-around - The method used to generate addresses in real mode in an 8086 or 8088 make it possible to form an address that goes beyond 1 MB internally. However, since these processors have only 20 physical address lines, addresses formed above 1MB effectively wrap around to the very bottom of the address space, to address zero.

semaphore - A special memory operand used by a task to determine if a resource shared with other processors or other tasks running on the same processor is currently available for access.

shadow RAM - RAM memory that has a copy of the contents of ROM. Information from ROM is copied into faster main memory during the power-up sequence to improve performance.

shared state - One of the states in the MESI protocol. Indicates that copies of the cache line may exist in other caches in the system and none of the caches with copies of the line have altered their copy since reading it from main memory.

shutdown - A microprocessor bus cycle type that is run when the processor encounters errors that it cannot handle. Similar to the halt bus cycle the processor stops after a shutdown cycle.

single transfer mode - The DMA transfer mode where the DMAC releases control of the buses after transferring each byte of data.

slowdown timer - An ISA timer used to slow the average execution time of fast processors, so that code appears to run at the same speed as it does on an 8MHz 80286 processor.

SMRDC# - The ISA signal that specifies that a memory read bus cycle is in process to a location within the first 1MB of address space.

SMWTC# - The ISA signal that specifies that a memory write bus cycle is in process to a location within the first 1MB of address space.

snarfing - A cache memory implementation where cache memory is updated along with main memory when the cache detects a snoop hit during a bus master write to main memory.

snoop hit - A match between an entry in the cache directory and the address being driven on the address bus by a bus master other than the microprocessor. Indicates that the bus master is accessing a main memory location that is also in the cache.

snooping - A technique used by cache memory controllers to monitor the system buses to detect an access to main memory that might cause a cache consistency problem.

spatial locality - One of the principles of locality that explain the nature of how code typically resides in main memory. Spatial locality suggests that information to be read from memory nest is likely to be adjacent or reside in a memory location that is close to the location previously accessed.

speaker data timer - An ISA timer that can be programmed to provide the speaker with a wide variety of audible frequencies.

SRAM - see **static RAM**

stack - An area of memory used as a temporary storage location for applications and the processor.

stale data - In a cache memory system, data in cache or in main memory that has not been updated to reflect a change in the other copy.

static RAM - Memory typically used in cache designs due to its fast speed, high cost, and greater power consumption than DRAM.

system address bus - SA bus. The ISA address bus consisting of SA19:SA0 that connects to 8- and 16-bit devices.

system control port A - An I/O location (92h) that contains status and control information. This port is typically known as the PS/2 compatibility port.

system control port B - An I/O location (61h) standard in all ISA machine that provide status and control information.

system data bus - The ISA data bus, consisting of SD7:SD0 (8- and 16-bit devices) and SD15:SD8 (16-bit devices).

system timer - Also called Timer 0, this timer connects to the master interrupt controller to provide system interrupts every 54.9ms.

tag RAM - see **cache directory**

TC - see **terminal count**

temporal locality - One of the principles of locality that explain the nature of how code typically resides in main memory. Temporal locality suggests that information recently read from memory is likely to soon be needed again by the microprocessor.

terminal count - TC. Also known as **transfer complete**. A signal on the ISA bus issued by the DMAC to notify a DMA device that the transfer has completed.

timer 0 - see **system timer**

timer 1 - see **refresh timer**

timer 2 - see **speaker data timer**

transfer complete - see **terminal count**

two-way set-associative cache - A cache organization that provides two cache memory banks (ways) in which information can be stored.

virtual address - An address that never appears on any bus in the system, but instead resides in the imagination of the system software engineer, and which is formed by a combination of the X86 protected-mode segment descriptors and logical addresses. Since there are up to 8K global descriptors and up to 8K local descriptors, and since each descriptor can describe 4GB of linear address space, the 80386 and higher processors can manage (8K + 8K) x 4GB = 64TB of memory. The virtual address can be thought of as consisting of the base address from the segment descriptor plus the logical address. Alternatively, it may be thought of as the segment selector (which is just an index), and the logical address (but not added together).

virtual memory page - For X86 processors, a 4KB range of information starting on a 4KB boundary. The information on mass storage devices is divided into pages by the operating system, as is the array of physical memory in the system. Used for virtual memory management using demand paging.

wait state - A delay of one bus clock period injected into a transaction because the currently-addressed device is not yet ready to complete the data transfer. When an X86 processor samples its READY# input deasserted at the end of data time, it inserts one wait state in the transaction and samples READY# again at the end of the wait state.

watchdog timer - A timer used in some ISA machines to detect ill-behaved programs in multitasking environments.

