

SECOND EDITION

COMPUTER SCIENCE

H A N D B O O K

EDITOR-IN-CHIEF

ALLEN B. TUCKER



CHAPMAN & HALL/CRC



Published in Cooperation with ACM, The Association for Computing Machinery

QA 76
C 515
2004

Library of Congress Cataloging-in-Publication Data

Computer science handbook / editor-in-chief, Allen B. Tucker—2nd ed.

p. cm.

Includes bibliographical references and index.

ISBN 1-58488-360-X (alk. paper)

1. Computer science—Handbooks, manuals, etc. 2. Engineering—Handbooks, manuals, etc.

I. Tucker, Allen B.

QA76.C54755 2004

004—dc22

2003068758

This book contains information obtained from authentic and highly regarded sources. Reprinted material is quoted with permission, and sources are indicated. A wide variety of references are listed. Reasonable efforts have been made to publish reliable data and information, but the author and the publisher cannot assume responsibility for the validity of all materials or for the consequences of their use.

Neither this book nor any part may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, microfilming, and recording, or by any information storage or retrieval system, without prior permission in writing from the publisher.

All rights reserved. Authorization to photocopy items for internal or personal use, or the personal or internal use of specific clients, may be granted by CRC Press LLC, provided that \$1.50 per page photocopied is paid directly to Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923 USA. The fee code for users of the Transactional Reporting Service is ISBN 1-58488-360-X/04/\$0.00+\$1.50. The fee is subject to change without notice. For organizations that have been granted a photocopy license by the CCC, a separate system of payment has been arranged.

The consent of CRC Press LLC does not extend to copying for general distribution, for promotion, for creating new works, or for resale. Specific permission must be obtained in writing from CRC Press LLC for such copying.

Direct all inquiries to CRC Press LLC, 2000 N.W. Corporate Blvd., Boca Raton, Florida 33431.

Trademark Notice: Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation, without intent to infringe.

Visit the CRC Press Web site at www.crcpress.com

© 2004 by Chapman & Hall/CRC



No claim to original U.S. Government works

International Standard Book Number 1-58488-360-X

Library of Congress Card Number 2003068758

Printed in the United States of America 1 2 3 4 5 6 7 8 9 0

Printed on acid-free paper

Contents

- 1 Computer Science: The Discipline and its Impact
Allen B. Tucker and Peter Wegner 1-1
- 2 Ethical Issues for Computer Scientists
Deborah G. Johnson and Keith W. Miller 2-1

Section I: Algorithms and Complexity

- 3 Basic Techniques for Design and Analysis of Algorithms
Edward M. Reingold 3-1
- 4 Data Structures
Roberto Tamassia and Bryan M. Cantrill 4-1
- 5 Complexity Theory
Eric W. Allender, Michael C. Loui, and Kenneth W. Regan 5-1
- 6 Formal Models and Computability
Tao Jiang, Ming Li, and Bala Ravikumar 6-1
- 7 Graph and Network Algorithms
Samir Khuller and Balaji Raghavachari 7-1
- 8 Algebraic Algorithms
Angel Diaz, Erich Kaltófen, and Victor Y. Pan 8-1
- 9 Cryptography
Jonathan Katz 9-1
- 10 Parallel Algorithms
Guy E. Blelloch and Bruce M. Maggs 10-1
- 11 Computational Geometry
D. T. Lee 11-1

12	Randomized Algorithms <i>Rajeev Motwani and Prabhakar Raghavan</i>	12-1
13	Pattern Matching and Text Compression Algorithms <i>Maxime Crochemore and Thierry Lecroq</i>	13-1
14	Genetic Algorithms <i>Stephanie Forrest</i>	14-1
15	Combinatorial Optimization <i>Vijay Chandru and M. R. Rao</i>	15-1

Section II: Architecture and Organization

16	Digital Logic <i>Miriam Leeser</i>	16-1
17	Digital Computer Architecture <i>David R. Kaeli</i>	17-1
18	Memory Systems <i>Douglas C. Burger, James R. Goodman, and Gurindar S. Sohi</i>	18-1
19	Buses <i>Windsor W. Hsu and Jih-Kwon Peir</i>	19-1
20	Input/Output Devices and Interaction Techniques <i>Ken Hinckley, Robert J. K. Jacob, and Colin Ware</i>	20-1
21	Secondary Storage Systems <i>Alexander Thomasian</i>	21-1
22	High-Speed Computer Arithmetic <i>Earl E. Swartzlander Jr.</i>	22-1
23	Parallel Architectures <i>Michael J. Flynn and Kevin W. Rudd</i>	23-1
24	Architecture and Networks <i>Robert S. Roos</i>	24-1
25	Fault Tolerance <i>Edward J. McCluskey and Subhasish Mitra</i>	25-1

Section III: Computational Science

26	Geometry-Grid Generation <i>Bharat K. Soni and Nigel P. Weatherill</i>	26-1
27	Scientific Visualization <i>William R. Sherman, Alan B. Craig, M. Pauline Baker, and Colleen Bushell</i>	27-1
28	Computational Structural Mechanics <i>Ahmed K. Noor</i>	28-1
29	Computational Electromagnetics <i>J. S. Shang</i>	29-1
30	Computational Fluid Dynamics <i>David A. Caughey</i>	30-1
31	Computational Ocean Modeling <i>Lakshmi Kantha and Steve Piacsek</i>	31-1
32	Computational Chemistry <i>Frederick J. Heldrich, Clyde R. Metz, Henry Donato, Kristin D. Krantzman, Sandra Harper, Jason S. Overby, and Gamil A. Guirgis</i>	32-1
33	Computational Astrophysics <i>Jon Hakkila, Derek Buzasi, and Robert J. Thacker</i>	33-1
34	Computational Biology <i>David T. Kingsbury</i>	34-1

Section IV: Graphics and Visual Computing

35	Overview of Three-Dimensional Computer Graphics <i>Donald H. House</i>	35-1
36	Geometric Primitives <i>Alyn P. Rockwood</i>	36-1
37	Advanced Geometric Modeling <i>David S. Ebert</i>	37-1
38	Mainstream Rendering Techniques <i>Alan Watt and Steve Maddock</i>	38-1
39	Sampling, Reconstruction, and Antialiasing <i>George Wolberg</i>	39-1

40	Computer Animation <i>Nadia Magnenat Thalmann and Daniel Thalmann</i>	40-1
41	Volume Visualization <i>Arie Kaufman and Klaus Mueller</i>	41-1
42	Virtual Reality <i>Steve Bryson</i>	42-1
43	Computer Vision <i>Daniel Huttenlocher</i>	43-1

Section V: Human-Computer Interaction

44	The Organizational Contexts of Development and Use <i>Jonathan Grudin and M. Lynne Markus</i>	44-1
45	Usability Engineering <i>Jakob Nielsen</i>	45-1
46	Task Analysis and the Design of Functionality <i>David Kieras</i>	46-1
47	Human-Centered System Development <i>Jennifer Tucker and Abby Mackness</i>	47-1
48	Graphical User Interface Programming <i>Brad A. Myers</i>	48-1
49	Multimedia <i>James L. Alty</i>	49-1
50	Computer-Supported Collaborative Work <i>Fadi P. Deek and James A. McHugh</i>	50-1
51	Applying International Usability Standards <i>Wolfgang Dzida</i>	51-1

Section VI: Information Management

52	Data Models <i>Avi Silberschatz, Henry F. Korth, and S. Sudarshan</i>	52-1
53	Tuning Database Design for High Performance <i>Dennis Shasha and Philippe Bonnet</i>	53-1

54	Access Methods <i>Betty Salzberg and Donghui Zhang</i>	54-1
55	Query Optimization <i>Yannis E. Ioannidis</i>	55-1
56	Concurrency Control and Recovery <i>Michael J. Franklin</i>	56-1
57	Transaction Processing <i>Alexander Thomasian</i>	57-1
58	Distributed and Parallel Database Systems <i>M. Tamer Özsu and Patrick Valduriez</i>	58-1
59	Multimedia Databases: Analysis, Modeling, Querying, and Indexing <i>Vincent Oria, Ying Li, and Chitra Dorai</i>	59-1
60	Database Security and Privacy <i>Sushil Jajodia</i>	60-1

Section VII: Intelligent Systems

61	Logic-Based Reasoning for Intelligent Systems <i>James J. Lu and Erik Rosenthal</i>	61-1
62	Qualitative Reasoning <i>Kenneth D. Forbus</i>	62-1
63	Search <i>D. Kopec, T.A. Marsland, and J.L. Cox</i>	63-1
64	Understanding Spoken Language <i>Stephanie Seneff and Victor Zue</i>	64-1
65	Decision Trees and Instance-Based Classifiers <i>J. Ross Quinlan</i>	65-1
66	Neural Networks <i>Michael I. Jordan and Christopher M. Bishop</i>	66-1
67	Planning and Scheduling <i>Thomas Dean and Subbarao Kambhampati</i>	67-1
68	Explanation-Based Learning <i>Gerald DeJong</i>	68-1
69	Cognitive Modeling <i>Eric Chown</i>	69-1

70	Graphical Models for Probabilistic and Causal Reasoning <i>Judea Pearl</i>	70-1
71	Robotics <i>Frank L. Lewis, John M. Fitzgerald, and Kai Liu</i>	71-1

Section VIII: Net-Centric Computing

72	Network Organization and Topologies <i>William Stallings</i>	72-1
73	Routing Protocols <i>Radia Perlman</i>	73-1
74	Network and Internet Security <i>Steven Bellovin</i>	74-1
75	Information Retrieval and Data Mining <i>Katherine G. Herbert, Jason T.L. Wang, and Jianghui Liu</i>	75-1
76	Data Compression <i>Z. Rahman</i>	76-1
77	Security and Privacy <i>Peter G. Neumann</i>	77-1
78	Malicious Software and Hacking <i>David Ferbrache and Stuart Mort</i>	78-1
79	Authentication, Access Control, and Intrusion Detection <i>Ravi S. Sandhu and Pierangela Samarati</i>	79-1

Section IX: Operating Systems

80	What Is an Operating System? <i>Raphael Finkel</i>	80-1
81	Thread Management for Shared-Memory Multiprocessors <i>Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy</i>	81-1
82	Process and Device Scheduling <i>Robert D. Cupper</i>	82-1
83	Real-Time and Embedded Systems <i>John A. Stankovic</i>	83-1

