# DECLARATION OF JUNE ANN MUNFORD

1. My name is June Ann Munford. I am over the age of 18, have personal knowledge of the facts set forth herein, and am competent to testify to the same.

2. I have been retained on behalf of the Petitioner to provide my expertise in the above-illustrated matter. I am being compensated for my services in this matter at my customary rate of $200.00 per hour plus reasonable expenses. My statements are objective, and my compensation does not depend on the outcome of this matter.

3. I declare under penalty of perjury that the foregoing is true and correct. I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code.

4. I earned a Master of Library and Information Science (MLIS) from the University of Wisconsin-Milwaukee in 2009. I have over ten years of experience in the library/information science field. Beginning in 2004, I



Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000002

have served in various positions in the public library sector including Assistant Librarian, Youth Services Librarian and Library Director. I have attached my Curriculum Vitae as Appendix CV.

5. During my career in the library profession, I have been responsible for materials acquisition for multiple libraries. In that position, I have cataloged, purchased and processed incoming library works. That includes purchasing materials directly from vendors, recording publishing data from the material in question, creating detailed material records for library catalogs and physically preparing that material for circulation. In addition to my experience in acquisitions, I was also responsible for analyzing large collections of library materials, tailoring library records for optimal catalog search performance and creating lending agreements between libraries during my time as a Library Director.

6. I am fully familiar with the catalog record creation process in the library sector. In preparing a material for public availability, a library catalog record describing that material would be created. These records are typically written in Machine Readable Catalog (herein referred to as "MARC") code and contain information such as a physical description of the material, metadata from the material's publisher, and date of library acquisition. In



Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000003

particular, the 008 field of the MARC record is reserved for denoting the date of creation of the library record itself. As this typically occurs during the process of preparing materials for public access, it is my experience that an item's MARC record indicates the date of an item's public availability.

7. Typically, in creating a MARC record, a librarian would gather various bits of metadata such as book title, publisher and subject headings among others and assign each value to a relevant numerical field. For example, a book's physical description is tracked in field 300 while title/attribution is tracked in field 245. The 008 field of the MARC record is reserved for denoting the creation of the library record itself. As this is the only date reflecting the inclusion of said materials within the library's collection, it is my experience from over 10 years of working with libraries across the country that an item's 008 field accurately indicates the date of an item's public availability within the library that created the record.

8. I have reviewed Exhibit 1006, *Memory Systems and Pipelined Processors* by Harvey G. Cragon.

9. Attached hereto as Appendix CRAGON01 is a true and correct copy of the MARC record for *Memory Systems and Pipelined Processors* as held by the



Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000004

Penn State University library. I secured this record myself from the library's public catalog. The MARC record contained within Appendix CRAGON01 accurately describes the title, author, publisher, and ISBN number of *Memory Systems and Pipelined Processors.*

10. Attached hereto as Appendix CRAGON02 is a true and correct copy of *Memory Systems and Pipelined Processors* as held by the Penn State University library. I secured these scans myself from the library's collection. In comparing Exhibit 1006 to Appendix CRAGON02, it is my determination that Exhibit 1006 is a true and correct copy of *Memory Systems and Pipelined Processors* by Harvey G. Cragon.

11. To secure these scans, I first searched for the book on Worldcat.org, a web resource that allows users to search the collections of multiple libraries. I used Worldcat to determine that I wanted to pursue a copy of *Memory Systems and Pipelined Processors* held by Penn State University Library, a place where an authentic copy of the book was likely to be found. I navigated to the Penn State University Library catalog, where I found the book by searching for its title and obtained the MARC record presented in this declaration as Appendix CRAGON01. I reviewed the MARC record from the Penn State University Library catalog, which acknowledged that



Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000005

the library's copy had been held for over 20 years, since December 18, 1997. After requesting the book, I traveled to the library's Greater Allegheny branch and secured Penn State University's copy of *Memory Systems and Pipelined Processors* using my resident borrower account. Upon inspecting Penn State's copy, I noted that the book was in a condition that raised no doubt as to its authenticity. A university library is precisely the place where one would expect a textbook such as *Memory Systems and Pipelined Processors* to be found. Indeed, the Penn State University Library is a place where an authentic copy of the book was likely to be found because Penn State University has a well-regarded library that catalogs and maintains copies of numerous textbooks such as *Memory Systems and Pipelined Processors*. I used this copy from the Penn State University Library to create the scans presented in the declaration as Appendix CRAGON02.

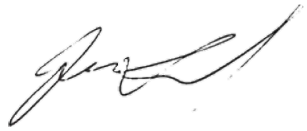
12. The 008 field of the MARC record in Appendix CRAGON01 indicates the date of record creation. The 008 field of Appendix CRAGON01 indicates the Penn State University library first acquired *Memory Systems and Pipelined Processors* as of December 18, 1997. This date is consistent with the publication date from the copyright page of *Memory Systems and Pipelined Processors*, which report a publication date of 1996. Considering



Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000006

this information, it is my determination that *Memory Systems and Pipelined Processors* was made available to the public by the Penn State University Library shortly after its initial acquisition as of December 18, 1997.

Dated: 10/23/2024



June Ann Munford



Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000007

# APPENDIX CV

June A. Munford
Curriculum Vitae

## Education

University of Wisconsin-Milwaukee - MS, Library & Information Science, 2009
Milwaukee, WI

- Coursework included cataloging, metadata, data analysis, library systems, management strategies and collection development.
- Specialized in library advocacy, cataloging and public administration.

Grand Valley State University - BA, English Language & Literature, 2008
Allendale, MI

- Coursework included linguistics, documentation and literary analysis.
- Minor in political science with a focus in local-level economics and government.

## Professional Experience

Researcher / Expert Witness, October 2017 – present
Freelance ● Pittsburgh, Pennsylvania & Grand Rapids, Michigan

- Material authentication and public accessibility determination. Declarations of authenticity and/or public accessibility provided upon research completion. Experienced with appeals and deposition process.

- Research provided on topics of public library operations, material publication history, digital database services and legacy web resources.

- Past clients include Alston & Bird, Arnold & Porter, Baker Botts, Fish & Richardson, Erise IP, Irell & Manella, O'Melveny & Myers, Perkins-Coie, Pillsbury Winthrop Shaw Pittman and Slayden Grubert Beard.

Library Director, February 2013 - March 2015
Dowagiac District Library ● Dowagiac, Michigan

- Executive administrator of the Dowagiac District Library. Located in

Southwest Michigan, this library has a service area of 13,000, an annual operating budget of over $400,000 and total assets of approximately $1,300,000.

● Developed careful budgeting guidelines to produce a 15% surplus during the 2013-2014 & 2014-2015 fiscal years while being audited.

● Using this budget surplus, oversaw significant library investments including the purchase of property for a future building site, demolition of existing buildings and building renovation projects on the current facility.

● Led the organization and digitization of the library's archival records.

● Served as the public representative for the library, developing business relationships with local school, museum and tribal government entities.

● Developed an objective-based analysis system for measuring library services - including a full collection analysis of the library's 50,000+ circulating items and their records.

November 2010 - January 2013
Librarian & Branch Manager, Anchorage Public Library ● Anchorage, Alaska

● Headed the 2013 Anchorage Reads community reading campaign including event planning, staging public performances and creating marketing materials for mass distribution.

● Co-led the social media department of the library's marketing team, drafting social media guidelines, creating original content and instituting long-term planning via content calendars.

● Developed business relationships with The Boys & Girls Club, Anchorage School District and the US Army to establish summer reading programs for children.

June 2004 - September 2005, September 2006 - October 2013
Library Assistant, Hart Area Public Library
Hart, MI

● Responsible for verifying imported MARC records and original MARC

cataloging for the local-level collection as well as the Michigan Electronic Library.

● Handled OCLC Worldcat interlibrary loan requests & fulfillment via ongoing communication with lending libraries.

**Professional Involvement**

Alaska Library Association - Anchorage Chapter
- Treasurer, 2012

Library Of Michigan
- Level VII Certification, 2008
- Level II Certification, 2013

Michigan Library Association Annual Conference 2014
- New Directors Conference Panel Member

Southwest Michigan Library Cooperative
- Represented the Dowagiac District Library, 2013-2015

**Professional Development**

Library Of Michigan Beginning Workshop, May 2008
Petoskey, MI
- Received training in cataloging, local history, collection management, children's literacy and reference service.

Public Library Association Intensive Library Management Training, October 2011
Nashville, TN
- Attended a five-day workshop focused on strategic planning, staff management, statistical analysis, collections and cataloging theory.

Alaska Library Association Annual Conference 2012 - Fairbanks, February 2012
Fairbanks, AK
- Attended seminars on EBSCO advanced search methods, budgeting, cataloging, database usage and marketing.

**Depositions**

2019 ● Fish & Richardson

Apple v. Qualcomm (IPR2018-001281, 39521-00421IP, IPR2018-01282 and 39521-00421IP2)

2019 ● Erise IP

Implicit, LLC v. Netscout Systems, Inc (Civil Action No. 2:18-cv-53-JRG)

2019 ● Perkins-Coie

Adobe Inc. v. RAH Color Technologies LLC (Cases IPR2019-00627, IPR2019-00628, IPR2019-00629 and IPR2019-00646)

2020 ● O'Melveny & Myers

Maxell, Ltd. v. Apple Inc. (Case No. 5:19-cv-00036-RWS)

2021 ● Pillsbury Winthrop Shaw Pittman LLP

Intel v. SRC (IPR2020-1449)

2022 ● Perkins-Coie

Realtek v. Future Link (IPR2021-01182)

2023 ● Fish & Richardson

Neuroderm Ltd. v. Abbvie, Inc (Case No. PGR2022-00040)

2023 ● Fish & Richardson

Nearmap US Inc. v. Pictometry International Corp. (IPR2022-00735)

2023 ● Fish & Richardson

Samsung Electronics v. MemoryWeb LLC (Case No. 39843-0136PS1)

2023 ● Pillsbury Winthrop Shaw Whitman LLP

Gravel Rating Systems v. Costco Wholesale Corp. (Civil Action No. 4:21-cv-149-ALM)

2024 ● Willkie-Farr

Netflix, Inc. v. VideoLabs. Inc. (IPR2023-00628)

**Limited Case and Clientele History**

Alston & Bird

- Ericsson

  v. Collision Communications (IPR2022-01233)

- Nokia

  v. Neptune Subsea, Xtera (Case No. 1:17-cv-01876)

- Universal Electronics Inc

  v. Roku Inc (IPR2022-00818)

Arnold & Porter

- Ivantis

  v. Glaukos (Case No. 8:18-cv-00620)

- Samsung

  v. Jawbone (Case No. 2:21-cv-00186)

Benesch Friedlander Coplan & Aronoff

- Voyis

  v. Cathx (Case No. 5:21-cv-00077-RWS)

Deschert LLP

- Smaxtec Animal Care

  v. ST Reproductive Technologies, LLC (IPR2024-00885)

Erise I.P.

- Apple

  v. Ericsson Inc. (IPR2022-00715)

  v. Future Link Systems (IPRs 6317804, 6622108, 6807505, and 7917680)

  v. INVT (Case No. 20-1881)

  v. Navblazer LLC (IPR2020-01253)

  v. Qualcomm (IPR2018-001281, 39521-00421IP, IPR2018-01282, 39521-00421IP2)

  v. Quest Nettech Corp (Case No. 2:19-cv-00118-JRG)

  v. Telefonaktiebolaget LM Ericsson (IPR2022-00275)

  v. Theta IP, LLC (IPR2024-00818)

- Fanduel

  v. CGT (Case No. 19-1393)

- Garmin

  v. Phillips North America LLC (Case No. 2:19-cv-6301-AB-KS)

- Netscout
    - v. Longhorn HD LLC (Case No. 2:20-cv-00349)
    - v. Implicit, LLC (Civil Action No. 2:18-cv-53-JRG)

- Sony Interactive Entertainment LLC
    - v. Bot M8 LLC (IPR2020-01288)
    - v. Infernal Technology LLC (Case No. 2:19-CV-00248-JRG)

- Unified Patents
    - v. GE Video Compression (Civil Action No. 2:19-cv-248)

Fish & Richardson

- Apple
    - v. AliveCor (3:21-cv-03958)
    - v. LBS Innovations (Case No. 2:19-cv-00119-JRG-RSP)
    - v. Koss Corporation (IPR2021-00305)
    - v. Masimo (IPR 50095-0012IP1, 50095-0012IP2, 50095-0013IP1, 50095-0013IP2, 50095-0006IP1, 50095-0135IP1)
    - v. Neonode (Case No. 21-cv-08872-EMC)
    - v. Qualcomm (IPR2018-001281, 39521-00421IP, IPR2018-01282, 39521-00421IP2)

- Dell
    - v. Neo Wireless (IPR2022-00616)

- Dish Network
    - v. Entropic Communications, LLC (Case No. 2:2023-CV-01043)

v. Realtime Adaptive Streaming (Case No. 1:17-CV-02097-RBJ)

v. TQ Delta LLC (Case No. 18-1798)

- Evapco Dry Cooling

  v. SPG Dry Cooling (IPR2021-00688)

- Genetec

  v. Sensormatic Electronics (Case No. 1:20-CV-00760)

- Huawei

  v. Bell Northern Research LLC (IPR2019-01174)

- Kianxis

  v. Blue Yonder (Case No. 3:20-cv-03636)

- LG Electronics

  v. Bell Northern Research LLC (Case No. 3:18-cv-2864-CAB-BLM)

- Metaswitch

  v. Sonus Networks (IPR2018-01719)

- Microsoft

  v. Throughputer Inc (IPR2022-00757)

- Mom Enterprises

  v. Ddrops Company (Case No. 1:22-cv-00332)

- MLC Intellectual Property

v. MicronTech (Case No. 3:14-cv-03657-SI)

- Nearmap Inc
  - v. EagleView Technologies (IPR2022-01009)

- Neuroderm Ltd.
  - v. Abbvie, Inc (Case No. PGR2022-00040)

- Realtek Semiconductor
  - v. Future Link (IPR2021-01182)

- Quectel
  - v. Koninklijke Philips (Case No. 1:20-cv-01710)

- Samsung
  - v. Aire Technology (IPR2022-00877)
  - v. Bell Northern Research (Case No. 2:19-cv-00286-JRG)
  - v. Communication Technologies Inc (IPR2022-01221)
  - v. Jawbone Innovations (IPR2022-00865)
  - v. MemoryWeb LLC (IPR2022-00885)
  - v. Dodots Licensing Solutions (IPR2023-00701)
  - v. Telefonaktiebolaget LM Ericsson (IPR2021-00615)

- Texas Instruments
  - v. Vantage Micro LLC (IPR2020-01261)

- Xilinx
  - v. Sentient Sensors LCC (Case No. 1:22-cv-00173)

Groombridge Wu

- Nearmap

  v. Eagle View Technologies Inc. and Pictometry International Corp. (Case No. 2:21-cv-00283-TS-DAO)

Irell & Manella

- Curium

O'Melveny & Myers

- Apple

  v. Maxell (Case No. 5:19-cv-00036-RWS)

- Disney Media & Entertainment Distribution LLC

  v. Digital Media Technology Holdings, LLC (Case No. 2:22-cv-01642)

- Micron Technology Inc.

  v. Besang Inc (IPR2023-00900)

- Samsung

  v. Daedalus Prime LLC (1335 Investigation)

Perkins-Coie

- Amazon

  v. Nokia (IPR2024-00847 and IPR2024-00848)

- Heru Industries

  v. The UAB Research Foundation (IPR2022-01148)

- Intel Corporation

  v. BeSang Inc. (IPR2023-00991)

  v. BitMICRO (Case No. 5:23-cv-00625)

  v. Telefonaktiebolaget L M Ericsson (IPR2024-00610)

- Realtek Semiconductor

  v. Future Link (IPR2021-01182)

- Twitter Inc

  v. VOIP-Pal.com (Case No. 3:20-cv-02397-JD)

- TCL Industries

  v. Koninklijke Philips NV (IPR2021-00495, IPR2021-00496 and IPR2021-00497)

Pillsbury Winthrop Shaw Pittman

- Intel

  v. FG SRC LLC (Case No. 6:20-cv-00315)

- Gravel Rating Systems

  v. Costco (Case No. 4:21-cv-149)
  v. Lowe's Home Centers (Case No. 4:21-cv-150)
  v. T-Mobile USA (Case No. 4:21-cv-152)
  v. Kohl's Inc. (Case No. 4:21-cv-258)
  v. Under Armor (Case No. 4:21-cv-356)

Quinn Emanuel

- ServiceNow, Inc.

  v. InQuisient, Inc. (Case No. 22-900-CJB)

- Wolfspeed

  v. The Trustees of Purdue University (Case No. 1:2021-cv-00840)

Sheppard, Mullin, Richter & Hampton LLP

- Cadence Design Systems Inc.

  v. Semiconductor Design Technologies (Case No. 3:23-cv-01001)

Stern, Kessler, Goldstein & Fox

- Preliminary Research

Willkie, Farr & Gallagher LLP

- Neurent

  v. The Foundry, LLC (IPR2024-00669)

- Netflix, Inc.

  v. VideoLabs, Inc. (IPR2023-00628)

# APPENDIX CRAGON01

# MARC View

| | | | |
|---|---|---|---|
| LEADER 01132nam a2200325 a 4500 | | | |
| 001 1464467 | | | |
| 003 SIRSI | | | |
| 005 20151215031026.0 | | | |
| 008 971218t19961996xxua b 000 0 eng d | | | |
| 010 | | | a\| 95022155 |
| 019 | | | a\| MARS |
| 020 | | | a\| 0867204745 |
| 035 | | | a\| LIAS2055796 |
| 035 | | | a\| (OCoLC)32626326 |
| 040 | | | c\| PSt d\| WaOLN d\| UtOrBLW |
| 050 | 0 | 0 | a\| QA76.5 b\| .C68 1996 |
| 100 | 1 | | a\| Cragon, Harvey G. |
| 245 | 1 | 0 | a\| Memory systems and pipelined processors / c\| Harvey G. Cragon. |
| 264 | | 1 | a\| Sudbury, Mass. : b\| Jones and Bartlett, c\| [1996] |
| 264 | | 4 | c\| ©1996 |
| 300 | | | a\| xv, 575 pages : b\| illustrations ; c\| 25 cm. |
| 336 | | | a\| text b\| txt 2\| rdacontent |
| 337 | | | a\| unmediated b\| n 2\| rdamedia |
| 338 | | | a\| volume b\| nc 2\| rdacarrier |
| 490 | 1 | | a\| Jones and Bartlett books in computer science |
| 504 | | | a\| Includes bibliographical references (pages 541-566) and indexes. |
| 650 | | 0 | a\| Computers, Pipeline. |
| 650 | | 0 | a\| Computer storage devices. |
| 830 | | 0 | a\| Jones and Bartlett books in computer science. |
| 949 | | | a\| QA76.5.C68 1996 w\| LC c\| 1 i\| 000032495621 d\| 4/9/2024 e\| 1/19/2022 l\| CATO-2 m\| UP-ANNEX n\| 5 q\| 1 r\| Y s\| Y t\| BOOK u\| 4/13/2001 z\| 6202 |

ASK A LIBRARIAN

PennState

Copyright @2019 The Pennsylvania State University. All rights reserved. Except where otherwise noted, this work is subject to a Creative Commons Attribution 4.0 license. Details and exceptions.

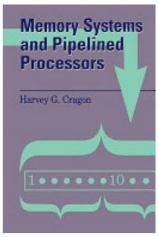
Legal Statements | PSU Hotlines

PENN STATE UNIVERSITY LIBRARIES

Libraries Home (814) 865-6368
Libraries Intranet (Staff Only)
Accessibility Help
Website Feedback
Policies and Guidelines
Staff Directory

CONNECT WITH PENN STATE UNIVERSITY LIBRARIES

Facebook
Twitter
Instagram




# Memory systems and pipelined processors / Harvey G. Cragon

| | |
|---|---|
| Author | Cragon, Harvey G. |
| Published | Sudbury, Mass. : Jones and Bartlett, [1996] |
| Copyright Date | ©1996 |
| Physical Description | xv, 575 pages : illustrations ; 25 cm. |

ASK A LIBRARIAN

Search inside at Google Books

## Availability

Browse Nearby on Shelf I Want It

### Library Storage (1 item)

| Call number | Material | Location |
|---|---|---|
| QA76.5.C68 1996 | Book | Being transferred between libraries |

**Series**

Jones and Bartlett books in computer science

**Subject(s)**

Computers, Pipeline

Computer storage devices

**ISBN**

0867204745

**Bibliography Note**

Includes bibliographical references (pages 541-566) and indexes.

View MARC record | catkey: 1464467

PennState

Copyright @2019 The Pennsylvania State University. All rights reserved. Except where otherwise noted, this work is subject to a Creative Commons Attribution 4.0 license. Details and exceptions.

Legal Statements | PSU Hotlines

PENN STATE UNIVERSITY LIBRARIES

Libraries Home (814) 865-6368

Libraries Intranet (Staff Only)

Accessibility Help

Website Feedback

Policies and Guidelines

Staff Directory

CONNECT WITH PENN STATE UNIVERSITY LIBRARIES

Facebook

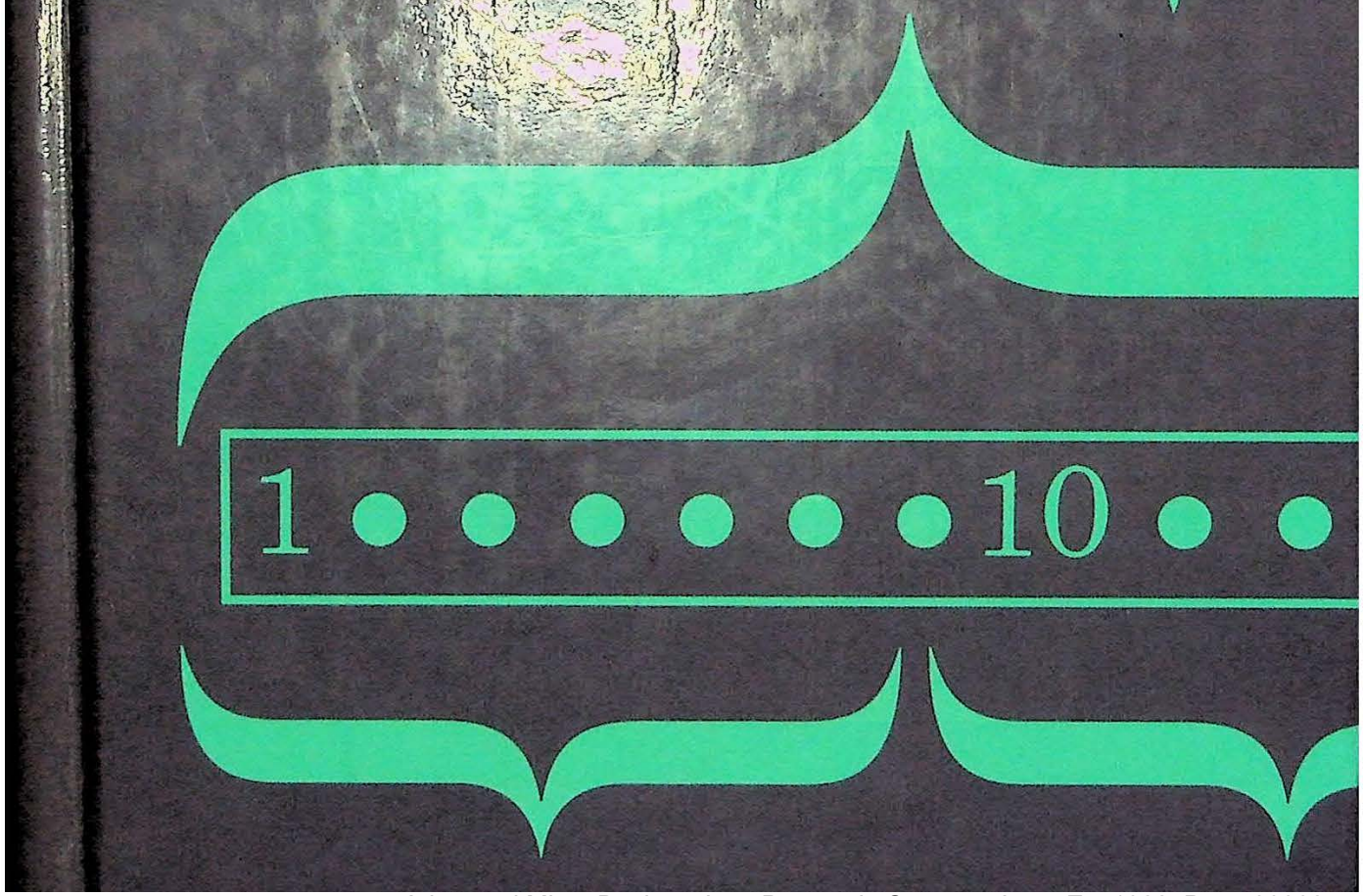
Twitter

Instagram

# APPENDIX CRAGON02

# Memory Systems and Pipelined Processors

Harvey G. Cragon




Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000025

# Memory Systems and Pipelined Processors

Harvey G. Cragon

The current widespread demand for high performance personal computers and workstations has resulted in a renaissance of computer design. To meet the challenge that this presents to students and professional computer architects, this graduate level text offers an in-depth treatment of the implementation details of memory systems and pipelined processors, the "microarchitecture" of modern computers and microprocessors.