Weitek - A type of numeric coprocessor that provides superior performance over the standard Intel coprocessors.

write-back cache - A type of write policy in which the cache controller only updates main system memory when necessary.

write miss - The processor initiates a memory write and the cache controller does not have a copy of the target memory location within its cache.

write policy - The method used to maintain cache consistency on memory write operations.

write-through cache - A type of write policy in which the cache controller updates main system memory immediately.

X-bus - The X-bus is located on the system board and is a buffered version of the standard ISA expansion bus. Devices such as the keyboard controller, CMOS RAM, and floppy controller typically reside on the X-bus.

ISA I/O Decode

ISA I/O cards reside in the I/O address range from 0100h through 03FFh. Many of these cards perform an inadequate I/O address decode. The card's I/O address decoder typically inspects A[9:5] to determine which of the 24, 32 location blocks of I/O space is currently being addressed. If the currently addressed I/O location is within the block of 32 locations assigned to the I/O expansion card, the card's logic examines address bits A[4:0] to determine if one of up to 32 I/O ports on the addressed expansion card is being addressed. Address bits A[15:10] are ignored by the I/O decoders. The I/O address range assigned for usage by expansion cards is 0100h through 03FFh. Within the group of address bits used by expansion I/O address decoders, A[9:5], bits A9 and A8 would therefore be either 01b (0100h through 01FFh range), 10b (0200h through 02FFh range), or 11b (0300h through 03FFh range). When the microprocessor places any address within these I/O address ranges on the address bus, an I/O card may respond.

As an example, assume that a machine has two IO cards installed. One of them performs an inadequate address decode using A[9:5] and has eight registers residing at I/O ports 0100h through 0107h. The other card performs a full decode using A[15:5] and has four registers residing at I/O ports 0500h through 0503h. Now assume that the microprocessor initiates a one byte I/O read from I/O port 0500h. The address placed on the bus is shown in the table that follows.

Example I/O Address

A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0

The board that occupies the 0500h through 0503h range looks at A[15:5] and determines that the address is within the 0500h through 051Fh block. It then looks at A[1:0] and determines that location 0500h is being addressed. Since this is an I/O read bus cycle, the card places the contents of location 0500h on the lower data path (this is an even address).

At the same time, the board that occupies the 0100h through 0107h range looks at A[9:5], a subset of the address seen by the other card's address decoder, and determines that the address appears to be within the 0100h through 011Fh

block. It then looks at A[2:0] and determines that location 0100h is being addressed. Since this is an I/O read bus cycle, the card places the contents of location 0100h on the lower data path (this is an even address).

Since both cards are driving a byte of data onto the lower data path, SD[7:0], data bus contention is occurring. This results in garbage data and possible hardware damage because two separate current sources are driving the lower data path. The problem occurs because the card residing in the 0100h through 0107h range looks at A[9:8] and thinks that this address is within the 0100h through 01FFh range. If the card were designed to perform a full address decode using A[15:5], the problem could have been avoided.

Addresses above 03FFh may be used as long as A[9:8] are always 00b, thus ensuring that the address does not appear to be in the 0100h through 01FFh, 0200h through 02FFh, or 0300h through 03FF ranges. The table that follows illustrates the usability or unusability of address ranges above 03FFh.