84	Process Synchronization and Interprocess Communication <i>Craig E. Wills</i>	84-1
85	Virtual Memory <i>Peter J. Denning</i>	85-1
86	Secondary Storage and Filesystems <i>Marshall Kirk McKusick</i>	86-1
87	Overview of Distributed Operating Systems <i>Sape J. Mullender</i>	87-1
88	Distributed and Multiprocessor Scheduling <i>Steve J. Chapin and Jon B. Weissman</i>	88-1
89	Distributed File Systems and Distributed Memory <i>T. W. Doepfner Jr.</i>	89-1

Section X: Programming Languages

90	Imperative Language Paradigm <i>Michael J. Jipping and Kim Bruce</i>	90-1
91	The Object-Oriented Language Paradigm <i>Raimund Ege</i>	91-1
92	Functional Programming Languages <i>Benjamin Goldberg</i>	92-1
93	Logic Programming and Constraint Logic Programming <i>Jacques Cohen</i>	93-1
94	Scripting Languages <i>Robert E. Noonan and William L. Bynum</i>	94-1
95	Event-Driven Programming <i>Allen B. Tucker and Robert E. Noonan</i>	95-1
96	Concurrent/Distributed Computing Paradigm <i>Andrew P. Bernat and Patricia Teller</i>	96-1
97	Type Systems <i>Luca Cardelli</i>	97-1
98	Programming Language Semantics <i>David A. Schmidt</i>	98-1

99	Compilers and Interpreters <i>Kenneth C. Louden</i>	99-1
100	Runtime Environments and Memory Management <i>Robert E. Noonan and William L. Bynum</i>	100-1

Section XI: Software Engineering

101	Software Qualities and Principles <i>Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli</i>	101-1
102	Software Process Models <i>Ian Sommerville</i>	102-1
103	Traditional Software Design <i>Steven A. Demurjian Sr.</i>	103-1
104	Object-Oriented Software Design <i>Steven A. Demurjian Sr. and Patricia J. Pia</i>	104-1
105	Software Testing <i>Gregory M. Kapfhammer</i>	105-1
106	Formal Methods <i>Jonathan P. Bowen and Michael G. Hinchey</i>	106-1
107	Verification and Validation <i>John D. Gannon</i>	107-1
108	Development Strategies and Project Management <i>Roger S. Pressman</i>	108-1
109	Software Architecture <i>Stephen B. Seidman</i>	109-1
110	Specialized System Development <i>Osama Eljabiri and Fadi P. Deek</i>	110-1
	Appendix A: Professional Societies in Computing	A-1
	Appendix B: The ACM Code of Ethics and Professional Conduct	B-1
	Appendix C: Standards-Making Bodies and Standards	C-1
	Appendix D: Common Languages and Conventions	D-1
	Index	Index-1

19

Buses

19.1	Introduction	19-1
19.2	Bus Physics	19-2
	Transmission-Line Concepts • Signal Reflections • Wire-OR Glitches • Signal Skew • Cross-Coupling Effects	
19.3	Bus Arbitration	19-5
	Centralized Arbitration • Decentralized Arbitration	
19.4	Bus Protocol	19-7
	Asynchronous Protocol • Synchronous Protocol • Split-Transaction Protocol	
19.5	Issues in SMP System Buses	19-9
	Cache Coherence Protocols • Bus Arbitration • Bus Bandwidth • Memory Access Latency • Synchronization and Locking	
19.6	Putting It All Together — CCL-XMP System Bus	19-15
19.7	Historical Perspective and Research Issues.....	19-17

Windsor W. Hsu
IBM Research

Jih-Kwon Peir
University of Florida

19.1 Introduction

The *bus* is the most popular communication pathway among the various components of a computer system. The distinguishing feature of the bus is that it consists of a single set of shared communication links to which many components can be attached. The bus is not only a very cost-effective means of connecting various components together, but also is very versatile in that new components can be added easily. Furthermore, the bus has a broadcasting capability which can be extremely useful. The downside of the shared communication links is that they allow only one communication to occur at a time and the bandwidth does not scale with the number of components attached. Nevertheless, the bus is very popular because there are many situations where several components need to be connected together but they need not all transmit at the same time. This kind of requirement maps naturally onto a bus, allowing a very cost-effective solution. However, there are cases where the bus does become a communication bottleneck. In such cases, very aggressive bus designs have been attempted, but there comes a point where the fundamental characteristic of the bus cannot be overcome and more expensive solutions such as point-to-point links have to be used.

Buses are used at every level in the computer system. For instance, within the processor itself, the bus is often the means of communication between the register file and the various execution units. At a higher level, the processor is connected to the memory subsystem through the *system bus*. Today's computers typically have a fast peripheral bus called a *local bus* which directly interfaces onto the system bus to provide a high bandwidth for demanding devices such as the graphics adaptor. Other less demanding peripheral devices are attached to the *I/O bus*. In the old days, the processor, memory subsystem, and I/O devices were all plugged onto the *backplane bus*, which is so called because the bus runs physically

along the backplane of the computer chassis. The various buses are each optimized for a particular set of performance requirements and cost constraints and may thus seem very different from one another. However, their underlying issues are fundamentally the same.

A major requirement for designing a bus or simply comprehending a bus design is a proper understanding of the electrical and mechanical behavior of the bus. As buses are pushed to provide higher data rates, physical phenomena such as signal reflection, crosstalk, skew, etc., are becoming more significant and have to be handled carefully. Because the communication medium of a bus is shared by multiple devices, at most one transmission can be initiated at any time by any device. The other devices can act only as receivers or **bus slaves** for the transmission. A device that is capable of initiating and controlling a communication is called a **bus master**. In order to ensure that only one bus master is talking at any one time, the bus masters have to go through a **bus arbitration** process before they can gain control of the bus. Once a bus master has been granted control of the bus, a **bus protocol** has to be followed by the master and the slave in order for them to understand one another. The specifics of the protocol can vary widely, depending on the functional and performance requirements. For instance, the protocol used in the system bus of a uniprocessor is dramatically different from that used in the system bus of a shared-memory **symmetric multiprocessor** (SMP).

In this chapter, an overview of the basic underlying physics of computer buses is presented first. This is followed by a discussion of important issues in bus designs, including bus arbitration and various communication protocols. Because of the increasing prevalence of SMP systems and the special challenges they pose for buses, a separate section is devoted to discussing special bus issues in such machines. The discussion is wrapped up with a case study of a modern SMP system bus design. Finally, a historical perspective of computer buses and the related research issues are given in the last section.

19.2 Bus Physics

Computer buses are becoming wider and are being run at higher frequencies in order to keep up with the phenomenal improvement in CPU performance. As the physical and temporal margins in bus designs are reduced, it is imperative that electrical phenomena such as signal reflections, skew, crosstalk, etc., be understood and properly handled. In this section, we introduce the basic ideas behind these phenomena. A more detailed discussion can be found in Giacomo [1990].

19.2.1 Transmission-Line Concepts

Electrical signals propagate with a finite speed that depends on the propagation medium. For instance, it takes approximately 5 ns for an electrical signal to travel 1 m along a typical copper wire. However, when analyzing electrical circuits, we often ignore their spatial properties. For example, we are seldom concerned with where each element of the circuit is located. We can do this because the circuits we usually encounter are lumped. In other words, their physical dimensions are small enough that for their particular applications, electromagnetic waves propagate across the circuits virtually instantaneously. However, this is not the case with high-speed buses. In this subsection, we introduce the basic concepts of the transmission-line model which are needed to understand the electrical behavior of today's buses.

In general, a connection is considered a transmission line if the propagation time of an electrical signal through it is a significant part of the rise time, fall time, or width of the signal. A good rule of thumb is to consider anything above $\frac{1}{4}$ as a "significant part." A transmission line has a characteristic impedance Z_0 . If the line impedance Z_L changes as a signal propagates down the line, part of the power travels backwards as a reflected signal. The reflection coefficient is given by $\Gamma = (Z_L - Z_0)/(Z_L + Z_0)$. In other words, the strength of the reflected signal increases with the magnitude of the impedance mismatch.

Consider the circuit in Figure 19.1. It contains a voltage source V_s with internal resistance R_s connected by a pair of transmission lines to a load of resistance Z_L . We assume that it takes T units of time for an electrical signal to propagate one-way across the circuit. At time $t = 0$, the switch is closed and a voltage

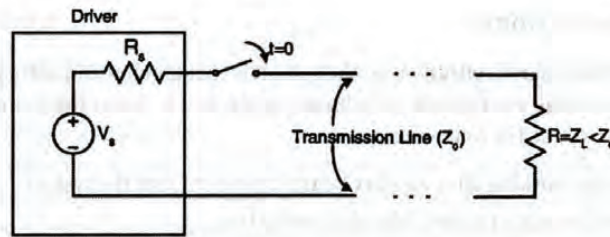


FIGURE 19.1 Circuit for transmission-line example.

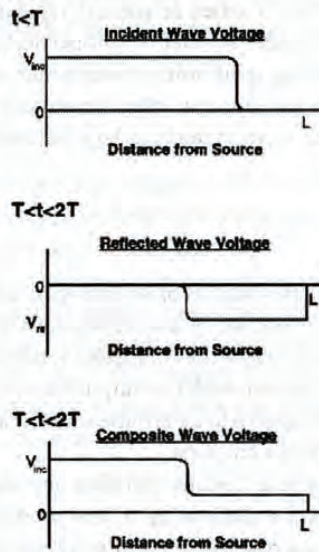


FIGURE 19.2 Transmission-line voltage waveform.

pulse is sent down the transmission lines toward the load. Figure 19.2 contains a short chronicle of the voltage waveform seen on the lines. For time $t < T$, the signal sees only Z_0 , the impedance of the line. Thus, the voltage of the incident wave $V_{inc} = V_s Z_0 / (R_s + Z_0)$. When the incident wave hits the load at time $t = T$, it sees an additional impedance Z_L which reflects the incident wave with a coefficient of $\Gamma = (Z_L - Z_0) / (Z_L + Z_0)$. In Figure 19.2, we assume that $Z_L < Z_0$ so that the reflected wave is negative with voltage $V_{re} = \Gamma V_{inc}$. If the transmission lines are not properly terminated at the driver, i.e., $R_s \neq Z_0$, there will be additional reflections before the system settles down. The composite signal at any point along the line is the instantaneous sum of all the incident and reflected signals. Note that all the signals are subject to line losses, especially in their high-frequency components. Thus, pulses eventually lose their shape.

