

THE CACHE MEMORY BOOK
SECOND EDITION

THE CACHE MEMORY BOOK

SECOND EDITION

JIM HANDY



Academic Press

San Diego New York Boston
London Sydney Tokyo Toronto



TK7895
M4H35
1998

This book is printed on acid-free paper. ∞

Copyright © 1998, 1993 by Academic Press, Inc.

All rights reserved.

No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopy, recording, or any information storage and retrieval system, without permission in writing from the publisher.

ACADEMIC PRESS, INC.
1250 Sixth Avenue, San Diego, CA 92101-4311

United Kingdom Edition published by
ACADEMIC PRESS LIMITED
24-28 Oval Road, London NW1 7DX

Library of Congress Cataloging-in-Publication Data

Handy, Jim.

The cache memory book / Jim Handy.—2nd ed.

p. cm.

Includes index.

ISBN 0-12-322980-4 (alk. paper)

1. Cache memory. I. Title.

TK7895.M4H35 1998

004.5'3—dc21

97-35868
CIP

Printed in the United States of America

97 98 99 00 01 EB 9 8 7 6 5 4 3 2 1

Dedication to the First Edition

*This book is dedicated to
my wife, Mary Pope-Handy, the love of my life,
who suffered arthritis, asthma, pneumonia, and two kids
in diapers to give me time to write.*

Dedication to the Second Edition
I dedicate the second edition, again, to my wife, Mary,
and to my young son Brian, and to my even younger daughter
Clair, all of whom supported my efforts. No man could ask for a finer
and more loving family than ours.

CONTENTS

Introduction to the Second Edition	v
Introduction to the First Edition	vii
Acknowledgments	xi
Chapter 1 What Is a Cache Memory?	1
1.1 CPU speed vs. system speed	1
1.2 The cost of fast main memory	3
1.3 The concept of locality	5
1.3.1 Locality in space	6
1.3.2 Locality in time	7
1.4 Fooling the CPU	8
1.5 Cache data and cache-tag memories	14
1.6 Thrashing: Getting the most out of the cache	22
1.7 Caches and the memory hierarchy	27
1.8 Reducing bus traffic	29
1.9 Reducing power consumption	34
1.10 An example cache	35
Chapter 2 How Are Caches Designed?	39
2.1 The CPU-to-main-memory interface	39
2.1.1 Why main memory is too slow for modern CPUs	39

2.1.2	How the CPU handles system bus delays	41
2.1.3	The cache's interface to the CPU	42
2.2	Choosing cache policies	49
2.2.1	Logical vs. physical	50
2.2.2	Associativity	51
2.2.3	Unified vs. split caches	63
2.2.4	Write-through vs. copy-back	64
2.2.5	Line size	67
2.2.6	Write buffers and line buffers	77
2.2.7	Noncacheable spaces	86
2.2.8	Read-only spaces	87
2.2.9	Other status bits	88
2.2.10	Primary, secondary, and tertiary caches	89
2.3	Statistical premises	94
2.3.1	The value of choosing policies empirically	94
2.3.2	Using hunches to determine policies	97
2.4	Software problems and solutions	98
2.4.1	Trade-offs in software and hardware interaction	98
2.4.2	Maintaining compatibility with existing software	99
2.5	Real-world problems	100
2.5.1	Parasitic capacitance and bus loading	100
2.5.2	Critical timing paths	104
2.5.3	Bus turnaround	108
Chapter 3	Cache Memories and RISC Processors	111
3.1	The RISC concept	111
3.1.1	The CISC bottleneck	112
3.1.2	The RISC architect's goal	114
3.1.3	Interaction of software and hardware	115
3.1.4	Optimizing compilers	115
3.1.5	Architectural trade-offs	117
3.2	Feeding instructions to a RISC CPU	121
3.3	Problems unique to RISC caches	122
Chapter 4	Maintaining Coherency in Cached Systems	123
4.1	Single-processor systems	124
4.1.1	DMA activity and stale cache data	126
4.1.2	Problems unique to logical caches	135
4.2	Multiple-processor systems	138

4.2.1	Two caches with different data	139
4.2.2	Maintaining coherency in multiple logical caches	141
4.2.3	Write-through caches as a solution	142
4.2.4	Bus traffic problems	143
4.2.5	Copy-back caches and problems unique to copy-back caches	144
4.3	Existing copy-back coherency protocols	155
4.3.1	The MESI protocol	156
4.3.2	Futurebus+	159
4.3.3	The MOESI protocol	165
4.3.4	N+1	169
4.3.5	Berkeley	172
4.3.6	University of Illinois	177
4.3.7	Firefly	180
4.3.8	Dragon	183
4.3.9	Others	186
Chapter 5	Interesting Cache Tricks	187
5.1	Efficiently feeding a superscalar machine	188
5.2	Primary and secondary caches on the same chip	191
5.3	Controlling the cache's consistency with its write buffers	193
5.4	Achieving a true LRU by using a stack	194
5.5	A "semiassociative" cache	196
5.6	Controlling aliases by using eight comparators	197
5.7	Write buffering vs. multiprocessing	198
Glossary		201
Index		225

CA; Corollary, Inc., of Irvine, CA; and the VME International Trade Association (VITA) of Scottsdale, AZ.

As all authors of technical books say, all errors or omissions are solely my own. If you spot anything which you find erroneous, difficult, or lacking, please let me know through the publisher.

CHAPTER 1

WHAT IS A CACHE MEMORY?

1.1 CPU SPEED VS. SYSTEM SPEED

The problem is simple. Designers are constantly trying to get the most out of their designs in the most cost-effective means. As faster versions of a particular CPU become available, the designer will often try to improve the throughput of an existing design by simply increasing the CPU clock frequency.

After a certain point, the speed of the system's main memory (sometimes called a **backing store**) becomes the limiting factor in the throughput of the system. This is illustrated in Figure 1.1. The *X*-axis represents CPU clock frequency, and the *Y*-axis the overall throughput of the system. For this example, we will maintain that the system is compute-bound; that is, the performance of the system is constrained by the performance of the CPU, rather than by the speed of an input/output (I/O) device (i.e., a disk drive). CPU caches are most important in compute-bound systems, since they serve solely to increase the efficiency of CPU-related operations.

We will also assume that the performance of the system's main memory as seen by the CPU is the same across the range of CPU clock frequencies.

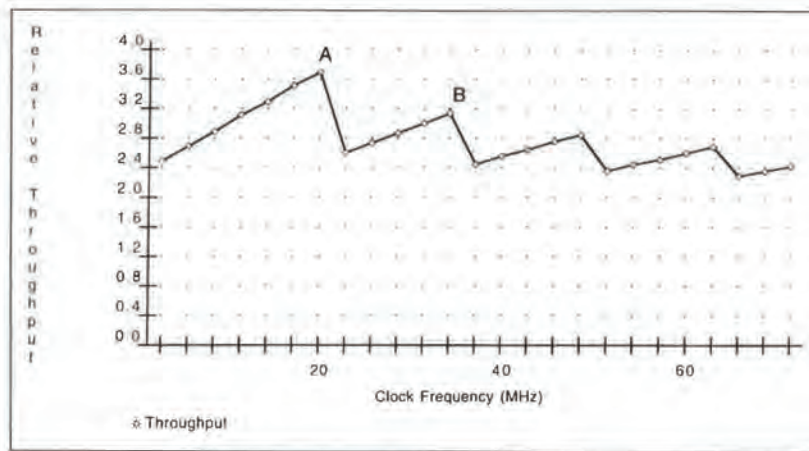


Figure 1.1 Throughput vs. clock frequency of a typical single-processor system.

In other words, if the main memory has an average access time of 60ns, this access time will be the same despite the CPU speed. (The 60ns figure is not a particularly fast speed even at the writing of this text, much less a few years from now. The author asks the reader to insert any appropriate number when the text is read, and anticipates that the problem will remain the same.) The value of the constant main memory access time assumption will be defended in Section 1.2 of this chapter.

At the early part of the chart, throughput increases linearly in proportion to the CPU clock speed, up to the peak at the point labeled A. This is intuitive. Suddenly, there is a drop to a lower level. Why is this? At clock frequencies up to about 20 MHz, the processor is free to run as fast as it can, without wait states from a 60ns main memory. Once the processor requires an access time of less than 60ns, a wait state must be inserted to account for the difference. A single wait state will suffice until the clock frequency requires an access time of less than 60ns, even with the single wait state. This is labeled point B. Once again, the throughput of the system suddenly drops, as it did with the addition of the first wait state. As the graph shows, increases in clock frequency merely take the throughput of the system through a narrow band, indicated by the sawtooth pattern in Figure 1.1, without any hope of improving performance beyond a certain limit set by the memory access time. Even worse is the trend of both the peaks and valleys of the sawtooth graph to decrease as the system clock frequency increases, a phenomenon I'm not sure I can fully explain.