Usable and Unusable I/O Address Ranges Above 03FFh

I/O Address Range	Usable or Unusable
x000h - x0FFh	usable
x100h - x1FFh	Unusable. Appears to be 0100h - 01FFh
x200h - x2FFh	Unusable. Appears to be 0200h - 02FFh
x300h - x3FFh	Unusable. Appears to be 0300h - 03FFh
x400h - x4FFh	usable
x500h - x5FFh	Unusable. Appears to be 0100h - 01FFh
x600h - x6FFh	Unusable. Appears to be 0200h - 02FFh
x700h - x7FFh	Unusable. Appears to be 0300h - 03FFh
x800h - x8FFh	usable
x900h - x9FFh	Unusable. Appears to be 0100h - 01FFh
xA00h - xAFFh	Unusable. Appears to be 0200h - 02FFh
xB00h - xBFFh	Unusable. Appears to be 0300h - 03FFh
xC00h - xCFFh	usable
xD00h - xDFFh	Unusable. Appears to be 0100h - 01FFh
xE00h - xEFFh	Unusable. Appears to be 0200h - 02FFh
xF00h - xFFFh	Unusable. Appears to be 0300h - 03FFh

Note: where x = any hex digit.

1

1.19318MHz clock, 469, 471
14.31818MHz clock, 350, 469
16-bit read from 16-bit device, 134
16-bit read from 8-bit device, 133
16-bit write to 16-bit device, 134
16-bit ISA bus master, 345
16-bit write to 8-bit device, 132

3

32KHz crystal oscillator, 441

8

80286 and 80386 bus cycle state names, 140
80286 bus cycle definition, 140
80286 microprocessor address space, 34
80287, 459
80386DX outputs while RESET is asserted, 115
80386SX microprocessor address space, 34
80387, 459
8042 local I/O ports, 450
8042 microcomputer, 449
8042 sample keyboard/mouse interface command list, 455
8042 test port, 452
8080 microprocessor, 29
8086, 33
8087 numeric coprocessor, 459
8088, 33
8237 DMA controller, 410
8259 programmable interrupt controller, 366
8-bit read from 16-bit device, 134
8-bit write to 16-bit device, 134
8-bit write to odd addressed location in 8-bit device, 131
8-Bit read from odd addressed location in 8-bit device, 129

A

A2 output, 465
A20 GATE, 79, 450
A20 MASK#, 79
access time, 134
address pipelining, 123
address bus, 17
address decoder, 18, 39
address enable, 345
address latch, 121
address translation, 438
AEN, 345, 424
alarm output of real-time clock and configuration RAM chip, 386
ALE (address latch enable), 121
alternate gate A20, 400
alternate hot reset, 400
atomic access, 94

B

BALE, 342
basic input/output system (BIOS) routines, 405
BASIC language, 322
battery, 441
buffered write-through cache, 293
bus control logic, 119
bus snooping, 295
BCLK, 341
BHE#, 83
big-endian byte ordering rule, 74
BIOS, 322
BIOS ROM, 42
BIOS routine, 43, 404
bit cell, 244, 315, 318
BLE#, 188
boot program, 322
boot ROM, 42, 116
BOUND range exceeded exception, 403
breakpoint, 403
buffered address latch enable, 342
bus clock, 341

bus concurrency, 278
bus high enable (BHE#), 83
bus low enable (BLE#), 188
bus master capability, ISA, 433
BUSY#, 461, 464

C

cache coherency, 292
cache FLUSH#, 313
cache handling of non-cacheable memory, 312
cache line, 283
cache memory, 279
cache memory controller, 274
cache read hit, 274
cache read miss, 275
cache ways, 305
cache, advantage of, 276
CAS# (column address strobe), 236, 242
CAS-before-RAS refresh, 248
CBR refresh, 248
cache and testing memory, 313
cache consistency, 279
cache directory, 281
cache handling of I/O information, 311
cache line size, 308
cache management logic, 281
cache performance, 278
cache size, 310
column address decoder, 237
column address latch, 236
column address strobe (CAS#), 242
channel or I/O check, 349, 400
channel ready, 342
CHCHK#, 349
checksum error, 331
checksumming, 322
CHRDY, 137, 342
clear interrupt enable instruction, 368
CLI, 368
CMD0, 465
CMD0#, 465
CMD1, 465
CMOS address decoder, 442
CMOS address port, 442
CMOS century part of time and date function, 446
CMOS checksum value, 446
CMOS data port, 442
CMOS diagnostic status byte, 446
CMOS diskette drive type byte, 446
CMOS equipment installed byte, 446
CMOS fixed disk drive type byte, 446
CMOS indicator for memory above 1MB, 446
CMOS low and high base memory bytes, 446
CMOS RAM memory, 441
CMOS RAM valid, 445
CMOS reset code byte, 446, 447
CMOS system information byte, 446
CMOS, accessing, 442
code cache, 291
column address, 236, 238, 239, 242
combined cache, 290
command outputs, 145
command port, 36
configuration RAM, 460
configuration RAM usage, 446
control bus, 19
control/alternate/delete soft reset, 111, 447
coprocessor chip-select, 465
coprocessor emulation, 460, 465
coprocessor error, 403
coprocessor error conditions, clearing, 463
coprocessor RESET, 463
coprocessor status register, 463
coprocessor's data register, 460
coprocessor's opcode, or instruction, register, 460
coprocessor, systems using Intel and Weitek, 468
coprocessor/processor communication signals, 464
copy protection scheme, 474