The text explores techniques for solving the design problems inherent in computers with high levels of concurrency, such as the demand for a memory system with low latency and high bandwidth, branching, providing precise interrupts, managing dependencies and insuring coherency. Additionally, it examines the difficulties presented by virtual memory in high performance computers. As a thorough compendium of both historical and contemporary implementation techniques, this is an essential sourcebook for computer architecture students and practicing professionals.

## Features

- Text is ideal for a one-semester course
- Provides references to the original ideas that allow readers to relevant literature
- Offers a complete treatment of the ways that returns can be implemented
- Includes a chapter on diversity of addressing

## About the Author

Harvey G. Cragon has held the Ernest Cockrell, Jr. Centennial Chair in Engineering at the University of Texas at Austin since 1984. He has over 30 years of industrial experience in architectural design and implementation of digital computers, including 25 years with Texas Instruments. Mr. Cragon's current research interests are in computer architecture design methodology and high-speed computers.

## Other Titles of Interest

*Computer Architecture: Pipelined and Parallel Processor Design*, Michael J. Flynn

*Algorithms and Data Structures in Computer Engineering*, E. Stewart Lee

ISBN 0-86720-474-5



Jones and Bartlett Publishers
One Exeter Plaza
Boston, MA 02116
800-832-0034 • 617-859-3900

PENN STATE UNIVERSITY LIBRARIES
A000032495621



9 780867 204742 >

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000026

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000027

6202

# AXIOMS FOR IMPLEMENTATION

Computer designers strive to provide the best performance consistent with cost objectives. I have collected a few rules for implementation that are called *Axioms of Implementation* and are listed below; the order does not signify relative importance.

**1** Functions that consume the most time should be the object of effort to speedup the total process while recognizing that the ultimate speedup is limited by the function(s) that cannot be speeded up.

{ "This axiom is a restatement of Amdahl's observation that "effort expended on achieving high parallel processing rate is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude."

**2** Spatial and/or temporal concurrency in hardware can improve performance.

{ Spatial concurrency is found in split caches, superscalar implementations that issue more than one instruction in a clock, multiple I/O channels, and the like. Temporal concurrency is found, for example, in pipelined processors, pipelined caches, and concurrent input/compute/output scheduling.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000028

ENGINEERING LIBRARY

3 Speculative operations, performed until a choice of path can be determined, will in most cases improve performance.

Examples are found in late select caches, speculative execution of instructions while waiting for a branch to be resolved, and concurrent speculative address translation in a virtual memory system. Optimistic synchronization is another example of a speculative operation.

4 With concurrency and speculative operations, special control and record-keeping structures are required for undoing operations that are not continued and must be abandoned.

Providing precise interrupts in a pipelined processor and undoing early state changes of unused speculatively executed instructions are examples.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000029

*Editorial, Sales, and Customer Service Offices*
Jones and Bartlett Publishers
One Exeter Plaza
Boston, MA 02116
1-800-832-0034
617-859-3900

Jones and Bartlett Publishers International
7 Melrose Terrace
London W6 7RL
England

Copyright © 1996 by Jones and Bartlett Publishers, Inc.

All rights reserved. No part of the material protected by this copyright notice may be reproduced or utilized in any form, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without written permission from the copyright owner.

Trademarked products mentioned in this book are listed on page 576.

**Library of Congress Cataloging-in-Publication Data**

Cragon, Harvey G.
Memory systems and pipelined processors / Harvey G. Cragon.
p. cm.
Includes bibliographical references and index.
ISBN 0-86720-474-5
1. Computers, Pipeline. 2. Computer storage devices. I. Title.
QA76.5.C68 1995
004.25'6--dc20 95-22155
CIP

*Acquistions Editor:* Carl Hesler
*Production Editor:* Anne S. Noonan
*Manufacturing Buyer:* Dana L. Cerrito
*Editorial Production Service:* Superscript Editorial Production Services
*Typesetting:* Scanway Graphics International
*Printing and Binding:* Braun-Brumfield
*Cover Design:* Beth Santos
*Cover Printing:* John P. Pow Company

Printed in the United States of America
99 98 97 96 95 10 9 8 7 6 5 4 3 2 1

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000030

# Contents

Preface xi

*1* Memory Systems 1

**1.0** Overview 1
**1.1** Memory Latency 4
**1.1.1** Hierarchical Memory Systems 5
**1.1.2** Memory Efficieny 7
**1.2** Memory Bandwidth 10
**1.3** Queues 13
**1.4** Summary 14

*2* Caches 15

**2.0** Overview 15
**2.1** Cache Organization and Placement Policies 19
**2.1.1** Set Index Allocation 27
**2.1.2** Late/Early Select Caches 29
**2.1.3** Control Bits: Tags, Valid, Dirty, Shared and LRU 31
**2.1.4** Nonblocking Caches 32
**2.2** Read Policies and Performance Models 34
**2.2.1** Transport Time 37
**2.2.2** Cache Miss Characteristics 39
**2.2.3** Sources of Miss and Hit Data 41
**2.2.4** Fetch Policies 42



Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000031

**2.2.5** Write Policies 45
**2.2.6** Replacement Policy 54
**2.2.7** Block Size Considerations 56
**2.2.8** Bus Design Considerations 57
**2.3** Instruction Buffering Schemes 60
**2.3.1** Instruction Buffers 61
**2.3.2** Instruction Queues 62
**2.3.3** Instruction Caches 68
**2.3.4** Published Instruction Caches Hit Data 69
**2.3.5** Contemporary Instruction Caches 72
**2.4** Data Caches 72
**2.4.1** Published Data Cache Miss Data 73
**2.4.2** Contemporary Data Caches 73
**2.4.3** Data Queues 78
**2.5** Unified Caches 78
**2.5.1** Published Performance Data 81
**2.5.2** Contemporary Unified Caches 82
**2.6** Split and Multilevel Caches 82
**2.7** Multilevel Caches 91
**2.7.1** Published Performance Data 94
**2.8** Other Cache Considerations 95
**2.8.1** Processor Control 95
**2.8.2** Transport Time Hiding or Reduction Methods 97
**2.8.3** Compiler Optimization 107
**2.8.4** Pipelined Caches 107

***3*** Virtual Memory 109

**3.0** Overview 109
**3.1** Virtual Memory Concepts 113
**3.2** Memory Mapping 117
**3.2.1** Creating Large Virtual Addresses 118
**3.2.2** One-Level Map Address Extension 118
**3.2.3** Register Extensions 120
**3.2.4** Summary 120
**3.3** Virtual Memory Organizations 121
**3.3.1** Page Allocation Systems 123
**3.3.2** Segmented Systems 153
**3.3.3** Paged Segmentation 156
**3.4** Translation Lookaside Buffers 162
**3.4.1** TLB Organization 165
**3.4.2** TLB Miss Ratio Data 174
**3.4.3** TLB Impact on Virtual Memory Performance 177
**3.4.4** Instruction Support for TLBs 177

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000032

**3.5** Virtual Memory Accessing Rules 178
**3.5.1** Read Accesses 178
**3.5.2** Allocation 180
**3.5.3** Write Accesses 184
**3.5.4** Location of Tables 184
**3.5.5** Table Walking Methods 185
**3.5.6** Instruction Set Support for Virtual Memory 187
**3.5.7** Memory Access Control 189
**3.5.8** Choice of Page Size 193
**3.5.9** Addressing the Disk 195
**3.6** Virtual Memory Performance Issues and Models 197
**3.6.1** Published Page Fault Ratio Data 199

***4*** Memory Addressing and I/O Coherency 201

**4.0** Overview 201
**4.1** Cache Addressing, Virtual or Real 203
**4.1.1** Real Address Caches 204
**4.1.2** Pipelined Real Caches 206
**4.1.3** Virtual Address Cache 209
**4.1.4** Restricted Virtual Caches 212
**4.1.5** Summary 213
**4.1.6** TLB Addressing 215
**4.2** I/O System Addressing Design Issues 215
**4.2.1** Cache Coherency 217
**4.2.2** Snoopy Controllers 220
**4.2.3** DMA I/O Configurations 222
**4.2.4** Processor Control Over Snoopy Controllers 225
**4.3** Other Considerations 227
**4.3.1** Memory-Mapped I/O 227
**4.3.2** Cache Coherency Protocols 227
**4.3.3** Snooping on Write Queues 229
**4.3.4** Summary 230

***5*** Interleaved Memory and Disk Systems 231

**5.0** Overview 231
**5.1** Interleaved Memory 231
**5.1.1** Performance Models 234
**5.1.2** Reducing the Effect of Strides≠ 1 237
**5.1.3** Superinterleaving 241
**5.1.4** Interleaving for Multiport Access 248
**5.1.5** Cache-Memory Transfers 249
**5.1.6** Nibble, Page, and Static Column Mode Drams 250

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000033

**5.1.7** Wide-Word Interleaved Modules 251
**5.2** Disk Systems 252
**5.2.1** Disk Requirements 254
**5.2.2** Disks in File I/O and Virtual Memory 265
**5.2.3** Disk Arrays 272
**5.2.4** Disk Caches 283

***6*** Pipelined Processors 287

**6.0** Overview 287
**6.1** Pipeline Performance Models 290
**6.2** Pipeline Design Considerations 294
**6.3** Other Pipeline Performance Models 298
**6.3.1** Pipeline Efficiency 300
**6.4** Pipeline Partitioning 301
**6.5** Contemporary Pipelined Microprocessors 305
**6.6** Exclusion or Synchronization Instructions 306
**6.6.1** Processors That Address Memory 309
**6.6.2** Load/Store Processors 311
**6.6.3** Synchronization Summary 315

***7*** Branching 317

**7.0** Overview 317
**7.1** Modeling Technique 318
**7.2** Pipeline Freeze Strategies 319
**7.3** Prediction Strategies 321
**7.3.1** Static Prediction Strategies 324
**7.3.2** Semistatic Prediction Strategies 326
**7.3.3** Dynamic Prediction Strategies 327
**7.4** Instruction Sequence Alteration Strategies 339
**7.4.1** Delayed Branch Strategy 339
**7.4.2** Delayed Branch with Abandon 341
**7.4.3** Delayed Branch with Fast Compare 343
**7.4.4** Branch Spreading with Condition Codes 343
**7.5** Fetch Multiple-Paths Strategies 344
**7.5.1** Fetch Two Paths 345
**7.5.2** Fetch All Paths 347
**7.5.3** Hardware Requirements 349

***8*** Dependencies 351

**8.0** Overview 351
**8.1** Instruction Issue and Release Techniques 355

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000034

**8.2** Data Register or Data Path Dependencies 358
**8.2.1** True Dependencies 358
**8.2.2** Resource Dependencies 375
**8.2.3** Other Methods 380
**8.3** Memory Dependencies 385
**8.3.1** Anti- and Output Dependencies 385
**8.3.2** True Dependencies 385
**8.3.3** Load Bypass 387
**8.3.4** Self-Modifying Code: True Dependencies 387
**8.3.5** MU5 Name Store 388
**8.4** Special Register Dependencies 390
**8.4.1** Address Registers 390
**8.4.2** Status Registers 391
**8.4.3** Program Counter 393
**8.4.4** Global Impact Registers 394
**8.5** Performance Models 397
**8.6** Summary Comments 401
**8.7** Structural Dependencies 402
**8.7.1** Single-Function Pipelines 403
**8.7.2** Multi-Function Dynamic Pipelines 408
**8.7.3** Arithmetic Pipelines 412

***9*** Exceptions and Interrupts 425

**9.0** Overview 425
**9.1** Precise Interrupts 429
**9.1.1** Interrupted Instruction and Saved Program Counter 433
**9.1.2** Handling Preceding Instructions 434
**9.1.3** Handling Following Instructions 455
**9.2** Performance and Cost of Precise Interrupts 461
**9.3** Precise Interrupt Comments 462

***10*** Enhanced Implementations 467

**10.0** Overview 467
**10.1** Superpiplined Processors 468
**10.1.1** Elementary Performance Model 469
**10.1.2** A Refined Model 471
**10.1.3** Superpipeline Design Issues 473
**10.1.4** Current Example of Superpipelining 474
**10.2** Superscalar Processors 475
**10.2.1** Elementary Performance Model 476

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000035

**10.2.2** Superscalar Design Issues 481
**10.2.3** Superscalar Branching Strategies 491
**10.2.4** Dependency Detection and Resolution 492
**10.3** Performance Comparison 494
**10.4** Superscalar Superpipelined Processor 497
**10.4.1** Processor Parallelism 497
**10.4.2** Instruction-Level Parallelism 499
**10.4.3** Processor Performance 500
**10.5** Very Long Instruction Word (VLIW) Processors 502

***11*** Vector Processors 507

**11.0** Overview 507
**11.1** Vector Operations 509
**11.2** Memory-Processor Interface 510
**11.2.1** Memory-to-Memory Architecture 511
**11.2.2** Load/Store Architecture 512
**11.3** Vector Performance 513
**11.3.1** Assessment by Vector Instruction Latency 514
**11.3.2** Streaming Rate 515
**11.3.3** Assessment by MFLOPS 518
**11.3.4** Assessment by Half Performance 520
**11.3.5** Vector-Scalar Code Mix Problem 521
**11.4** Vector Instructions 523
**11.5** Mask Operations 532
**11.6** Dependencies 534
**11.6.1** Intrastatement Dependencies 534
**11.6.2** Interstatement Dependencies 535
**11.6.2** Chaining 536
**11.7** Benchmark Performance 538

References 541

Subject Index 567

Processors Index 574

Trademarks 576

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000036

# Preface

*Every difficulty slurred over will be a ghost to disturb your sleep.*

— Chopin

*The greatest difficulties lie where we are not looking for them.*

— Johann Wolfgang von Goethe

The widespread demand for high-performance personal computers and workstations has stimulated a renaissance in computer design. VLSI permits many millions of transistors to be placed on a chip that will sell for a few hundred of dollars. The two forces of demand and low cost combine to stimulate innovative designers to revisit old schemes and devise new ones to satisfy the ever-increasing demands for more performance. The performance of microprocessors has increased at a compound rate of about 30 percent per year over the last decade. Positive feedback is supporting the demand for more performance; new software systems require more processor performance, while more processor performance permits more complex software systems. Many of the techniques discussed in this book had their birth in mainframe and supercomputer designs. However, with the success of the microprocessor, the architecture of large mainframe class computers is being brought into question.

This book describes and discusses the design details of high-speed memory systems and pipeline processors that underpin the modern microprocessor. I undertook the project of writing this book for several closely related reasons. First, I believed that there was a need for a graduate level course on the nitty-gritty details of computer system implementation and I could find no suitable text. A compendium of the implementation techniques used today by many designers has not been collected into one book. In addition, I wanted the experience, over a



Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000037

# 1

# Memory Systems

## 1.0 Overview

There are many models that are used to evaluate the performance of a processor. One of the most insightful, the model for the time required to execute a task, is

task time = number of instructions executed × clocks per instruction (CPI) × clock period.

The number of instructions executed is usually a function of the instruction set design and is outside the scope of this book, but the second and third terms are the subject of this book. The design of the memory system and the pipeline processor have a direct bearing on the CPI that can be realized from a processor. As will be pointed out, there are CPI design considerations that affect the clock period; the CPI can be reduced at the expense of a longer clock and otherwise. The important metric is the product of these two terms, not their individual values. The interactions between the memory system and the processor will be evident as each of these major subfunctions is described.

The performance of a computer depends in large measure on the interface between the processor and memory. If this interface is not correct, a significant increase in CPI can result. The processor makes demands upon the memory for instructions and data, and this demand must be satisfied in order to realize the full potential of the processor. The following two quotes from [FLYN66] detail two parameters of the memory that are important.

> Latency or latent period is the total time associated with the processing (from excitation to response) of a particular data unit at a phase in the computing process.



Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000038

| | SPARC Integer | SPARC Floating Point | MIPS Integer | MIPS Floating Point | IBM S/360 | VAX | AVERAGE |
|---|---|---|---|---|---|---|---|
| Instruction | 0.79 | 0.80 | 0.76 | 0.77 | 0.50 | 0.65 | 0.71 |
| Data read | 0.15 | 0.17 | 0.15 | 0.19 | 0.35 | 0.28 | 0.22 |
| Data write | 0.06 | 0.03 | 0.09 | 0.04 | 0.15 | 0.07 | 0.07 |

**TABLE 1.1** Memory reference distribution.

Bandwidth is an expression of the time rate of occurrence. In particular, computational or execution bandwidth is the number of instructions processed per second, and storage bandwidth is the retrieval rate of operand and operation memory words (words/second).

The techniques used to provide low-latency and high-bandwidth memory for a processor depend heavily upon the nature of the references made to the memory by the processor. Table 1.1 shows the fraction of memory references classified as Instruction Fetch, Data Read, and Data Write for a number of processors. Notice that instruction fetches represent between 50% and 80% of all references. Data reads represent 15% to 35% of all references, while data writes represent less than 10% for all processors other than the IBM S/360.

The implication of these reference statistics is that the priorities of a memory system design are: first the instruction fetches, second the data reads, and last the data writes. The obvious solution to the problem of providing both low latency and high bandwidth is to use higher performance memory devices. However, the history of increasing the bandwidth of memory devices (circuits) over a ten-year period is not encouraging. Published data from the International Solid State Circuits Conference provide normalized information on the reduction of access times of dynamic RAMS and static RAMS shown in Figure 1.1.

Note that while clock periods have decreased by a factor of approximately 23 (a reduction of 27% per year over the last decade), SRAM cycle times have decreased by only a factor of 18 (a reduction of 25% per year) and DRAMS by a factor of 13 (a reduction of 23% per year). These data show that memory performance increases have not kept pace with reductions in processor clock periods.

One conclusion from the data of Figure 1.1 is that the required decrease in latency and the increase in memory bandwidth have come, and will continue to come, not from improved memory technology but from improved memory system design. Memory performance will probably continue to lag processor clock cycle-time improvements, and more processor concurrency will further increase the demand on memory. Thus

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000039

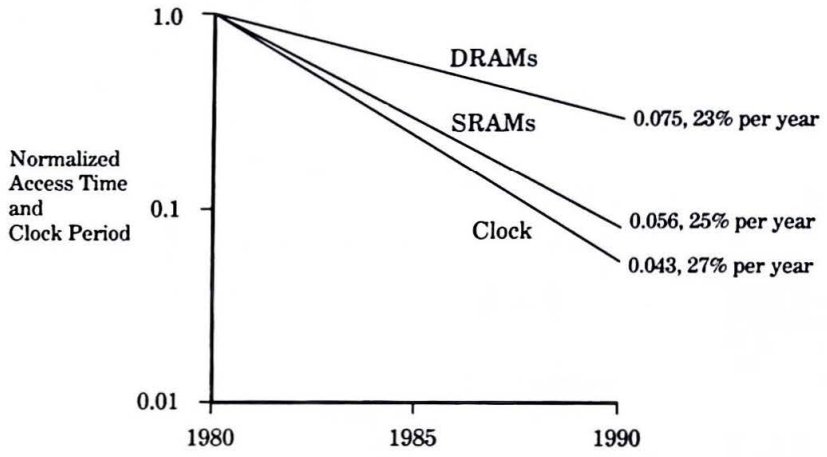



**FIGURE 1.1** Memory and processor cycle-time improvements.

innovative memory architectures are mandatory if processor performance is to continue to increase.

The need for increased memory performance has occurred in a period of significant memory cost decreases. Memory costs have been decreasing dramatically since the early commercial computers. A 1-M byte of add-on memory for IBM S/360 computers cost $1M in 1965! In 1994, a 1-M byte of memory for a personal computer costs approximately $50; a dramatic cost reduction of approximately 20,000-fold.

The relative cost relationship between fast and slow memory components has remained relatively constant over this period and can be expressed by Grosch's Law [GROS53]. This law is an empirical observation that states "giving added economy only as the square root of the increase in speed—that is, to do a calculation ten times as cheaply you must do it one hundred times as fast." The converse has often been stated as if you double the price of a computer or computer component, the performance will increase by a factor of four. Grosch's law can be confirmed by noting the relative cost and performance of SRAM and DRAM chips. Note that Grosch's law, as posited, applies over time and does not consider learning-curve effects. However, the law is frequently applied to one time period as when looking at members of a computer family.

The ideal memory for a computer would be one that removes the memory bandwidth and latency constraints. Such a memory would have zero access and cycle time (zero latency and infinite bandwidth), be of infinite size, and have zero cost. Obviously, we will never have such an ideal memory. Over three decades, designers have attempted to approach the zero latency of this ideal memory by the architectural path of a memory hierarchy.

The following two sections introduce the issues of supplying low-

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000040

| Latency Clocks per Instructions | Normalized Performance |
|---|---|
| 0.0 | 1.00 |
| 0.2 | 0.83 |
| 0.4 | 0.71 |
| 0.6 | 0.62 |
| 0.8 | 0.55 |
| 1.0 | 0.50 |

Note: Basic CPI = 1.

**TABLE 1.2** The effect of memory latency on processor performance.

latency and high-bandwidth memory. Chapters 2, 3, and 4 give details of low-latency memory in the forms of caches and virtual memory. Chapter 5 describes the techniques for providing high-bandwidth memory using interleaved low-bandwidth memory components.

## 1.1 Memory Latency

The issue of memory latency is quite important with regard to pipelined processors. Consider the effect of memory latency on VAX 11/780 performance. If this processor had a perfect memory, with a bandwidth of one clock per reference and no additional latency, an instruction execution would take approximately 9.5 clocks. Actual memory latency adds approximately 0.5 clocks, giving a CPI of approximately 10 clocks, which is an increase in the CPI of approximately 5%. However, with a pipelined processor that may have a CPI of 1.5 with a one cycle memory, 0.5 additional clocks (1 clock is accounted for in the basic CPI figure) represent a 33% increase in CPI. The reason latency is important is that when a memory request is made, the processor may be idle until that request is satisfied. Table 1.2 shows the impact of latency on the performance of a processor.

As expected, performance will decrease by 50% when the memory reference latency is equal to the processor performance without latency. Current memory technology is capable of satisfying memory bandwidth requirements for even the most powerful computers. But the satisfaction of latency is a major design problem. With increasingly faster processors, the memory performance bottleneck becomes even more pronounced.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000041

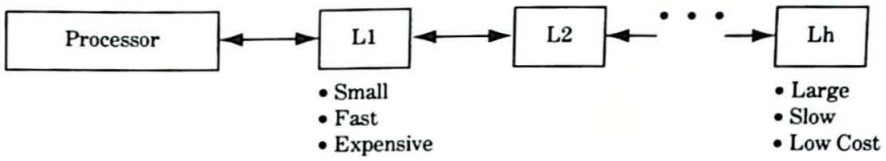



**FIGURE 1.2** Hierarchical memory.

### 1.1.1 *Hierarchical Memory Systems*

*Hierarchical memory* uses a hierarchy of memory units, as shown Figure 1.2, to satisfy the low-latency requirement of a memory system. Starting with the registers in the processor, the storage capacity in each level increases and the speed and the cost per bit decrease. For most references, the referenced item is found in a memory with low latency. Frequently referenced items are stored in high-speed, costly memory, and infrequently referenced items are stored in a low-speed, low-cost level. The weighted average latency is decreased by storing instructions and data in the level that is consistent with the frequency of the referenced item.

Interestingly, von Neumann [BURK46] noted in 1946 the requirement for a three-level memory hierarchy. The three levels are the processor internal registers, the main memory, and bulk storage on magnetic wire recorders. Hierarchical memory systems permit the designer to trade latency or access time against cost. The design goal is that the processor will see a memory that is only slightly slower than the fastest memory in the hierarchy.

This book does not directly address the processor registers, although they are a member of the hierarchy. The register-level design considerations are basically the concern of the instruction set designer. However, indirect effects on the performance of the memory hierarchy are found in such items as the memory bandwidth required to support context switching a very large register file.

Three terms that apply to all levels of the memory hierarchy are defined as follows.

The *hierarchical level* is usually denoted by an index beginning with 1 for the level nearest the processor and progressing upward. But, in relative terms, the "highest level" is the level closest to the processor and the "lowest level" is the slowest memory of the system. In the discussions to follow, the convention of the index 1 is used for the highest level, with indices $2 \cdot \cdot h$ denoting the lower levels.

*Multilevel inclusion* is the property that the contents of each level of the hierarchy $h$ is always found at level $h + 1$ [LAM79], [BAER87].

*Main memory* is the largest random access memory in the system.

Early computers had four levels of memory: registers, main memory,

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000042

disks or drums, and archival storage such as magnetic tape. For these computers, the main memory was magnetic core with rather long access and cycle times. Thus, the first introduction of another level in the hierarchy was a cache between the processor and the main memory. Further, these first caches were only instruction buffers that evolved to a unified cache holding both instructions and data.