Of course, this problem can be fixed through the use of fast main memory. The only question remaining is what speed of main memory must be used to support the fastest possible operation?

1.2 THE COST OF FAST MAIN MEMORY

Fast main memory is not inexpensive. As a general rule, the difference between the price of a fast memory chip and the next fastest speed grade can range up to 50% or 100%! All this to gain an extra 20% in speed!

If the world were a fair place, a 20% increase in speed on a dynamic RAM (which I'll call a DRAM for the remainder of this book) would support a 20% increase in speed in the CPU. Unfortunately, this is far from what really happens.

Most systems insert some sort of buffer or two between the CPU and the DRAM, and another buffer or two between the DRAM data output and the CPU data input pins (Figure 1.2). These buffers serve not only to increase the small current available from the CPU to drive the address inputs of the many DRAMs in the system, but also to isolate the CPU from the memory during those times when another device must have control of the main memory. Examples of such devices include DRAM refresh circuitry and DMA devices such as small computer system interface (SCSI) ports and disk I/O. The buffers have a fixed propagation delay, that is, the propagation delay of these devices does not decrease with advances in semiconductor processing anywhere near as drastically as does the access time of DRAMs and static RAMs (which we'll call SRAMs from now on), or as fast as micro-processor clock speeds increase.

Other static times are write cycle data and address set-up and hold times (with respect to write pulse edges), which are usually the same for any speed grade of DRAM or SRAM. A typical SRAM might have a write data set-up time of around 5ns, with a hold time of 0ns. The problem here is that, although manufacturers try to reduce these figures in the specifications of their faster parts, the improvements are far from proportional to any improvements in read access time. As a result, a larger and larger proportion of the overall CPU clock cycle time gets consumed by these static times. In a shrinking clock period, once these static set-up and hold times are removed, the time left for a memory access must shrink far faster than the clock period in order to satisfy the needs of the CPU.

Figure 1.3 shows the relationship between the clock frequency of an uncached processor system and the access time required of a memory subsystem in order to support zero-wait read cycles on that system. For this example, the system of Figure 1.2 was used, based on a 386 processor since this

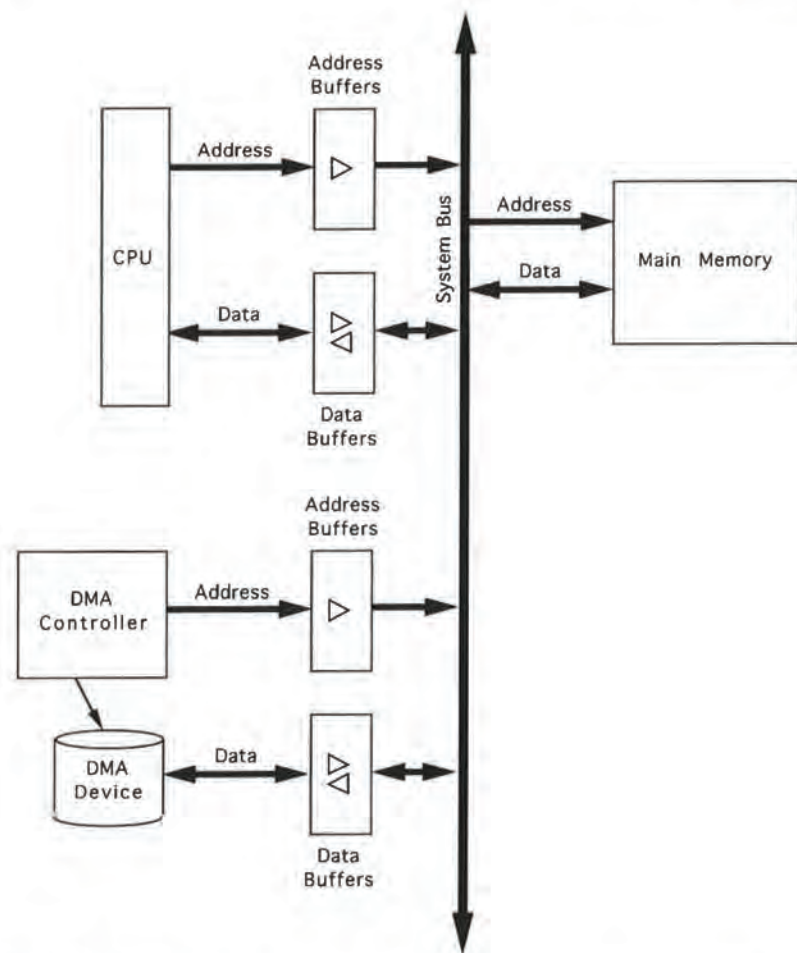


Figure 1.2. A typical processor system, with processor, main memory, a direct memory access (DMA) device, and the buffers required to allow the DMA and processor to gain alternate access to the main memory.

processor does not have an internal cache, with all buffers offering a 5ns propagation delay, and no other delays in the system such as those caused by pin capacitance or PC board loading (these will be spelled out in detail in Chapter 2, Section 2.5). To be fair, I've included both modes of operation for the 386, pipelined and nonpipelined. For either case, the trend

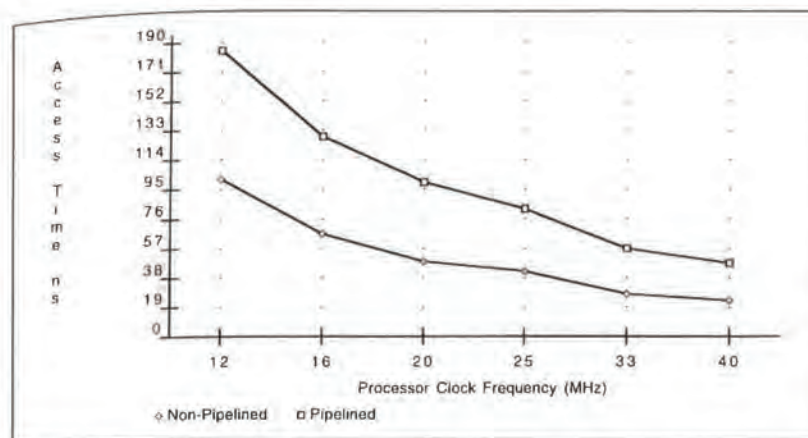


Figure 1.3. This graph shows the access time of a zero-wait-state main memory plotted as a function of the clock cycle time of the CPU for the system of Figure 1.2. The slope of the curve is very steep, showing how the speed of the main memory required in order to support increases in processor speed outpaces the speed increases of dynamic RAMs brought about by advancing both the CPU and the DRAM to the same process technology.

asymptotically approaches a 0ns access time, but the timing for pipelined operation reaches zero at a higher clock frequency.

1.3 THE CONCEPT OF LOCALITY

We are all very practical designers. When asked to design a system which performs to a certain target specification, we try to achieve the specification for the lowest reasonable cost. The arguments presented in the preceding two sections would seem to indicate that it would become prohibitively expensive to design a system which took advantage of the fastest CPU speeds available today.

This is where statistics come to the rescue. Back in the 1960s, researchers at IBM discovered that nearly all code was extremely repetitive in nature, a fact which can be used to the advantage of computer throughput. If anything repetitive can be stored in a small, very high-speed memory, then the effect of wait states can be limited only to the less repetitive portions of the program which can be stored in slower, less expensive memory.

What portion of a program is repetitive? This question is answered by the principle of **locality**, both spatial (locality in space) and temporal (locality in time).

1.3.1 Locality in Space

Locality in space, spatial locality, or locality of reference are terms for the phenomenon that most computer code executes out of a small area repetitively. This space is not necessarily in a single address range of main memory, but may be spread around quite significantly.