D

DAK0#:DAK3#, 345
DAK2#, 412
DAK4#, 417

DAK5#:DAK7#, 345
Dallas Semiconductor real-time clock and configuration RAM chip, 441
data bus, 19
data bus contention (address conflicts), 41
data bus transceivers, 124
data cache, 291
DATA ENABLE, 126
data port, 37
date and time, 441, 470
data bus steering logic, 126
default ready timer, 135
direct-mapped cache, 301
dynamic RAM (DRAM) memory, 235
dedicated cache, 290
descriptor, 101, 162
descriptor privilege level (DPL), 105
device ROMs, 330
divide-by-zero condition, 402
DMA acknowledge, 412
DMA acknowledge for channels 0 through 3 and 5 through 7, 345
DMA address latch, 424, 431
DMA ADSTB signal, 424
DMA AEN signal, 424
DMA block transfer mode, 416
DMA bus cycle, 420
DMA byte or word transfers, 423
DMA channels, 410
DMA clock, 420
DMA compressed timing, 421
DMA controller (DMAC), 409, 414
DMA controllers cascaded, 419
DMA demand transfer mode, 416
DMA direction of transfer with reference to memory, 410
DMA extended write option, 422
DMA master and slave controllers, 417
DMA memory address logic, 425, 426
DMA page register, 423, 424
DMA read transfer, 414
DMA request, 412
DMA request for channels 0 through 3 and 5 through 7, 345
DMA single transfer mode, 415
DMA start memory address, 410
DMA transfer count, 410
DMA transfer count exhausted, 413
DMA transfer modes, 414
DMA transfer rate, 431
DMA transfer types, 414
DMA verify transfer, 414
DMA write transfer, 414
DMA, EOP (end-of-process), 413
DMA, fly-by transfer, 409
DMA, TC (terminal count reached), 413
DMAC, 409
DMAC control register, 431
DMAC initialization during POST, 431
DMAC mode register, 431
DMAC states, 420
don't care bits, 46
double exception detected, 403
DPL, 105
DRAM, 235
DRAM addressing sequence, 236, 242
DRAM access time, 250
DRAM address decoder, 240
DRAM addressing logic, 238
DRAM addressing multiplexer, or MUX, 240
DRAM bank, 251
DRAM bank width, 253
DRAM memory chip block diagram, 237
DRAM parity, 256
DRAM refresh logic, 471
DRAM refresh, 244
DRAM, burst mode, 265
DRAM, cycle time, 250
DRAM, destructive read, 250
DRAM, enhanced page-mode, 264
DRAM, interleaved memory architecture, 269
DRAM, nibble mode, 265
DRAM, page-mode, 259
DRAM, pre-charge delay, 250
DRAM, static column RAM (SCRAM), 267
DRQ0:DRQ3, 345
DRQ2, 412
DRQ4, 417
DRQ5:DRQ7, 345
DT/R#, data transmit or receive, 126

E

ECC memory, 259
EEPROM, 319
EM bit, 460, 466
ENABLE LOWER, 126
ENABLE UPPER, 126
EOI (end-of-interrupt) command, 379
EOI (end-of-interrupt) command, for slave, 388
EOP (end-of-process), 413
EPROM, 318
ERROR# output of the numeric coprocessor, 386, 463, 464
error-checking-and-correcting memory, 259
exception 7 interrupt request, 466
exception 7 interrupt service routine, 466
exception interrupts, 402
extended memory block-move return, 447