When the circuit is in steady state, the above analysis should agree with the lumped circuit model. For simplicity, let us assume that the transmission lines are properly terminated at the driver, i.e., $R_s = Z_0$. In this case, there is only one reflection and the circuit settles down at time $t = 2T$. Thus, we would expect the steady-state voltage to be the sum of V_{inc} and V_{re} . This works out to $V_s Z_L / (Z_L + Z_0)$, a result which agrees with that predicted by the lumped-circuit model. An intuitive way to reason about transmission lines is to think in terms of feedback. In the beginning, the signal has no clear idea of how much impedance it will encounter in the circuit. Thus, it makes a guess based on the impedance it has already seen. As the signal propagates, it learns more about the circuit, and this information is fed back in the form of reflections, so that eventually Kirchhoff's circuit laws are satisfied.

19.2.2 Signal Reflections

The consequence of having signal reflections in the system is that glitches and extra pulses may appear on the bus. This may cause some unexpected and obscure problems. Some of the more common symptoms of reflection problems include the following:

- A board that stops working after another board is plugged into the system.
- A board that works only in a particular slot on the bus.
- A system that works only when the boards are arranged in a specific order.

Reflections are typically most significant at the various sources and loads on the bus. We can reduce the magnitude of these reflections by matching the impedance of the sources and loads to that of the lines.

This can be accomplished by adding a series or parallel resistance or by using a clamping diode. Impedance matching is complicated by the fact that the properties of bus drivers change as they switch on or off. For better impedance matching, small voltage swings and low-capacitance drivers and receivers are helpful. Note that reflections can also occur at other impedance discontinuities such as interboard connections, board layer changes, etc. To accurately model all these effects, computer simulations using tools such as SPICE are often needed.

19.2.3 Wire-OR Glitches

Wire-OR logic is a kind of logic where the outputs of several open-collector gates are connected together in such a way as to realize a logical OR function. A sample circuit is shown in Figure 19.3. Notice that the voltage on the line is low as long as any one of the transistors is turned on. Thus this circuit implements the logical NOR function or the OR function with the output asserted low. Wire-OR is very useful in bus arbitration. For instance, it enables the system to determine whether any bus master wishes to use the bus. Most buses use wire-OR logic for at least a few lines.

However, wire-OR lines are subject to a fundamental phenomenon known as *wire-OR glitch*. During an active to high-impedance transition, a glitch of up to one round-trip delay in width may appear on a wire-OR line. This phenomenon is a result of the finite propagation speed of electrical signals on a transmission line. Consider the case where only transistors 1 and n are initially turned on. Suppose that transistor n is now turned off. The current that it was previously sinking continues to flow, thus creating a signal which propagates along the line.

A more detailed explanation of the wire-OR glitch is given in Gustavson and Theus [1983]. Various ways of dealing with it are discussed in Gustavson and Theus [1983] and Taub [1983a, 1983b]. In the IEEE Futurebus, the wire-OR glitch problem is mitigated by the fact that the bus specification imposes constraints on when devices can switch on or off, effectively setting a limit on the maximum glitch duration [Taub 1984].

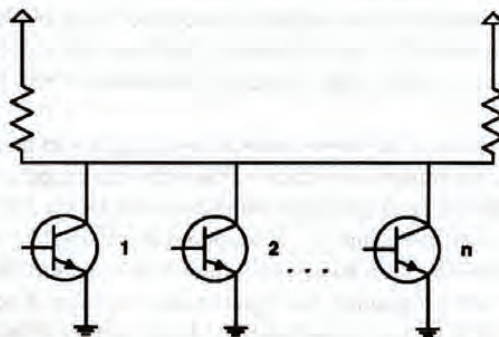


FIGURE 19.3 Wire-OR circuit.

19.2.4 Signal Skew

Another important electrical phenomenon in buses is *signal skew*. Because of differences in transmission lines, loading, etc., slight differences in the propagation delay of different bits in a word are inevitable. These differences are known as signal skew.

In a transmission, the receiver must be able to somehow determine when all the bits of a word have arrived and can be sampled. The effect of skew is to reduce the window during which the receiver can assume that the data are valid. This effectively limits the data rate of the bus. A wide bus consists of more parallel lines and is thus subject to more skew. In general, skew can be reduced by paying meticulous attention to the impedance of the bus lines. Synchronous buses have to deal with the additional problem of clock to data skew. An approach that has been taken to minimize this skew is to loop back the clock line at the end of the bus. When doing a data transfer, the clock signal that propagates in the same direction as the data transfer is used. Rambus uses this technique to minimize skew with respect to its aggressive 250-MHz clock [Rambus 1992].

19.2.5 Cross-Coupling Effects

A signal-carrying line sets up electrostatic and magnetic fields around it. In a bus, the lines run parallel and close to one another. Thus the fields from nearby lines intersect, causing a signal on one line to affect the signal on another. This is called crosstalk or coupling noise.

A simple way to reduce this effect is to spatially separate the bus lines so that the fields do not interfere with one another. However there is clearly a limit to how far we can carry this. Another way to reduce both the mutual capacitance and inductance of the lines is to introduce ground planes or wires near the bus lines. But this has undesirable side effects such as increasing the self-capacitance of the lines. An approach commonly taken to reduce coupling effects is to separate the lines with an insulator that has a low dielectric constant. Typically, combinations of these techniques are used in a bus design.

19.3 Bus Arbitration

Buses are ubiquitous in computer systems because they are a cost-effective and versatile means of connecting several devices together. The cost-effectiveness and versatility of buses stems from the fact that a bus has only one communication medium, which is shared by all the devices. In other words, at most one communication can occur on a bus at any one time. This implies that there must be some mechanism to decide which bus master has control of the bus at a given time. The process of arbitrating between requests for bus control is called bus arbitration. Bus arbitration can be handled in different ways depending on the performance requirements and cost constraints.

19.3.1 Centralized Arbitration

In *centralized arbitration*, there is a special device, the central arbiter, which is in charge of granting bus control to one of the bus masters. The fact that there is only one arbiter means that the centralized scheme has a single point of failure. In general, centralized arbitration can be further divided into two schemes, depending on how the bus masters are connected to the arbiter.

In the first scheme, the central arbiter is connected to each of the bus masters through private two-way connections. Because the bus requests can be made independently and in parallel by the bus masters, this is sometimes known as *centralized independent requests arbitration* or *centralized parallel arbitration*. Notice that the connections in this system form star networks emanating from the central arbiter. One such network carries the bus request signals from the bus masters to the arbiter. Another carries the bus grant signal from the arbiter back to one of the bus masters. Various arbitration policies can be implemented in the arbiter, making this a very flexible scheme. This scheme is fast because there are direct connections between any bus master and the arbiter. However, all these direct connections lead to a high implementation

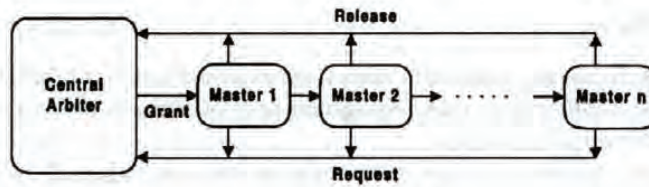


FIGURE 19.4 Daisy-chain bus arbitration.

cost. Furthermore, the use of direct connections means that the arbitration signals do not appear on the bus. This makes bus monitoring for debugging and diagnostic purposes difficult.

The second centralized arbitration scheme is known as *centralized serial priority arbitration*. In this scheme, there is a single bus grant signal line, which is routed through each of the bus masters as shown in Figure 19.4. This form of connection is known as a *daisy chain*. Hence, centralized serial priority arbitration is more commonly referred to as *daisy-chain arbitration*. In this scheme, there is a common wire-OR bus request line. A bus master may take control of the bus only if it has made a request and its incoming grant line is asserted. A bus master that does not wish to use the bus is required to forward the bus grant signal along the daisy chain. Notice that this implies an implicit priority assignment — the nearer a bus master is to the arbiter, the higher is its priority. The main advantage of daisy-chain arbitration is that it requires very few interconnections and the interface logic is simple. However, bus allocation in this scheme is slow because the grant signal has to travel along the daisy chain. Furthermore, the implicit priority scheduling may cause low-priority requests to be locked out indefinitely. Finally, as in centralized parallel arbitration, daisy-chain arbitration does not facilitate debugging and diagnosis. The VMEbus uses a variation of this scheme with four daisy-chained grant lines, which enable it to implement a variety of scheduling algorithms [Giacomo 1990].

19.3.2 Decentralized Arbitration

In *decentralized arbitration*, each bus master has its own arbitration and allocation logic. The responsibility of deciding who has control of the bus is distributed among the bus masters. Thus, this scheme is also known as *distributed arbitration*.

Typically, the bus contains n arbitration wire-OR lines and each bus master is assigned a unique n -bit arbitration number according to some priority scheme. During arbitration, if a master wishes to use the bus, it drives the arbitration lines with its arbitration number. If a master detects that the arbitration lines are carrying a higher priority number, it stops driving the less significant bits of its number. When the system settles down, the arbitration lines will indicate the winner, which is the participating master with the highest priority. The time for the system to settle down can be specified as a fixed time in the bus protocol. An alternative is to use an additional wire-OR line to indicate whether competing bus masters have completed arbitration. As with any priority scheme, the possibility of access starvation of the low-priority masters has to be considered. In the IEEE Futurebus, bus masters are divided into priority and fairness modules depending on whether they have any particularly urgent needs such as having to meet real-time constraints [Taub 1984]. A priority module can issue bus requests whenever it needs to. A fairness module, after winning an arbitration, will have to wait for all pending requests to be serviced before it can issue another request [Taub 1984]. Note that this does not guarantee but only helps to ensure that every module will be able to get a portion of the bus bandwidth.