For the sake of illustration, an assembly code program is shown in Figure 1.4. This program is not written in any sort of rigorous manner, but should be adequate to show the general idea of locality in space. The program is a much simplified version of the kind of code which is found in all comput-

ADDRESS	LABEL	INSTRUCTION	COMMENT
0000 1000	START:	LOAD SP,#F000 0007	;Load stack pointer with a value ;which will assure a stack thrash.
0000 1004		LOAD R1,#0000 0063	;Set up loop counter to loop 99 times ;(63 hexadecimal).
0000 1008	LOOP:	JSR SUB1	;Jump to a subroutine with a ;thrashable address.
0000 100C		DJNZ R1,LOOP	;Repeat the above process 99 times ;before proceeding.
0000 1010		BLAH	;Continue this program
0000 1014		BLAH	
0000 1018		BLAH	
;*****			
;* *			
;* The following is the subroutine called by the main program shown above *			
;* *			
;*****			
0001 1008	SUB1:	LOAD R2,(R1+#1000)	;Get a character pointed to by ;R1+1000 hex
0001 100C		CMP R2,"A"	;Is the character an upper-case A?
0001 1010		JNZ SKIP	;No. Leave the character as-is
0001 1014		LOAD R2,"z"	;Yes. Change it to a lower-case z.
0001 1018		STORE (R1+#1000),R2	
0001 101C	SKIP:	RETURN	;Return to calling routine.

Figure 1.4. An example piece of assembly code to demonstrate locality.

1.3 THE CONCEPT OF LOCALITY

ers. First, a loop is used to look through a 99-character string stored in memory. Second, a subroutine contains a piece of code which might be useful to other parts of the main program. During the entry and exit of the subroutine, the stack is used to temporarily store and retrieve the calling routine's program counter. The function of this piece of code is to scan a string for an uppercase letter "A" and to change all such occurrences within that string into a lowercase "z."

The first perfectly obvious fact is that the code is written so that the same set of instructions, mostly those within the subroutine, is used repeatedly while the processor is running through the loop. The principle of spatial locality refers to the fact that the calling routine, the subroutine, and the stack space all exist in three very narrow areas starting at the three addresses 0000 0008, 0000 1000, and 0001 1008.

1.3.2 Locality in Time

The second principle of locality, **locality in time, temporal locality, or time of reference**, refers simply to the fact that the instructions in this example execute in close sequence with each other, rather than being spread through time. A processor is much more likely to need to access a memory location which it accessed ten cycles ago than one which it accessed ten thousand cycles ago (which is only 0.12 milliseconds on a 166 MHz Pentium). This appears awfully obvious, but without locality in time and in space, caches would not be able to work.

Combining both spatial and temporal locality, it is simple to state that the program in the example will spend a while executing the same few instructions 99 times before moving to another piece of code which will probably also be repeated. If the reader were to look at a real program, a much longer one than the example used here, repetitive sequences of a similar size to this would be found to lie within repetitive loops which were much larger, which in turn would lie within even larger loops, ad infinitum.

It would be a good idea to take advantage of both temporal and spatial locality by assuring that the repetitive portion of the program executes from a very fast memory while it is being used and resides in slower, less expensive memory when it is waiting to be used. At first glance, the most obvious way to achieve this is to divide the memory space into both faster and slower sections and have the operating system copy portions of code from the slower portions into the faster portion as required. This is exactly how a virtual memory system works, with the slower memory being the mass (magnetic) media and the faster portion being the main memory or DRAM. An alternative is to have a small, fast memory in addition to a full complement

of slower main memory and to use specialized hardware to assure that the currently useful parts of main memory are copied into the small, fast memory. This is what we call a **cache memory**.

The software-based technique is excellent, inexpensive, and works well for memory mapping, but it is not used for cache management for two good reasons. First, cost considerations usually force caches to be very small, and mapping memory spaces into and out of the cache via software would consume too much time at frequent intervals, thus offsetting the speed gain of the cache. (Typically, a piece of software at least the size of our example loop in Figure 1.4 would be needed to move the example loop into the fast memory, and a small, fast memory would need to have its contents replaced relatively often.) Second, most caches are designed to accelerate already existing systems where the operating system and other software cannot be reconfigured to support a memory which is split into faster and slower portions. Software in such systems must be kept unaware that the cache even exists. All existing programs must run similarly, although faster, on cached systems. A cache which achieves this end is called a **software transparent cache**.

1.4 FOOLING THE CPU

What seems to make sense, then, is to design a memory controller which will move data back and forth between slow and fast memory in hardware, leading the CPU to believe that the same address is being accessed at any time, only sometimes that location accesses faster than other times. Although fooling the CPU like this sounds tricky, it is not. Once the reader gets over the basic concept, simple cache designs become almost trivial.

I like to use an analogy which a colleague once made to me, of a clerk's desk next to a phenomenal number of filing cabinets, possibly at an uncomputerized credit bureau (see Figure 1.5). The clerk's mundane job is to answer the telephone and read items from a credit applicant's file in response to the caller's queries. The files are stored by the applicant's telephone number (we'll use the U.S. style of a seven-digit number for the sake of this argument). In a typical day, the clerk will answer 50 to 100 calls and will get up from the desk and search the files 50 to 100 times. However, since the same credit applicant may have applied for a loan on the same day or two at five or ten banks, several of these calls will be for the same data as was requested in a prior call either that day or a few days before. (Astute readers will note that this analogy uses temporal locality, but ignores spatial locality.)

Now let's assume that the clerk is very clever and sees that there is an unused file drawer in the kneehole of the desk. Why not keep copies in the desk's file drawer of those files which are currently in high demand? The

photocopy machine is close to the filing cabinets, so making an extra copy to take back to the desk is no problem. But who knows which files will be requested repeatedly and which will be requested only once? Being unable to predict the future, the clerk will simply make a copy of every document which is pulled from the file.

This starts off very easily. When a file is requested, the clerk puts a copy of it into the desk's file drawer and makes a note of the fact that it is in the drawer on a sheet of paper hung upon the wall. This sheet of paper is the directory to the desk's file drawer. When answering the phone, the clerk first looks at the directory to determine whether or not a copy of the requested file resides in the desk's file drawer. If it does, the clerk need not get up, but can save time and effort simply by pulling that file from the drawer. The directory also lists where a certain piece of information may be found within the file drawer. On those occasions when a copy of the file is not in the file drawer, the clerk will get up, copy the file from the filing cabinet bank, and add it to the drawer, making a note to that effect on the directory.

Everything works very well until the file drawer gets full. What then? Being very practical, the clerk decides to put the date and time of the last access to a directory entry alongside of that entry. When a new file is put into the desk's file drawer, the directory entry with the oldest date and time will be erased, the file will be discarded, and the new file and directory entry will be put in its place. (In another interesting analogy, Intel once likened the same problem to that of a shopper who went to the supermarket for some items, but preferred to store often-used items in the refrigerator at home.)

This is the concept behind cache memory design. The massive bank of filing cabinets represents the system's main memory, and the clerk is the CPU. The desk's file drawer is called the **cache data memory** or **data RAM**, and the directory is aptly called the **cache directory**. The exact implementation just described is rarely used, since the hardware requirements are more expensive than some very good alternatives. Several tricks are used in exact cache implementations, and a considerable amount of this book will be devoted to those tricks.

There are four fundamental parts of a cache memory, all shown in Figure 1.6. The cache data memory is the small, fast memory which is used to store replicas of instructions and data which could be accessed more slowly from main memory. When the size of the cache is discussed, the **cache size** is equal to the number of bytes in the cache data memory alone. The directory is not included in this number. The cache directory is a list of the main memory addresses of the data stored in corresponding locations of the cache data memory (i.e., if data in a certain cache data location comes from main memory address 0000 0000, then the cache directory would contain a representation of 0000 0000). So with each cache location, not only is data