F

first level cache, 289
four-way set-associative cache, 307
fully associative cache, 299
fixed disk activity LED, 400
flag register, 374
floating gate, 318
floating-point, 459
floating-point calculations, 466
floating-point processor, 459
floppy disk controller, 387
FLUSH#, 313
FORCE A20, 79, 451

G

GDT, 101
GDTR, 102
general protection software exception, 403
global descriptor table (GDT), 101
global descriptor table register (GDTR), 102

H

halt, 150, 230
halt or shutdown bus cycle, 150
hard disk controllers, 387
Hi/Lo byte copier, 130
hidden refresh, 248
high memory area, 79, 327
HLDA, 412
HMA, 79, 327
HOLD, 412, 475
hold time, 149
hot reset, 450

I

I/O address decoder, system board, 47
I/O address range assigned to system board devices, 52
I/O bus, 335
I/O check, 349
I/O ports, 30, 36
I/O read command, 340
I/O size 16, 344
I/O write command, 340
ICW1, 397
ICW2, 371, 397
ICW3, 397
ICW4, 397
IDT, 101
IDTR, 102
interleaved memory architecture, 269
IMR, 397
IN instruction, 32
in-line code fetch, 14
in-service register, 371
instruction cache, 291
INT instruction, 403
INTA, 369
integer math, 465
interrupt acknowledge, 369, 374
interrupt cascade bus, 387
interrupt controller, 366
interrupt controller cascading, 383
interrupt controller, slave, 387

interrupt descriptor table (IDT), 101
interrupt descriptor table register (IDTR), 102
interrupt ID, 371
interrupt instruction, 403
interrupt mask register, 397
interrupt on overflow, 403
interrupt priority scheme, 368, 384
interrupt request, 365
interrupt request (IRQ) lines, 347, 366
interrupt request level 0, 384
interrupt request level 7, 384
interrupt request register (IRR), 371, 399
interrupt return instruction, 379
interrupt service routines, 43, 322, 366, 369
interrupt servicing flowchart, 367
interrupt servicing, clearing the interrupt enable flag, 378
interrupt servicing, jumping to the ISR, 378
interrupt servicing, saving instruction pointer, 374
interrupt table, 369
interrupt table entry assignments, 388
interrupt table entry number, 371
interrupt table initialization, 371
interrupt type code, 371
interrupt vector, 371
interrupts, dedicated IRQ lines, 386
interrupts, edge-triggered mode, 397
interrupts, end-of-interrupt, or EOI, command, 397, 398
interrupts, exception, 402
interrupts, initialization command words, or ICWs, 396
interrupts, initialization sequence for the master interrupt controller, 397
interrupts, initialization sequence for the slave interrupt controller, 398
interrupts, in-service register, 398
interrupts, master's interrupt ID, 397
interrupts, operation command words, or OCWs, 396
interrupts, programming the 8259, 396
interrupts, software interrupts, 402
interrupts, special fully-nested mode, 397
interrupts, special mask mode, 399
INTR, 368
invalid opcode exception, 403
invalid task state segment, 403
IO16#, 128
IORC#, 340
IOWC#, 340
IRET (interrupt return) instruction, 379, 404
IRQ lines, 347
IRQ0, 386, 472
IRQ0 frequency, 470
IRQ0 interrupt service routine, 470
IRQ0 latch, 469, 473
IRQ0 through IRQ7, 386
IRQ0, resetting, 470
IRQ1, 372, 386, 449
IRQ10, 387
IRQ11, 387
IRQ12, 387, 450
IRQ13, 386, 463, 464
IRQ13 interrupt service routine, 463
IRQ14, 387
IRQ15, 387
IRQ2, 383, 386, 387
IRQ2 redirect, 387, 389
IRQ3, 386
IRQ4, 386
IRQ5, 386
IRQ6, 387
IRQ7, 387
, 391, 395
IRQ8, 386, 444
IRQ8 through IRQ15, 386
IRQ9, 387
IRR, 371
ISA address bus, 338
ISA bus cycle definition, 339
ISA bus cycle timing, 341
ISA bus master capability, 433
ISA command lines, 340
ISA connector, 337
ISA connector, 16-bit, 336
ISA connector, 8-bit, 336
ISA device size lines, 343
ISA DMA transfer rates, 431
ISA DMA-related signals, 345
ISA error reporting signal, 349