Further evolution of the hierarchy consists of split caches for the private storage of instructions and data and, more recently, disk caches. With the advent of VLSI having more than one million transistors on a chip, the cache moved onto the processor chip and another level of caching was introduced between the on-chip cache and the main memory. Today, a memory system may consist of six levels: processor registers, on-chip cache(s), cache(s), main memory, disk cache, and disk.

Concurrent with the evolution of caching techniques, designers have been working on the management of the lower levels of the hierarchy. The programmer would, in general, like to see a single-level high-speed memory. This statement, however, is qualified because of the need to segregate various programs and data sets in the address space for file management and protection reasons. Nevertheless, the goal of the hierarchical memory is to provide the appearance of a large, fast single-level memory. The system, hardware and software, manages the memory system so that frequently referenced items will be found in the highest speed memory when needed.

Early systems employed *static* memory management; that is, the programmer determined the sequence of the required items and explicitly managed the memory hierarchy by overlays. A number of researchers, at Rice University, MIT, and Cambridge University [DENN70], were concurrently developing the ideas of virtual memory that provided for the automatic dynamic management of the memory hierarchy. Virtual memory gives the programmer the desired view of a flat memory, the size of which is equal to the processor's virtual address space. Even though the physical high-speed memory is much smaller than the virtual address space, the burden of managing overlays is removed from the programmer.

It is well known that a programmer can trade memory space for performance. For example, loops can be unrolled into a nonlooping program at the expense of a larger program. With virtual memory, the programmer must be aware that even though virtual address space is almost unlimited, as a larger address space is used, the memory system's performance will degrade. Thus, a programmer must use virtual memory system care. I note the fact of the increase in processor management of cache functions as discussed in Section 2.8.1. Thus, with some contemporary processors, we have gone almost full circle to static management of some of the memory hierarchy.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000043

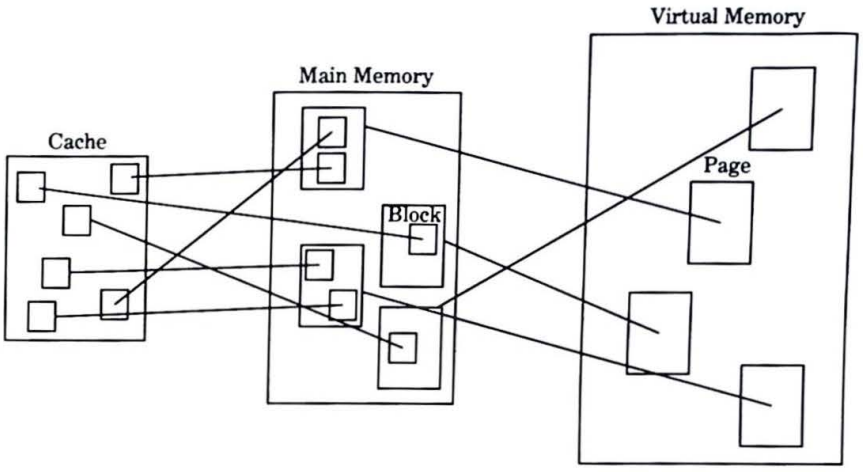



**FIGURE 1.3** Hierarchical memory allocations.

A hierarchical memory allocates selected elements of a slow memory into a higher speed memory. Selected elements of this memory are then allocated in the next higher speed memory and so on. Figure 1.3 shows this process. Pages are taken from virtual memory space and allocated into main memory. Blocks from these pages are then allocated into the cache.

The hardware and software associated with a hierarchical memory must provide the following functions:

1. Detect a miss at the referenced level.
2. Identify the items to be vacated at the referenced level and perform the move to a lower (slower) level if required.
3. Map addresses between levels.
4. Read the new items from the lower (slower) level to the referenced level.
5. Complete the reference to the referenced level.

These functional requirements can be satisfied with hardware, software, or combinations of the two. I will describe the specifics of the design of these functions in Chapters 2 and 3.

### 1.1.2 *Memory Efficiency*

The access times of three levels of a hypothetical hierarchical memory are tabulated below. In addition, the ratios of access times are shown. Ratios are commonly used to normalize to the access time of the fastest member of the hierarchy.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000044

| Level | Access Time | Access Time Ratio |
|---|---|---|
| Cache | 50 ns | 1 |
| Main Memory | 200 ns | 4 |
| Disk | 20 ms | 100,000 |

These access times are the parameters of a two-level model for the effective access time at either the cache or main memory levels. Other models are described in Chapter 2.

$$\text{Effective Access Time } (t_{ea}) = t_h + P_{miss}(t_{h+1} - t_h)$$

where $t_h$ is the access time of the highest level, $t_{h+1}$ is the access time of the next level, and $P_{miss}$ is the number of references to the cache that are not found in the cache divided by the number of references to the cache.

The effective access time model can be normalized to the access time of the highest level or cache as

$$t_{ea} = 1 + P_{miss}\ (R - 1) \quad \text{where } R = [t_{h+1}/t_h]\,.$$

The value of $R$ is the ratio of the access time of two levels that is rounded up to an integer because the memory cycle time is usually an integer multiple of the basic clock period. Furthermore, the cache cycle time is an integer multiple, frequently 1, of the clock period. The effective access time of the memory, as determined by this model, can be viewed as the weighted average memory latency—the average time required to satisfy a memory request.

Chapter 2 discusses in more detail the derivation of the $t_{ea}$ model, with variations, and the values for $P_{miss}$ in real systems. At this point, however, we will consider the efficiency of a hierarchical memory system. That is, the fraction of total time during which the memory is performing useful references when the total time is the useful time plus idle time when misses are being serviced:

$$\text{memory efficiency} = 100 \times \frac{1}{1 + P_{miss}(R - 1)} = \frac{100}{1 + P_{miss}(R - 1)}\,.$$

If $P_{miss} = 0$ or $R = 1$ or both, the referenced memory is said to be 100% efficient. That is, no undesirable latency due to a miss is associated with

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000045

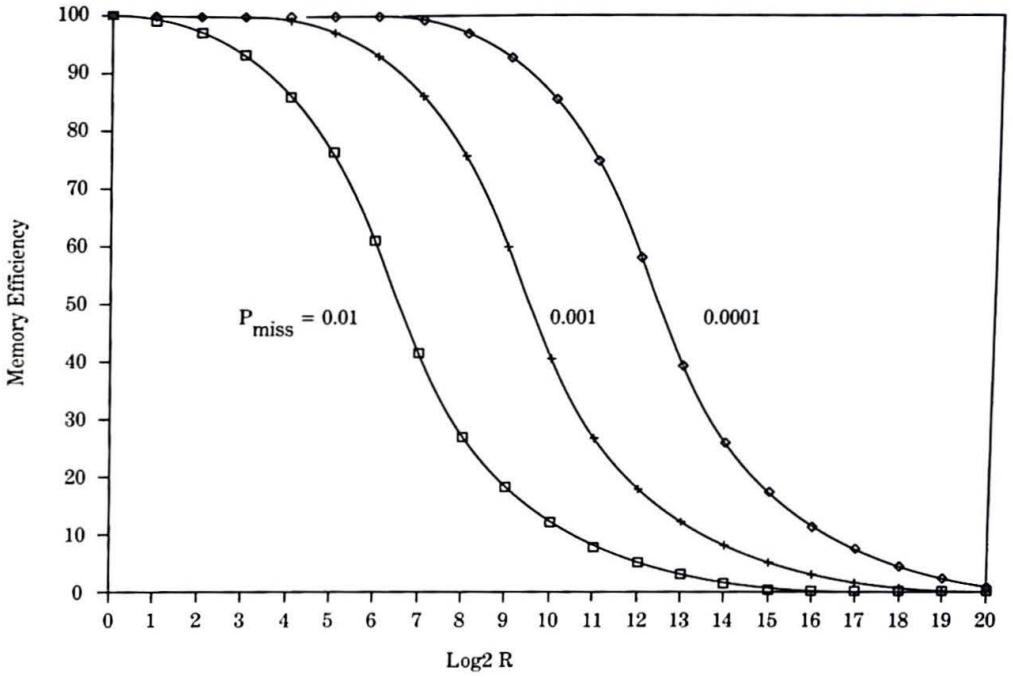



**FIGURE 1.4** Hierarchical memory efficiency.

an access. Figure 1.4 shows a plot of memory efficiency for the locus o different values of $P_{miss}$ as $R$ is varied from 1 to 1,000,000. Note that for small values of $R$, the memory efficiency is approximately 100% regardless of the value of $P_{miss}$. Also, for large values of $R$, the memory efficiency is approximately 0% regardless of the value of $P_{miss}$.

Low values of memory efficiency are observed for large values of $R$, the domain of virtual memory level, and multiprogramming. A multiprogramming operating system will swap out the executing process for another process when a miss occurs at the referenced level. With large values of $R$, the management of the interface is performed by a combination of hardware and software because the time constraints are not overly demanding. At any design point, the efficiency of a memory can be improved by an equal percentage improvement in either $P_{miss}$ or $R$. Practical considerations, however, limit these options. $P_{miss}$, in a virtual memory system, is limited by the workload, which is discussed in Chapter 3. Reducing the value of $R$ is the only practical alternative under the condition when $P_{miss}$ can be reduced no further.

Caches usually have small values of $R$ and corresponding high values of efficiency. Thus, job swapping is precluded and requires that the management of the hierarchy is via hardware with a minimum of overhead. Because tasks are not swapped at the cache level, small $P_{miss}$ values *and* small $R$ values are design goals of cache systems. But as

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000046

Chapter 2 discusses, improvements in $P_{miss}$ at the cache level are usually easier to obtain than improvements in $R$.

The concepts of managing the interface between different levels of the hierarchy are similar, if not identical. The major differences are in the implementation method, that is, the use of hardware or software or both; however, the terminology changes from level to level. The discussion of various types and levels of caches will precede the discussion of virtual memory.

## 1.2 Memory Bandwidth

Sufficient memory bandwidth is required to support the processing rate of the computer. Each instruction must be fetched, and an executing instruction may require one or more operands or may store a result or both. There must be a balance between the demand for and the availability of memory bandwidth. The memory/processor balance is illustrated in Figure 1.5, which plots the processing rate of a given processor/memory combination as the processor clock frequency is changed. Clock frequency is used as a surrogate for increased execution rate and thus memory bandwidth demand. The assumption is that the bandwidth of the memory system is fixed, as indicated by the arrow. At this point, the available memory bandwidth is reached and any further increase in the clock frequency will not yield a performance improvement.

The performance of a memory bound or limited processor can be increased by increasing the memory's available bandwidth. Alternatively, some architectures are devised to reduce the memory bandwidth

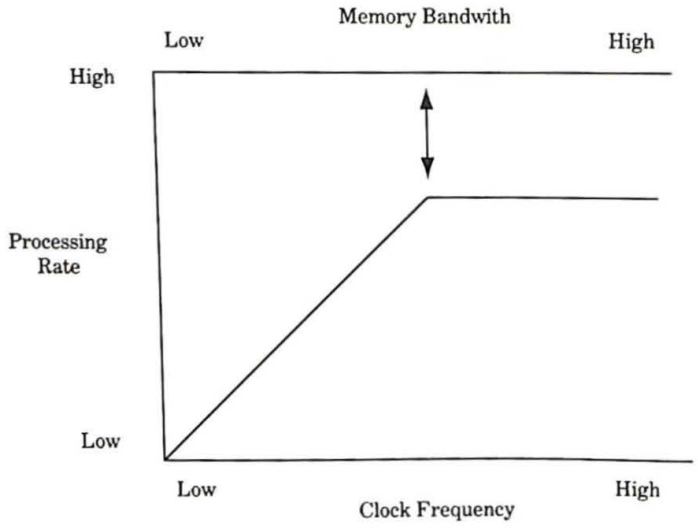



**FIGURE 1.5** Processor/memory bandwidth.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000047

requirement and, thereby, obtain increased performance with a limited bandwidth memory. Most CISC architectures are based on this approach. However, these architectures have been microprogrammed, and the bandwidth of the microstore can be included in the aggregate available bandwidth. Microprogram instruction sets are usually designed for high performance, and the bandwidth is not fully used. Thus, it is misleading to add merely the microprogram bandwidth to the instruction bandwidth of a CISC architecture and expect that bandwidth to equal the instruction bandwidth of a pipelined processor. Nevertheless, a well-designed processor memory is one with a close match between bandwidth demand and bandwidth availability. If the system is either processor bound or memory bound, inefficiencies result.

A well-known case of the negative effect of insufficient memory bandwidth is found in the Cray-1 and is discussed in Chapter 11. The Cray-1 had only one memory port, which resulted in a significant reduction of the processing rate for some problems. This defect was corrected in the Cray X-MP, which has three memory ports.

The bandwidth demand of CISC processors is illustrated by the demand of the VAX 11/780. This processor required, on average, 10 clocks of 250 ns (nanoseconds) to execute an instruction, or 2.5 μs. per instruction. Each instruction processed requires the fetching of an instruction and approximately 1.8 operands. Weicker [WEIC84] indicates that an average instruction is 3.8 bytes. With a four-byte operand, the average number of bytes referenced per instruction is

$$\begin{aligned}\text{bytes referenced per instruction} &= \text{inst. bytes/inst.} + \text{operand bytes/inst.} \\ &= 3.8 + 1.8 \times 4 = 11.\end{aligned}$$

Thus, the bandwidth demand placed on the memory by the processor is approximately $11/2.5 \times 10^{-6} = 4.4$ million bytes per second.

On the other hand, a modern pipelined processor, as shown in Figure 1.6, must fetch an instruction every clock period and, for load and store instructions (approximately 30% of all instructions), there is a memory

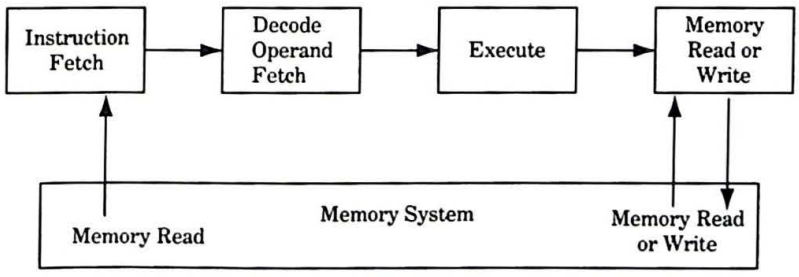



FIGURE 1.6 Pipelined processor.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000048

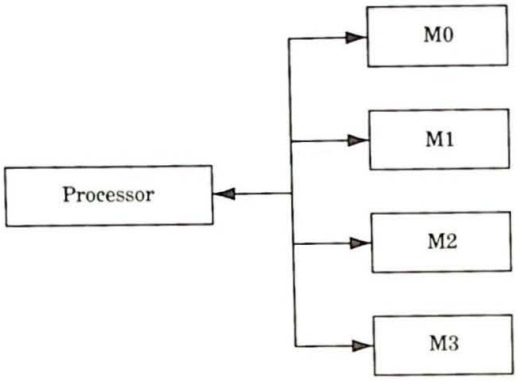



**FIGURE 1.7** Interleaved memory.

reference of either a read or a write. With a clock period of, for example, 25 ns, and 1.3 four-byte references per clock, the memory must support approximately $52 \times 10^6$ references per second, or have the bandwidth of $200 \times 10^6$ bytes per second.

This back-of-the-envelope memory bandwidth analysis shows an increase in memory bandwidth demand by approximately a factor of 50 in the period from 1980 to 1990, an increase of approximately 48% per year. The modern pipelined microprocessor places a heavier demand on memory bandwidth that must be provided by innovative memory architectures and increased circuit speed.

*Interleaved memory* uses a number of low-speed and low-cost memory modules to achieve increased bandwidth by anticipating the distribution of accesses over all of the memory modules, as shown in Figure 1.7. That is, increased bandwidth can be obtained by accessing the memory modules in parallel. Chapter 5 discusses interleaved memory in detail.

Because the memory modules are accessed in parallel, the bandwidth (BW) of an interleaved memory approaches

$$\text{BW}(I) = \text{bandwidth of one module} \times \text{number of modules.}$$

In other words, two modules will have twice the bandwidth of one module because the two modules can be accessed in parallel and the referenced items, for reads or writes, are passed over the bus to/from the processor. With a high degree of interleaving, bandwidth can be increased to almost any desired value, keep in mind; however, that the latency of the first reference is still that of referencing the first module. Thus, interleaving does not solve the latency problem, only the problem of memory bandwidth. More realistic performance models are discussed in Chapter 5.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000049

## 1.3 Queues

Queues have been used in the memory hierarchy beginning with the Stretch [BLOC59]. We find both fetch queues and write queues in many processors today. Fetch queues provide a technique for obtaining high bandwidth and low latency from an interleaved memory that has only high bandwidth such as early high-performance computers, the Stretch being an example. The Stretch processed variable length instructions thus presenting a varying demand (in bytes per clock) to the memory system. A queue is ideal for providing variable length instructions while hiding the latency of an interleaved memory.

The IBM 360/91 used fetch queues for instructions, target instructions, and data. The Intel ix86 family of processors as well as the VAX family have variable-length instructions and use instruction queues. Superscalar processors, which are described in Chapter 10, have the characteristic of variable-length instructions; the PowerPC 601 and the Intel Pentium are examples and use instruction fetch queues.

Instruction fetch queues continued to be used after the introduction of caches. The queue can hide, in many cases, the latency of the cache when there is a cache miss. Data queues require that there be a load prefetch capability in the processor. In general, instruction prefetch has been extensively studied but the study of data prefetch is relatively recent in the literature. The design and use of instruction and data fetch queues is described in Chapter 2. Write queues, which are also discussed in Chapter 2, have been used to hide the write time when a write to memory is required. By writing into a write queue, the processor can resume execution and the actual write (which may have a long latency) is hidden from the processor.

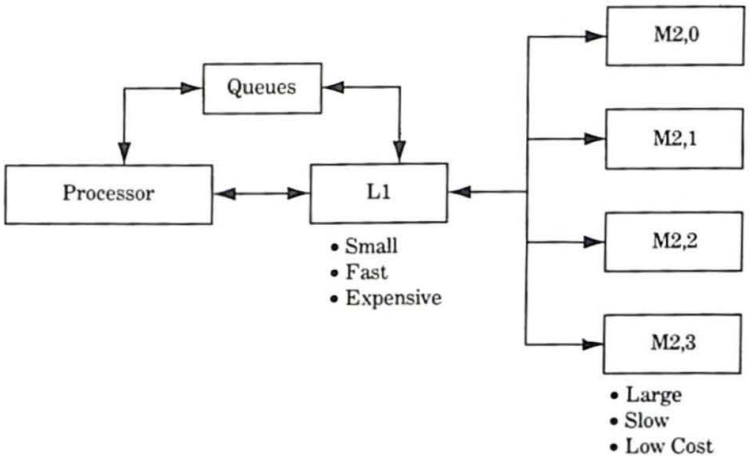



**FIGURE 1.8** Memory system components.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000050

## 1.4 Summary

As later chapters point out, the modern memory system combines queues with interleaved and hierarchical memories as shown in Figure 1.8. The cache, noted as M1, provides low-latency access to frequently used items while interleaved memory modules, noted as M2,n, are used to provide a high-bandwidth stream to the cache when an item is not found in the cache. Queues hide latency and smooth the flow of variable-length instructions.

In addition to the memories shown in Figure 1.8, interleaved disks can provide a high-bandwidth stream between the disk system and the main memory. The technology of interleaving disks (RAID) is discussed in Chapter 5.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000051

# 2

# Caches

## 2.0 Overview

The primary purpose of caches is to reduce the latency of references to memory. Ideally, the total latency should be one processor clock cycle for instruction reads or data reads or writes. The basic idea behind the design of all members of the memory hierarchy, including caches, is that the referenced addressable units (AU), instructions, and/or data will frequently be found in a high-speed memory. For any AU, a reference to a lower-speed memory will be required, which adds to the latency of the access.

Caches and related topics have been the subject of intense research for over thirty-five years. Cocke [COCK59] described a cache-like memory for the Stretch computer that is named a "virtual memory." This memory provided a form of lookahead and lookbehind for the data stream. Bloom [BLOO62] and Wilkes [WILK65] first described buffer memories that can be recognized as caches. The body of published material is immense and includes 403 citations in Smith's 1986 bibliography [SMIT86]. A comprehensive bibliography today would probably have in excess of 1,000 citations. For this reason, I am unable to cite every idea that has been posited concerning caches. Despite the large number of citations, however, I discuss a number of classifications and ideas in this text that may be outside the mainstream of current cache design practice.

Przybylski points out that "The entire cache design space is incredibly diverse" [PRZY90a]. He further states that "about 130 parameters are needed to fully specify a two-level cache system." This chapter catalogues the design space into taxonomies and provides analytic performance models to assist in making design tradeoffs.

The early literature referred to a cache as a *buffer* [GIBS67]. Over time, however, the name *cache*, first used by Conti et al. [CONT68], has become the preferred name. The various cache design issues addressed



Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000052

in this chapter concern the reduction of delays that add to the basic—usually one cycle—latency of a cache.

Three major dimensions of cache design are discussed in this book, the first two of which are addressed in this chapter. They are

1. Organization or structure of the cache(s).
2. The set of rules that is followed when the cache is accessed and managed. These rules are known as the *policy*.
3. The cache address space. In Chapter 2, the address presented to a cache is viewed as just an address. The logical characteristic (real or virtual) of the cache address space is discussed in Chapter 4.

In addition to the organization component of cache design, the processor architecture and the program workload both contribute to the domain of cache design. These two additional components are outside the scope of this book and will only be briefly mentioned. The thrust of all innovations in cache design has been to design the cache so that the cache organization complements the characteristics of the address stream and increases the probability that a requested AU will be found in the cache. Recent research in compiler optimization for reordering the address stream to match a cache design is briefly discussed in Section 2.8.3.

Because of their importance to the operation of caches, I begin with a discussion of locality concepts, the three most frequently noted of which follow.

1. *Temporal locality*. Information recently referenced by a program is likely to be used again soon.
2. *Spatial locality*. Portions of the address space near the current locus of reference are likely to be referenced in the near future.
3. *Sequential locality*. A special case of spatial locality in which the address of the next reference will be the immediate successor of the present reference.

I believe that Gibson [GIBS67] was the first person to note these two distinct addressing patterns. Gibson points out that if references are purely random "any word is equally likely to be referenced by the CPU, variations of block size and replacement algorithms have no effect on the number of word references not found in the local store." These and other results [DENN70, DENN72] were produced through research on the virtual memory system and were later translated into cache concepts. Smith [SMIT82], however, was the first to use the terms "spatial" and "temporal" locality. Temporal locality is often referred to as "look backward." That is, an instruction sequence such as a loop will reuse instructions. Spatial locality is often referred to as "look forward." A program counteraccesses instructions in sequence. Thus, the next

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000053

instruction will be fetched from a location adjacent to the present instruction and referenced next. Accessing a data vector also displays spatial locality even when the stride is greater than 1.

As caches have developed over the years, a rich and varied terminology has emerged. Various researchers and computer manufacturers have created new terms for identical concepts. Also, different terms are used at different levels of the hierarchy to identify identical concepts. I will try to point out the different terms by source in the following sections.

Caches are classified into the following four types according to their function or the nature of the information held in the cache.

1. An *instruction cache* holds only instructions. The design is optimized for the instruction stream only. Instruction buffers and instruction queues are sometimes used in place of instruction caches. Instruction caches are discussed in Section 2.3.
2. A *data cache* holds only the data stream. Data caches are discussed in Section 2.4.
3. A *unified cache* holds both instructions and data. The design is optimized for a mixed address stream. Unified caches are discussed in Section 2.5.
4. A *split cache* is a two-cache system, one for instructions and one for data. The two cache designs may be different for the two address streams. Split caches are forms of memory systems referred to as Harvard architectures [CRAG80] and are discussed in Section 2.6.

Processor designers face challenging problems in designing caches for VLSI processors. Usually there is a chip area budget for caches that should be used most effectively. Some design alternatives are

1. a large instruction cache,
2. the area split between an instruction cache and a data cache,
3. a large unified cache, and
4. a multilevel split cache.

Because each of the cache types may be designed for different address streams, their designs may be different. Therefore, the following sections of this book will examine each cache type individually.

Before proceeding, a comment on terminology is in order. The words *fetch* and *read* are generally synonymous. However, fetch is frequently used in the literature to refer to an *instruction fetch* while read refers to *data reads* and the method for handling a cache miss is sometimes referred to as the *fetch policy*. The vocabulary of this chapter is as follows.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000054

*Read*, a read access is made by the processor to a cache for instructions or data.
*Write*, a write access is a processor write into the cache and/or main memory.
*Fetch*, a read access to memory to load a block into the cache.
*Access*, a generic reference to the cache or memory.

Caches can be classified by the interconnection topology; the selection and interconnection of the various cache components. A multilevel cache is one that has a first-level cache (either instruction, unified or split) and a second-level unified cache. The second-level cache is designed to accept the missed references from the first-level cache(s). Each of the two levels can be

N Nil, not implemented;
I Instruction cache, buffer or queue;
D Data cache;
S Split cache;
U Unified cache.