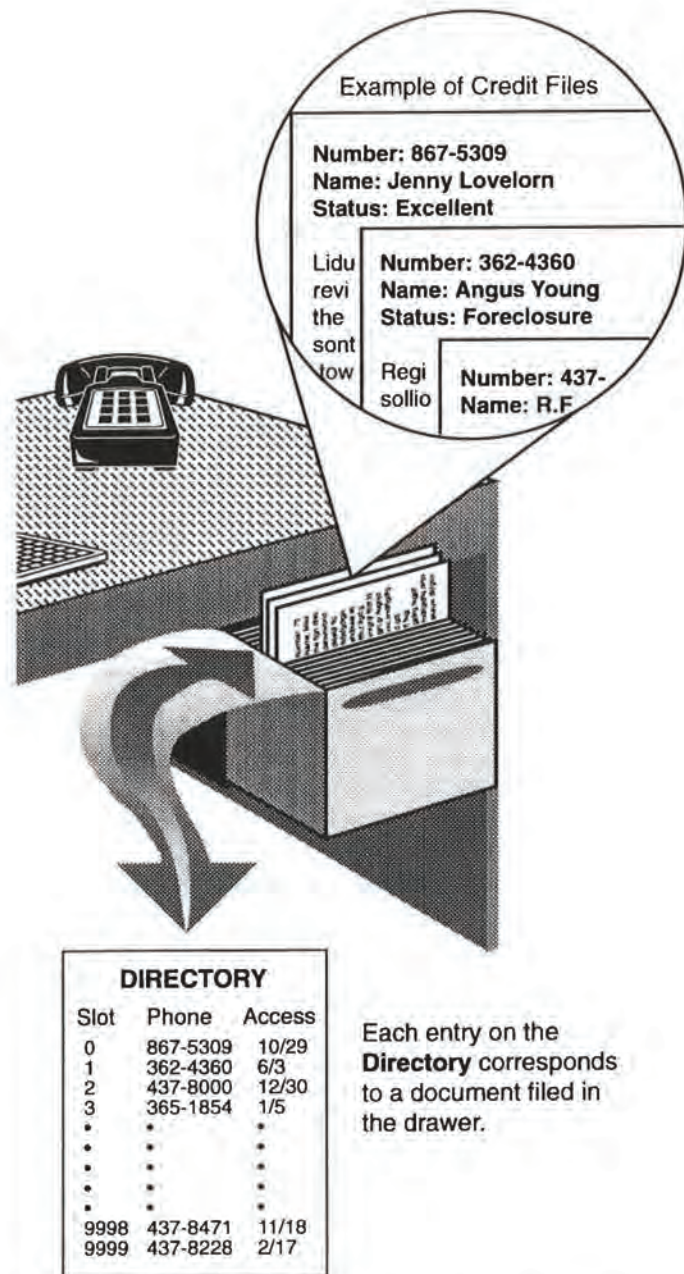
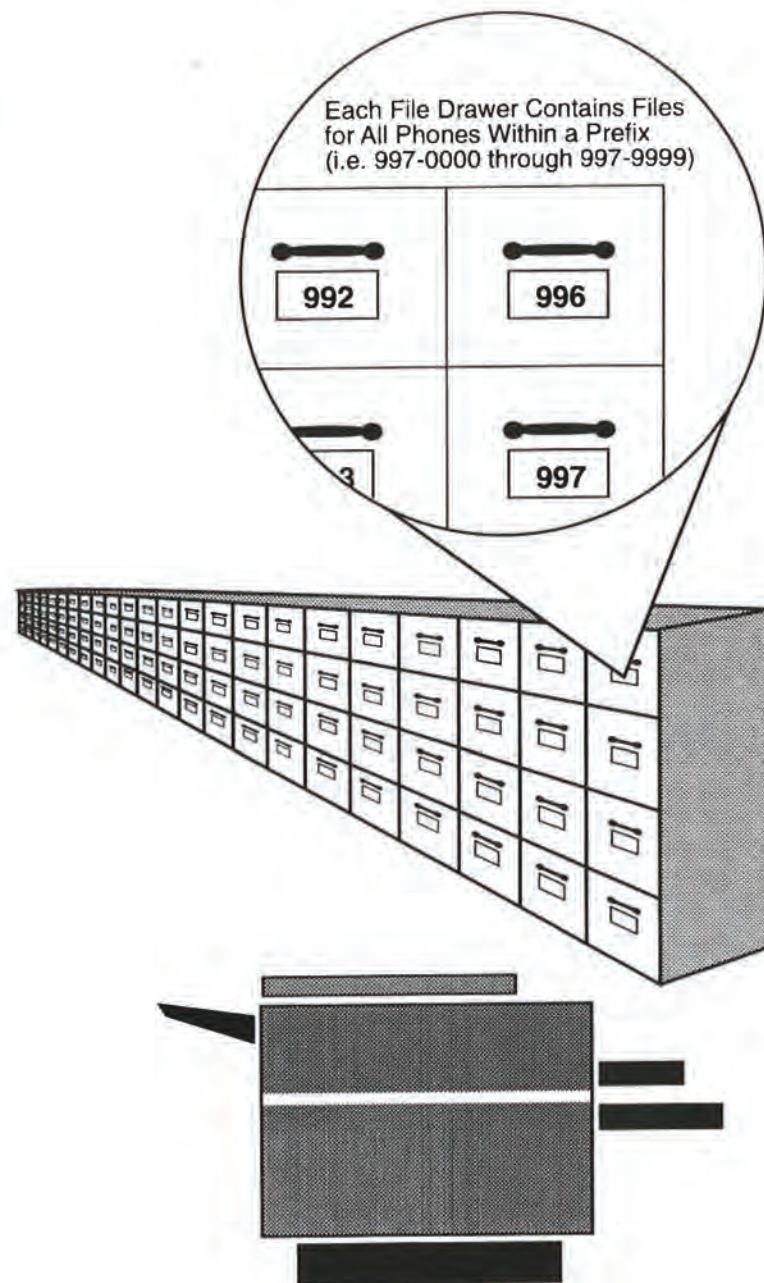


Figure 1.5. A clerk at a credit bureau is responsible for giving callers credit reports from a bank of 250 filing cabinets, representing the entire set of telephone numbers within an area code.



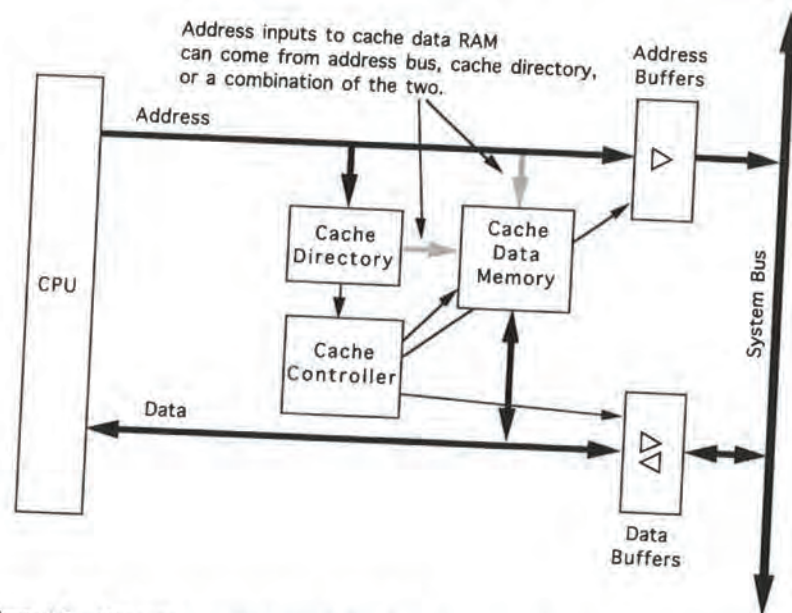


Figure 1.6. The cache memory consists of a directory, a cache data memory, buffers (to isolate the cache subsystem from the system bus), and the cache controller.

stored, but so is an address, making the combined directory and data RAMs behave like a single, very wide memory.

The bus buffers serve a very important function, even though they are basically the same chips which would have been used in the noncached system shown in Figure 1.2. These buffers are now controlled in such a way that if the cache can supply a copy of a main memory location (this is called a **cache hit**), then the main memory is not allowed to put its data onto the CPU's data pins. In many cache designs, the address is never even sent to the main memory unless the cache has indicated that it does not contain a replica of the data requested by the CPU (a **cache miss** or sometimes **fault**). So during all cache hit cycles, the CPU converses with the cache, and in all cache misses, the CPU converses with main memory. The data bus between the CPU, the cache data RAM, and the system bus buffers is sometimes referred to as the cache's **data path**.

Tying the whole system together is the cache's control logic, which is called the **cache controller** or **cache management logic**. It is this logic which implements the algorithm which moves data into and out of the cache data

memory and the cache directory. This is the crux of cache designs, and the cache logic embodies the implementation of some hotly debated decisions called the **cache strategy**. When a strategy is decided upon and implemented, it becomes the cache's **policies**. Policies will be evaluated in some detail in Chapter 2. The control logic also determines when to turn the bus buffers on and off and when to read from and write to the cache data RAM. In some systems, the system bus interface is sufficiently complex that the system bus side of the cache controller is subdivided into its own portion of the cache control logic, in which case it may be called the **storage control element (SCE)**.

Cache size and policies both affect the **hit rate** of the cache (the percent of CPU cycles which are cache hits). The **miss rate**, conversely, is the remaining percentage of CPU cycles. Throughput of a cached system can be computed based upon the average number of wait states per bus transaction. This number equals the miss rate times the average number of wait states on the system bus (plus the hit rate times the cache latency, if the cache requires the use of wait states).