The major disadvantage of the distributed scheme is that it requires relatively complex arbitration logic in each bus master and several arbitration lines on the bus. However, this scheme allows very fast bus allocation and a flexible assignment of priorities to the bus masters. Distributed arbitration is also more fault-tolerant than the centralized schemes in that the failure of a single bus master does not necessarily affect the operation of the bus. Variations of this scheme are widely used in buses such as the IEEE Futurebus, Nubus, Multibus II, Fastbus [Borrill 1985], and the Powerpath-2 System Bus of the SGI Challenge [Galles and Williams 1994].

The scheme described above is sometimes known as *distributed arbitration by self-selection* because each master decides whether it has won the race, effectively letting the winner select itself. Some computer networks, such as Ethernet, use another form of distributed arbitration which relies on collision detection [Metcalfe and Boggs 1976].

19.4 Bus Protocol

The bus is a shared resource. In order for it to function properly, all the devices on it must cooperate and adhere to a protocol or set of rules. This protocol defines precisely the bus signals that have to be asserted by the master and slave devices in each phase of the bus operation. In this section, we discuss some of the key options in designing bus protocols.

19.4.1 Asynchronous Protocol

In a communication, the sender and receiver must be coordinated so that the sender knows when to talk and the receiver knows when to listen. There are two basic ways to achieve proper coordination. This subsection discusses the asynchronous protocol. The next describes the synchronous design.

An asynchronous system does not have an explicit clock signal to coordinate the sender and receiver. Instead a *handshaking protocol* is used. In a handshaking protocol, the sender and receiver can proceed to the next step in the bus operation only if both of them are ready. In other words, both parties have to shake hands and agree to proceed before they can do so.

Figure 19.5 shows a basic handshaking protocol. Assume that *ReadReq* is used to request a read from memory, *DataReady* is used to indicate that data are ready on the data lines, and *Ack* is used to acknowledge the *ReadReq* or *DataReady* signals of the other party. Suppose a device wishes to initiate a memory read. When memory sees the *ReadReq*, it reads the address on the address bus and raises the *Ack* in acknowledgment of the request (step 1). When the device sees the *Ack*, it deasserts *ReadReq* and stops driving the address bus (step 2). Once memory sees that *ReadReq* has been deasserted, it drops the *Ack* line to acknowledge that it has seen the *ReadReq* signal (step 3). A similar exchange is carried out when memory is ready with the data that has been requested (steps 5–7). Notice that the two parties involved in the protocol take turns to respond to one another. They proceed in lockstep. This is how the handshaking protocol is able to coordinate the two devices.

The handshaking protocol is relatively insensitive to noise because of its self-timing nature. This self-timing nature also allows data to be transferred between devices of any speed, giving asynchronous designs the flexibility to handle new and faster devices as they appear. Thus, asynchronous designs are better able to scale with technology improvements. Furthermore, because clock skew is not an issue, asynchronous buses can be physically longer than their synchronous counterparts. The disadvantage of asynchronous designs lies in the fact that the handshaking protocol adds significant overhead to each data transfer and is thus slower than a synchronous protocol. As in any communication between parties with different clocks, there

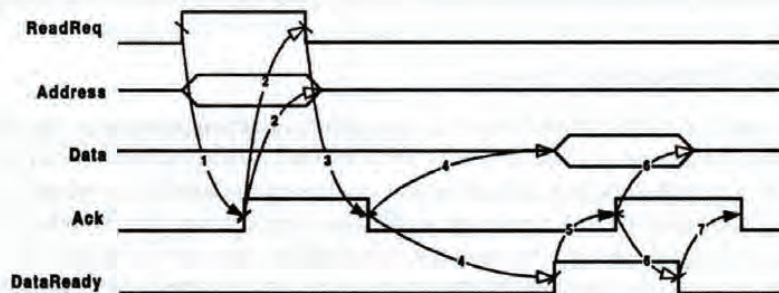


FIGURE 19.5 A basic handshaking protocol.

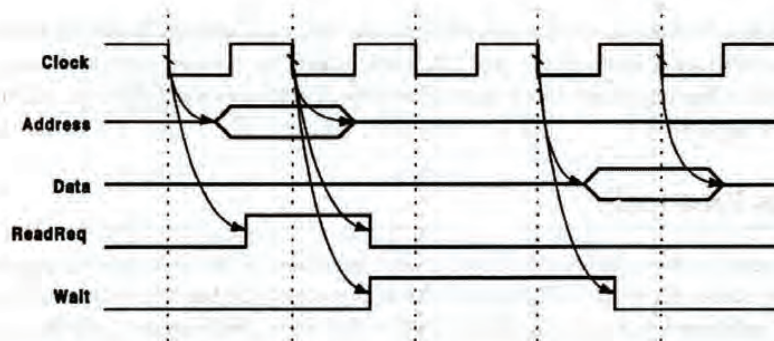


FIGURE 19.6 A basic synchronous protocol.

is an additional problem of synchronization failure when an asynchronous signal is sampled. Asynchronous designs are typically used when there is a need to accommodate many devices with a wide performance range and when the ability to incrementally upgrade the system is important. Thus, many of the asynchronous buses such as VMEbus, Futurebus, MCA, and IPI are backplane or I/O buses [Giacomo 1990].

19.4.2 Synchronous Protocol

In synchronous buses, the coordination of devices on the bus is achieved by distributing an explicit clock signal throughout the system. This clock signal is used as a reference to determine when the various bus signals can be assumed to be valid. Figure 19.6 shows a basic synchronous protocol coordinating a memory read transaction. Notice that all the signal changes happen with respect to the clock. In this particular example, the system is negative-edge triggered, which means that the signals are sampled on the falling edge of the clock.

An important design decision in synchronous buses is the choice of the clock frequency. Once a frequency is selected, it becomes locked into the protocol. The clock frequency must be chosen to allow sufficient time for the signals to propagate and settle throughout the system. Allowances must also be made for clock skew. Thus, the clock frequency is limited by the length of the bus and the speed of the interface logic. All things being equal, shorter buses can be designed to run at higher speeds.

The main advantage of the synchronous protocol is that it is fast. It also requires relatively few bus lines and simple interface logic, making it easy to implement and test. However, the synchronous protocol is less flexible than the asynchronous protocol in that it requires all the devices to support the same clock rate. Furthermore, this clock rate is fixed and cannot be raised compatibly to take advantage of technological advances. In addition, the length of synchronous buses is limited by the difficulty of distributing the clock signal to all the devices at the same time. Synchronous buses are typically used where there is a need to connect a small number of very tightly coupled devices and where speed is of paramount importance. Thus, synchronous buses are often used to connect the processor and the memory subsystem.

19.4.3 Split-Transaction Protocol

In order to increase the effective bandwidth of a bus, a **split-transaction protocol** can be used. The basic observation behind this protocol is that the bus is not being used to transmit information throughout the entire duration of a transaction, but only at the start and toward the end of the transaction. The idea is thus to split a bus transaction into a request transaction and a reply transaction so as to allow the bus to be released for other uses in between the request and reply stages. Figure 19.7 illustrates the idea. Clearly, this protocol only makes sense when the system has more than one bus master and the memory system is sophisticated enough to handle multiple overlapping transactions. This protocol is sometimes also known as a *connect/disconnect protocol*, *pipelined protocol*, or *packet-switched protocol*.

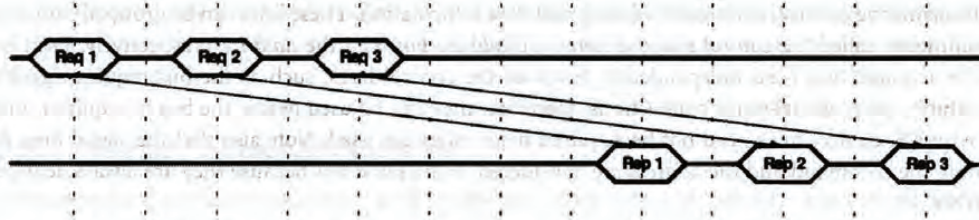


FIGURE 19.7 A split-transaction protocol.

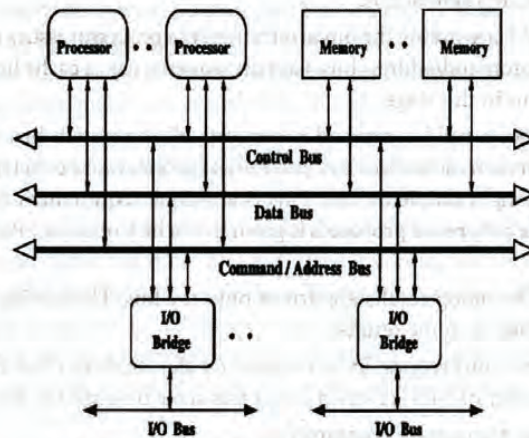


FIGURE 19.8 A typical multiprocessor system bus.

Although the split-transaction protocol allows more efficient utilization of bus bandwidth than a protocol that holds on to the bus for the whole transaction, it usually has a higher latency because the bus has to be acquired twice — once for the request and once for the reply. Furthermore, the split-transaction protocol is expensive to implement because it requires that the bus transactions be tagged and tracked by each device. Split-transaction protocols are widely used in the system buses of shared-memory SMPs, because bus bandwidth is a big issue in these machines.

Notice from Figure 19.7 that even with split transactions, the bus bandwidth is not totally utilized. This is because some bus cycles are needed to acquire the bus and to set up the transfer. Furthermore, some buses require a cycle of turnaround time between different masters driving the bus. Another way to increase effective bus bandwidth is thus to amortize this fixed cost over several words by allowing the bus to transfer multiple contiguous words back to back in one transaction. This is known as a *burst protocol* or *block transfer protocol* because a contiguous block of several words is transferred in each transaction.

19.5 Issues in SMP System Buses

Microprocessor-based symmetric multiprocessors (SMPs) with shared snooping buses have become an industry standard for building midrange departmental servers [Peir et al. 1993, Galles and Williams 1994]. In this section, we devote special attention to the bus issues that arise in SMP designs.

A typical SMP bus architecture (usually referred to as a *SMP system bus*, or simply *system bus*) is illustrated in Figure 19.8. SMP system buses require high bandwidth and low latency to connect multiple processors, memory modules, and I/O bridges. A system bus is composed of independent signal lines

for transmitting control, command/address, and data information. These lines can be grouped into what is commonly called the *control bus*, the *command/address bus*, and the *data bus*, respectively. Each bus can be acquired and used independently. Some of the control lines, such as the bus request signal to the arbiter, are point-to-point connections; therefore, they can be used before the bus is acquired. Also, the wire-OR control lines need not be acquired before they are used. Note also that the signal lines for sending the command and the address are considered as the same bus because they are always acquired together.