ISA interrupt request (IRQ) lines, 347
ISA interrupt-related signals, 347
ISA miscellaneous signals, 350
ISA reset signal, 344
ISR, 366
ISRs, 369

J

jump instruction, 13

K

key matrix, 450
keyboard, 449, 450
keyboard BIOS routine, 457
keyboard break scan code, 450
keyboard command port,, 451
keyboard data port, 450
keyboard interrupt request, 450
keyboard interrupt request line, 386
keyboard interrupt service routine, 372, 450
keyboard make scan code, 450
keyboard scan, 450
keyboard scan code, 450
keyboard/mouse command/status port, 454
keyboard/mouse data port, 456
keyboard/mouse interface, 449

L

LA bus, 122
LA23:LA17, 338
latchable address bus, 122, 338
LDT, 101
LDTR, 102
line, 283, 308
little-endian byte ordering rule, 74
least recently used (LRU) algorithm, 307
local data bus, 126
look-aside cache, 287
local descriptor table (LDT), 101
local descriptor table register (LDTR), 102
look-through cache, 283

M

M/IO#, 33
M16#, 128, 344
machine status word (MSW) register, 116
maskable interrupt request line (INTR), 368
master interrupt controller, 383
MASTER16#, 345, 435
MC146818 real-time clock and configuration RAM chip, 441
memory address bus, 236, 241
memory parity error, 399
memory read command, 340
memory size 16, 344
memory write command, 340
memory-based semaphore, 93
miscellaneous ISA signals, 350
MODE command, 475
Motorola MC146818 real-time clock and configuration RAM chip, 441
mouse, 449, 450
mouse BIOS routine, 457
mouse interface, 387
mouse interrupt request, 450
MOV instruction, 33
MP bit, 460, 466
MRDC#, 340
MROM, 317
MS-DOS function calls, 404
MSW register, 460
multitasking operating system, 472
MUX, 240
MWTC#, 340

N

NCA#, 312
NMI, 349, 399, 442, 473
no wait state, 343
non-specific EOI, 379
NOWS#, 138, 343
NPRD#, 465
NPS1#, 464
NPS2, 465
NPWR#, 465

numeric coprocessor, 459
numeric coprocessor control bits in the MSW register, 460
numeric coprocessor installed, description of operation, 460
numeric coprocessor, Weitek, 467
numeric processor read, 465
numeric processor select 1, 464
numeric processor select 2, 465
numeric processor write, 465

O

OCW1, 397
option ROM, 43
OSC, oscillator signal, 350
OUT instruction, 32
overrun condition, 369

P

page fault, 403
parallel port 1, 387
parallel port 2, 386
parity, 256
parity generator/checker, 256
PASS A20, 451
PEACK#, 461, 464
PEREQ, 461, 464
PIC, 366
POST, 322
POST/BIOS ROM, 42
POWERGOOD, 113
power-on reset, 447
power-on restart address, 12, 116
power-on self-test, or POST, 116, 322
page-mode DRAM, 259
principle of locality, 277
prefetcher, 54
processor extension acknowledge (PEACK#), 461, 464
processor extension not available exception, 403
processor extension request (PEREQ), 461, 464
processor extension segment overrun exception, 403
program, 12
PROM, 317
protect enable, 62
protected mode, 62, 98, 162
protected mode to real mode, 447

R

RAM parity check, 400
RAM refresh logic, 433
RAM, dual-ported, 312
RAS#, 236, 242
RAS/CAS delay, 236, 242
RAS/CAS generation logic, 240
RAS-only refresh, 245
read, 12
read bus cycle, 141
read-only memory, 315
ready line, 141
ready timer, 135
READY#, 141
READYO#, ready output, 464
real mode, 369
real-time clock bytes, 443
real-time clock chip, 470
real-time clock function, 443
refresh, 474
refresh and ISA bus masters, 439
refresh interval, 249
refresh logic, 245, 433
refresh request, 400
refresh request signal, 471
refresh row counter, 245
refresh timer, 471
refresh timer, reprogramming, 471
REFRESH#, 247, 350
requestor privilege level (RPL), 102, 105
RESDRV (reset drive) signal, 344
reset code byte, 447
reset register values, 115
RESET signal, 113
ROM address decoder, 42
ROM checksum, 322
ROM chip-enable, 321