As an example, let us assume that all system bus accesses take three wait states, and the cache responds in zero wait states. With a cache memory offering a 90% hit rate (an easily achieved number), the average number of wait states for the cached system would be $10\% \cdot 3 = 0.3$ wait states per memory cycle. Fractional wait states are perfectly valid in this sort of calculation, since this is an average between wait state and zero-wait cycles. The same system with a much smaller cache might exhibit a typical hit rate of only 80%, and the average number of wait states would be double that of the larger cache, or 0.6 wait states. The jump in throughput is even more pronounced if DRAMs with slower cycle times are used, requiring even more wait states per bus cycle. An interesting way of looking at this is to expand the problem to see where all the system clock cycles are consumed. Let's say we have a processor whose average zero-wait instruction execution time is two cycles, and cache misses execute with three wait states for a total of $3 + 2 = 5$ cycles. Ten instructions in the system with an 80% hit rate cache would consume $10 \cdot 0.8 \cdot 2 + 10 \cdot (1 - 0.8) \cdot (2 + 3) = 26$ cycles, only 16 of which would execute from the cache. In other words, 10/26 or almost 40% of the time the CPU would be attempting to get through 20% of the instructions. This also means that the bus will be in use 40% of the time, the implications of which we will examine in Section 1.8.

The process of caching follows a certain pattern. At power-on, the cache contains random data and is disallowed from responding to a request from the CPU. When the processor reads data from a main memory location, the cache data RAM is ordered to copy that location's contents, while the corresponding cache directory location is told to copy the address requested by

the CPU. This same sequence would occur for any further cycles until a loop was encountered. Once the processor reached the end of the loop, it would again output the address of the first location, which in all probability would still be in the cache data memory. This time, however, the data from that location would be supplied to the processor from the cache, rather than from the main memory, as would the rest of the instructions in the loop. It becomes pretty obvious how much of a help this would be. In our example program, the instructions in the loop would operate at main memory speeds only during the first go-round, then would subsequently execute much faster from cache for the next 98 times through the loop/

This covers cached read cycles. What happens during write cycles depends greatly upon the cache policies chosen, and will be examined in depth in Chapter 2.

From the CPU's perspective, the data always comes from main memory, although at some times it arrives faster than at other times.

1.5 CACHE DATA AND CACHE-TAG MEMORIES

So we now know how simple it is to reroute the CPU's main memory request to be satisfied by the cache, but the design of the cache directory and cache data memories has yet to be explained. These might be pretty intimidating to the first-timer. The choice of directory architecture has a strong impact upon the design of the cache data memory, so the directory will be investigated first.

Certain college-level courses explain that the cache directory consists of a content addressable memory of **CAM**. This is a directory type which corresponds well to our example of the clerk in the credit bureau. A CAM is a backwards memory which outputs an address when data is presented to certain inputs. The address shows where a matching entry has been found within the CAM. All CAM locations are examined simultaneously for matching data, and if a match is found, its address is placed upon the address output pins. This is particularly useful when a 32-bit processor address needs to be converted to a much smaller cache address. The 32-bit processor address is presented to the data input pins of the CAM, and a shorter address pops out of the address output pins. In truth, CAMs are almost never used to implement cache designs, but they facilitate the explanation of a cache directory, so we'll examine CAMs here first, then see what the alternatives are.

One big question that comes out of this is "what is a CAM?" A CAM is made of a series of address registers, each of which contains a comparator which compares the address contained within the register to the address currently on the compare address bus (see Figure 1.7). This is achieved by

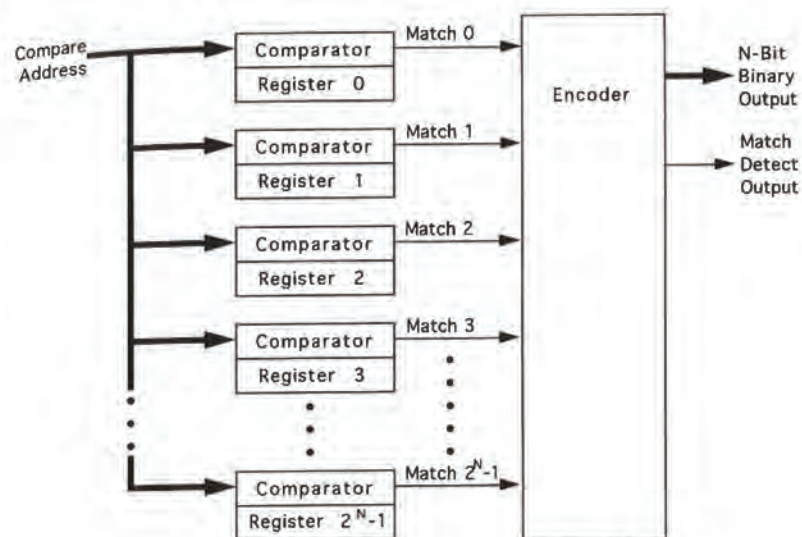


Figure 1.7. In a content addressable memory (CAM), an address presented to the compare address bus is compared simultaneously with the contents of every memory location. If there is a match at any location, the address of that location is sent to the N -bit binary output pins.

having a separate comparator attached to each and every address register. Although this sounds like a lot of hardware, it can actually be manufactured relatively easily in silicon by using a specialized, seven-transistor, static RAM cell which is less than twice as complex as the industry's standard four-transistor cell. The comparator outputs are encoded by a simple priority encoder into a binary representation of the address of the matching entry. During a write cycle, both an address and data are presented to the CAM, and a write pulse is generated. During a read cycle, however, data is input to the CAM, and an address is output. The output address is the address of the CAM location which contains matching data. If no data match is found, a no-match signal is output from the CAM.

The reader might ask "Why aren't CAMs widely available?" The simple reason is that much simpler schemes perform nearly as well as the CAM approach, yet can be constructed from standard static RAMs and allow the use of less expensive cache data RAMs. These alternate approaches will be discussed here briefly and in depth in Chapter 2. Cache designers are not ready to pay as much for CAMs as CAMs would have to cost to attract the interest of semiconductor manufacturers. The author knows of only two kinds of CAMs available on the market today. The first is a mature product, an ECL

4 × 4 bit device which would have been designed into mainframe caches before faster (70ns) static RAMs became available. The other is a device designed specifically for LAN address filters, which requires a 48-bit compare address to be input in three multiplexed chunks over a relatively long period of time (i.e., 100ns).

Back to the example: Let's assume that the system has just been initialized and that no cache locations contain valid copies of main memory locations. Where in the cache should we put the first location accessed by the processor? Some processors seek their first instruction after a reset from the first main memory address, or address zero. Other processors start off at the top of the memory space (FFFF FFFF). Yet others seek a vector at one of these two addresses, then jump to the address pointed to by that vector.

If the cache directory is a CAM, then *any* main memory address can be mapped into *any* directory location. This approach is called **fully associative**, a term which simply means that any main memory address has the full freedom to be replicated anywhere within the cache. The method employed to determine which cache location should be used to store a copy of a main memory address is referred to as **mapping**, or **hashing**. (A fully associative approach is just one of many hashing algorithms which we will examine in this book.) An appropriate (hopefully unused) directory location would have to be assigned whenever something was to be put into the cache. The cache controller does this by assigning an address representing the directory location to be used to store the incoming data. This address is fed into both the CAM and the cache data memory while the main memory is being accessed. The data contained at the address of the main memory access (which is a completely independent value from the directory address just mentioned) is then written as data into the directory while the main memory data is both fed into the CPU and written into the cache data RAM (see Figure 1.8). Later, when the processor's address again matches the address stored in the CAM, the CAM outputs the address of the cache data RAM location which contains matching data. This address is fed into the cache data RAM, which consists of a standard static RAM, and the data from that addressed location becomes available to the CPU.