In general, each system bus request traverses through a number of stages, each of which may take one or more bus cycles.

- *Bus arbitration*: The requesting processor needs to go through an arbitration process in order to gain access to the shared system bus.
- *Command issuing*: After winning the bus arbitration, the processor issues the command along with an address on the command/address bus. Certain requests, e.g., a cache line writeback, also require access to the data bus in this stage.
- *Cache snooping*: Once a valid command is received, all the system bus masters (processors, I/O bridges) search their own cache directory and initiate proper cache coherence activities to maintain data coherence among multiple caches. This is a unique requirement for SMP system buses. A description of **cache coherence protocols** is given in Cache Coherence Protocols subsection of this section.
- *Acknowledgment*: The snoop results are driven onto the bus. The issuing processor has to update its cache directory based on the results.
- *Data transfer*: When a bus request incurs a data transfer, such as a line-fill request issued upon a cache miss, the transfer of data is carried out at this stage through the data bus.
- *Completion*: The bus transaction is completed.

Several important and unique issues in SMP system bus design are outlined as follows.

1. *Bus physics*: Due to the high bandwidth requirement, SMP system buses face the challenge of a high speed and wide data bus, and have to overcome heavy signal loading to accommodate a reasonable number of ports for connecting multiple system devices.
2. *Cache coherence*: Cache memory is a critical component in SMP systems. In order to maintain data coherence among multiple caches, a cache coherence protocol must be incorporated into the bus protocol.
3. *Bus arbitration*: Each of the processors should be given an equal opportunity to gain access to the bus. Logic to prevent, detect, and resolve starvation and deadlock is typically needed.
4. *Bus bandwidth*: Besides faster and wider data buses, two other bandwidth-increasing features are important: pipelining bus requests using the split-transaction bus design, and transferring large data blocks in a *burst* transfer mode. As described before, the processing of a system bus request involves several distinct stages. Pipelining these requests can increase the effective bus bandwidth significantly.
5. *Memory access latency*: Memory access latency is a classic performance bottleneck. Techniques such as replicated arbiter and **bus parking** [Mahoney 1990] can reduce the latency in bus arbitration. Furthermore, early DRAM access before resolving cache coherence issues can reduce memory access latency. Such early DRAM accesses, however, may have to be canceled depending on the snoop results.
6. *Synchronization and locking*: SMP system buses need to provide an efficient way of implementing atomic read-modify-write instructions. Techniques such as address-based locking and nonblocking caches can reduce the interference between locking requests and other normal bus operations.
7. *Reliability and fault tolerance*: Parity or ECC can be used to detect and correct transmission errors. When an error is uncorrectable, the requester will receive a negative acknowledgment and will retry the request. The timeout scheme is commonly included to detect the loss of a command.

In the following, these important issues will be discussed in detail. Because the techniques for handling bus physics issues for the SMP system bus are basically the same as those in other buses, we will omit further discussion.

19.5.1 Cache Coherence Protocols

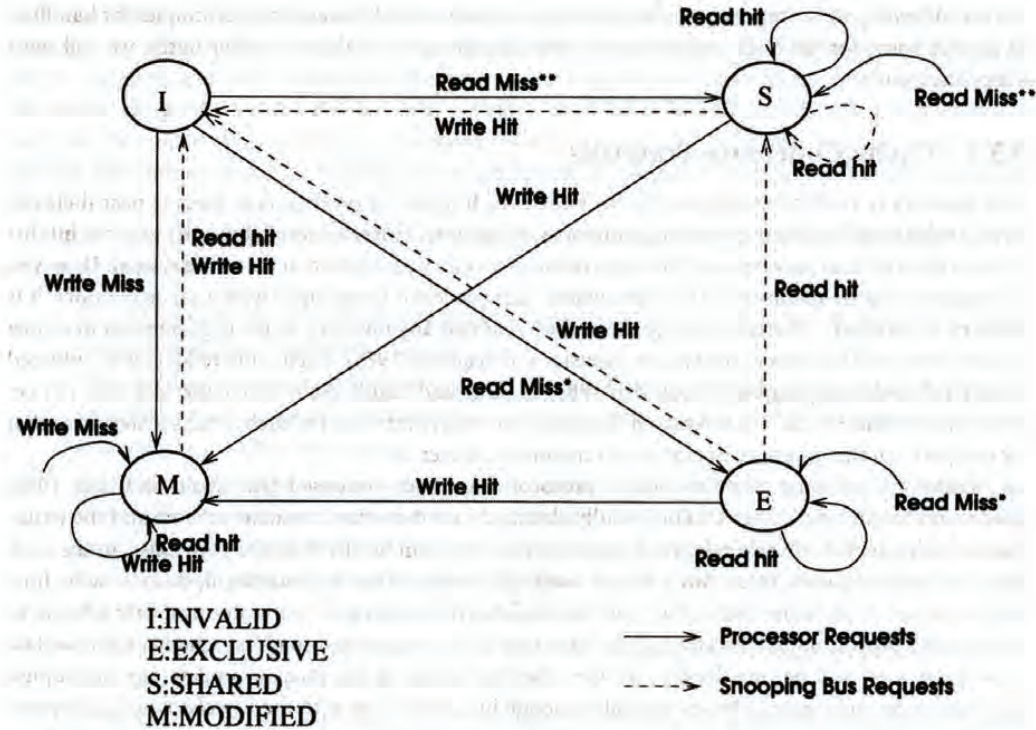
Cache memory is a critical component in SMP systems. It helps the processors to execute near their full speed by substantially reducing the average memory access time. This is achieved through fast cache hits for the majority of memory accesses and through reduced memory contention in the entire system. However, in designing a shared-memory SMP system where each processor is equipped with a cache memory, it is necessary to maintain coherence among the caches such that any memory access is guaranteed to return the latest version of the data in the system [Censier and Feautrier 1978]. Cache coherence can be enforced through a shared **snooping bus** [Goodman 1983, Sweazey and Smith 1986]. The basic idea is to rely on the broadcast nature of the bus to keep all the cache controllers informed of each other's activities so that they can perform the necessary operations to maintain coherency.

A number of snooping cache coherence protocols have been proposed [Archibald and Baer 1986, Sweazey and Smith 1986]. They can be broadly classified into the write-invalidate scheme and the write-broadcast scheme. In both schemes, read requests are carried out locally if a valid copy exists in the local cache. For write requests, these two schemes work differently. When a processor updates a cache line, all other copies of the same cache line must be invalidated according to the write-invalidate scheme to prevent other processors from accessing the stale data. Under the write-broadcast scheme, the new data of a write request will be broadcast to all the other caches to enable them to update any old copies. These two cache coherence schemes normally operate in conjunction with the *writeback* policy, because the *writethrough* policy generates memory traffic on every write request and is thus not suitable for a bus-based multiprocessor system.

The MESI (modified-exclusive-shared-invalid) write-invalidate cache coherence protocol with the writeback policy is considered in the following discussion. With minor variations, this protocol has been implemented in several commercial systems [Intel 1994, Greenley et al. 1995, Levitan et al. 1995]. In this protocol, each cache line has an associated MESI state recorded in the cache *directory* (also called cache tag array). The definitions of the four states are given in Table 19.1. When a memory request from either the processor or the snooping bus arrives at a cache controller, the cache directory is searched to determine cache hit/miss and the coherence action to be taken. The state transition diagram of the MESI protocol is illustrated in Figure 19.9, in which solid arrows represent state transitions due to requests issued by the local processor, and dashed arrows indicate state transitions due to requests from the snooping bus.

TABLE 19.1 Four States in MESI Coherence Protocol

State	Description
Modified (M)	The M-state indicates that the corresponding line is valid and is exclusive to the local cache. It also indicates that the line has been modified by the local processor. Therefore, the local cache has the latest copy of the line.
Exclusive (E)	The E-state indicates that the corresponding line is valid and is exclusive to the local cache. No modification has been made to the line. A write to an E-state line can be performed locally without producing any snooping bus traffic.
Shared (S)	The S-state indicates that the corresponding line is valid but may also exist in other caches in a multiprocessor system. Writing to an S-state line updates the local cache and generates a request to invalidate other shared copies.
Invalid (I)	The I-state indicates that the corresponding line is not available in the local cache. A cache miss occurs in accessing an I-state line. Typically, a line-fill request is issued to the memory to bring in the valid copy of the requested cache line.



Read Miss*: No other cache has a copy of the requested line, or the line is found in another cache in the M state.

Read Miss**: The requested line is found in other caches in the E or S state.

FIGURE 19.9 State transition diagram of MESI coherence protocol.

In general, when a read request from the processor hits a line in the local cache, the state of the cache line will not be altered. A write hit, on the other hand, will change the line state to M. In the meantime, if the original line state is S upon the write hit, an invalidation request will be issued to the snooping bus to invalidate the same line if it is present in any of the other caches. When a read miss occurs, the requested line will be brought into the cache in different states depending on whether the line is present in other caches and on its state in these caches. When a write miss occurs, the target line is fetched into the local cache; the new state becomes M and any copy of the cache line in any other cache is invalidated. For the requests from the snooping bus, a write hit always causes invalidation. A read hit to an M-state line will result in a writeback of the modified line and a transfer of ownership of the target line to the requesting processor. A read hit to an E- or S-state line for a snooping read request will cause a state transition to S.

19.5.2 Bus Arbitration

There are two issues in bus arbitration that are especially important for SMP system buses. The first is to ensure that the system is fair, deadlock-free, and starvation-free. In general, the fairness issue can be handled by using a first-come-first-served (FCFS) priority scheme, which will guarantee that requests are serviced in order of arrival. In fact, a simple random priority scheme may be adequate to provide a fair arbitration scheme for all the bus masters. The deadlock problem typically arises in a split-transaction bus where it is possible for a cycle of dependencies to form among the requests and replies. This can be handled

by distinguishing between requests and replies and always ensuring that replies are able to make forward progress. At first sight, it might seem that a fair arbitration policy should be starvation-free. However, this is not the case, because when overlapping bus requests are allowed, a request that has been granted by the bus may have to be later rejected due to interference with other requests. On a subsequent retry, the request may again encounter another conflict. In the pathological case, such a request may retry indefinitely. This situation is worse if there are lock requests and other resource conflicts on the system bus.