Some reasonable, or known, two-level cache topologies are shown in Figure 2.1. A more complete description of various topologies is found in Section 2.6.

Historically, caches have been viewed as a pure implementation technique for improving performance that is transparent to the program. However, following this view, the program can misuse the cache with a performance penalty when there is no recognition of the presence or absence of a cache. This view of the cache has, of necessity, changed; software support for caches is now becoming the norm. There are things the program can or should explicitly do with the cache. For example, two of these actions are: (1) prefetch a cache block and (2) clear the cache to maintain coherency. These program actions are specific to individual architectures and are discussed in Section 2.8.1.

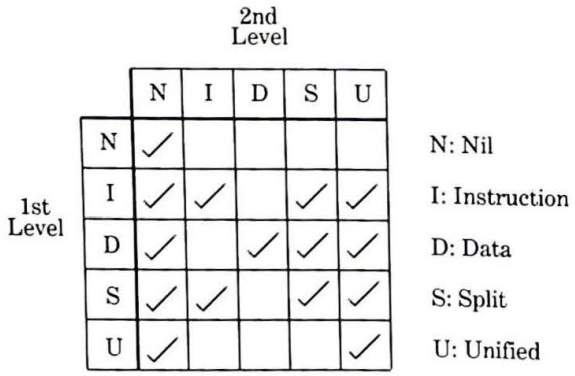



**FIGURE 2.1** Common cache topologies.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000055

The preceding discussion concerns caching of instructions and/or data. As I note in the following sections, caches for addresses play an important role in processor implementation. For example, speeding up virtual memory name translations with address caches is discussed in Chapter 3. Address caches used to improve branch performance are discussed in Chapter 7. In addition, the special cache designs needed to support superscalar processors are discussed in Chapter 10.

## 2.1 Cache Organization and Placement Policies

The mapping of addresses between the main memory and the cache is known as the *placement policy*. An address mapping is required because AUs taken from scattered main memory locations will be placed and accessed in a cache with a relatively small contiguous address space. The requirement for cache address mapping leads to the major design considerations of the cache, its organization and placement policy. These in turn are determinants of the $P_{\text{miss}}$ (miss ratio) of a cache for a given workload. Two definitions of $P_{\text{miss}}$ are in common use:

1. The number of references to the cache that are not found in the cache divided by the number of references to the cache. $P_{\text{miss}}$ is the probability that a reference will miss in the cache. This definition of $P_{\text{miss}}$ is used in this book.
2. The number of cache references not found in the cache divided by the number of instructions executed [KAPL73, ALEX86].

The first definition is used to evaluate the design of a cache alone, while the second definition allows the cache design to be evaluated per the instruction-level performance of the processor.

The main memory and cache are partitioned into equal-sized groupings of AUs, say for example, 16 bytes or 64 bytes [GIBS67]. The name *block* is used for these groupings even though other names, such as *fetch unit, line*, and *transfer size*, have wide usage. A memory address is composed of two fields, the *block name* and the *displacement*, within the block as shown in Figure 2.2.

The block size is the number of AUs transferred between the main memory and the cache. When the cache is accessed for either a read or a write, a mapping function is employed to locate the block holding the requested AU in the cache, if it is present. To accomplish this, a cache performs three major functions:

1. Translates the block name to a cache address.
2. Determines the presence or absence of the block containing the requested AU.
3. Provides the data storage component.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000056

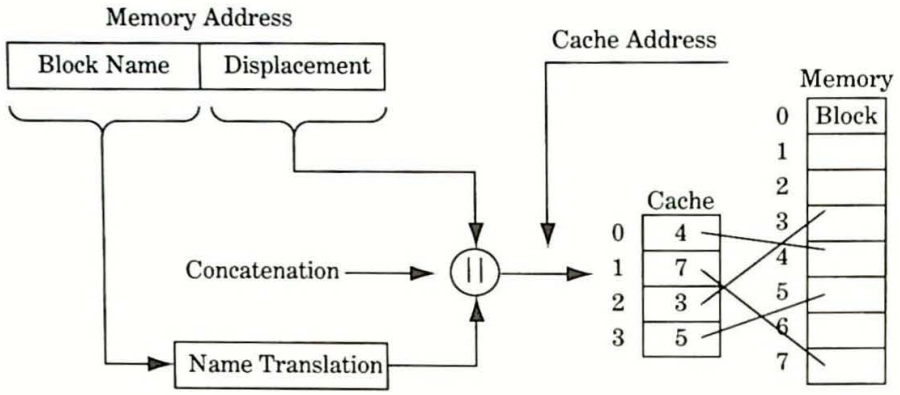



**FIGURE 2.2** Memory partition into blocks.

Current practice is to integrate functions 1 and 3 into one structure that is referred to as a *late select* cache. When the functions are disjointed, the cache is referred to as an *early select* cache; these organizations are discussed in Section 2.1.2.

As previously noted, the placement policy is the cache organization, as shown in Figure 2.3. There is considerable diversity in the terminology of cache organization in the literature. The terminology used here is taken from [GIBS67] and the IBM S/360/85 [LIPT68], the first known commercial implementation of a cache.

The descriptions of the cache organization start with the smallest allocation.

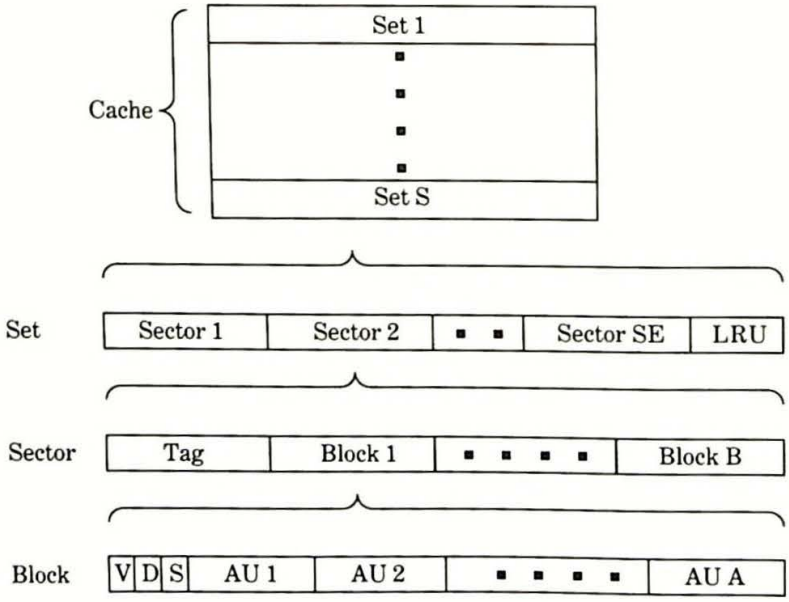



**FIGURE 2.3** Cache organization.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000057

An *addressable unit* (AU) is the unit addressed by the processor, a byte, word, or other unit.

A *block* contains $A$ AUs with control bits (valid and dirty described in Section 2.1.3).

A *sector* contains $B$ blocks plus a tag. The number of bits in a tag is a function of the size of the memory address and the parameters of the cache. A sector is also known as an *entry* [SMIT82].

A *set* contains $SE$ sectors and, in some designs, LRU bits. $SE$ is the degree of associativity.

A *cache* contains $S$ sets. A set is a congruence class, to be described later.

As noted previously, the main memory is divided into blocks that are the unit of transfer between the main memory and the cache on a fetch operation. These units are also known as the *fetch unit* [PBZY90] or *transfer size*. The fetch unit is a multiple of the block size and is equal to or less than a sector size: $B \leq \text{FU} \leq SE$, with the most common fetch unit being a block. Depending upon the write policy, the write unit may be, and usually is, different from the fetch unit.

Note that each of the parameters ($S$, $SE$, $B$, $A$) are always integers and are usually even powers of two. This 4-tuple is used to denote the design of a cache and is similar to the 4-tuple of [PRZY90a] and the 3-tuple of [AGAR89]. The number of AUs in the cache, or cache capacity, is equal to the product of the values of the tuple; that is,

$$\text{number of AUs} = (S \times SE \times B \times A).$$

The original use of a block was reported by Gibson [GIBS67]. Th grouping of blocks into sectors is used on the IBM S/360/85 to spread the cost of tag bits over a larger number of blocks. In later years, other terms were used to express the partitioning into sectors and blocks. These include *block frame and block* [HILL87] and *block and sub-block* [HILL84]. IBM recently introduced the term *page*, which refers to all of the sectors within a set [GROH90]. Table 2.1 provides a list of the synonyms of the terms used in this book.

Cache organization is illustrated with the following figures, which show the most common cache organizations. The first organization shown in Figure 2.4 is a canonical cache, which is used to illustrate the relationships between the various cache allocations; the organization is (8, 2, 2, 2). This example of the canonical cache has eight sets and two sectors; each sector has two blocks, and each block has two AUs.

The organization of the cache and the translation of the memory address to the cache address are tightly bound. As shown in Figure 2.5, the cache address is formed by allocating the memory address into fields.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000058

| Text | Przybylski [PRZY90] | Hill [HILL87] | Intel i486 | PowerPC 601 |
|---|---|---|---|---|
| block | sub-block | block | line | sector |
| sector | block | block frame | block | set |
| set | set | set | set | line |
| cache | cache | cache | cache | cache |
| fetch unit: block | fetch unit | sub-block | line | sector |

**TABLE 2.1** Cache organization terminology.

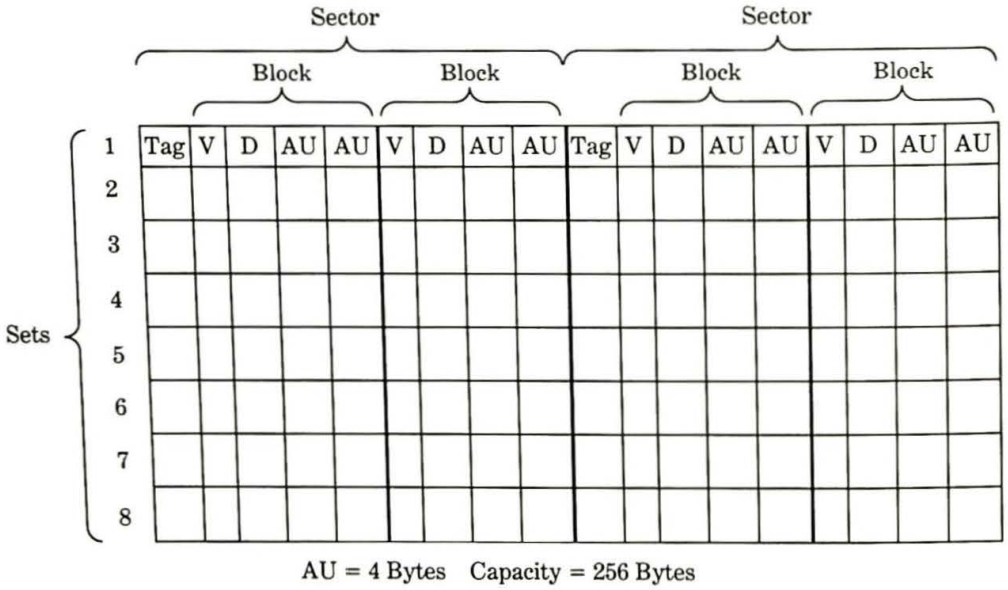



**FIGURE 2.4** Canonical cache.

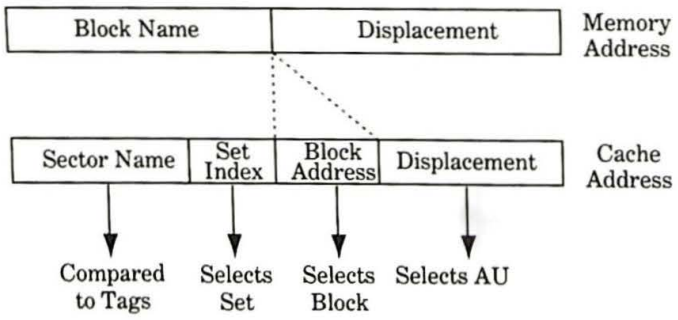



**FIGURE 2.5** Cache address formation.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000059

The memory address block name is allocated to:

1. the sector name, which is compared to the tags in sector(s) of the selected set,
2. the set index, which selects a set, and
3. a block address, which selects a block within a selected sector. As many caches have only one block per sector, this field can be of zero length.

The displacement is allocated to the displacement in the cache address.

This type of addressing is called *congruency mapping* [MATT70], which is defined as:

> Two integers are congruent to modulus $m$, if, after division by the modulus $m$, both have the same remainder $r$ [GARN59].

Because the block name is divided into two fields, the set index and sector name, the set index is the remainder when the block name is divided by $S$, the number of sets in the cache. Thus, a block is divided into congruence classes. Stated another way, *a congruence class* consists of those AUs that have the same LSBs in their set index and different MSBs in their sector names. The remainders are found by division, that is, right shifting the block name, a trivial operation when base 2 design is followed. Note that memory is also divided into congruence classes as shown in Figure 2.2.

If there is only one block per sector, the block address field is of zero length. For these designs, is the proper name for this entity a sector or a block? This naming problem is resolved by using the term *sector-block*.

One cache design that has more than one block per sector, and hence a block address field, is the MC68030 instruction and data caches, which is illustrated in Figure 2.25. These caches are organized with four blocks per sector, requiring two block address bits. With a block of 4 bytes, or one word, no further subdivision by a displacement is required.

If there is only one set, that is, the organization has only one congruence class, there is no set index field. The block name from the memory address becomes the sector name of the cache as illustrated by the IBM S/360/85 shown in Figure 2.21. As described below, this organization is called an *associative cache*.

There are three classes of cache designs described below; the descriptions include the structure of the cache and the address field subdivisions.

1. An *associative cache* has one set and is organized as (1, *SE, B, A*). One can view an associative cache as having a tag comparison on each sector with no indexing into the cache. An example of this organization

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000060

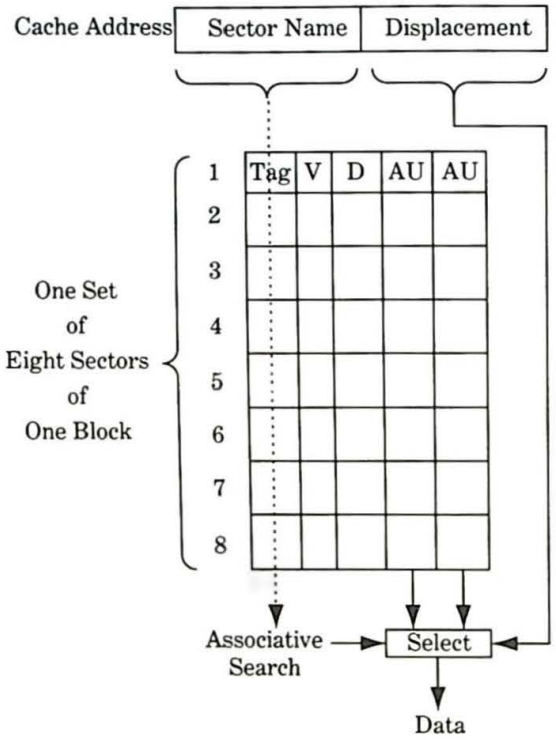



**FIGURE 2.6** Associative cache.

for (1, 8, 1, 2) is shown in Figure 2.6. Mattson [MATT70] referred to the associative cache as *unconstrained* because any block from main memory can be placed in any sector in the cache. Conti [CONT69] named this organization *a fully associative buffer*; however, the term *full* or *fully* is not used in this text. However, if there is more than one block per sector, congruence mapping into the sector is required to identify the block, which was named a *sector buffer* by Conti.

Associative or unconstrained mapping does not require a set index in the address, because the tags are accessed by an associative search. In other words, the block name of the memory address and the values held in the tag field are the same. Thus, all blocks are in the same congruence class (the remainder is zero).

2. An *n-way set associative cache* is organized as ($S$, $n$, $B$, $A$) with $S \geq 2$. An example of an $n$-way set associative cache, organized as (4, 2, 1, 2), is shown in Figure 2.7. The associative cache shown in Figure 2.6 has been divided into four sets, reducing the number of comparison circuits from eight to two. The number of tag comparison circuits is equal to the value of SE and is the degree of set associativity of the cache. [MATT70] refers to the set associative cache as *partially constrained* because a block from main memory can only be placed in one

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000061

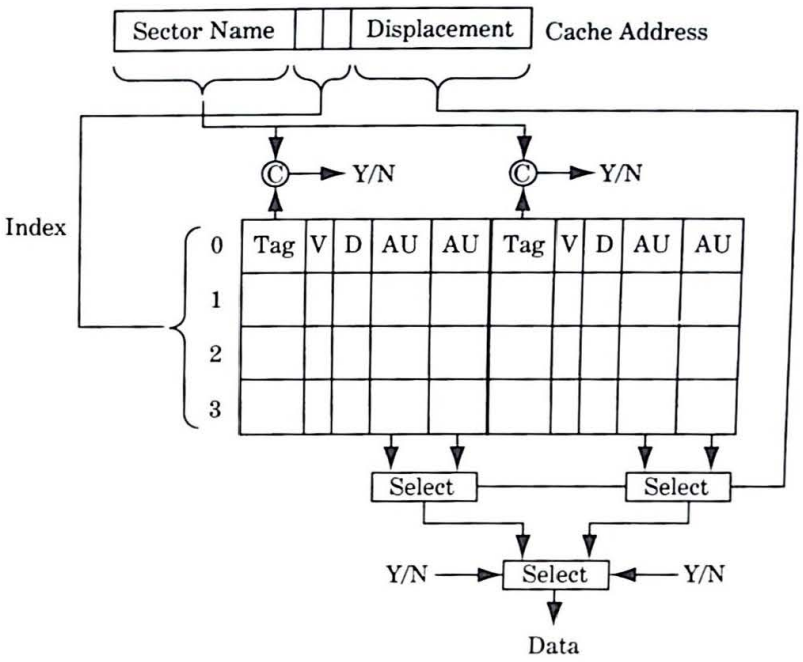



**FIGURE 2.7** Two-way associative cache.

of the sectors of its equivalent set. Conti named this organization a *set associative buffer*.

Caches with $n$-way set associativity have always, to my knowledge, used an even power of two for $SE$: 1, 2, 4, . . . . Smith [SMIT78a] provides information on selecting the degree of associativity for both caches and main memory. A recent development, the Texas Instruments Superscalar SPARC [WEIS92], has a 5-way set associative data cache. Having prime number associativity may tend to reduce set conflict misses. There seems to be no problem in addressing a set that has five sectors; the sector is selected by comparing the tag bits. There is no addressing modulo 5 required as is the case with prime number memory interleaving, which is discussed in Chapter 5.

3. A *direct cache* [BELL74] is organized as ($S$, 1, 1, $A$); an example of a direct cache organized as (8, 1, 1, 2) is shown in Figure 2.8. This direct cache is a further division of the 2-way set associative cache of Figure 2.7. With this cache, there are eight sets and only one comparison circuit; thus, it is 1-way set associative. Mattson [MATT70] referred to the direct cache as *fully constrained* because a block from main memory can only be placed in its equivalent set of one sector with one block. Conti named this direct cache organization a *direct mapped buffer*.

The following is a qualitative assessment of the effect on performance for each of the parameters of the cache organization. See Przybylski [PRZY90a] for a more complete set of "rules of thumb."

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000062

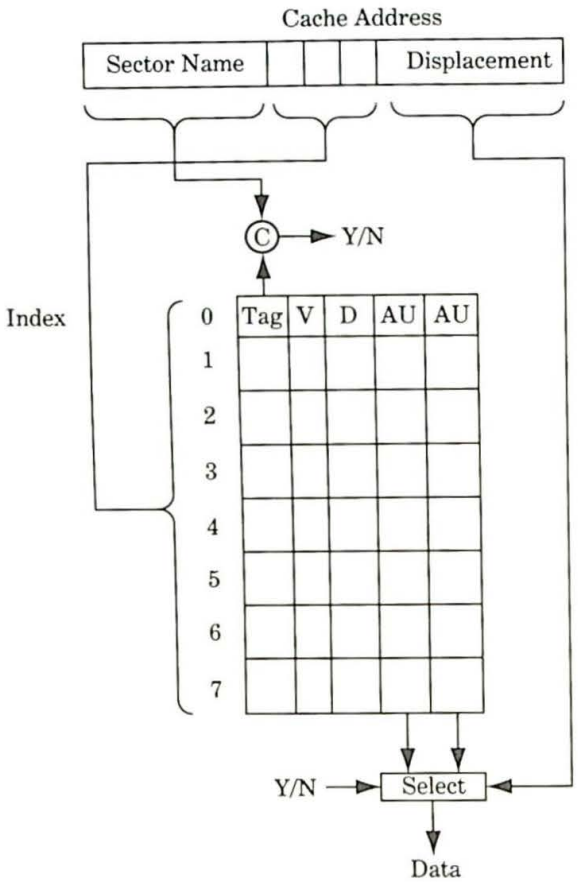



**FIGURE 2.8** Direct cache.

Cache size is a first-order determinant of cache performance. As noted above, the size of the cache is the product of the terms of the 4-tuple. However, as cache size increases there can be a detrimental effect on the processor clock period. The dominant effect of cache size will be noted when $P_{\text{miss}}$ data is examined.

The size of the AU is a function of the addressing method of the processor. In other words, the processor may do byte addressing or word addressing.

The number of AUs in a block influences the miss penalty by reducing the number of AUs that are either displaced or overwritten upon a miss. The transport time for a miss is a factor of the number of AUs in a block.

The number of blocks, $B$, in a sector is set to improve temporal locality of the cache by not requiring the replacement of a complete sector. Most contemporary caches, however, have $B = 1$.

The number of sectors, $SE$, is the degree of set associativity; it is also the number of tag comparators in the cache. Experimental data

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000063

presented later in this chapter indicates that $SE$ should be in the range of 4 to 16. As each sector has a tag that is compared to a portion of the address, the number of tag comparison circuits is equal to $SE$. The number of tag bits is also a function of $SE$, $S$, and the address space of memory. Hennessy [HENN90] provides a cache rule-of-thumb: "The miss rate of a direct-mapped cache of size $X$ is about the same as a 2- to 4-way set associative cache of size $X/2$." Hill [HILL88] points out that higher levels of associativity tend to lengthen logic path delays and increase the clock period. Because of this, a cache with a higher $P_{\text{miss}}$ can actually give the smallest $t_{\text{ea}}$.

The number of sets $S$ is important if $S \neq 1$, that is, not an associative cache. Usually $S$ is determined or selected to give the desired cache size after the other parameters are set.

## 2.1.1 *Set Index Allocation*

Figure 2.5 shows the set index field of the cache address positioned in the LSBs of the block name field of the memory address. Other options for the location of the set index field are illustrated in Figure 2.9. The examples shown are for a three-bit block name divided into two sets.

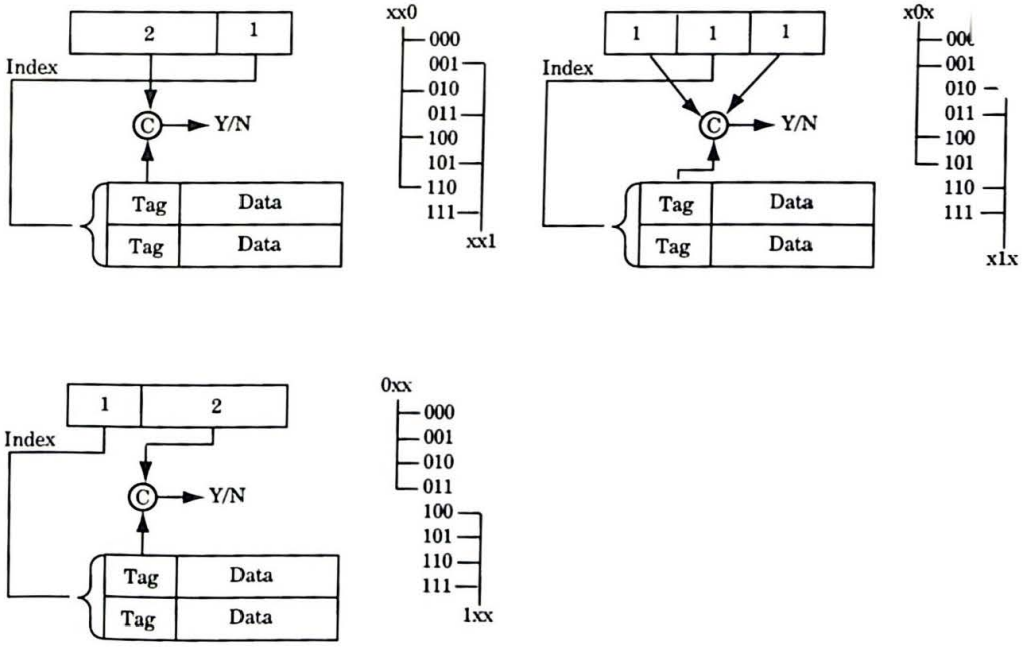



**FIGURE 2.9** Set index bit positions.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000064

The set index is one bit and can be positioned as shown in the figure. Also shown is a map of the addresses in the cache for each placement. The three illustrated options are:

1. With the one-bit set index in the LSB position, the sectors alternate in the cache sets, modulo the number of sectors in a set. This placement distributes the sectors uniformly across the cache addresses and gives congruent mapping.
2. If the one-bit set index is placed at the MSB position, the sectors are grouped together in such a way that, for this example, two adjacent sectors cannot be resident in the cache at the same time.
3. Placing the one-bit set index in the center of the address provides interdigitated groups of sectors.