Partly due to the lack of availability of large CAMs, several ingenious alternatives to the CAM approach have been devised. These alternatives also tend to be significantly less complex than the CAM approach. Returning to the file clerk analogy, let's put together an organizing system which will take advantage of the fact that the desk drawer is the same size as any of the drawers in any of the filing cabinets. The phone numbers used are all seven digits long and follow the format 867-5309. Furthermore, let's say that the filing cabinets are arranged so that each drawer represents a prefix (the first three digits of the telephone number), and each is capable of carrying the

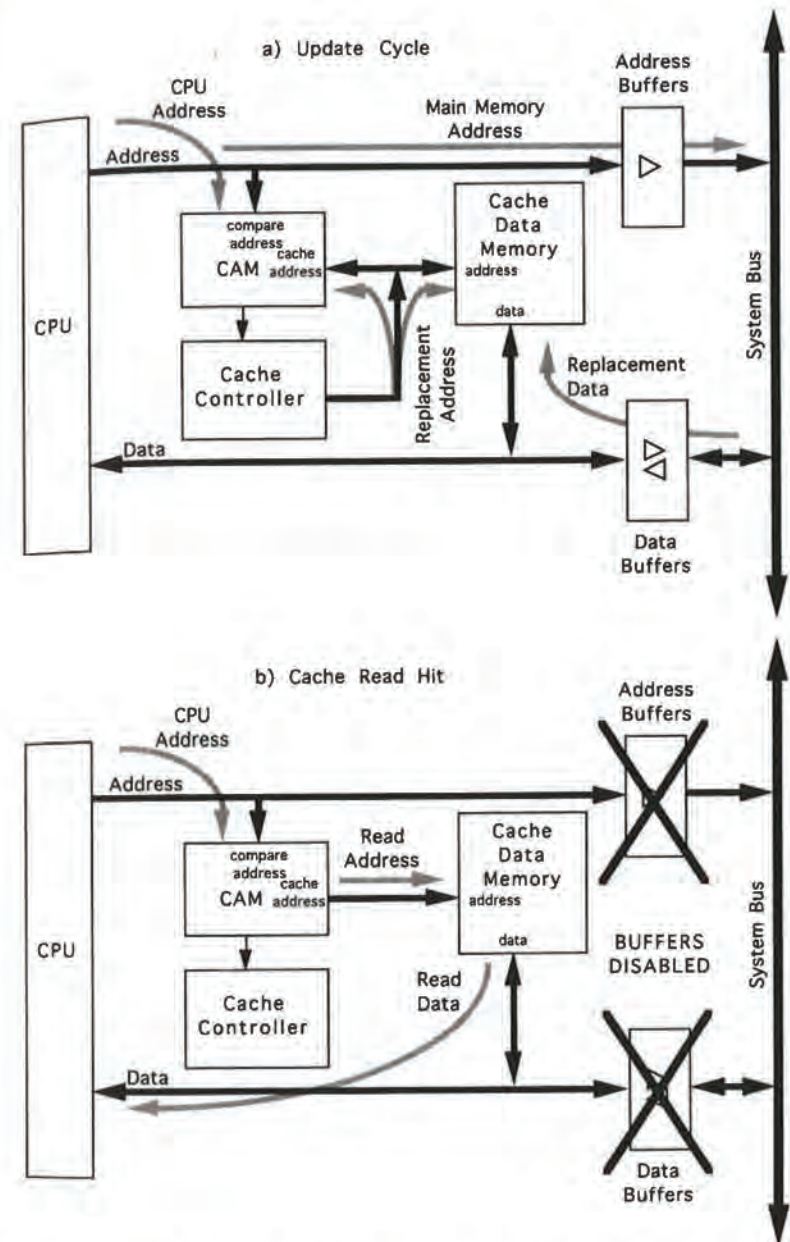


Figure 1.8. During the update of an entry in a fully associative cache (a), the cache controller feeds the address which is to be replaced into both the CAM and the cache data memory. When there is a cache hit (b), the CAM outputs this address to the cache data memory, causing its data to be fed into the CPU.

files for all 10,000 numbers or credit applicants within that prefix (from -0000 to -9999). So the first drawer would have numbers 000-0000 through 000-9999 and so forth. The clerk could also arrange the desk drawer into 10,000 slots, so that each slot would only be allowed to contain a file whose suffix (last four digits) matched the slot number; in other words, the number 867-5309 would only be allowed to be placed within desk drawer slot 5309. The directory would still be a paper on the wall containing 10,000 entries describing the phone numbers of the data stored in each of the desk drawer slots; however, each entry would be at a location which matched the suffix of the phone number, so only the prefix would need to be written into the directory (so for the number 867-5309, the prefix 867 would be written onto line 5309). Furthermore, the clerk would no longer have to choose where to put a new cache entry, so the directory entries would no longer need to contain the time of their last access. This is the most popular hashing algorithm in today's cache designs and is called **set associative**, since, for the sake of clarity, the last four digits of the telephone number have been named the cache's set address, and within any set the entry is associative (can have any prefix).

With this new method, the clerk can begin to look into the desk drawer while at the same time looking up the entry in the directory. There is no longer a need to cross-reference the directory to determine which desk drawer location contains the file. If the prefix (867) at the correct set address (5309) matches the one being requested by the caller, then the clerk will pull the file from desk drawer slot 5309 almost instantly. In a set-associative design, the address which is output from the processor is split into the equivalent of a prefix and a suffix at a location determined by the size and architecture of the cache. The lower address bits, those corresponding to the suffix or last four digits of the telephone number in the example, are called the **set** bits since they contain the set address. Some call these the **index** bits, but this terminology is in direct conflict with the terminology used in virtual memory translation, so, in the interests of clarity, it will not be used in this text. The remaining (upper) bits are compared against the directory entry for the selected set and are referred to as the **tag** bits.

You may recall a statement I made when first describing the clerk analogy, that this analogy does not account for spatial locality. This is easy to see, because the clerk might at any time receive calls for absolutely any file, representing any telephone number, in any order. Computer code never runs in such a random fashion, but tends to be full of loops and repetition, and whenever the program counter is not jumping around on a loop or a sub-routine call, code will execute sequentially. The set-associative approach takes advantage of this spatial locality by placing sequential instructions not at entirely random cache locations, but at sequential locations. As just illus-

trated, the set-associative cache does not allow a main memory location to be mapped into any cache address, but instead the cache is constrained so that the lower address bits of the cache location must match the lower address bits of the matching main memory address. Since the processor will be running through a small group of sequential addresses, this looks as if it should pose no problem, and the set-associative cache should run every bit as efficiently as the more complex fully associative design. We will see otherwise in Section 1.6.

An earlier paragraph stated that a fully associative cache (one with a CAM) would require the use of more expensive cache data RAMs than a set-associative cache. Why? In a fully associative cache, all address bits are filtered through the directory before being fed to the cache data RAM. This means that during the cached CPU's fastest cycle, a cache read hit, the processor address must first travel through the directory CAM and be translated into a cache address, then that address must be passed through the cache data RAM before the read data can be made available to the CPU (see Figure 1.9a). This means that the delay between the processor's address output and the data returning from the cache is at least equal to the combined propagation delays of the CAM and the RAM. Thus, if data needs to be returned to the processor within, say, 30ns, then that 30ns must be divided between the CAM and the RAM. Today's 150 MHz microprocessors require this transaction to occur in less than 20ns. A system with an 8ns CAM (good luck!) and an 8ns cache data RAM would be hard-pressed to use any logic in this critical path.

The set-associative cache reduces this CAM plus RAM delay into a single RAM access. In Figure 1.9b, the CPU address is shown to run directly to both the cache directory and to the cache data RAM. For the sake of simplicity, the design shown uses a cache directory which is made of the same depth of standard static RAM as is the cache data RAM. (Alternatives to this will be explained in painful depth in Chapter 2.) Let's assume that the cache is 8,128 (8K) locations deep, requiring 13 address bits to be fully accessed. The lower 13 CPU address bits are then routed to both the cache directory and the cache data RAMs simultaneously. While the cache data RAM finds its replica of the data stored at a main memory location with matching lower address bits (but goodness knows which upper address bits), the directory looks at a replica of the upper address bits stored at the location with the same lower address bits. This is pretty confusing, but, based on the desk analogy, just remember that the last four digits of the telephone number 867-5309 are 5309, so slot 5309 is accessed in the file drawer. The directory then must tell us whether there is a match between the stored prefix (867) or whether some other prefix's number (984-5309) is being accessed. To settle the delay issue, though, notice that during the processor cycle time, the