ROM chip-select, 321
ROM memory\: theory of operation, 315
ROM output enable, 321
ROM programmer, 316
ROM scan, 330
ROM, device ROM signature, 331
ROM, device ROMs, 330
ROM, electrically eraseable programmable read-only memory (EEPROM), 319
ROM, eraseable programmable read-only memory (EPROM), 318
ROM, erasing contents of EPROM, 319
ROM, flash EEPROM, 320
ROM, fusible-link PROM, 317
ROM, masked ROM (MROM), 317
ROM, system board, 322
row address, 236, 238
RPL, 102, 105
read from even addressed location in 8-bit device, 128
refresh timer, 245
row address decoder, 237
row address strobe (RAS#), 242
row latch, 236
RTC 24-hour mode, 444
RTC alarm interrupt enable, 444
RTC alarm interrupt flag, 445
RTC BIOS control, 445
RTC chip, 441
RTC chip's status registers A through D, 444
RTC date mode, 444
RTC daylight-savings enable, 444
RTC interrupt request flag, 445
RTC periodic interrupt enable, 444
RTC periodic interrupt flag, 445
RTC rate selection bits, 444
RTC set bit, 444
RTC square-wave enable, 444
RTC status register A, 444
RTC status register B, 444
RTC status register C, 445
RTC update in progress, 444
RTC update-ended interrupt enable, 444
RTC update-ended interrupt flag, 445

S

SA19:SA0, 338
SBHE#, 338
scan code, 450
SCRAM, 267
SD15:SD8, 339
SD7:SD0, 339
security lock latch, 400
segment descriptor, 101
segment not present, 403
segment wrap-around, 452
segment wraparound, 79
self refresh, 248
self-modifying code, 16
semaphore, 93
serial port 1, 386
serial port 2, 386
set interrupt enable instruction, 368
setup time, 149
shadow RAM, 324
shared resource, 92
shutdown, 108, 151, 185, 230
single step, or trap, interrupt, 403
slave ID, 387, 398
slave ID register, 383, 387
slave interrupt controller, 383, 387
slowdown timer, 474
SMRDC#, 340
SMWTC#, 340
snoop hit, 295
software interrupt instruction, 403
software interrupts, 402
software-enforced coherency, 298
speaker data timer, 471
split cache, 290
SRAM, 271
second level cache, 289
snarfing, 295
snooping, 295
spatial locality, 277
static RAM (SRAM), 271
system address (SA) bus, 121
stack, 374

stack segment overrun or not present exception, 403
stale data, 279
status enable, 464
status port, 37
STEN, 464
STI (Set Interrupt Enable) instruction, 368
synchronous DRAM, 268
system address (SA) bus, 338
system bus high enable, 338
system configuration information, 446, 450
system control port A, 400, 473
system control port B, 400, 471, 473
system data (SD) bus, 126
system data bus, lower path, 339
system data bus, upper path, 339
system memory read command, 340
system memory write command, 340
system timer, 386, 469
system timer frequency, 469
system timer interrupt, 472

T

T1, 140
T2, 140
Tc, 140
TC (DMA terminal count), 345, 413
time delay device, 239
timer 0, 469
timer 0 frequency, 469
timer 1, 471
timer 2, 471
timer 2 (speaker timer) output, 400
transfer complete, 345
trap interrupt, 403
Ts, 140
tag RAM, 281
temporal locality, 277
two-way set-associative cache, 304

U

UMB, 327
unified cache, 290
upper memory blocks, 327

W

watchdog timer, 472
watchdog timer status, 400
Weitek interpretation of the memory address, 468
Weitek numeric coprocessor, 467
Weitek-specific driver routine, 467
wraparound, 79
write, 12
write bus cycle, 146
write policy, 292
write-back cache, 294
write-through cache, 293