The organization placing the set index field in the LSB position is the most common. As noted above, this organization places contiguous sectors into different sets of the cache, thereby reducing set conflict

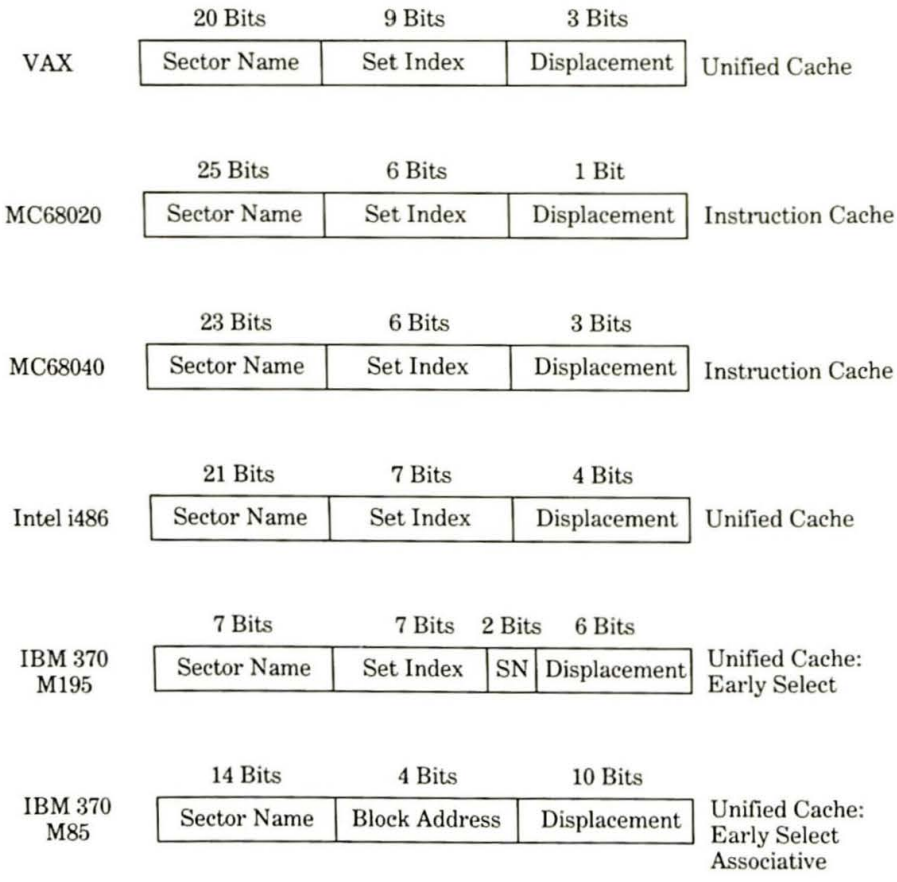



**FIGURE 2.10** Cache address field examples.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000065

misses. The author knows of no published research on the subject of how index placement affects $P_{miss}$.

Designers of cache systems have chosen to use a number of set index position schemes. Some of these options used in real systems are illustrated in Figure 2.10. These examples illustrate the diversity of index position design choices. Of particular interest is the IBM S/360/195. With this cache, the set index bits are central bits in the block name field of the address. The other illustrated designs differ primarily in the number of set index bits, not their placement.

### 2.1.2 *Late/Early Select Caches*

There are two methods for determining if a referenced AU is in the cache and, if present, accessing that AU.

1. *Early select.* With this selection method, the cache is organized with separate memories for tag/translation address and data memory and is known as early select because the determination of the presence of an AU in the cache is determined early in the cache cycle before the data memory is accessed. When a set is addressed, the tag/translation address memory is accessed and, if there is a valid tag comparison, the translated address is output. The translated address is then used as ar index into a high-speed memory (cache) as shown in Figure 2.11. This method is used in the IBM S/360/85 [MATI77], which is discussed further in Section 2.5. Because of the mapping that is performed between the cache address and the cache data memory, early select caches are associative.

2. *Late select.* With this selection method, the cache is organized with a one-to-one correspondence between the sectors and the tags. Late select organizations are illustrated in Figures 2.6–2.8 and are further described in Figure 2.11. When a set is addressed, the tag and data are read in parallel; the tag is then compared with the sector name and, if valid, the data is gated out. This selection method is known as late select because the actual selection of the AU is late in the cache cycle after the data memory is accessed.

The cache address for an early select cache is formed by concatenating the base address found in the translation map with the displacement. Concatenation is used because of the use of congruence mapping. As discussed in Section 2.2.5, a cache write is implemented as a form of an early select cache. Early select is required because the determination of the presence of a block must be made before the write can proceed.

The time steps required for an early select cache are

1. Read the tag/translation-address memory.
2. Compare the tag(s) with the sector name (hit/miss).

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000066

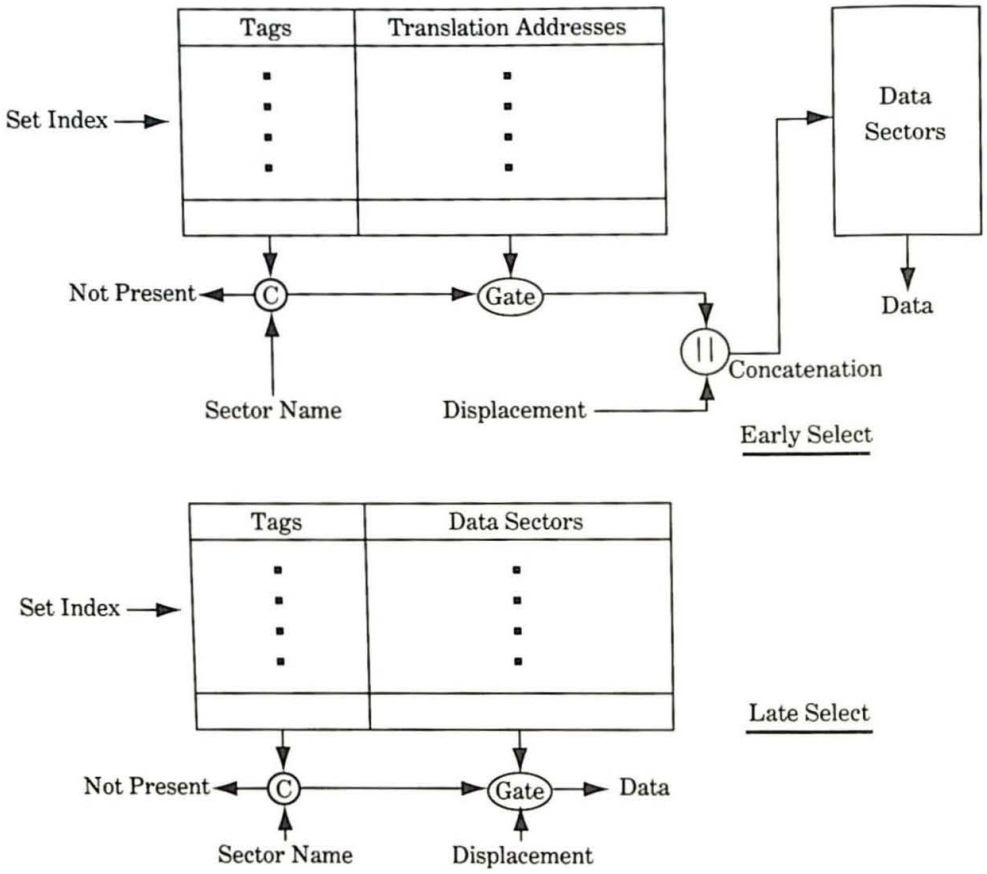



**FIGURE 2.11** Early and late select caches.

3. Compose the address.
4. Read the data memory.

The time steps required for a late select cache are:

1. Read the tag and data memories.
2. Compare the tag(s) with the sector name (hit/miss).
3. Gate out the data.

The use of early select for the IBM S/360/85 cache is based upon the flexibility of an associative allocation and the technology available at the time. The data storage portion of the cache uses high-speed core memory technology that was inappropriate for the main memory due to its high cost. The tag memory, being small, could be implemented with registers and logic circuits. Because it is fast, the delay time through the tag memory is small compared to the time through the data storage memory. That is, times 1 + 2 + 3 are small compared to time 4. Matick

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000067

[MATI77] states that the IBM S/360/85 cache requires three computer clocks to accomplish the four steps noted above.

All known contemporary caches are of the late select type. The use of late selection has come about because of the speed advantage and the introduction of cost-effective semiconductor memories that support the more homogeneous design of late select. However, as described in Chapter 3, virtual memory systems are early select.

A possible organization that combines the good points of early and late selects is the most recently used sector (MRUS) that keeps track of the last selected sector, giving a shorter logic path length than the late select path. This recently used sector, with its block, is conditionally read from the data section and presented to the processor. If the path through the tag memory indicates that the sector read is incorrect, the conditional read is aborted.

### 2.1.3 *Control Bits: Tags, Valid, Dirty, Shared, and LRU*

As shown in Figure 2.3, each sector has a tag containing the name of the stored sector. Each block has a number of control bits and a set can have LRU bits. The use of the tag bits was discussed in the previous section. The idea of sectors having multiple blocks in a sector stems from the desire to reduce bus traffic by the partial reading or writing of a sector and to reduce the number of tag bits needed for a cache of a given size.

The valid, dirty, and shared bits provide for control over a block that is resident in the cache. In general, all caches have valid bits, while dirty bits are used only with a write back cache and a shared bit is used with processors intended for multiprocessor applications.

*Valid bits* are used to indicate that the block contains valid information associated to its main memory address. The valid bit serves two functions. The first function is to indicate to the processor that the requested information is valid and not just a random collection of bits. For example, if a sector has multiple blocks, the tags can indicate the presence of the sector, but one or more of the blocks may not have been fetched. If a block that has not yet been fetched is addressed in this sector, the valid bit will indicate that the information contained should not be used. Usually valid bits are reset to zero when the contents of the cache are undefined as at the initial loading of a program into main memory.

A second use of the valid bit is described in Chapter 4. This use is associated with maintaining data coherency in the caches of multiprocessors. With one method of coherency maintenance, when noncoherence is

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000068

| # Sectors | # Blocks | LRU Placement |
|---|---|---|
| $SE = 1$ | $B = 1$ | none required |
| $SE > 1$ | $B = 1$ | place in set |
| $SE = 1$ | $B > 1$ | place in sector |
| $SE > 1$ | $B > 1$ | place in sector or set |

**TABLE 2.2** LRU bits placement.

detected, the AU in the cache is invalidated and a future reference to that AU results in a cache miss.

*Dirty bits (modified bits* or *flags)* indicate the blocks that have been written into and subsequently modified by the processor. Dirty bits are not required for instruction caches or write through data caches. By using dirty bits on write back caches, the write back traffic on the bus can be reduced to only those blocks that have been modified by writes.

*Shared bits* are used in the management of a cache in a multiprocessor configuration. Data is shared between processors and, if cached, must be managed in order to maintain coherency. This subject is discussed in more detail in Chapter 4.

For a write back cache with a valid and dirty bit, a block can be in one of three of the four possible states: valid and clean, valid and dirty, and invalid. For this third state, the dirty bit is a don't care.

*Least recently used* (LRU) *bits* provide support for selecting a block for either replacement or overwrite. As shown in Figure 2.3, a set or a sector (if $SE = 1$) may have a field of bits devoted to managing the replacement policy of the cache. This field is called the least recently used (LRU) field, and its placement is a function of the organization of the cache as shown in Table 2.2.

When there is a capacity miss (which is discussed in Section 2.2.3), one of the blocks in the sector must be evicted if the cache is write back or overwritten if write through. The LRU bits provide the information to make this selection. Examples of the use of LRU replacement policy are illustrated with the caches of MC88200 (Figure 2.19) and the i486 (Figure 2.24).

### 2.1.4 *Nonblocking Caches*

Simple caches exhibit the characteristic that a read miss or a write will lockup or stall the cache until the request is serviced. A cache lockup prevents other requests from being serviced and places the processor into a wait state. Kroft [KROF81] first proposed a lockup-free instruction cache. This work has been extended with write buffers to provide lockup-free or nonblocking caches for both reads and writes.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000069

### Nonblocking Reads

There are a number of cases where a nonblocking read is beneficial to the processor performance. In these cases, there must be a pool of read requests that can be serviced out-of-order and effectively used by the processor. Software prefetching and processors that can support out-of-order execution are examples and are discussed elsewhere. Nonblocking can be useful for both instruction and data fetches.

Additional cache hardware is required to provide nonblocking capability. From [KROF81] this hardware is:

1. an unresolved miss information/status holding register (MSHR) for each miss that will be handled concurrently or is pending;
2. one $n$-way comparator, in which $n$ is the number of MSHR registers, for registering hits $n$ data in transit from main memory;
3. an input stack to hold the total number of received data words possibly outstanding;
4. MSHR status update and collecting networks; and
5. the appropriate control unit enhancement to accommodate 1 through 4.

From this list of hardware additions it can be seen that providing a single nonblocking read is relatively inexpensive; however, providing the ability to queue up a number of reads can become unworkable. Also, the additional hardware will inevitably add gate delays that will give a slower cache and a longer hit time. The only known nonblocking caches can queue only one nonblocked read.

### Nonblocking Writes

A cache can be prevented from being placed in a wait state on a write by means of a write buffer or write queue. A buffer or queue permits writes to occur without wait states until the queue is full. The first known, to the author, example of a write buffer is the *store data buffer* (SDB) of the IBM S/360/91 [TOMA67] that held three double-precision floating point values. Store operations placed the result into the SDB and forwarded the result to destination registers as described in Chapter 8.

There are two types of write buffers: (1) for write through caches and (2) for write back caches. The Intel i486, an example of write through, has a four-level write buffer queue feeding the main memory. This write queue eliminates, in most cases, stalls of the processor on consecutive writes as measurements on the i486 show that the average number of pending writes is 3 [CRAW90]. This type of write buffer must be addressable, usually by some form of associative search, for resolving true dependencies. The IBM S/360/91 [ANDE67], for example, can buffer

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000070

three writes that have pending true dependencies (discussed further in Chapter 8). A programming style that is rich in procedure calls necessitates pushing a number of parameters on the stack that results in a number of consecutive store instructions. Thus a write buffer can significantly improve procedure call performance.

The IBM RS/6000, an example of a cache with a write back policy, has write buffers named *store back buffers* (SBB). Because the data cache is write back, all writes occur via the cache. When a block is evicted from the cache, the SBB takes a 128-byte block from the cache in one clock cycle and transfers the block to the main memory over the memory bus in 8 clock cycles [IBM90a]. The write buffers of the PowerPC 601, an architectural derivative of the RS/6000, illustrate how snooping (described in Chapter 4) is incorporated into the write operation.

## 2.2 Read Policies and Performance Models

The purpose of a cache is to provide a low-latency memory for the processor. Thus, a designer is interested in having some method of evaluating the efficacy of a design in matching the characteristics of an anticipated address stream. The following sections describe basic performance models and the cache design parameters of the models. It is helpful in modeling the operation and performance of caches to consider the following three cycles or states of operation noted by [MATI89].

1. The Read Access Cycle or State.
2. The Write Access Cycle or State.
3. The Reload Cycle, Penalty, or State (called *transport time* in this book).

The most commonly used measure of cache performance is its effective access time. The effective access time ($t_{ea}$) for a read is the weighted average access time of the AUs found in the cache and the AUs found in memory.

$$t_{ea} = (1\text{-}P_{miss}) \times \text{cache access time} + P_{miss} \times \text{transport time.}$$

Transport time is the time required to access an AU that is not found in the cache. Other terms for transport time are *miss delay, miss penalty, cache miss penalty*, and *reload time*. Transport time is discussed further in Section 2.2.1.

There are four read policies that can be modeled with variations of this simple cache performance model. These models are based on (1) the time at which the access is made to the main memory and (2) whether

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000071

or not the AU fetched from the main memory is passed to the processor in parallel with the cache or if a second reference to the cache is required. These variables are

Main memory reference.

Sequential. Memory reference is made after the miss is detected.

Concurrent. Memory reference is made in parallel with the reference to the cache.

Fetch method.

Simple fetch. AU is stored in cache, and a second cache reference is required.

Forward. AU is stored in the cache and forwarded to the processor.

Timing diagrams of these four cases are shown in Figure 2.12, where the cache cycle time is normalized to 1 and the transport time is mea-

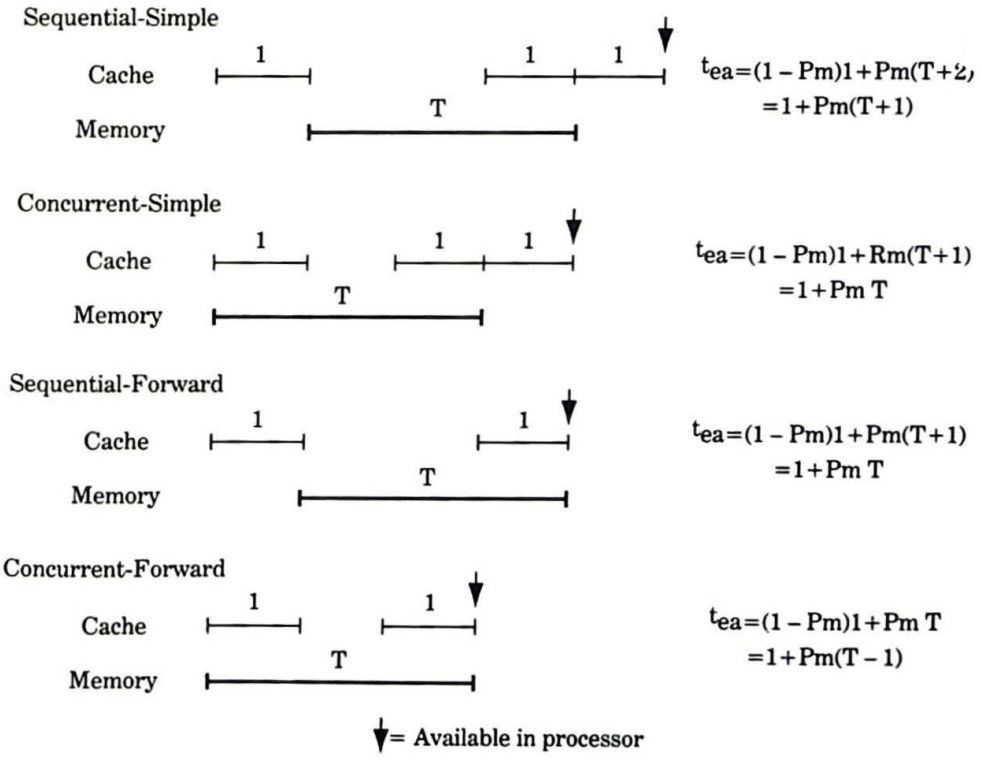



**FIGURE 2.12** Four cache models.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000072

sured in cache cycles. Consider the *sequential-simple* fetch design; a reference to the system accesses the cache first. If the requested AU is found, the effective access time is the normalized cache access time of 1. If the AU is not found, the main memory is accessed and the requested AU(s) are stored in the cache in $T$ cycles. The cache is then accessed again taking another cache cycle time. The down arrow indicates when the AU is available in the processor. The assumption is made that the cache access time is equal to the cache cycle time.

The *concurrent-simple* fetch design accesses both the cache and the main memory at the start of the access cycle. This design has the accessed AU available in the processor one clock earlier than the sequential-simple design when there is a cache miss. The operation of the other two designs can be followed from Figure 2.12. The forwarding designs pass the referenced AU to the processor in parallel with writing it into the cache.

The $t_{ea}$ models of the weighted average access times for the cache and the main memory clearly indicate the added latency that is a result of a cache miss. The *concurrent-forward* design gives the best performance—that is, the smallest $t_{ea}$—because the two memories are accessed concurrently and the transfer from the main memory is forwarded to the processor. However, this performance advantage is at the expense of a main memory cycle and a bus request for every cache access, which for some systems is an unacceptable load on the main memory bus. Thus, all known cache systems employ the *sequential-forward* design. A version of these models was first used by Lee [LEE69]. In addition to the effective access time, $t_{ea}$, there are two other metrics that are used for measuring the effectiveness of a cache design.

The *bus traffic ratio* is the ratio of bus traffic with a cache to the bus traffic without a cache [HILL84]. It should be noted that this metric can be greater than 1 in some cases. This situation occurs when the bus traffic servicing misses exceeds the bus traffic reductions due to hits in the cache. Bus traffic is discussed further in Section 2.2.4.

The inverse of the first metric is known as the *bus traffic reduction factor* (BTRF) [ALEX86].

Caches with more than one block per sector have interesting read hit and miss properties. A hit requires that the tag matches and that the addressed block is valid. A miss can occur if there is not a tag match or if a tag matches but the addressed block is not valid. On a cache miss because of the failure to match on a tag, a sector is selected by an LRU strategy and all 16 of the tag bits are reset. The selected sector tag is updated, and the sector in the data memory is then filled on a block-by-block basis as required by each subsequent reference that misses. For the second case, if there is a match on a tag but the requested block is

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000073

not valid, that block is fetched and the valid bit is set. The write policy for this organization is discussed in Section 2.2.5.

### 2.2.1 *Transport Time*

A *transport time* model describes the time required to respond to a cache miss. Transport time, measured in either clocks or in time, is a function of the method of accessing main memory and transferring a block to/from the cache. The transport time model is based on the interleaved memory configuration as shown in Figure 2.13. For interleaved memory systems, including bus systems, the model for the transport time is composed of the following parameters:

| | |
|---|---|
| $T$ | transport time. |
| $a$ | latency in clocks required to access the first AU of the block. |
| $b$ | number of clocks required to produce each subsequent AU of th block. |
| $A$ | number of AUs in a block. |
| $Wd$ | width of the path to the cache in AUs and the number of AUs per memory word. |

From these parameters, the transport time required to transfer a

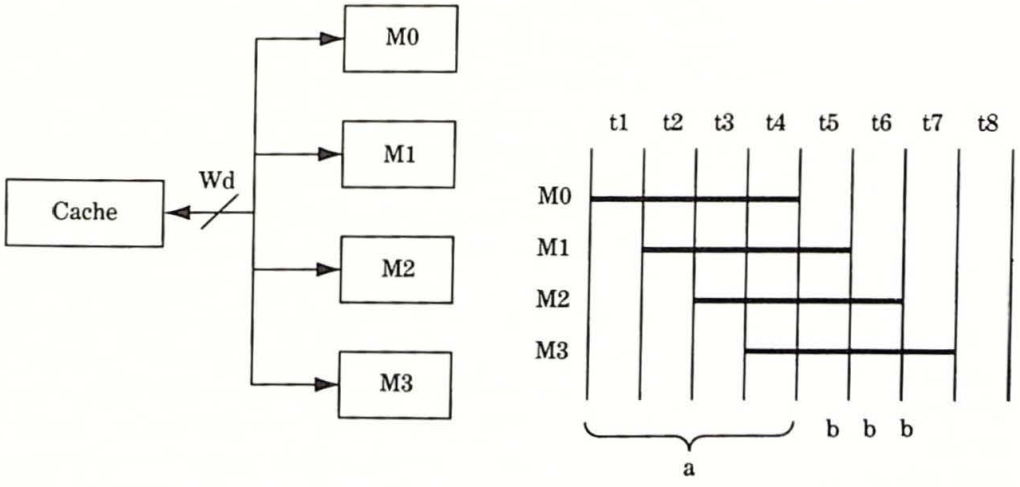



**FIGURE 2.13** Transport time model.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000074

block between the main memory and the cache is

$$T = a + b\left(\frac{A}{Wd} - 1\right).$$

This transport time model is typical of interleaved memories that supply blocks to the cache. An example is found in the IBM S/360/85, which has $a = 4$, $b = 1$, $A = 32$, and $Wd = 8$. The clock period is 80 ns. Thus, transport time is 7 clocks or 560 ns.

The transport time model will be used in a modified form in later sections when write policies are described, because some fetch and write policies require that a block must be moved from the cache to main memory (*evicted*) before the desired block is moved from the main memory to the cache. For those cases, $T$ is multiplied by $(2 - P_c)$, where $P_c$ is the probability that a block is clean, that is, the block has not been modified.