There are two general solutions to this starvation problem. The first solution eliminates any request rejection to prevent the starvation from happening. Whenever a conflict occurs, the request is queued at the location where the conflict is encountered and the queued request will be processed once the conflict condition disappears. This solution implies a variable-length bus pipeline, which requires extra handshaking on the system bus. In addition, excessive queues must be implemented to handle the conflict situation. The second solution depends on the ability to detect the starvation condition and to resolve the condition once it occurs. This method, in general, counts the number of times each request has been rejected. When a certain threshold is exceeded, emergency logic is activated to ensure that the starved request is serviced quickly and successfully.

The second important issue in bus arbitration is to minimize the latency for acquiring the bus. There are two techniques for doing this. The first technique is to replicate the arbiter in each processor. Each arbiter receives the request signals from all the bus masters just like in the centralized arbiter design. However, after the arbitration is resolved, each replicated arbiter only needs to send the grant signal to the local processor. The delay through a long wire across multiple chips in the centralized implementation can thus be avoided. The second latency-reduction technique is bus parking, which essentially implements a round-robin priority among the bus masters with a token being passed around. Once a bus master owns the token, it can access the bus without arbitration. This scheme is effective, in general, with a small number of bus masters.

19.5.3 Bus Bandwidth

An SMP system bus provides the only pathway for multiple processors to access the memory and the I/O devices. Therefore, the bandwidth of the system bus effectively determines the number of processors that can be supported. The simplest system bus design is to allow only one request at a time. A second command cannot be issued before the current bus request completes. This so-called *simple bus* design has very limited bus performance. A split-transaction bus, on the other hand, allows the overlapping of multiple bus requests using the pipelining technique. When a bus transaction is split, it only occupies the bus when the bus is really needed. For example, when a request is at the cache snooping stage or at the stage of accessing memory, the bus is available for other bus transactions. Even when the request is at the data transfer stage, the command/address bus is free to accept another request which does not require the data bus in the same cycle. The split-transaction approach utilizes the critical system bus much better. Most of the modern SMP system buses employ split-transaction designs to maximize the bus bandwidth [Peir et al. 1993, Galles and Williams 1994].

Even with the aggressive split-transaction design, the system bus can support a very limited number of today's high-performance microprocessors. For instance, consider a 50-MHz system bus connecting multiple 100 MIPS processors together. Assume that each processor has on-chip instruction/data caches as well as an external second-level cache; together, the instruction-per-miss rate is 50. In this case, each processor will generate bus traffic for handling two million cache misses every second. As a result, 25 processors will produce enough traffic to use up 100% of the bus bandwidth. This calculation does not allow for the fact that about 30% of the cache lines being replaced are typically modified and have to be written back to memory. In addition, this calculation does not take into account bus traffic due to I/O instructions, cache coherence transactions, etc. All these factors will further limit the number of processors that can be effectively supported by the system bus.

In the above discussion, two ideal conditions are assumed. The first assumption is that the bus utilization can reach 100%. In reality, the rule-of-thumb is that heavy queuing will occur when the bus utilization

reaches above 60%. The second assumption is that each bus request only occupies a single bus cycle. This is very difficult to achieve for a split-transaction bus running at 50 MHz. Typically, both the arbitration stage and the cache snooping stage may require more than one cycle. Even though these two stages do not occupy the bus directly, they need to access other related critical components such as the arbiter and the cache directory, respectively. Realistically, it takes a minimum of two cycles to initiate a new command on the system bus. This two-cycle-per-request design is already very aggressive, because the subsequent command is issued before the current command is acknowledged. In order to avoid potential interference between the active commands, protection hardware must be implemented. In addition, the data bus must be able to transfer a cache line in two cycles. For instance, a 128-bit data bus is required when the Intel Pentium processor is used in the above example, because the Pentium has 32-byte cache lines. Furthermore, any *idle* cycles during the data transfer must be eliminated. This typically requires a high-performance memory system with multiple modules each with independent memory banks.

Current projections suggest that processor performance will continue to improve at over 50% a year [Hennessy and Patterson 1996]. It is unlikely that improvements in the system bus will be able to keep up with the processor curve. Thus, the number of processors that the snooping bus can support is expected to decrease even further. There has been some work to extend the single bus architecture to multiple interleaved buses [Hopper et al. 1989] or hierarchical buses [Wilson 1987]. There have also been proposals to abandon the snooping bus approach and to use a directory method to enforce cache coherence [Lenoski et al. 1992, Gustavson 1992]. The detailed descriptions of these proposals are beyond the scope of this chapter.

19.5.4 Memory Access Latency

When a requested data item is not present in a processor's caches, the item must be fetched from memory. The delay in obtaining the data from memory may stall the requesting processor even when advanced techniques, such as out-of-order execution, nonblocking cache, relaxed consistency model, etc., are used to overlap processor execution with cache misses [Johnson 1990, Farkas and Jouppi 1992, Gharachorloo et al. 1990]. Therefore, it is important to design the system bus to minimize the number of cycles needed to return data upon a cache miss. Techniques such as replicated arbiter and bus parking to reduce arbitration cycles have been described in the subsection on Bus Arbitration above.

Another way to reduce the memory latency is to trigger DRAM access once the command arrives at the memory controller. This early DRAM access may have to be canceled if the requested line turns out to be present in a modified state in another processor's cache. Such a condition will only be known after the acknowledgment. Early DRAM access is very useful because the chance of hitting a modified line in another cache is not very high. In the unlikely case that the requested line is in fact modified in another cache, a cache-to-cache data and ownership transfer can be implemented to send the requested data directly to the requesting processor. The cache-to-cache transfer normally has higher priority and may bypass other requests in the write buffer of the processor that owns the modified line. In some implementations, the memory is also updated with the modified data during the cache-to-cache transfer.

19.5.5 Synchronization and Locking

The basic hardware primitive for implementing synchronization and locking is an atomic read-modify-write instruction such as Test&Set and Compare&Swap. These instructions must be guaranteed to read the contents of a memory location, test and modify the data, and write the result back to the same memory location in an indivisible sequence of operations. In a bus-based SMP system, the Test&Set instruction can be executed in two separate steps: a read-lock operation followed by a write-unlock operation. The read-lock operation reads the memory word and at the same time sets up certain hardware lock signals or registers so that no other requests for the same memory word will be permitted. After the memory word has been modified, the write-unlock operation returns the new result back to the memory location and releases the hardware lock.

There are two ways of implementing the hardware lock on the system bus. The first is to lock the bus completely during the period from the read-lock operation to the write-unlock operation; no other request is allowed in between. This approach has poor performance but may be suitable for a simple bus design which allows only one request at a time anyway. The second approach is to implement **address locking** on the system bus. When a read-lock operation is issued, an invalidation request is sent across the system bus to knock out any copy of the cache line from the other caches. In the meantime, the address of the cache line is recorded in a *lock register* to prevent any snooping on the same cache line until the lock is released by the write-unlock operation. Depending on the data alignment and the size of the synchronization variable, a read-lock operation may have to lock two cache lines at a time.

The address locking method minimizes the interference of a Test&Set instruction with other system bus requests, because only those requests that access the same cache line as the Test&Set will be rejected. Multiple lock requests, each from a different processor, are permitted as long as they target different cache lines. However, the lock request is still relatively expensive because the read-lock operation has to be broadcast to all the other processors and the issuing processor cannot proceed until confirmation is received from each processor.

19.6 Putting It All Together — CCL-XMP System Bus

CCL-XMP [Peir et al. 1993], an Intel Pentium-based SMP system, was designed and developed at the Computer and Communication Laboratories (CCL) of the Industrial Technology Research Institute (ITRI) of Taiwan under a collaborative effort with Acer, ICL, and Intel. The initial target system consists of eight 66-MHz Pentium processors, each with a 256-Kbyte second-level cache. The system bus, operating at 33 MHz, has independent control, command/address, and data buses. The MESI protocol is incorporated to enforce coherence among multiple caches. The system bus can arbitrate and accept one request every two cycles. A dual 64-bit data bus sustains a data transfer rate of over 400 Mbyte/s. Based on odd-even line interleaving, each data bus is connected to one memory module. In order to maximize the bus bandwidth, each memory module is divided into four independent banks. Furthermore, each bank is designed as a two-way interleaved DRAM array to provide zero-wait-state operation in the burst-mode data transfer at 33 MHz. CCL-XMP requires a powerful I/O subsystem. Both VESA local bus and PCI bus can be connected to the system bus through high-performance I/O bridges. The I/O bridges act as both a bus master and slave to transfer data between the system bus and the I/O buses.

The CCL-XMP system bus architecture is very similar to the general SMP bus architecture shown in Figure 19.8. Each request traverses through the same number of stages as described in Section 19.5. In this section, some of the relevant features of the CCL-XMP system bus design are described.

The CCL-XMP system bus is a synchronous bus. All the signals are driven at the rising edge of the clock and latched and sampled at the next rising edge. TTL voltage levels are used, and most signals are tristate with the exception of a few wire-OR control signals. The first version of the motherboard measures 32 cm × 32 cm. The system bus is 15 cm in length and supports eight bus slots. A single 64-bit data bus is included in the first prototype to support up to six Pentium processors. The clock tree is carefully laid out on the motherboard to limit the clock skew to within 2 ns. The total delay between the sender and the receiver through the system bus is less than 28 ns. This includes the delay of latches (flip-flops), the output TTL pad (with 120-pF bus loading), the input pad, boundary scan, clock skews, and other combinational logic. This is tolerable under the 33-MHz clock rate. In designing CCL-XMP, it is more challenging to manage the delay between the system bus interface chip and the second-level cache controller, which is operating at 66 MHz.

Figure 19.10 shows the detailed operation of a line-fill request on the CCL-XMP system bus. Arbitration of the command/address bus takes two cycles. Once the bus is granted, the memory read command and address are issued to the bus in the third cycle. It takes two cycles for all the bus masters (including I/O bridges) to search their cache directories and to update the respective line states upon a hit. At the same time, the memory controller initiates the DRAM access on the arrival of the read command. In the sixth cycle, an acknowledgment is sent back to the requester from each device based on the snooping result.