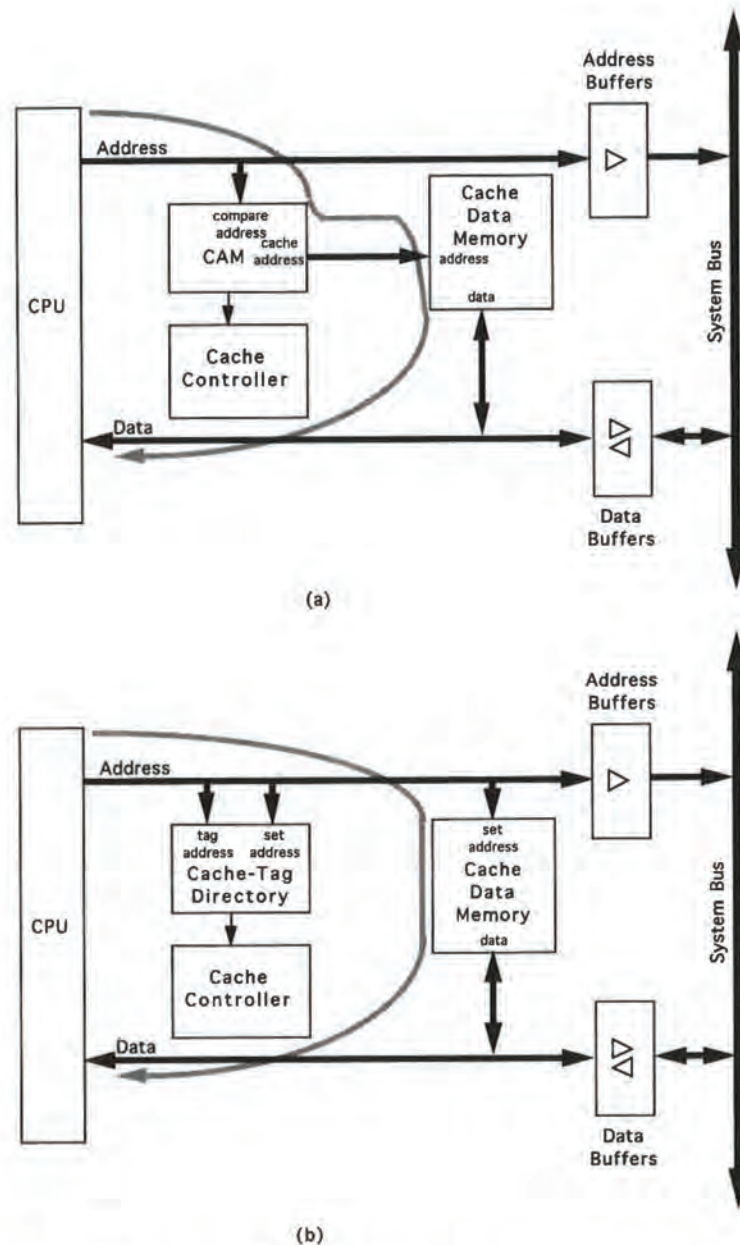


Figure 1.9. The critical speed path, from CPU address out to data in, runs through both the CAM and the cache data memory in fully associative cache designs (a). Set-associative caches (b) allow the CPU address to be fed directly to the cache data memory without going through any other devices.

cache directory and the cache data RAM are being examined *simultaneously*, with the output of the cache directory being used simply to tell the CPU whether to proceed at full speed (a hit) or to wait while a main memory access is used to fetch the data which was requested by the CPU but did not reside within the cache.

The directory in a set-associative cache is considerably simpler than that of a fully associative cache. The fully associative cache's directory not only had to keep track of whether or not there was a valid entry for a main memory location contained within the cache (a fact which has not yet been described, but will be in Chapter 2), but also had to track where that entry resides in the cache data RAM, while the set-associative directory must only look to see whether the cache entry with the same set bits as the CPU's output address came from the location which has the same tag bits, and whether that location is valid. This is a simple comparison. Does or does not the tag at location 5309 match 867 for the phone number 867-5309, and is the location valid? Obviously, the tag comparison can be performed using exclusive ORs feeding into an AND gate, the function of a simple address comparator chip (Figure 1.10). Validity is equally as simple and will also be described in depth in Chapter 2. Set-associative caches assume (until told otherwise) during a processor read cycle that the data in the cache is what the processor is requesting. The cache data RAM starts the read cycle with its data outputs turned on, and its address pins will be tied directly to the processor's less significant address output pins (see Figure 1.9b). The cache-tag RAM, meanwhile, is looking up the address' more significant bits to see if the cache entry is from the correct address. If there is a match between the requested address and the address of the cached copy of the main memory data, then the cache controller gives the processor a Ready signal to accept the data from the cache data RAM and allows things to continue as they are. If the cache contains data at the same set address which is from a different part of main memory (i.e., the tag bits don't match), then the cache controller declares to the processor that a cache miss occurred by not asserting the Ready input, then reads main memory data into the cache, allowing the CPU to continue as the main memory data and address are being written into the CPU. The cache data RAM in a set-associative cache need only have an access time of slightly less than the processor's minimum address to data cycle. The cache-tag RAM would have a similarly slow access time, less the delays needed for the comparator and the downstream logic. Slower static RAMs are usually much less expensive than faster ones, so the designer can save a lot of money by going with a set-associative design. Some static RAM manufacturers provide products which have a comparator contained within a standard SRAM. This provides the same function at a certain advantage in speed, component count, and complexity of the downstream logic. Whether

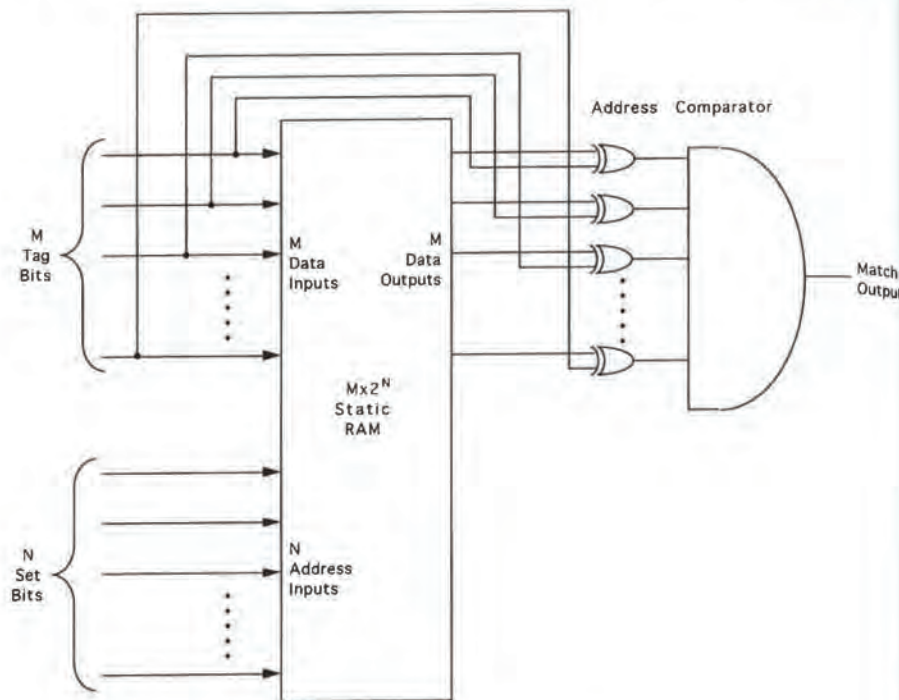


Figure 1.10. In a cache-tag RAM, the stored data at a certain RAM address (addressed by the set bits of the address bus) is compared against the upper address bits, or tag bits of the address bus. This comparison is performed with a simple address comparator consisting of exclusive OR gates feeding into an AND.

the directory is made from a simple RAM with a separate comparator or with a chip which includes both, the directory in a set-associative cache is called the **cache-tag** memory simply because it is where the address tags are stored. There are a few other terms for a cache-tag RAM, notably **tag RAM**, and **cache address comparator**, **address comparator**, **cache comparator**, or simply **comparator**.

1.6 THRASHING: GETTING THE MOST OUT OF THE CACHE

The entire argument in the preceding section ignored a phenomenon called **thrashing**. When a cache thrashes, a commonly used location is dis-

placed by another commonly used location. Every time the CPU can't find what it needs within the cache, it must perform a main memory access at the slower speed of main memory. This makes these accesses at least as slow as they would have been had there been no cache.

Say the processor must examine two addresses in close sequence which contain the same set bits. How can this happen? There are several ways. First, most code is constructed out of calling routines and subroutines. There is a reasonable possibility that some of the code in the subroutine will have the same set bits as the code in the calling routine, since the calling routine may be nowhere near the subroutine. This is especially true for small caches which have fewer set bits, which makes the odds of a conflict higher. Next, consider what happens when a subroutine is called. The program counter and possibly lots of other registers are pushed onto the stack. Where is the stack? What are the possibilities that the stack's set bits match the set bits of some of the code in the calling routine and the subroutine? Another place where the set bits might get confused is when the program uses a pointer, say, to move a block of values or to examine a text string. Last, think of what happens when an interrupt is serviced. Will the interrupt's service routine (or some subroutine called by that service routine) have set bits which match the set bits of a calling routine address? Any of these scenarios can cause the cache to thrash.

For an example, let's reexamine the piece of code shown in Figure 1.4. Three addresses have the last three digits of 008 hexadecimal: the beginning of the loop in the calling routine, the stack pointer, and the beginning of the subroutine. If our cache uses these three last digits as a set address, or uses a subset of these three digits, then the cache will thrash. The next paragraph and the diagram in Figure 1.11 will illustrate this in detail. The stack pointer is a kind of "gotcha" in this instance, since it causes the CPU to overwrite other items in the cache. I'm pointing this out, because a lot of designers tend to forget that instructions which push and pop items perform reads and writes that you have to think about in order to remember. It is macho for a cache designer to always know where the stack pointer is, especially under a multitasking operating system.