Because the transfers to/from the main memory and the cache are by way of the bus, another important performance model concerns the *bus traffic*, which is defined as

$$\begin{aligned}\text{bus traffic} &= \text{bytes per second across the cache–main memory bus}\\ &= \text{requests/inst.} \times \text{inst./clock} \times \text{clock freq.}\\ &\quad \times \text{block size} \times P_{\text{miss}}\ .\end{aligned}$$

This model assumes that there is one bus transaction for each cache miss—a correct assumption for read misses but not for write misses. Nevertheless, the model gives some indication of the bandwidth required of a bus. For example, a processor with requests/inst. = 2, inst./clock = 0.66, clock freq. = 10 MHz, block size = 16 bytes, and $P_{\text{miss}} = 0.05$ will consume 10.66 million bytes per second of bus capacity. This model will be modified in the following sections to accommodate the write policy employed for the cache.

VLSI technology makes possible large caches with small values or $P_{\text{miss}}$. Thus, a designer is quite concerned with reducing the transport time without the complications of interleaving. When interleaving is not used, the transport time is simply the number of memory cycles (in time or clocks) required to transfer the block. Today, semiconductor manufacturers are producing DRAMS with organizations that support the efficient transfer of a block to a cache. These organizations, discussed further in Section 5.1.6, are known as *nibble mode, word (page) mode*, and static *column mode* devices.

In the late 1980s, VLSI implementation of caches became practical and changed the long-held assumption of independence of cache cycle

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000075

time and transport time. Transport times have been reduced to a few clocks (2–5), and processor clock periods are a few tens of nanoseconds. Jouppi disputes this assertion, however, and says that transport times are lengthening [JOUP90].

### 2.2.2 Cache Miss Characteristics

Traditionally, the three major cache parameters (cache access time, $P_{miss}$ and $T$) have been viewed as orthogonal. That is, one parameter can be changed independently of the other two. There is evidence that this orthogonal view is no longer correct. For example, a larger cache may provide a lower $P_{miss}$ but will have a longer access time. And, as $T$ is an integer multiple of the cache access time, there can be an increase in $T$. The issue of cache design becomes one of minimizing the effective access time of the cache as the three parameters are varied. This design approach is the subject of current research by [HILL88, HILL87, PRZY88, PRZY90, JOUP90].

There are a number of conditions, both transient and steady state, under which misses occur in a cache. From the discussion to follow on the various causes for misses, a cache designer should be cautious about accepting only one value of cache miss. $P_{miss}$ can vary widely due to the instantaneous state of the processor and the nature of the program that is in execution.

Transient misses occur due to initial loads, context switches, and operating system calls. Two types of transient miss conditions have been noted by [AGAR88].

*Cold start misses* arise because the program starts executing with an empty cache, the same as compulsory misses.

*Warm start misses* arise after a context switch that has evicted a portion of the cache's contents.

A warm start miss is also known as the *cache reload transient* [THIE87]. Thiebaut and Stone studied cache transients and developed an analytical model to measure its influence on cache performance.

The steady-state misses are those that occur after the transients have subsided. These misses are classified by [HILL87].

*Compulsory misses* arise from references to previously unreferenced blocks. The compulsory $P_{miss}$ is equal to the $P_{miss}$ of an infinite cache.

*Capacity misses* arise because the cache is finite. Space must be provided in the cache for the referenced AUs.

*Set conflict misses* arise because of too many references to the AUs in the same set or congruence class. Space must be provided for the referenced AU.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000076

A classification of factors that cause cache misses is given by Agarwal et al. [AGAR89] and reviewed here.

*Startup effects* are the same as cold start misses and compulsory misses.

*Nonstationary behavior* of the cache is caused by the slow change of a program's working set over time.

*Intrinsic interference* is the same as set conflict misses.

*Extrinsic interference* involves the invalidation of cache locations of one program by another program in a multiprogramming system.

The transient behavior of cache misses was studied by Voldman and Hoevel [VOLD81]. They proved information on miss spectrum for instructions and data as well as intermiss distances. Their results clearly show the effect of the change in locality resulting from a context switch on the cache $P_{miss}$. Additional information is given in [VOLD83] that shows the fractal nature of the change in locality.

The cache performance models discussed previously and shown in Figure 2.12 use some form of weighted average $P_{miss}$. However, with capacity misses and set conflict misses, transport time may increase to $2T$ to accommodate the time needed to write a block and then access a block. For a write back cache (discussed in Section 2.2.5), the fetch policy requires that a dirty block is written into the main memory for a capacity or set conflict miss. This write is not required for a write through policy.

The sequential-forward model will be modified to account for the need to evict or write a dirty block to the main memory before the missed block is fetched. Clean data blocks do not have to be evicted even on a capacity or set conflict miss. This model applies only to data references under the assumption that instruction fetches do not need to evict a block; instead, the fetched block can overwrite a cache block.

$$\begin{aligned} t_{ea} &= (1 - P_{miss}) \times 1 + P_{miss}(P_c(T + 1) + (1 - P_c)(2T + 1) \\ &= 1 + P_{miss}T(2 - P_c) \end{aligned}$$

where $P_c$ is the probability that the block is clean (not dirty).

The effect of various forms of miss is illustrated by considering the performance model for a sequential-forward, write back cache. This model, which accommodates the block writes that occur with capacity and set conflict misses, is

$$\begin{aligned} t_{ea} &= 1 + P_{miss}T \\ &= 1 + P_{mcomp}T + (P_{mcap} + P_{mset})T(2 - P_c) \end{aligned}$$

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000077

where

$$P_{mcomp} = \text{probability of a compulsory miss,}$$
$$P_{mcap} = \text{probability of a capacity miss,}$$
$$P_{mset} = \text{probability of a set conflict miss,}$$
$$1 = P_{mcomp} + P_{mcap} + P_{mset}.$$

This model includes the effect that, for a write back cache, a compulsory miss requires only one transport time while capacity and set conflict misses require two transport times if the block is dirty in the slot that must be available for allocating the fetched block.

### 2.2.3 *Sources of Miss and Hit Data*

A cache designer has a number of sources for $P_{miss}$ data. The first, and most abundant, source of general data is published data by [SMIT82], [SMIT87], and [HILL84]. In addition, [HILL87] also reports miss data for the three classes of steady-state misses noted above. Transient miss data is provided by [STRE83] and [THIE87]. Miss data for the SPEC benchmarks suit executing on a number of processors was published by Gee et al. [GEE91].

Davidson [DAVI87] studied cache misses on the basic and two modified versions of the VAX instruction set. A number of benchmarks are simulated, and the results indicate that for relatively small caches (8K to 32 Kbytes) there can be as much as a 60% difference in $P_{miss}$ for the same program. For large caches (64 Kbytes), the influence of code density disappears as the cache can hold the complete working set. These results indicate that code density can be a significant design consideration for on-chip caches that are of limited size. The point should be made that $P_{miss}$ is not independent of the underlying processor architecture. That is, the instruction set and resulting code density can be a significant determinant of $P_{miss}$. For a given size of cache, the higher the code density, the lower the $P_{miss}$. While instruction set design is outside the scope of this book, I suggest consulting the research of [MITC86] and [STEE89] in addition to [DAVI87].

Miss data for operating systems and multiprogramming workloads is provided by [AGAR88]. This data is collected by trace-driven simulation for various configurations of caches with various workloads and measurements on actual caches when executing benchmarks.

Mogul and Borg [MOGU91] reported on the effects of context switches on the performance of a cache. Their results are important because of the increased context switch rate that results from program execution under highly modularized operating systems. They simulated

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000078

three caches: a level-one instruction cache, a level-one data cache, and a level-two unified cache. Their results show that the cost of a context switch can be in the tens of thousands of cycles due to the reduction in cache performance or the increase in memory latency.

The empirically derived formulas by [HIGB90] are another source of miss data source. These formulas are the product of curve fitting to trace-driven simulation data. Higbie also gave formulas for evaluating the average access time of a cache as a function of its organization. These formulas are useful for making first-order performance approximations of a cache design before expensive simulations are conducted. The discussions of each of the cache types in Sections 2.3 to 2.6 include identification of the published $P_{miss}$ data for the various cache types.

In conclusion, $P_{miss}$ data is usually obtained by simulating a cache, testing it with an address trace, and recording the $P_{miss}$ results. Special methods of performing cache simulations have been effective in significantly reducing the simulation time for a large number of cache designs [HILL87, MATT70]. Some new microprocessors, such as the Intel Pentium, have counters on the chip that can log cache misses permitting in situ and at-speed miss data collection.

### 2.2.4 *Fetch Policies*

In addition to the organizational aspects of cache design, there is the issue of the *fetch policy* that can influence the effectiveness of a cache by changing $P_{miss}$. It is stated as the algorithm for determining (1) the conditions that trigger a fetch from the main memory into the cache, (2) the order of moving AUs from the main memory to the cache, and (3) when the CPU can resume processing after a cache miss. Fetch policy is also called *fetch strategy* [PRZY90].

The policies discussed in this section are used for both instruction and data caches. However, most instruction caches only have need for a fetch policy, since writes into an instruction cache are usually not allowed. Recall, from Table 1.1, that approximately 90% of all memory references are reads and 10% writes. Our primary attention is devoted, therefore, to reducing fetch time even at the expense of write time.

Before fetch policies are discussed, a historical note is in order. The first caches were quite small by modern standards. Because the caches were small, capacity misses are large and "fine tuning" of the cache organization and fetch policy was employed to improve $P_{miss}$ as much as possible. For example, Gibson [GIBS67] examined the impact of block size in a small cache.

However, as the absolute size of the cache has increased and with the use of two-level caches, fine tuning of block size and fetch policies

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000079

are no longer the dominant considerations they once were. The use of direct caches with $SE = 1$ and $B =$ one word, as found in the MIPS R2000, indicates current thinking in cache design. Also, Przybylski's [PRZY90] simulations indicate that "fine tuning" of the fetch policy can yield only marginal improvements in cache performance. Nevertheless, it is important to consider all aspects of fetch policy design as some of the issues remain relevant.

### Conditions for a Fetch from Main Memory

This section briefly discusses the three common conditions for initiating a fetch from the main memory to the cache. The first two conditions apply equally to instruction and data references to split or unified caches.

*Fetch on miss.* This condition is the common case of accessing a block from the main memory into the cache only on a cache miss, sometimes known as *demand fetching*. For caches having more than one AU per block, prefetching occurs because more than the missed AU is fetched.

*Software prefetching.* With software prefetching, a block is fetched under program control or automatically before an AU is needed. When software prefetching is used to preload a conventional cache, it is anticipated that there will be a decrease in $P_{\text{miss}}$ for the program. Software prefetching with a performance model is described in Section 2.8.1.

Software prefetching is a much studied but, as yet, little implemented idea. The reason for this is discussed in Section 2.8.1, where a performance model for software prefetching is developed and its performance is evaluated. Special cases of software prefetching can be found, such as instruction buffering, where prefetching is effective.

*Hardware prefetching.* A special form of prefetching is hardware driven and is used for instruction and data stream prefetching into queues (described in Section 2.3.2) and prefetching of branch target instructions (described in Chapter 6). A special case of hardware prefetching is *instruction prefetching*, which is defined as: "Instruction words ahead of the one currently being decoded are fetched from the main memory before the instruction decoding unit requests them" [GROH82]. This definition is general enough to cover prefetching into queues as well as into pipeline stages.

### Ordering Main Memory to Cache Moves

For cache having more than one AU per block, the effective access time, $t_{ea}$, can be improved by the use of an efficient fetch policy. There are three known design options for handling multi-AU block on a fetch. These options assume that a cache read miss can occur on any address within the block, including the first address. Recall that the block is the unit of transfer between the main memory and the cache. Timing dia-

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000080

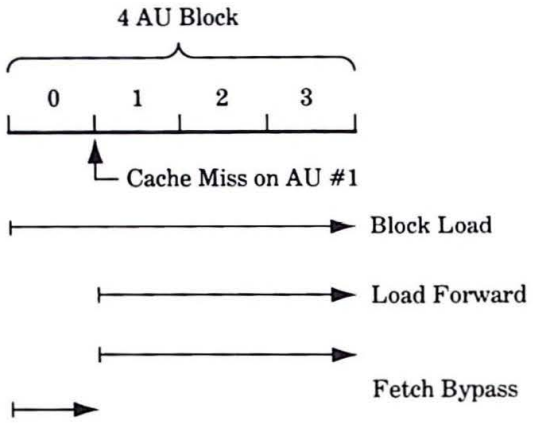



**FIGURE 2.14** Load policies.

grams of the following three options are shown in Figure 2.14, illustrated with a miss occurring on the access of AU#1 of a four-AU block.

1. *Block load*. The complete block is loaded starting with the first AU in the block [HILL84].
2. *Load forward*. The AUs forward of the miss and within the block are loaded. The AUs behind the miss are not loaded. This option is also named *loading partial blocks on miss* [HILL87]. Note that valid bits are needed for each AU and the valid bits of only the loaded AUs are set. With a valid bit, a miss can occur either because of a tag that does not compare or a valid bit in a block that is not set.
3. *Fetch bypass*. The block is loaded starting with the missed AU; then the AUs behind the missed block are loaded [SMIT87]. Fetch bypass is also known as a *wraparound load*. This option is employed on the IBM S/360/85 [LIPT68]. Unfortunately, the term *fetch bypass* has another usage with the IBM RS/6000; see Section 2.4.2.

For systems that have a bus with a high level of utilization, selecting a load policy that reduces bus usage may be beneficial. Note that for options 1 and 3 the complete block is always loaded into the cache. Option 2 loads only a partial block, thereby reducing the bus traffic ratio, usually at the later expense of an increased $P_{\text{miss}}$. The higher $P_{\text{miss}}$ may then increase the bus traffic, for no net average reduction. Hill [HILL84] performed simulations that suggest that the load forward policy reduces the bus traffic ratio in the range of 10% to 20% at the cost of an increase in $P_{\text{miss}}$ of less than 10%. The net effect of these changes is negligible and is probably not worth the complication of the load forward design.

There are further complications with a fetch policy. For example, what happens with a sequential-forward or concurrent-forward policy if there is another cache miss while the block, or partial block, from the

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000081

previous miss is still being loaded into the cache? There are two options for handling this situation [PRZY90].

1. For a nonblocking cache, hold the second miss until the first block load is completed. This strategy is called *nbdwf*.
2. The block load from the first miss is aborted and the second is started. This strategy is called *adwf*.

The temporal clustering of cache misses for a particular cache design and workload is a major factor in selecting an option; [PRZY90] provides some insight. As the block size increases, the temporal clustering decreases. However, the decrease is not significant for relatively small blocks, leading to the conclusion that the simple *nbdwf* option is the best choice.

### Resumption of Processing

The processor can resume processing when the requested AU is available. There are two variations of each of the three load policies that concern how the missed AU is passed to the processor. These two policies are introduced in Section 2.2 and shown in Figure 2.12.

1. *Simple fetch*.The block is loaded into the cache, with any of the three policies above, and the reference to the cache is reinitiated.
2. *Forward*. As the block is being loaded into the cache, the missed AU is passed to the processor that resumes processing while the balance of the block is filled. This policy is named *out-of-order fetch* [HENN90], *continuation*, or *early restart* [PRZY90].

In general, the forward policy reduces the number of cycles required to service completely a cache miss and resume processing. However, this policy is more difficult to implement than the simple policy and may not be worth the cost for systems with small $P_{\text{miss}}$ and relatively short transport time. And, as noted above, there can be cache misses because of the early restart of processing.

With a nonblocking cache (discussed in Section 2.1.4) and a pool or queue of reads, the processor can resume processing immediately after the miss service is initiated. Thus, the simple fetch policy may be satisfactory.

## 2.2.5 *Write Policies*

Section 2.2.4 discussed fetches from the main memory to the cache. When there is a data write from the processor, the cache must have a policy for handling a number of situations. Consider first a write to a sector or block that may or may not be in the cache. An *early select* strategy must

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000082

be used because the cache must be checked for a hit or miss before the write is initiated. What this means is that writes are accomplished by a read–test–modify–write protocol. A *late select* policy, if one could be devised, would cause an invalid write that would incorrectly change the value of an AU in the cache.

Instruction caches are, in general, read-only and do not require a write policy. For data writes, policies for all cases of hits and misses and writing into the cache and main memory are needed.

## Write Hit

After the early select process determines that the write will hit in the cache (that is, the tags match), the write must update both the cache and the main memory. For write hits there are three strategies.

1. *Write through (store through)*. For this policy, the AU is written into the cache and into the main memory. For performance enhancement, a write buffer can be employed to hold the AU before it is written to the main memory. The write buffer permits the processor to continue while the write to the main memory is pending.

Write through has a number of characteristics that must be accommodated. (1) There can be a coherency problem because the value of the AU in the cache can be different from the value in the main memory while the AU is in the buffer. (2) The write through policy requires that every write to a location causes a memory bus transaction. And, (3) as there can be a sequence of writes to a particular location, there will be more bus activity than with the write back policy.

2. *Write back (copy back, store in, nonstore through, swapping)*. For the write back policy, the AU is written into the cache and the dirty bit is set if it has not already been set. At some future time when the sector or block must be evicted from the cache for a read miss, only the dirty block(s) must be written to the main memory. Dirty bits would not be required if the full sector was always moved to the main memory on a miss. However, writing a full sector imposes a significant performance penalty because of unnecessary moves. With dirty bit control, only the last write to a location is updated into the main memory, which results in lower bus activity than with the write through policy.

A read miss may require the eviction of a block if the block is dirty. Thus, the selection of a write back policy can impact the transport time required to service a future read miss. There is a coherency problem with the write back policy because the value of the AU is different in the cache and the main memory until the write back is accomplished. Coherency is discussed in Chapter 11.

3. *Write once*. This policy treats the first write as write through and then reverts to write back for all subsequent writes to the same block.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000083

Write once reduces the need for invalidations or updates for maintaining coherency, which is a problem with multiprocessor systems, while reducing the bus traffic found with a pure write through policy.

### Write Miss

The first-level design options for write misses (that is, the tags do not match) concerns whether on not the block into which the write will occur is allocated (fetched from the main memory) or not allocated. The second-level design options are *write through* and *write back* as with write hits. The allocation design options are as follows.

*Write-allocate* can be used by either the write through or write back policies and treats a write miss similarly to a read miss. The sector or block that is the target of the write is fetched into the cache. One of the replacement policies described in Section 2.2.6 must be observed. After the block is fetched, either write through or write back policy can be followed.

*No-write-allocate (write around, write direct)* is used only with the write through policy. With this policy, the write is made directly to the main memory or to the main memory via a buffer as in the write through policy.

### Writes into Write Through Caches

Write through caches with multiple AUs per block have an advantage with writing [HENN90]. If there is a hit (tag match) the write to the cache, and to the main memory, can proceed regardless of the state of the valid bit. However, if there is not a tag match, the AU can be written, the tag updated, the valid bit set, the other valid bits in the block reset, and the AU written into the main memory. The reason this works is that there is a valid image of the block in the main memory for the block that does not hit.

### Write Policy Taxonomy

With two policy options for a write hit (not considering the write once policy) and three policy options for a write miss, there are six possible write designs, as shown in Table 2.3.

Four combinations of policies are the most used and are indicated by designs A, B, C, and D. Note that options B and D use allocate, and the $P_{miss}$ of future reads may be reduced because a requested AU may now be present in the cache. On the other hand, a future read may have a higher $P_{miss}$ due to the eviction of a block that will be needed at that time. There is no published research on this issue known to the author. Cases A and C require that the block is fetched with read misses only

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000084

| Design | Write Hit | Write Miss |
|---|---|---|
| A | Write through | No write-allocate, write through |
| B | Write through | Write-allocate, write through |
| | Write through | Write-allocate, write back |
| C | Write back | No write-allocate, write through |
| | Write back | Write-allocate, write through |
| D | Write back | Write-allocate, write back |

**TABLE 2.3** Write policy taxonomy.

since write misses do not cause a block to be fetched from the main memory.

With the four designs, there is another design option. This option is the inclusion of a write buffer or queue discussed in Section 2.1.4. All known contemporary cache designs use a write buffer. Recall that the write buffer used with a write through cache (designs A and B) buffers processors writes while a write back cache (designs C and D) uses a write buffer with the eviction of a block.

## Write Policy Performance Models

Simple models are developed to assess the performance of the four cache designs. There are two performance issues that are a consequence of the selected write policy: (1) the effective write time of the cache, and (2) the bus traffic. These issues are discussed separately later in this section.

Simple flow charts for the four write designs are shown in Figure 2.15. The models are for a unified cache and assume the following.

1. All hits take one clock to read from the cache.
2. The transport time of a block to/from the main memory is $T$ bus cycles and the ordering is block load.
3. For misses where a dirty block is moved from the cache to the main memory before the block is fetched, the transport time is $2T$ bus cycles.
4. All writes are buffered and take 1 cycle; the write buffer is never full.
5. Write to the main memory of a single AU requires w cycles on the bus.
6. Dirty bits are set in parallel with the cache and/or the main memory buffer write.

### *Cache Access Time Models*

For the four cache designs A–D described previously and illustrated in the flow charts, Table 2.4 describes the write time in cycles. Because of

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000085

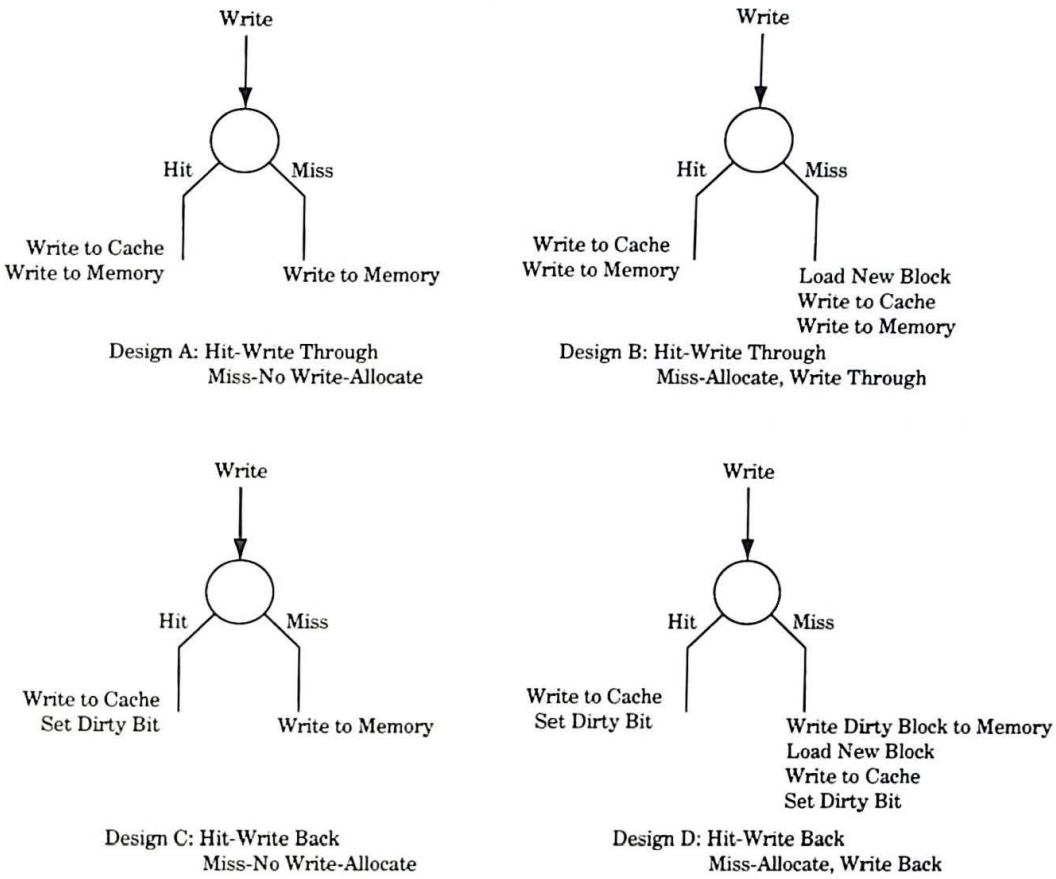



**FIGURE 2.15** Write design flow charts.

the use of write buffers, all writes take one cycle. All actual writes to memory are overlapped and do not delay the processor.

Note that while the hit events for designs C and D set the dirty bits, no eviction transport time is charged because the transport time is accounted for on a subsequent read that requires that the block be evicted. The effective write time $t_w$ is the same for all cache designs and is based on the effective use of write buffers to hide all transport times

| Design | Write Hit | Write Miss | Examples |
|---|---|---|---|
| A Hit-WT, Miss-NWA | 1 | 1 | i486, MC68040* |
| B Hit-WT, Miss-A, WT | 1 | 1 | MC68030* |
| C Hit-WB, Miss-NWA | 1 | 1 | |
| D Hit-WB, Miss-A, WB | 1 | 1 | MC68040* |

*Depending upon the mode.

**TABLE 2.4** Cache write cycles.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000086

| Write Design | Read | | Write | |
|---|---|---|---|---|
| | Hit | Miss | Hit | Miss |
| A Hit-WT, Miss-NWA | 1 | $1 + T$ | 1 | 1 |
| B Hit-WT, Miss-A, WT | 1 | $1 + T$ | 1 | 1 |
| C Hit-WB, Miss-NWA | 1 | $1 + T(2 - P_c)$ | 1 | 1 |
| D Hit-WB, Miss-A, WB | 1 | $1 + T(2 - P_c)$ | 1 | 1 |

$P_c$ is the probability that a block is clean; that is, the block has not been modified.