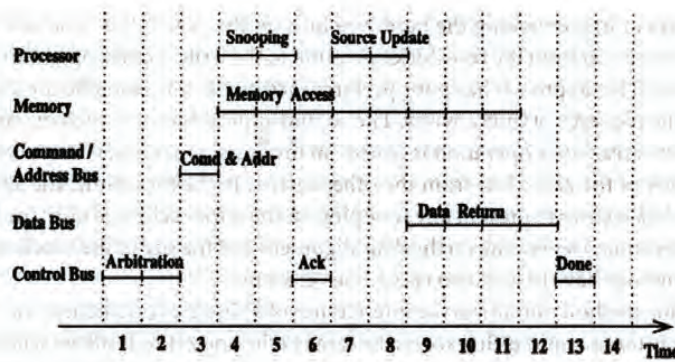


FIGURE 19.10 The pipeline cycles of a line-fill request on the CCL-XMP system bus.

The requester then updates its cache directory according to the acknowledgment and the cache coherence protocol. The memory controller begins arbitrating for the data bus two cycles before the data are fetched out of the DRAM array. A read request is given priority for using the data bus over a write request. After the data bus has been granted, the target 64-bit data is driven onto the bus in the ninth cycle. This is followed by three consecutive cycles to transfer the entire cache line to the requester. Finally, the command completes in the thirteenth cycle.

The CCL-XMP system bus uses replicated arbiters to achieve a two-cycle arbitration. Each arbiter receives n request signals, one from each bus master; but the grant signal is only sent to the local processor to avoid chip-crossing and long wiring delays. Basically, the arbiter latches the requests from all the bus masters at the end of the first cycle. In the second cycle, a resolution of the bus priority is carried out, and the winner is notified by the local arbiter. In addition, the CCL-XMP system bus uses the concept of *arbitration group* to achieve fairness without implementing FIFO queues. An arbitration group consists of the bus masters that are simultaneously requesting access to the system bus. The arbiter will serve all the members in the group according to some priority scheme. Any other master outside the current group is not allowed to join the group until all the requesters in the group have been granted the bus. After that, a new arbitration group can be formed.

Emergency logic is included in the arbiter design to resolve the starvation conditions. Each bus master has an associated *urgent counter*. The counter is incremented when a command receives a negative acknowledgment on the bus. A command can be rejected for several reasons, including resource busy, address locking, or some erroneous conditions. When the counter exceeds a certain threshold, the processor will raise its urgent request control line to the arbiter. The arbiter will then give the highest priority to the urgent requester and activate the emergency logic for resolving any blocking condition for the urgent request until the command is executed successfully and the urgent line is dropped.

The CCL-XMP system bus is a split-transaction bus; it allows the pipelining of multiple requests to sustain maximum bus bandwidth. Figure 19.11 illustrates two consecutive cache line-fill instructions on the command/address bus and the data transfer cycles for the two cache lines on the data buses. These two requests must have originated from different processors, because the Pentium processor only allows one outstanding memory request at a time. When the two requested lines are located in different memory modules, concurrent data transfers are possible. However, the data transfers have to be serialized when both accesses hit the same module. Note that there is a dead cycle on the data bus between the two cache line transfers on the same data bus. This is required to prevent a potential signal conflict. After the read command is issued, it takes six cycles including ECC check for the memory controller to return the target 64-bit data. Because the command bus can accept one command every two cycles, a dual 64-bit bus is designed to balance the performance. The dual data bus can transfer two 32-bytes of data every five bus cycles (including the dead cycle). This provides a sustained bus bandwidth of over 400 Mbyte/s.

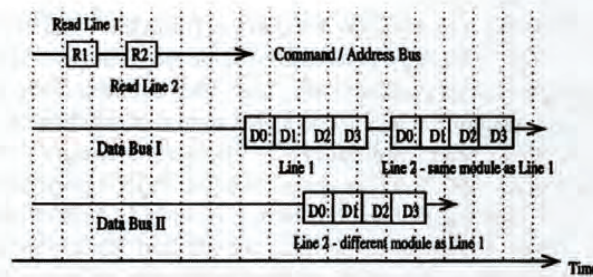


FIGURE 19.11 Split transactions on the CCL-XMP system bus.

CCL-XMP supports cache-to-cache data transfer with memory reflection when a request hits a modified line located in another cache. The memory copy is updated during the data transfer to the requester. The cache-to-cache transfer request has higher priority and may bypass other requests in the write buffer or the replacement writeback queues. This design reduces the penalty in accessing a modified line located in another processor's cache. The update of the memory copy provides the flexibility to transfer the ownership of the requested line to the requester and to switch the requested line to a shared state when there are subsequent read requests for the same line from other processors.

A pair of lock registers are implemented in each cache snooping controller to support address locking. At any given time, each processor can lock two cache lines to implement atomic read-modify-write memory instructions. A request is negative-acknowledged when a snoop hits an address in the lock register. The rejected request will be retried at a later time. Several protection registers are incorporated into the cache snooping controller to protect a cache line in a transient state from being accessed. For example, in addition to the lock registers, a current register (CR) is used to protect the active command until it completes. Also, a writeback register (WBR) records the modified line that is in the process of being written back to memory. As in the case of the lock register, a request will be rejected if it hits these protection registers. Parity bits are added to both the command/address bus and the data bus for detecting transmission errors.

19.7 Historical Perspective and Research Issues

Computer systems based on a shared backplane bus are very popular because they are both cost-effective and highly flexible. The early to mid 1980s saw the establishment of a number of standard buses such as the VME bus [Pri-Tal 1986], Fastbus [Gustavson 1986], Nubus [Taylor 1989], Multibus II [Mahoney 1990], and Futurebus [Borrill 1986]. All these bus architectures have been standardized by the IEEE to allow different vendors to design various boards that can be attached to each bus. Although the option of connecting multiple processors was considered in all these bus designs, only the Futurebus fully supports cache coherence protocols for both writethrough and writeback caches. Other important aspects, such as synchronous/asynchronous communication protocols, bus arbitration, bus bandwidth, locking mechanisms, TTL/BTL/ECL bus interfaces, connectors, pins, etc., are also significantly different among these open bus architectures. A comprehensive comparison can be found in [Borrill 1985].

The continuing search for higher-performance system interconnection has led to three more recent IEEE standards: the IEEE 896.x Futurebus+ [Aichinger 1992], the P1394 High-Performance Serial Bus [Teener 1992], and the P1596 Scalable Coherence Interface (SCI) [Gustavson 1992]. Futurebus+ is an expanded version of the original Futurebus (IEEE 896.1-1987). The work was initiated by the navy in 1988 and later gained widespread support from the working groups of the VMEbus and Multibus as well as many of the major computer companies. The Futurebus+ was designed to be a truly open standard. The standard defines in detail the architectural, electrical, and mechanical specifications. The data path of Futurebus+ varies from 32 to 256 bits wide to provide better performance scalability. The centralized arbitration mechanism is used for its simplicity and high performance. **Live insertion** is adopted to answer the needs of the market

for high-availability and fault-tolerant computers. In addition, Futurebus+ can be reconfigured as an I/O bus. Using the standard "Profile B" bridge, a Futurebus+ can be connected to other open-system buses.

Parallel backplane buses need large physical connectors. Such connectors are usually costly and are the primary source of failure. In addition, a point-to-point, unidirectional link can provide much faster transmission speed. The serial buses were introduced based on these advantages. The P1394 serial bus was first presented at IEEE CompCon 1992. It was designed for low cost, yet it provides the data transmission speed and low latency needed for a peripheral bus or as a possible backup of the backplane bus. The transmission protocol has three layers: transaction, link, and physical. Read, write, and lock are the three transactions supported, and each transaction can be divided into four stages: request, indication, response, and confirmation. The link layer provides a half-duplex data-packet delivery service. Each packet delivery needs to go through three stages: an arbitration stage to gain the access of the physical bus, a packet transmission stage to deliver the packet to the physical layer, and an acknowledgment stage to confirm the transmission with the receiver. The last stage is only required for synchronous package delivery. The split-transaction protocol and a Test&Set instruction are implemented in the P1394 serial bus to enable higher transaction rates and to properly handle locking activities.

As microprocessor performance increased at the rate of over 50% per year [Hennessy and Patterson 1996], it soon became apparent that the use of shared buses as the fundamental interconnection in multiprocessor systems creates a performance bottleneck. In July 1988, the P1596 SCI working group was formed with the aim of defining a scalable interconnect architecture for future shared-memory cache-coherent multiprocessor systems. There are a number of fundamental issues that must be solved in order to achieve this goal. First, signal speed has to be made independent of the size of the system. Second, multiple signal paths (links) have to be used so that multiple independent transfers can take place concurrently. Third, multiple cache coherence activities must be allowed to occur in parallel to overcome the bottleneck in the snooping-bus approach. To resolve the signal-speed problem, SCI uses point-to-point, unidirectional buses with low-voltage differential signals. *Ring* is the most common structure to connect multiple processing nodes (processors) through the fast SCI bus. *Distributed directory* is chosen to maintain cache coherence. The main memory and the directory which records the status of the associated lines in cache are distributed among all the nodes. Instead of maintaining full presence bits to indicate the caches where each line is located, a double link list is constructed to link all the copies of each cache line. Certain cache coherence actions, e.g., invalidation of shared copies of a cache line, need to go through the link list in a sequential fashion.

While all these standardization efforts were under way, a different approach was being taken by Rambus Inc. Rambus Inc. was set up in 1990 to develop technology that will enable systems to keep up with the increasing bandwidth requirements of processors [Rambus 1992]. The core of the Rambus technology is a proprietary chip-to-chip bus, the Rambus Channel. The Rambus Channel consists of a small number of very high-speed lines clocked at an aggressive 250 MHz. Data are transferred on both edges of the clock, allowing nine data lines to achieve a peak transfer rate of 500 Mbyte/s. A block-oriented protocol is used to allow effective utilization of this peak bandwidth. The key to achieving the high data rate lies in paying special attention to the physics of the bus. First, the Rambus interface consists of only 32 carefully terminated transmission lines. The small number of lines helps to reduce bus noise and power consumption, which is significant given the high clock rate. Second, clock-to-data skew is minimized by having a clock-to-master signal and a clock-from-master signal. The appropriate clock is chosen depending on the direction of the data transfer. Third, the channel is designed to operate with low-voltage swings, which further reduces power consumption. Finally, the maximum bus length is limited to about 10 cm and vertical surface-mount packages are used to enable devices to be densely packed onto the bus.