**TABLE 2.5** Memory access cycles.

$$t_w(A) = t_w(B) = t_w(C) = t_w(D) = 1 \text{ cycle}.$$

We now consider the overall effect of the write policy on the effective cache access time by modeling the effective cache access time for both reads and writes. The read policy is sequential-forward. The memory access cycles are shown in Table 2.5.

The effective access time for both reads and writes can be modeled by forming the weighted average of the clocks required for reads and writes. A unified cache is assumed even though split caches can also be evaluated by selecting the proper parameters. New terms for these models are defined as

$P_r$ Probability that a memory access is a read
$P_{mr}$ Probability that a read access misses
$P_w$ Probability that a memory access is a write
$P_{mw}$ Probability that a write access misses

$$P_r + P_w = 1.$$

Designs A and B provide the same effective cache access time; designs C and D also have the same effective access times

$$t_{ea}(A) = t_{ea}(B) = P_r(1 + P_{mr}T) + P_w,$$
$$t_{ea}(C) = t_{ea}(D) = P_r(1 + P_{mr}T(2 - P_c)) + P_w.$$

*3us Utilization*

The effective access time of the cache is not the only performance measure that should be considered. Each of the four designs presents a different traffic density or bus utilization. High bus utilization can be a significant problem in a shared memory multiprocessor because of contention on the bus from a number of processors. A write back policy

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000087

is expected to result in lower bus utilization. The models to follow show the number of clock cycles that each reference to the cache will generate; bus activity is measured as

BCR = the weighted average number of bus clock cycles per memory reference.

If a split cache is evaluated, BCR will be the sum of the BCRs for each of the two caches. The model of bus utilization makes the assumption that the cache is always active; that is, the processor is completely overlapped with cache access time. Thus, the bus utilization fraction is

$$\text{bus utilization (BU)} = \text{BCR}/t_{ea} \text{ (A, B, C, or D)}.$$

The transport time model determines the time to move a block between the cache and the main memory. For write through and no-write-allocate caches only a single AU, not a block, is usually written to the main memory and a bus and the main memory usually handle single AU writes differently than a block write. While write buffers hide a single AU write time from the processor, the number of writes is a factor in determining bus traffic. Therefore, a parameter is needed to quantify single AU writes to the main memory; specifically it is defined as $w$, the number of clocks required to write an AU to the main memory and is usually less than $T$ because a single AU is written to the main memory over the bus rather than a complete block.

Table 2.6 shows the contribution made to BCR for each of the four events and each of the four designs. These models are derived from the flow charts of Figure 2.15 and Tables 2.4 and 2.5. The models are the same with and without buffers because a buffer can only delay, but not eliminate, bus activity.

| Write Design | Read | | Write | |
|---|---|---|---|---|
| | Hit | Miss | Hit | Miss |
| A Hit-WT, Miss-NWA | 0 | $T$ | $w$ | $w$ |
| B Hit-WT, Miss-A, WT | 0 | $T$ | $w$ | $T(2 - P_c) + w$ |
| C Hit-WB, Miss-NWA | 0 | $T(2 - P_c)$ | 0 | $w$ |
| D Hit-WB, Miss-A, WB | 0 | $T(2 - P_c)$ | 0 | $T(2 - P_c)$ |

**TABLE 2.6** Bus clock cycles, BCR.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000088

The BCRs for each of the four designs are

$$\begin{aligned}
\text{BCR(A)} &= P_r P_{mr} T + P_w w, \\
\text{BCR(B)} &= P_r P_{mr} T + P_w (w + P_{mw} \text{T}(2 - P_c)), \\
\text{BCR(C)} &= P_r P_{mr} T(2 - \text{P}_c) + P_w P_{mw} w, \\
\text{BCR(D)} &= T(2 - P_c)(P_r P_{mr} + P_w P_{mw}).
\end{aligned}$$

For evictions, transport time can be reduced for a cache that has more than one AU per block if dirty bits are given to each AU. These dirty bits can reduce the bus traffic to $w$ clock cycles for each AU transported, not $T$ cycles for the complete block. Needless to say, instruction caches do not require dirty bits because there will be no writes for systems that prohibit self-modifying code.

## Write Policy Performance Impact

The models developed in the previous section will now be used to evaluate the four cache designs for access time and bus utilization. The parameters of the bus design are from Figure 2.13 plus the word write time: $a$ is the bus latency, $b$ is the transfer rate, and $w$ is the write time. The following assumptions are made on the workload and bus parameters.

$$\begin{aligned}
P_r &= 0.9, & a &= 2, \\
P_{mr} &= 0.1, & b &= 1, \\
P_w &= 0.1, & A &= 16, \\
P_{mw} &= 0.15, & Wd &= 4, \\
P_c &= 0.9, & w &= 2.
\end{aligned}$$

By applying the models developed in the previous section, we obtain values for $t_{ea}$, BCR, and BU as shown in Table 2.7 for the four designs. First, the value of $T$ is determined as

$$T = 2 + 1\left(\frac{16}{4} - 1\right) = 5 \text{ clocks}.$$

These results indicate that, as expected, designs A and B have the

| Write Design | $t_{ea}$ | BCR | BU |
|---|---|---|---|
| A Hit-WT, Miss-NWA | 1.45 | 0.65 | 0.45 |
| B Hit-WT, Miss-A, WT | 1.45 | 0.73 | 0.50 |
| C Hit-WB, Miss-NWA | 1.49 | 0.52 | 0.35 |
| D Hit-WB, Miss-A, WB | 1.49 | 0.57 | 0.38 |

**TABLE 2.7** Cache design model results, clocks.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000089

best performance and the highest bus utilization. These results compare favorably to simulation results of the cache for the i486 described by [GROC89]. The i486 has a unified cache that is write through on hit and write direct on miss (design A). Their simulated results show BU in the range of 0.38 to 0.55 for a suit of five benchmarks with the same bus design.

Simulations were also performed to assess the impact of changing the bus parameters for design A. The bus transport time parameters $a$, $b$, and $w$ are varied; $Wd$ is 4 and $A$ is 16. For each of the bus configurations the execution clocks are normalized to a bus with $a = 0$ and $w = 0$. The simulations provided a count of the number of execution clocks with different bus configurations; the results are compared to the models in Table 2.8. The simulation-measured number of execution clocks as the bus parameters is changed. As a surrogate for processor performance, the model for $t_{ea}$ is used also to normalize $a = 0$ and $w = 0$.

The model results compare favorably with the simulation results, that is, ±8%. The bus designs with the greatest difference are the 2-2-2 and 4-1-4. One reason for the differences is that the simulation measured the effect of processor execution clocks rather than just the cache clocks. In an actual system the processor and cache are not completely overlapped, and the cache is the idle part of the time. For example, the processor may have to wait on a write to the buffer to update in the main memory; this is a delay that is in the simulator but not in the model. Nevertheless, the models do indicate the effect of the bus design on the performance of a cache. The values of $a$ and $b$ are more important than $w$.

[GROC89] also performed a simulation that evaluates the bus utilization for four bus designs with both write through and write back policies. The columns of Table 2.9 labeled SIM are the simulation results, while the columns labeled MOD are the model results. The workload

| Bus Designs ($a$ $b$ $w$) | Normalized Execution Clocks | Normalized $t_{ea}$ |
|---|---|---|
| 2 1 2 | 1.17 | 1.14 |
| 3 1 2 | 1.21 | 1.21 |
| 4 1 2 | 1.26 | 1.28 |
| 2 1 3 | 1.21 | 1.14 |
| 2 2 2 | 1.28 | 1.38 |
| 4 1 4 | 1.38 | 1.28 |

**TABLE 2.8** Simulation and model results comparison.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000090

| Bus Design | Write Through | | Write Back | |
|---|---|---|---|---|
| *a b w* | SIM | MOD | SIM | MOD |
| 2 1 2 | 0.40 | 0.45 | 0.27 | 0.35 |
| 4 1 3 | 0.53 | 0.57 | 0.35 | 0.44 |
| 6 2 4 | 0.68 | 0.71 | 0.48 | 0.57 |

**TABLE 2.9** Bus utilization.

and cache design parameters are the same as used in the examples above; only the bus parameters are varied.

The modeled values of bus utilization are quite close to those found by simulation, validating somewhat the analytic models. The model results are, in all cases, conservative, showing greater utilization that is indicated by simulation. As would be expected, a slow bus exhibits higher utilization for both write through and write back designs.

### 2.2.6 *Replacement Policy*

When there is a read miss or a write-allocate, it may be necessary to evict a block from the cache to the main memory to make room for the block that is to be fetched.

*Replacement policy* determines the block or sector to be evicted; hence, it is only needed for write back data or unified caches as write through data caches and instruction caches can have a block overwritten as the main memory is consistent with the cache. The cache organizations and their constraints, also noted in Section 2.1, on a replacement policy are as follows.

*Direct caches* ($S$, 1, 1, $A$) offer no eviction alternatives. There is only one block into which the new block can be placed.

Even though *direct caches* ($S$, 1, $B$, $A$) may have more than one block per sector, a selection of blocks is not required as the sectors and blocks are uniquely addressed.

*Set associative caches* ($S$, $SE$, 1, $A$), ($S$, $SE$, $B$, $A$) have more than one sector within a set and each sector may have more than one block. Thus the replacement choice is one of selecting the sector and/or block based on a use factor.

The fetched block can replace any block of an *associative cache* (1, $SE$, 1, $A$). The selection techniques for identifying a sector or block are the same as for the set associative cache. The set chosen for eviction is determined by a use factor.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000091

For the caches with replacement policies based on a use factor, there are a number of methods that can be used to select the sector or block for eviction. These methods have been extensively discussed by [PUZA85] and others and were first investigated in the context of virtual memory; much of the relevant literature is found in that area. The primary concern when selecting a method is to reduce the bus traffic that will result if a sector or block is moved to the main memory only to be needed again in a short time. In other words, attempts are made to identify temporal locality and only move a sector or block that will not (one hopes) be needed in the near future. These selection methods are briefly discussed below for blocks or sectors; the term *block* will be used for this discussion.

The block to be evicted can be selected by a *random choice*. A random choice can be made in a number of ways, including sampling a pseudorandom number generator, such as the low-order bits of the real time clock, in order to obtain the designator of the block to be evicted.

The block can be selected based upon the criteria of *least recently used* (LRU). The LRU policy is based on the idea that the least recently used block is least likely to be used in the future. Implementation of this method requires that an activity file be maintained for each block; the activity file is probed to find the block to evict.

The LRU strategy has the unfortunate property that a block can be evicted just before it is requested again. The reason that this untimely eviction can happen is that cache misses do not have a uniform distribution within the address space but can be clustered. The blocks associated with these clusters stand a good chance of being evicted from the cache just when they are needed. Pomerene et al. [POME84, STON93] described a technique that anticipates premature eviction and creates a *shadow directory* that assists in improving the replacement policy of LRU.

With the *first in first out* (FIFO) method, the block that has been in the cache the longest is assumed to be the least likely to be needed again and can be evicted. The FIFO method requires a queue for each sector or set. As a block is referenced, its name is placed on the top of the queue. When a miss occurs, the block name on the bottom of the queue is selected for eviction. Large caches require a large number of queues, which can result in a slower cache.

The LRU and FIFO policies become quite complicated to implement for set associative caches with multiple block sectors. Simulations and performance measurements of real systems indicate that the simple random policy is usually adequate for most cases. On the other hand, processors such as the MC88100 and i486 have caches with modified LRU policies that are discussed in Sections 2.4 and 2.6.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000092

### 2.2.7 *Block Size Considerations*

The design issue of block size was mentioned in previous sections. Two consequences of block size selection are important: (1) block size effect on $P_{\text{miss}}$ and (2) block size effect on transport time. These two effects combine to make block size a significant determinant of small cache performance and were studied extensively by Smith [SMIT87].

The effect of block size on $P_{\text{miss}}$ is considered first. For increasing block size, $P_{\text{miss}}$ will decrease because of spatial locality. In effect, prefetching larger blocks of information decreases future misses. On the other hand, as the size of the block becomes large with respect to the size of the cache, $P_{\text{miss}}$ will begin to increase. This increase in $P_{\text{miss}}$ occurs because the capacity of the cache is divided into a smaller number of uniquely AUs. This leads to dominance of capacity misses as a result of information loaded into the cache that is never used and is given the name *memory pollution* [SMIT78].

Smith [SMIT87] gave data on $P_{\text{miss}}$ as a function of block and cache sizes for unified, instruction and data caches; an excerpt of this data is shown in Table 2.10. The excerpted Smith data shows that $P_{\text{miss}}$ decreases monotonically with an increase in block size. However, for very small caches (the data is not shown in this table) there is a block size that gives a minimum $P_{\text{miss}}$. Since large caches dominate today's designs, small caches are of little practical interest.

The second consideration of block size selection is the impact on miss transport time (discussed in Section 2.2.1) as well as the resulting impact on the weighted average cache access time. Recall that for the sequential-forward cache

$$t_{\text{ea}} = 1 + P_{\text{miss}}T \text{ and } T = a + b(\tfrac{A}{Wd} - 1), t_{ea} = 1 + P_{miss}(a + b(\tfrac{A}{Wd} - 1)).$$

| Cache Size | Block Size (bytes) | | | | | |
|---|---|---|---|---|---|---|
| Unified cache | 4 | 8 | 16 | 32 | 64 | 128 |
| 2048 bytes | 0.405 | 0.258 | 0.170 | 0.124 | 0.098 | 0.093 |
| 8192 bytes | 0.232 | 0.135 | 0.080 | 0.050 | 0.033 | 0.025 |
| 32768 bytes | 0.124 | 0.070 | 0.040 | 0.024 | 0.014 | 0.009 |
| Instruction cache | | | | | | |
| 2048 bytes | 0.391 | 0.234 | 0.150 | 0.098 | 0.068 | 0.057 |
| 8192 bytes | 0.172 | 0.100 | 0.060 | 0.037 | 0.230 | 0.016 |
| 32768 bytes | 0.091 | 0.052 | 0.030 | 0.017 | 0.010 | 0.007 |
| Data cache | | | | | | |
| 2048 bytes | 0.256 | 0.169 | 0.120 | 0.094 | 0.083 | 0.089 |
| 8192 bytes | 0.214 | 0.129 | 0.080 | 0.053 | 0.039 | 0.032 |
| 32768 bytes | 0.108 | 0.065 | 0.040 | 0.025 | 0.017 | 0.012 |

**TABLE 2.10** $P_{\text{miss}}$ as a function of cache size and block size.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000093

| Cache Type-Size | Block Size (bytes) | | | | | |
|---|---|---|---|---|---|---|
| Unified cache | 4 | 8 | 16 | 32 | 64 | 128 |
| 2048 bytes | 160 | 110 | 100 | 105 | 112 | 200 |
| 8192 bytes | 75 | 50 | 35 | 30 | 30 | 32 |
| 32768 bytes | 50 | 32 | 24 | 20 | 18 | 21 |

**TABLE 2.11** Nanoseconds/memory reference.

Smith [SMIT87] provided results of simulations that show the effect on $t_{ea}$ as the cache and block sizes are varied and for various values of $a$ and $b$. Table 2.11 shows an excerpt of results for a unified cache with $a = 360$ ns and $b = 15$ ns.

These results clearly show, due to the interaction between $P_{miss}$ and transport time, that there is a clear minimum in the time for a memory reference as the block size is varied. For the small cache of 2 Kbytes, the minimum occurs with a block size of 16 bytes. For the larger 32-Kbyte cache, the minimum moves out to 64 bytes. However, for the larger caches, the minimum is very shallow, which indicates considerable latitude in block size selection. This data suggests that larger block sizes will be more common with the larger cache sizes of today. For example, the RS/6000 64-Kbyte data cache has a 128-byte block.

Hill and Smith [HILL84] reported on the effect of sector and block sizes on the bus traffic ratio. They pointed out that an instruction miss can fetch more instructions than will be used by the processor. Thus, it is possible to have a bus traffic ratio that is greater than one. Caches with a one-word block ($A = 1$) and an AU equal to the instruction word length will always have a bus traffic ratio of one or less for instruction caches.

As previously noted, the bus traffic ratio may be a major design concern for shared memory multiprocessors that have a multiplicity of caches on one bus to the main memory. As the bus traffic is proportional to the product of $P_{miss}$ and the transport time, the bus traffic ratio can increase with block size. The reason for this is that, as the block size increases, the transport time may increase faster than $P_{miss}$ decreases due to the larger block size. For this and other reasons, processors intended for multiprocessor use may have sets with one sector of one block of one word [STEN90]. The penalty of an increased $P_{miss}$ is viewed as acceptable so as not to saturate the bus.

### 2.2.8 *Bus Design Considerations*

In previous discussions, there have been a number of references to the bus that connects the cache to the main memory. A complete discussion

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000094

of buses is not possible in this book; however, a few comments are still needed. See [LEVY78, BORR81, VAND89, WARD90] for more details. An excellent collection of papers on various buses is found in [DIGI90]. The engineering aspects of bus design are carefully described in [DELC86].

Computers of the 1950s and 1960s had their major functional modules interconnected by ad hoc connection paths. The designers of the PDP-5 [BELL78] introduced the concept of the bus, which provided for modular interconnection of the functional modules by a standard interface known as a *BUS*.

There are two basic low-level control methods with buses: synchronous and asynchronous. Examples of both types can be found in systems and standards. For example, the *Futurebus* (IEEE896) is asynchronous and the *NuBus* (IEEE1196) is synchronous. Another low-level architectural feature is whether or not the addresses are multiplexed with the data blocks on the bus. Because the receiving device must decode addresses before the data can be accepted or generated, it is natural that address/data multiplexing be used. However, multiplexing adds timing and control complexity to the bus to accommodate saving conductor paths. Examples of these two methods are found in the *Futurebus*, which is multiplexed, and the *VMEBus*, which is not multiplexed.

Regardless of the control method, there are two performance parameters of interest in relation to cache–main memory interconnections. These are the latency and transfer rate, in other words, the components of the transport time model. Unlike the multiplicity of ad hoc buses used on the early computers, a single interconnection bus must be time shared among the various requesters and bus designers strive to provide the smallest latency and the highest bandwidth possible between the sources and sinks. A bus transfer consists of the following steps:

1. Arbitration for bus mastership.
2. Transmission of the address.
3. Receipt of the address, decoding, and acknowledgment.
4. Transfer of data.

The first three steps constitute the latency of the transaction while step 4 gives the transfer rate. The total time of the transaction is the sum of the latency and the data transfer time. Because the latency is usually larger than the transfer time of a single AU, block transfers are employed if at all possible. The bandwidth of a bus is derived from the transport time model of Figure 2.13.

$$\text{Bandwidth} = \frac{A}{\text{latency} + \text{transfer time}}.$$

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000095

It follows from the above that the maximum bandwidth of a bus has an upper limit of $A$/transfer time that is achieved only when the block becomes very large and the latency is totally prorated. However, as was discussed earlier, a typical block size is only 4 to 64 AUs. Thus, low latency is a prime requirement for a good bus.

A number of manufacturer-supported buses have been developed over the years. A brief, and incomplete, list includes:

| Manufacturer | Bus Names |
|---|---|
| DEC | *Unibus* |
| | *VAX SBI* |
| IBM | AT Bus |
| | Micro Channel |
| Intel | Multibus |
| Motorola | VMEBus, IEEE P1014/d1.2 |
| Texas Instruments, MIT | NuBus |
| Several | Futurebus, IEEE |

As processors and memories have become faster, bus technology has not kept pace. System designers are therefore reverting back to the techniques of the 1950s and 1960s and employing special dedicated buses that do not need the generality of the buses discussed above.

An example of a dedicated bus is found in the Compaq *Deskpro* 386/20 [VANN88]. An IBM AT compatible bus interconnects the main memory to the DMAs. A special 32-bit bus connects the cache to the main memory. With a dedicated bus, the design is specific to the task of transferring blocks between the main memory and cache. The number of loads on the bus is fixed as are the physical distances involved. Thus, the latency can be minimized and the bandwidth improved. A dedicated graphics bus is common today with high-performance personal computers.

Borrill [BORR81] has classified the various levels and use of buses. By his taxonomy, a dedicated bus is known as a component-level bus and named *Level* 0. The bus connecting the main memory to DMAs is named *Level* 1 and is more commonly known as the *backplane bus*.

The bus between a cache and the main memory is usually optimized for fetching a block. However, a nontrivial fraction of bus transactions are writes of a single AU that result from the use of a write through policy. Recall from Table 1.1 that reads represent, on average, 93% of memory references while writes represent 7%. If a unified cache has a $P_{\text{miss}}$ of 0.05 for reads and all writes are sent by the bus to the memory (as with a write through policy), then writes constitute approximately

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000096

60% of the bus transactions. With the i486, for example, writes on the bus constitute 70% of the bus transactions [CRAW90].

The IEEE Futurebus has been adopted by a number of manufacturers for new processors such as the Intel Pentium and the PowerPC 601. This bus, in conjunction with the MESI cache coherency protocol [GALL91], supports multiprocessors and is discussed in Section 4.2.5.

Another newcomer in the field of buses is the RAMBUS [WILS92]. The bus structure is integrated into the DRAM and can provide rapid block transfer. The DRAMS operate in a mode similar to the nibble mode described in Section 5.1.6. Advertised performance values for byte transfers are $a = 48$ ns, $b = 2$ ns. The proponents of these devices suggest that by sufficiently reducing the transport time between the main memory and the cache, the need for second-level caches is eliminated.

Bus latency and bandwidth are being recognized as limiting factors in the performance of processors for some workloads. This limit is discussed in Chapter 11 with regard to the CRAY-1 and CRAY X-MP memory buses. The commonly used SPEC benchmarks, except for one case, are processor bound and do not hit the bus limit to performance. However, as contemporary microprocessors tend toward the execution of programs with poor locality, the bus performance limit will become critical.

## 2.3 Instruction Buffering Schemes

Because the major source of memory references is instruction reads, computer designers first applied hardware assistance to instruction latency reduction and increased instruction bandwidth. This section discusses instruction *buffering schemes* that are represented by three very distinct techniques: *instruction buffers*, *instruction queues*, and *instruction caches* [KURI91]. These three buffering schemes are designed specifically to support instruction reads, and their organization and policies are based on the particular characteristics of instruction addressing patterns.

By employing an instruction buffering scheme, two paths to the main memory are provided: the instruction path and the data path. The effect can be a substantial improvement in performance for processors that require high-memory bandwidth. Therefore, if, for any reason, only one buffering scheme can be implemented, it should generally be as an instruction buffer, which can provide another path to the main memory.

Instructions are read, except for branches, from adjacent memory locations giving a high degree of spatial locality. Also, scientific programs are characterized by an extensive use of loops that provide a high degree of temporal locality. In the sections to follow, buffers, queues, and caches

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000097

that support one or both of these locality characteristics will be described. The three instruction buffering schemes to be discussed are:

1. *instruction buffers* that catch and hold previously used instructions for reuse, providing look behind;
2. *instruction queues* for prefetching instructions, providing look ahead;
3. *instruction caches*, providing both look ahead and look behind.

Instruction buffer schemes are unique in that only a fetch policy is required; the processor does not write into the instruction buffer, queue, or cache. Thus, instruction buffering schemes do not have the complications of supporting writes. However, for systems executing self-modifying code, a unified cache is usually employed in conjunction with branch instructions that flush the instruction prefetch in the hierarchy so that modified instructions can be written directly into the cache without a coherency problem. The Intel Pentium caches, to be described later, are exceptions to this generalization.

Another form of an instruction buffer scheme is known as a *branch target cache* (BTC), sometimes known as a *target instruction buffer*, that stores the stream of an instruction path that may be taken. BTCs are discussed in Chapter 7.

### 2.3.1 *Instruction Buffers*

The CDC 6600 is a scientific supercomputer that uses instruction buffer ing. This computer, designed by Seymour Cray in the early 1960s, has an instruction buffer that supports temporal locality [THOR70]. The use of a buffer that can hold a loop eliminates further references to the main memory for instructions. The loop is loaded into the buffer on the first instance of fetching an instruction in the loop; thus, this type of instruction buffer is referred to as a *loop catcher*.

The buffer is a push-down-stack-like structure, called an *instruction stack* by CDC, to save instructions for reuse. New instructions are fetched, placed in execution, and pushed on the top of the stack. When the program counter reads the next instruction, the address is compared with the addresses of instructions available in the buffer by an associative search. If the instruction is in the buffer, it is read from the stack; if it is not in the stack, it is fetched from the main memory. When the buffer is full, additional fetches from the main memory will cause the buffer to overflow and, hence, some of the instructions to be lost. Note that an associative cache with a FIFO replacement policy is functionally equivalent to the CDC 6600 instruction buffer.

The instruction buffer consists of eight 60-bit registers that are able to hold sixteen 30-bit instructions or thirty-two 15-bit instructions or

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000098

combinations of the two. Thus, a large fraction of loops found in scientific programs can be made resident in the stack, thus eliminating subsequent fetches from the main memory. The access time to the stack is one clock (100 ns) while a fetch from the main memory requires a minimum of 4 clock cycles; in some cases the latency is larger because of the time required to access the interleaved core memory.

The Cray-1 and subsequent Cray processors employ similar instruction buffers. The major difference is found in the size of the buffer, which has been increased to 256 16-bit instructions. The buffer is interleaved four ways so that concurrent fetches into the buffer from the main memory and instruction reads by the processor can be sustained. The interleaving matches the interleaving of the main memory, which results in a fetch rate of four 16-bit instructions per clock.

### 2.3.2 *Instruction Queues*

An instruction queue solves many of the problems associated with supplying an instruction stream without delays in two ways.

1. An instruction prefetch queue can hide the memory system latency by prefetching instructions. The latency can be either the latency of an interleaved memory or the delay latency caused by a cache miss. If enough instructions can be queued up, the delay of a cache miss is hidden as the processor can continue to fetch instructions from the queue. Two contemporary examples of this use of instruction queues are found in the Intel Pentium and the PowerPC 601. In the case of the Pentium, there are two queues—one stores the in-line instruction stream and the other stores the target instruction stream, as described in Chapter 7.

2. Some instruction queues provide read latencies of one clock for many variable-length instructions or multiple instructions when the queue is not empty. A smooth flow of variable-length instructions can be supplied to the decoder, making the instruction fetch into the decoder a single-clock event for most instructions. Otherwise, instruction fetches are multiple-clock operations without the prefetch queue. For RISC processors with fixed-length instructions, this type of queue and instruction prefetching is not required.

Instruction queues work because of spatial locality of instruction streams while temporal locality is usually not supported. Instruction queues are found in two topologies, as shown in Figure 2.16. One topology uses an instruction queue between the processor and an interleaved memory, while the other topology has a cache between the queue and the interleaved memory. As will be discussed later, the management policies and performance models of these two topologies are different.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000099

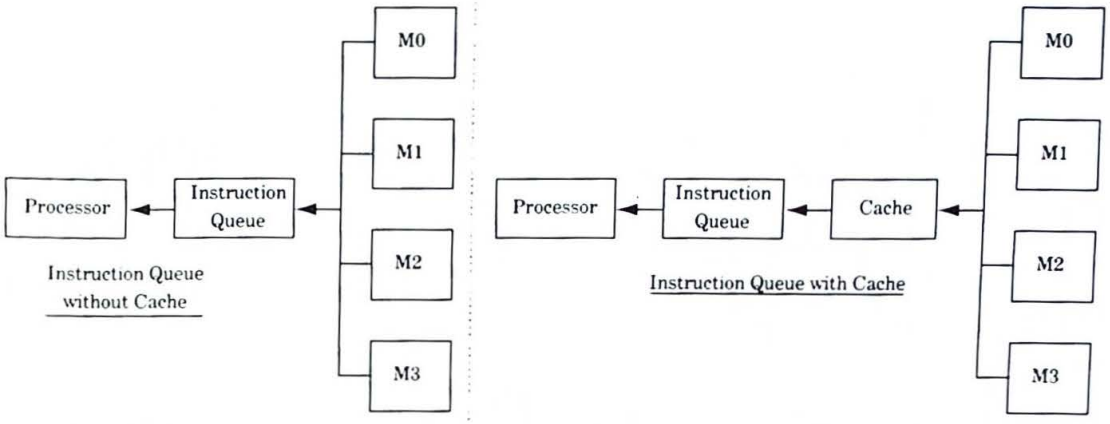



**FIGURE 2.16** Instruction queue topologies.

An instruction queue that smoothes the flow of variable-length instructions has interesting properties. Instructions fill the queue in sequential order at the top. At the bottom of the queue, there is logic that permits a variable number of bytes, or instructions, to be extracted broadside from the queue. Up to a design limit, four bytes for example instructions with one, two, three, or four bytes can be extracted in one clock and passed to the processor. The span of the extraction logic is controlled by the instruction fetch stage and may require a partial decode of an instruction byte to determine the number of subsequent bytes to extract.

The IBM 7030 Stretch is the first known use of an instruction queue. This computer was developed for The Lawrence Livermore Laboratory starting in 1956 [BLOC59]. This processor has a 64-bit bus to an interleaved memory and instruction lengths of 32 and 64 bits. The queue holds four 32-bit instructions. Sixty 4-bits (two 32-bit instructions) are fetched in one bus cycle into the queue. When the first instruction is read from the queue by the processor, a reference is made to the main memory to fetch another 64 bits. The queue supplies one instruction, single or double length, in one cycle. In the absence of branches, the instruction processing rate of the processor is exceeded by the fetch bandwidth and there are no delays for instruction fetches.

The IBM S/360/91 extended the use of an instruction queue while continuing the ability to smooth variable-length instruction issue to the execution units. The instruction queue is 8 deep with each level having words of 64 bits, holding approximately 16 instructions [ANDE67]. The instruction queue can be placed in "loop mode," which halts prefetching and locks in loops to provide "look behind" supporting temporal locality.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000100

The main memory of this processor consists of interleaved memory modules, which will be discussed in Chapter 4.

The VAX 11/780 has an instruction prefetch queue, which is named an *instruction prefetch buffer* [LEON87]. The maximum length of a VAX instruction is 50 bytes; however, the weighted average VAX instruction length is 3.8 bytes; 96% are 3 bytes or less and 98% are 8 bytes or less [FITE90]. The instruction queue holds 8 bytes; thus, the queue can hold approximately two instructions when it is full, and instructions are removed from the queue 1 byte at a time. When there is space for a 4-byte block load from the cache, the prefetch is triggered and the block is loaded broadside into the queue. This flow of instructions continues until a branch is encountered.

The VAX 9000 expanded the instruction queue of the VAX 11/780 to 25 bytes [MURR90], a queue that holds approximately six 3-byte instructions. More importantly, the decoder can read up to 4 bytes (the op-code and three specifiers) in one clock from the queue by conditionally decoding the next instruction.

An interesting case study on the use of instruction queues is found in the ix86 family. The first member of this family, the i8086, employed a queue (named a buffer by Intel) of 6 bytes. This depth was selected by simulation and by considerations of the chip area required for the queue [MCKI79]. The i186 and i286 retained the 6-byte queue but added a second queue that holds three decoded instructions, and the fetch path to the main memory was extended to 16 bits.

The size of the ix86 processor queues has increased with the evolution of this processor family. The i386 has a similar instruction prefetch mechanism. However, the prefetch is 4 bytes rather than 2, and the queue is 8 bytes rather than 6. Simulation results [GROC89] show that the instruction queue of the i386 had $P_{\text{miss}} = 0.05$ while the i486, described later in this chapter, has $P_{\text{miss}} = 0.03$. The meaning of $P_{\text{miss}}$ in a queue is that the requested AU is not in the queue and that a reference to a lower level in the memory system is required. In other words, the prefetch queue has not been able to keep up with the flow of instructions into the processor.

The i486 employs instruction prefetching into a 32-byte queue from a unified on-chip cache [CRAW90]. This queue is required because instruction can vary in length between 1 and 15 bytes, with an average length of 3 bytes. The first stage of the pipeline prefetches instructions into one of two 16-byte queues. Three bytes are read in parallel from the queue into the first decoder stage of the pipeline; the prefetch stage and its queues operate independently from the balance of the pipeline. With this system, a large number of the simple instructions can be decoded at a rate of one per clock even though they are of variable length.

The average input and output data rates from the queues are equal,

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000101

but the queue is required to smooth out the flow due to the variable-length instructions if one instruction per clock is to be fetched. To achieve this smoothing effect, the i486 path between the on-chip cache and the instruction queue provides approximately 9 times the required instruction bandwidth. This path transfers 16 bytes in one clock into the queue. The i486 CPI is approximately 1.75, and each instruction is approximately 3 bytes. Thus, instruction fetching from the queue consumes 3/1.75 = 1.7 bytes per clock, giving a bandwidth multiplier of 16/1.7 = 9.4. Because the input is driven by a prefetch type request, in the steady state, the average input bandwidth to the queue is equal to the consuming bandwidth of the instruction processor.

The PowerPC 601 [MOTO93] instruction queue holds up to eight instructions prefetched from the in-line instruction stream. The Power PC 601 is a superscalar processor, which will be discussed in Chapter 10, and can issue up to three instructions in one clock. The queue control attempts to keep the queue full at all times by fetches from the on-chip unified cache. When there is a cache miss, the main memory access will occur only if the top five entries are empty. This strategy delays main memory references until a useful number of instructions, that is a block of 16 bytes, is needed, thus reducing the number of bus access cycles.

Simple models for two systems using interleaved memory without a cache are developed in the following paragraphs. The two systems are for double buffers and a single queue that has concurrent loads and stores.

## Double Buffer Model

A simple model for the size of a double buffer instruction queue that is serviced from an interleaved memory, without a cache, is developed here. Three parameters are of interest in an instruction queue:

1. the rate at which the lower level of the memory hierarchy can fill the queue,
2. the rate at which the processor consumes instructions,
3. the size of the queue.

If the queue is too small or the fill rate is too low, the inherent processing rate of the processor cannot be sustained. A simple match of the steady-state bandwidths may indicate that a queue is not needed. However, the real problem is not the steady-state rate at which the queue is filled but, for an interleaved memory, the transport time required to fill the queue or, for a cache system, the time to fill the queue if there is a cache miss.

For the model, the queue is assumed to be a double buffer, as with

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000102

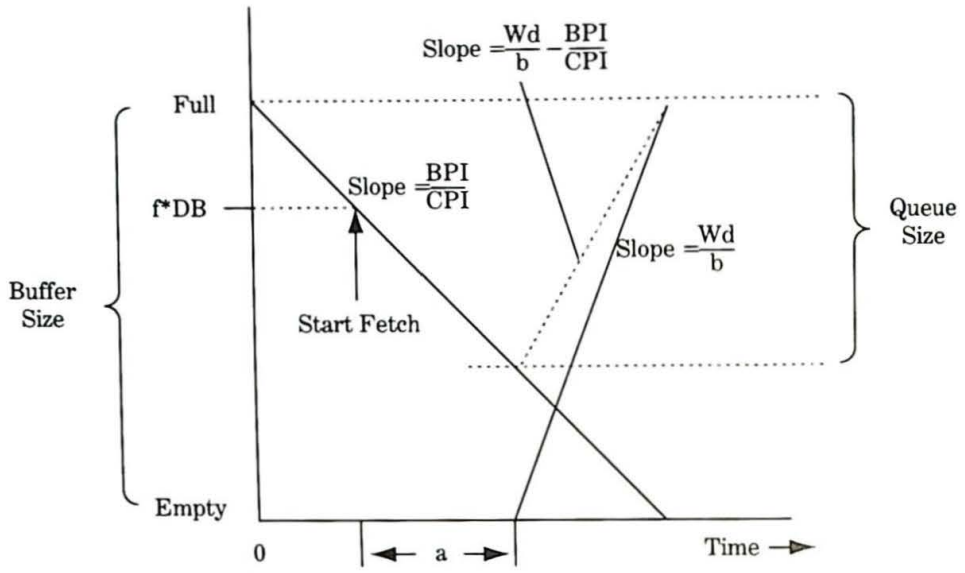



**FIGURE 2.17** Queue fetch model.

the Pentium. One buffer is supplying instructions while the other buffer is being filled from the memory. As shown in Figure 2.17, one buffer is assumed to be full at $t_0$ ($DB = 1$) and the processor is extracting instructions at the rate:

$$\text{extraction rate} = \frac{\text{bytes per instruction}}{\text{clocks per instruction}} = \frac{\text{BPI}}{\text{CPI}}.$$

When the queue has been depleted to a point $f \times DB$ ($f$ is a design parameter), an access to the interleaved memory begins. The second buffer must be filled from the interleaved memory before the first buffer is emptied by the processor. After a delay of $a$, the second buffer fills at the rate $Wd/b$; these variables are taken from the transport time model.

From the geometry of Figure 2.17, the following time equality (the time to fill the second buffer must equal the time to empty the first buffer after the fill starts) must hold:

$$a + \frac{b \times DB}{Wd} = \frac{f \times DB \times CPI}{BPI}.$$

And, solving for $DB$,

$$DB = a \Big/ \left(\frac{f \times CPI}{BPI} - \frac{b}{Wd}\right).$$

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000103

For an example of estimating the size of the buffers, assume the following:

$$a = 8, \quad b = 1, \quad Wd = 8, \quad CPI = 1.25, \quad BPI = 3, \quad f = 0.85.$$

Using these system parameters, each of the buffers should be 34.9 bytes. This result agrees with the actual design of the Pentium, which has a double buffer of 32 bytes each. If the first buffer is emptied before the second buffer is full, the processor must stall, an effect that has been called a queue miss.

## Queue Model

While a double buffer is easier to control, it requires more bits than is required for a true queue as a queue will fill as it empties. As shown in Figure 2.17, the queue issues a request after the processor has depleted the queue to $f$ and starts to refill after the latency $a$ at the net rate shown in the dotted line (the difference between the fill and extraction rates of the interleaved memory). The queue must only be large enough to be empty when the fill begins, and the fill rate must be greater than the empty rate. A model for the size of a queue can be found from the geometry of Figure 2.17 as

$$Q = a \Big/ \left[ \frac{f \times CPI}{BPI} - 1 \Big/ \left( \frac{Wd}{b} - \frac{BPI}{CPI} \right) \right].$$

For an example of estimating the size of a queue, the approxi. parameters of the IBM S/360/195 are

$$a = 8, \quad b = 1, \quad Wd = 8, \quad CPI = 1.55, \quad BPI = 4, \quad f = 0.9.$$

Using these parameters, the model shows a queue size of 61.9 bytes compared to the actual IBM S/360/195 queue that has 64 bytes. Thus, the model gives a reasonable estimate for the size of an instruction queue. This model can be extended to assess the sensitivity of the processor performance as the queue size is varied (recall the discussion about the simulations performed on the i8086).

Table 2.12 gives a summary of instruction queue designs. The queue depth, queue width, memory data path, and whether or not instructions are decoded in the queue are noted. A desirable property is to decode and identify branch instructions in the queue so that branch delays can be reduced (a feature discussed in Chapter 7).

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000104

| Processor | Queue Depth | Queue Width | Cache/Mem Path Width (*Wd*) | Decoded Instruction Queue |
|---|---|---|---|---|
| IBM 7030 (Stretch) | 2 | 4 bytes | 4 bytes | No |
| IBM S/360/91 | 8 | 8 bytes | 8 bytes | Yes |
| VAX 11/780 | 8 | 1 byte | 1–4 bytes | No |
| i086 | 3 | 2 bytes | 2 bytes | No |
| i386 | 8 | 2 bytes | 4 bytes | Yes |
| i486 | 16 | 2 bytes | 4 bytes | Yes |
| Pentium | 1 | 64 bytes* | 32 bytes | No |
| PowerPC 601 | 8 | 4 bytes | 32 bytes | Yes |

*Double buffers, 32 bytes each.

**TABLE 2.12** Processor prefetch queues.

### 2.3.3 *Instruction Caches*

Since the early 1980s, instruction caches have received accelerated research interest, because VLSI technology provides sufficient chip area for the inclusion of an instruction cache on the chip with the processor. The organization of an instruction cache must now comprehend not only $P_{miss}$ but also the efficiency of layout in order to minimize the chip area devoted to the instruction cache. Also, timing considerations are different with on-chip caches due to the elimination of delays between the processor chip(s) and the cache chip(s).

Instruction caches for VLSI implementation were first discussed by [KATE84] while a separate cache chip was proposed for the RISC II processor [PATT83]. Hill and Smith [HILL84] extended the evaluation by considering on-chip instruction caches. Small instruction caches have given way to larger, more general split caches due to the ability to include relatively large caches on a processor chip.

The RISC research group at Berkeley [PATT83] considered a number of alternative cache designs and issues such as fault tolerance, remote program counters, and Huffman coding of instructions. A cache chip was proposed that is organized as (64, 1, 1, 4). These chips could be configured as either direct or set associative organizations. Simulation results indicate that the set associative configuration provides a slightly better $P_{miss}$. Each block has a fault tolerant bit that could be used to disable a block precluding its use if an AU is faulty.

Hill and Smith [HILL84] reported on their research into instruction caches for the PDP-11, Z8000, VAX-11, and IBM S/370 architectures. The term *minimum cache* is used to identify a cross between an instruction buffer and a cache. A minimum cache is a buffer that can, unlike pure buffers, recognize branch addresses. The authors use a descriptor

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000105

that is different than the one introduced in this chapter to describe their caches. They use the 2-tuple ($bx$, $sx$) where $bx$ is the number of bytes in the sector (block) and $sx$ is the number of bytes in the block.

Agarwal et al. [AGAR87] investigated alternative organizations for the instruction cache of the MIPS-X processor. In their paper, three cache organizations are analyzed:

1. a set associative cache that uses approximately half of the available space for tag bits; this organization is named a "c cache";
2. an associative cache that is named a "buffer";
3. a set associative cache that reduces the area used for tag bits from that of 1; this organization is named a "hybrid buffer."

Each of these cache variants is evaluated; the block size is varied and $P_{\text{miss}}$ data obtained. Trial layouts are also made to assess the chip area required for implementation.

Another instruction cache design is described by Hill [HILL87] for the Berkeley SPUR single-chip microprocessor. The issues and tradeoffs of instruction caches and target instruction buffers (TIB) are examined. TIBs are discussed in Chapter 6; the discussion in this section will deal with the instruction cache only. Hill's research is noteworthy in that circuit implementation issues are considered as well as the more conventional $P_{\text{miss}}$ data for the various configurations. As the clock periods decrease, small design details can, if improperly implemented, reduce performance more than changes in cache organization.

The instruction cache for SPUR would, of necessity, be relatively small and have a relatively high $P_{\text{miss}}$. Hill believed that cold starts and multiprogramming effects would be small relative to the naturally high $P_{\text{miss}}$. Thus, multiprograms with cold starts are not directly simulated. However, as a surrogate for real programs, multiprogramming effects are modeled by evicting all of the blocks in the cache after every 30,000 references. Three mutually exclusive policy modes are implemented for the SPUR instruction cache: (1) disabled, (2) enabled without prefetching, and (3) enabled with prefetching. Prefetching in this case refers to fetching the next AU from a miss. The policy modes are program-selectable in order to permit a closer match of the program to the cache policy.

### 2.3.4 *Published Instruction Cache Hit Data*

In this section we discuss published $P_{\text{hit}}$ data for instruction caches. Smith and Goodman [SMIT83] published early results directed to the design of instruction caches based upon a Markov model for the behavior

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000106

of instruction references. Smith and Goodman considered three replacement algorithms—Random, LRU, and FIFO—for direct-mapped and set associative caches. The Markov models gave qualitative indicators concerning the replacement method and the cache organization. Their model results are also compared to experimental results of three benchmarks written in C, running on a VAX -11 under UNIX.

The conclusions from their experimental results are that the replacement method affects the value of $P_{hit}$ in the second or third decimal place. For this reason, a designer may opt for the simpler design. The second conclusion is that for a given block size, $A = 16$ bytes in this case; $P_{miss}$ for a cache of a given size is not influenced by the number of sets and the number of sectors.

Table 2 of the Smith and Goodman study provides the results that are reproduced in Table 2.13 for caches organized as ($S$, $SE$, 1, 16). The figure displays the $P_{hit}$ for the evaluated combinations of $S$ and $SE$. 45-deg lines from lower left to upper right are lines of equal cache size; these caches also have approximately equal $P_{hit}$. For example, the design $S = 1$ and $SE = 512$ has the same capacity (8 Kbytes) as the cache with $S = 512$ and $SE = 1$, given that the block sizes are equal. Furthermore, these caches have the same $P_{hit}$ of 0.998. Likewise, other cache designs that are on a diagonal have approximately the same value of $P_{hit}$.

A conclusion that can be drawn from this data is that the effectiveness of an instruction cache, as measured by $P_{hit}$, is primarily based upon its size—all other factors are of second-order importance. A further

| | Number of Sets | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| No. of Sec./Set | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 |
| 1 | | | .699 | .727 | .779 | .829 | .905 | .959 | .990 | .998 |
| 2 | | .697 | .731 | .777 | .838 | .915 | .980 | .996 | .998 | |
| 4 | .700 | .726 | .768 | .839 | .921 | .989 | .998 | .998 | | |
| 8 | .727 | | | | | | | | | |
| 16 | .768 | | | | | | | | | |
| 32 | .816 | | | | | | | | | |
| 64 | .901 | | | | | | | | | |
| 128 | .944 | | | | | | | | | |
| 256 | .998 | | | | | | | | | |
| 512 | .998 | | | | | | | | | |

Block size is 16 bytes.

**TABLE 2.13** Smith instruction cache $P_{hit}$ data.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000107

observation, for the benchmarks used, is that there is a practical maximum size of an instruction cache. That is, increases in $P_{hit}$ will be relatively small as the size of the cache is increased. When all of a program can be stored in a cache, there is little reason to increase the cache size further. An increase in instruction cache size may even be counterproductive as its cycle time may increase as well.

However, this conclusion can be seriously in error because the benchmarks are quite simple and the study only investigated the effect of capacity and compulsory misses, not the effect of context switching. These effects can significantly change the view of instruction cache size on $P_{hit}$ [THIE87].

Recent research into the effect of instruction cache size and organization ($S$, $SE$, 1, 16) on $P_{hit}$ rates used large caches and the SPEC benchmarks [GEE91, GEE93]. These results are published as $P_{miss}$ but are shown, in excerpted form, as $P_{hit}$ in Table 2.14. The values of $P_{hit}$ shown are with a composite workload of floating point and integer programs. Notice that as the cache becomes larger, the influence of organization on $P_{hit}$ is as muted as with the smaller instruction caches shown in Table 2.13. For example, a cache with $S = 4K$ and direct access has $P_{hit} = 0.997$, while the same size cache organized as $SE = 512K$ and $S = 8$ has $P_{hit} = 0.9993$. The difference is probably insignificant for practical purposes.

It is interesting to consider the effect of doubling an instruction cache size using the data of Table 2.14. For example, if the initial organization has $S = 128$ and $SE = 2$, the $P_{hit}$ is 0.976. Doubling the size by doubling the number of sets or the degree of set associativity gives approximately the same $P_{hit}$ (0.983 and 0.984). The design choice then is based on the availability of set index bits in the address and implementation difficulty as the resulting $P_{hit}$ will be approximately the same.

| | Number of Sets | | | | | | |
|---|---|---|---|---|---|---|---|
| No. of Sec./Set | 64 | 128 | 256 | 512 | 1K | 2K | 4K |
| 1 | .949 | .961 | .969 | .98 | .988 | .993 | .997 |
| 2 | .966 | .976 | .983 | .99 | .992 | .997 | .9998 |
| 4 | .977 | .984 | .991 | .992 | .999 | .9998 | .9999 |
| 8 | .984 | .991 | .992 | .9993 | .9998 | 1.0 | 1.0 |

Block size is 16 bytes.

**TABLE 2.14** Instruction cache $P_{hit}$, SPEC benchmarks, overall.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000108

### 2.3.5 *Contemporary Instruction Caches*

There are few examples of contemporary instruction caches that are not used in conjunction with a data cache. The reason for this is that the rapid increase in circuit density has permitted the inclusion of a split cache on the chip; small on-chip instruction caches were quickly made obsolete by split on-chip caches. For some contemporary processors, the instruction and data caches are identical in organization so as to reduce the design task.

One example of a solo instruction cache is found in the Motorola MC68020. The MC68020 is designed with a small instruction cache so as to reduce the number of instruction fetches from the main memory [MACG84]. With an instruction cache, accesses of instructions and data can overlap when there is an instruction hit on the cache. The chip area was limited, and the area that was available could best be used as an instruction cache.

The MC68020 cache is a direct cache organized as (64, 1, 1, 2) [MOTO90b]. The instruction AU of the MC680x0 processor is 32 bits, 4 bytes, and each sector-block contains two of these AUs. The policy of the cache can be selected from a number of options under program control that is exercised by 4 bits in the Cache Control Register. Bit 0 enables/disables the cache. Bit 1 freezes the cache so that valid data cannot be evicted. Bit 2, named the Clear Entry bit, permits the location in the address to be invalidated. Finally, bit 3 clears the cache of all data and tags. The requirement for these programmable policy functions is discussed in Section 2.8.1.

An example of a unique instruction cache that works with a data cache is found in the IBM RS/6000. This instruction cache has special properties that are needed to support superscalar operation and is described in Chapter 11. I note that the Motorola MC88110 superscalar processor uses almost symmetrical instruction and data caches [DIEF92]. The difference being that the data cache has snoop tags that the instruction cache does not need. Snoopy controllers are discussed in Chapter 4.

## 2.4 Data Caches

Data caches are provided, usually in combination with an instruction buffer, queue, or cache, to reduce the latency of data fetches. Unlike instruction caches, data caches must support writes as well as reads. As shown in Table 1.1, approximately 29% of all memory references are data reads and approximately 7% of all references are data writes. However, writes to a data cache constitute approximately 25% of all data cache references while reads are 75% of all data cache references. Thus, improving reads is more important than improving writes. The design

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000109