Although flexible and cost-effective, the bus is fundamentally not scalable in performance. This becomes increasingly apparent as processors become faster and SMP designs start pushing the performance limits of buses with fewer and fewer processors. Widening the data bus will not help, because of the bottleneck in the command/address bus. Using multiple buses [Hopper et al. 1989] to achieve more than one transfer at a time results in a complex bus interface design to maintain cache coherence with bus snooping mechanisms. The Scalable Coherence Interface uses a distributed-directory approach to build coherent shared-memory

multiprocessors with high-performance SCI rings. Although the SCI approach allows concurrent coherence activities, the cost of the large directory and the latency of coherence transaction remain serious problems. The Stanford DASH project [Lenoski et al. 1992] advocates the same distributed directory method using mesh-connected networks. Again, the latency in accessing remote memory may cause severe performance degradation [Kuskin et al. 1994]. The "right" way to build a scalable cache-coherent shared-memory multiprocessor remains an active research area.

An issue intimately related to bus performance is the performance of the memory subsystem. The standard RAS/CAS DRAM interface is designed for low cost and high density. In order to achieve sufficient memory bandwidth for today's demanding processors, an interleaved memory subsystem is often required. For SMPs, the memory interleaving is typically very aggressive. Recently, a couple of improvements to the existing DRAM interface have been announced. These include the extended-data-out (EDO) mode and the pipelined burst mode (PBM) [Kumanoya et al. 1995]. In addition, several new DRAM interfaces promising higher bandwidth and lower latency have been proposed. These include the synchronous DRAM, cached DRAM, Rambus DRAM, and others designed specifically for graphics application. The interested reader is referred to [Przybylski 1993, Kumanoya et al. 1995] for more information on these novel DRAM interfaces.

Defining Terms

Address locking: A mechanism to protect a specific memory address so that it can be accessed exclusively by a single processor.

Bus arbitration: The process of determining which competing bus master should be granted control of the bus.

Bus master: A bus device that is capable of initiating and controlling a communication on the bus.

Bus parking: A priority scheme which allows a bus master to gain control of the bus without arbitration.

Bus protocol: The set of rules which define precisely the bus signals that have to be asserted by the master and slave devices in each phase of a bus operation.

Bus slave: A bus device that can only act as a receiver.

Cache coherence protocol: A mechanism to maintain data coherence among multiple caches so that every data access will always return the latest version of that datum in the system.

Cache line: A block of data associated with a cache tag.

Live insertion: The process of inserting devices into a system or removing them from the system while the system is up and running.

Snooping bus: A multiprocessor bus that is continually monitored by the cache controllers to maintain cache coherence.

Split-transaction bus: A bus that overlaps multiple bus transactions, in contrast to the *simple bus* that services one bus request at a time.

Symmetric multiprocessor (SMP): A multiprocessor system where all the processors, memories, and I/O devices are equally accessible without a master-slave relationship.

References

- Aichinger, B. P. 1992. Futurebus+ as an I/O bus: profile B, pp. 300–307. In *Proc. 19th Int. Symp. on Computer Architecture*.
- Archibald, J. and Baer, J. L. 1986. Cache-coherence protocols: evaluation using a multiprocessor simulation model. *ACM Trans. Comput. Systems* 4(4):273–298.
- Agarwal, A., Simoni, R., Hennessy, J., and Horowitz, M. 1988. An evaluation of directory schemes for cache coherence, pp. 280–289. In *Proc. 15th Int. Symp. on Computer Architecture*.
- Borrill, P. 1985. MicroStandards special feature: a comparison of 32-bit buses. *IEEE Micro* 5(6):71–79.
- Borrill, P. 1986. Futurebus: the ultimate in advanced system buses, pp. 210–216. In *Proc. Buscon '86 West*.
- Censier, L. and Feautrier, P. 1978. A new solution to coherence problems in multicache systems. *IEEE Trans. Comput.* C-27(12):1112–1118.

- Chaiken, D., Fields, C., Kurihara, K., and Agarwal, A. 1990. Directory-based cache coherence in large-scale multiprocessors. *IEEE Comput.* 23(6):49–59.
- Farkas, K. and Jouppi, N. 1992. Complexity/performance tradeoffs with non-blocking loads, pp. 211–222. In *Proc. 21st Int. Symp. on Computer Architecture*.
- Galles, M. and Williams, E. 1994. Performance optimizations, implementation, and verification of the SGI challenge multiprocessor, pp. 134–143. In *Proc. 1994 Hawaii Int. Conf. on System Science, Architecture Track*.
- Gharachorloo, K., et. al. 1990. Memory consistency and event ordering in scalable shared-memory multiprocessors, pp. 15–26. In *Proc. 17th Int. Symp. on Computer Architecture*.
- Giacomo, J. D. 1990. *Digital Bus Handbook*. McGraw-Hill, New York.
- Goodman, J. 1983. Using cache memory to reduce processor-memory traffic, pp. 124–131. In *Proc. 10th Int. Symp. on Computer Architecture*.
- Greenley, D. et. al. 1995. Ultrasparc: the next generation superscalar 64-bit SPARC, pp. 442–451. In *Proc. COMPCON'95*.
- Gustavson, D. B. 1984. Computer buses — a tutorial. *IEEE Micro* 4(1):7–22.
- Gustavson, D. B. 1986. Introduction to the Fastbus. *Microprocessors and Microsystems* 10(2):77–85.
- Gustavson, D. B. 1992. The scalable coherent interface and related standards projects. *IEEE Micro* 12(1):10–22.
- Gustavson, D. B. and Theus, J. 1983. Wire-OR logic on transmission lines. *IEEE Micro* 3(3):51–55.
- Hennessy, J. and Patterson, D. 1996. *Computer Architecture, a Quantitative Approach*, 2nd ed. Morgan Kaufmann, San Francisco.
- Hopper, A., Jones, A., and Lioupis, D. 1989. Multiple vs wide shared bus multiprocessors, pp. 300–306. In *Proc. 16th Int. Symp. on Computer Architecture*.
- IBM Corp. 1982. IBM 3081 Functional Characteristics, GA22-7076. IBM Corp., Poughkeepsie, NY.
- Intel Corp. 1994. *Pentium Processor User's Manual*, Vols. 1, 2. Intel Corp. Order nos. 241428, 241429.
- Johnson, M. 1990. *Superscalar Microprocessor Design*. Prentice-Hall, Englewood Cliffs, NJ.
- Kumanoya, M., Ogawa, T., and Inoue, K. 1995. Advances in DRAM interfaces. *IEEE Micro* 15(6):30–36.
- Kuskin, J., et. al. The Stanford FLASH multiprocessor, pp. 302–313. In *Proc. 21st Int. Symp. on Computer Architecture*.
- Lenoski, D. et. al. 1992. The Stanford Dash multiprocessor. *IEEE Comput.* 25(3):63–79.
- Levitan, D., Thomas, T., and Tu, P. 1995. The PowerPC 620 microprocessor: a high performance superscalar RISC microprocessor, pp. 285–291. In *Proc. COMPCON'95*.
- Mahoney, J. 1990. Overview of Multibus II architecture. *SuperMicro J.* No. 4, pp. 58–67.
- Metcalfe, R. M., and Boggs, D. R. 1976. Ethernet: distributed packet switching for local computer networks. *Commun. ACM* 19(7):395–404.
- Peir, J. K., et al. 1993. CCL-XMP: a Pentium-based symmetric multiprocessor system, pp. 545–550. In *Proc. 1993 Int. Conf. on Parallel and Distributed Systems*.
- Pri-Tal, S. 1986. The VME subsystem bus. *IEEE Micro* 6(2):66–71.
- Przybylski, S. 1993. DRAMs for new memory systems. parts 1, 2, 3. *Microprocessor Rep.*, Mar. 8, pp. 18–21.
- Rambus. 1992. *Rambus Architectural Overview*. Rambus, Inc., Mountain View, CA.
- Stenstorm, P. 1990. A survey of cache coherence schemes for multiprocessors. *IEEE Comput.* 23(6):12–25.
- Sweazey, P. and Smith, A. J. 1986. A class of compatible cache consistency protocols and their support by IEEE Futurebus, pp. 414–423. In *Proc. 13th Int. Symp. on Computer Architecture*.
- Taub, D. M. 1983a. Overcoming the effects of spurious pulses on wired-OR lines in computer bus systems. *Electron. Lett.* 19(9):340–341.
- Taub, D. M. 1983b. Limitations of looped-line scheme for overcoming wired-OR glitch effects. *Electron. Lett.* 19(15):579–580.
- Taub, D. M. 1984. Arbitration and control acquisition in the proposed IEEE 896 FutureBus. *IEEE Micro* 4(4):28–41.
- Taylor, B. G. 1989. Developing for the Macintosh NuBus, pp. 143–175. In *Proc. Eurobus/UK Conference*.

- Teener, M. 1992. A bus on a diet — the serial bus alternative: an introduction to the P1394 high performance serial bus, pp. 316–321. In *Compcon '92*.
- Wilson, A. 1987. Hierarchical cache/bus architecture for shared memory multiprocessors, pp. 244–252. In *Proc. 14th Int. Symp. on Computer Architecture*.
- Zalewski, J. 1995. *Advanced Multimicroprocessor Bus Architecture*. IEEE Computer Society Press.

Further Information

Advanced Multimicroprocessor Bus Architectures by J. Zalewski [Zalewski 1995] contains a comprehensive collection of papers covering bus basics, physics, arbitration and protocols, board and interface designs, cache coherence, various standard bus architectures, and their performance evaluations. The bibliography section at the end provides a complete list of references for each of the topics discussed in the book.

Digital Bus Handbook by J. D. Giacomo [Giacomo 1990] is a good source of information for the details of the various standard bus architectures. In addition, this book has several chapters devoted to the electrical and mechanical issues in bus design.

The bimonthly journal *IEEE Micro* and the *Proceedings of International Symposium on Computer Architecture* are good sources of the latest papers in computer architecture area including computer buses.