First, the cache loads address 0000 1000, the "load stack pointer" instruction. This gets put into cache location 000, and at tag location 000 the tag address 0000 1 is written. All this happens at main memory cycle times, so the cache does not accelerate this particular cycle. Similarly, the next two instructions are loaded at main memory access times into cache addresses 004 and 008, and the tags for these two newly cached copies of these instructions are also filled with 0000 1 at tag addresses 004 and 008. The instruction at location 0000 1008 is a "jump to subroutine" (JSR), so the program counter is pushed onto the stack at stack location F000 0008, and, in

1) BEFORE OPERATION OF ROUTINE

Set Address	Cache Tag	Cache Data
000	?	?
004	?	?
008	?	?
00C	?	?
010	?	?
014	?	?

2) ADDRESS 0000 1000: LOAD STACK POINTER

Set Address	Cache Tag	Cache Data
000	0000 1	LOAD SP, #F000 0008
004	?	?
008	?	?
00C	?	?
010	?	?
014	?	?

3) ADDRESS 0000 1004: LOAD REGISTER 1

Set Address	Cache Tag	Cache Data
000	0000 1	LOAD SP, #F000 0008
004	0000 1	LOAD R1, #0000 0063
008	?	?
00C	?	?
010	?	?
014	?	?

Figure 1.11. The contents of a set-associative cache during execution of the code shown in Figure 1.4. Note that location 008 never retains the cache data long enough to offer any speed advantage.

this example cache design, will immediately overwrite the stored instruction from memory address 0000 1008 (the JSR instruction), once again at the slower speed of a main memory access. Upon entering the subroutine, the first instruction of the subroutine is fetched from address 0001 1008, once again at a main memory access time, and immediately replaces the newly cached copy of the program counter stored at cache location 008. When the return instruction is encountered, the pushed program counter is restored not from cache, but from main memory, since it was bounced out

4) ADDRESS 0000 1008: JUMP TO SUBROUTINE

Set Address	Cache Tag	Cache Data
000	0000 1	LOAD SP, #F000 0008
004	0000 1	LOAD R1, #0000 0063
008	0000 1	JSR SUB1
00C	?	?
010	?	?
014	?	?

5) CONTINUATION OF JUMP TO SUBROUTINE: PUSH PROGRAM COUNTER

Set Address	Cache Tag	Cache Data
000	0000 1	LOAD SP, #F000 0008
004	0000 1	LOAD R1, #0000 0063
008	F000 0	0000 100C (PC Value)
00C	?	?
010	?	?
014	?	?

6) FIRST SUBROUTINE INSTRUCTION, ADDRESS 0001 1008: LOAD REGISTER 2

Set Address	Cache Tag	Cache Data
000	0000 1	LOAD SP, #F000 0008
004	0000 1	LOAD R1, #0000 0063
008	0001 1	LOAD R2, (R1+#1000)
00C	?	?
010	?	?
014	?	?

Figure 1.11. (continued)

of the cache by the first instruction of the subroutine. No cache speed advantage is realized for this stack address. Finally, the calling loop is reentered, and the JSR instruction from address 0000 1008 is again fetched from main memory, instead of cache, once again at a slower main memory cycle time. This whole process of the three items continually replacing each other in the cache is repeated 99 times. Obviously, locations with the lower address bits matching 008 have not benefited at all from the use of cache!

The main point of this illustration is to show that thrashing is commonplace in small caches, but fades in importance in larger caches, as the number of set bits increases, causing the likelihood of confusion between the set

7) AFTER LAST SUBROUTINE INSTRUCTION, POP PROGRAM COUNTER

Set Address	Cache Tag	Cache Data
000	0000 1	LOAD SP, #F000 0008
004	0000 1	LOAD R1, #0000 0063
008	F000 0	0000 100C (PC Value)
00C	0001 1	CMP R2, "A"
010	0001 1	JNZ SKIP
014	0001 1	LOAD R2, "z"

8) CALLING ROUTINE ADDRESS 0000 100C: LOOP BACK TO 0000 1008

Set Address	Cache Tag	Cache Data
000	0000 1	LOAD SP, #F000 0008
004	0000 1	LOAD R1, #0000 0063
008	F000 0	0000 100C (PC Value)
00C	0000 1	DJNZ R1, LOOP
010	0001 1	JNZ SKIP
014	0001 1	LOAD R2, "z"

9) REPEAT ADDRESS 0000 1008: JUMP TO SUBROUTINE

Set Address	Cache Tag	Cache Data
000	0000 1	LOAD SP, #F000 0008
004	0000 1	LOAD R1, #0000 0063
008	0000 1	JSR SUB1
00C	0000 1	DJNZ R1, LOOP
010	0001 1	JNZ SKIP
014	0001 1	LOAD R2, "z"

Figure 1.11. (continued)

bits of one address and another to decrease. You might want to consider the two extreme arguments to this. How likely is thrashing in a system with only a single set bit (the least significant bit of the address)? How likely would thrashing be in a system with as many set bits as the CPU had address outputs (the cache is as large as the largest possible main memory)? The second point is that thrashing is a phenomenon of set-associative caches, where cache lines are overwritten based on their set bits rather than on some more reasonable algorithm. A fully associative cache would have handled the problem just illustrated as easily as it would have conducted a transaction

with no match between any address bits. In Chapter 2, we will examine cache architectures which fall between the set-associative and the fully associative models.

Another buzzword relating to thrashing is the **compulsory miss**. Some cache misses will occur no matter how good the cache design is, simply because the cache is being asked to provide data which it had no reason to have previously contained. Since there is no way to avoid such misses, they are called compulsory misses. Thrashing never causes compulsory misses. Just like most of the buzzwords in this book, though, there are some alternatives to the nomenclature compulsory miss, namely, **stationary miss** and **nonconflict miss**.

1.7 CACHES AND THE MEMORY HIERARCHY

Memory **hierarchy** or **storage hierarchy** refers to the distinct stages in the memory's price per bit, size, and performance. The most expensive memory in a computer consists of the registers within the processor itself. These devices are designed to perform extremely quickly, and their structure allows one register to be written to at the same time that one or more different registers is being read. That structure consumes several transistors per bit, and the cost as well as the speed are related to the number of transistors required by each bit. Access to the registers is via on-chip buses, so the access time for register data is not hampered by any requirements for the signals to go off chip. Most processors have only a few registers, or a few tens of registers, and these registers are the width of the data path within the chip. Register access times are usually under 2ns.

In most noncached systems, the next level of memory is the main memory. This is usually made out of dynamic RAMs (DRAMs) which have an access time several times as long as the access time of the registers. Dynamic RAMs are made using a single transistor per bit, with a small percentage of additional transistors required for the support circuits. The structure of a DRAM makes it the cheapest possible form of semiconductor memory. In contemporary microprocessor-based systems, a typical DRAM main memory might be somewhere between 16 and 512 megabytes. Access times to main memory today consume in the order of 70ns.

The next two levels are comprised of rotating media. Magnetic disk is usually used to store whatever is not capable of residing within main memory. Today's hard magnetic disks exhibit access times in the tens of milliseconds and contain from hundreds to thousands of megabytes. Hard disks are often backed up with removable media, which is often either tape or diskette. Either of these is significantly less expensive than hard disk, but both have ac-

cess times of several seconds. Since this level is removable, the size is limited only by the user's desire to keep an inventory of tapes or diskettes.

Cache memory fits between the first and second levels of the description. Several microprocessors today contain an internal cache which is made of standard static RAM bits. Unlike registers, these locations cannot be written to and read from simultaneously, so certain operations will perform more slowly in on-chip cache than they would using internal registers. A register bit may require ten or more transistors to implement, whereas the SRAM cell used in the internal cache can be made of cells containing between four and six transistors per bit. Since the microprocessor is manufactured using the same chip, the on-chip cache is limited in size, mostly to keep the processor's cost from going through the roof. Die costs are not proportional to die size, but increase at more than a second-order rate. On-chip cache shares the advantage of not requiring an off-chip access at the expense of access time, so they are pretty fast. Today's microprocessors usually contain caches of between 8 and 64K bytes.

External caches are used both in systems which have and do not have on-chip caches. The external cache which is connected to an uncached processor serves to accelerate main memory accesses, as has just been explained. Such caches usually range from 256K bytes to 16M bytes and are made using economical SRAM chips constructed of four-transistor-per-bit memory cells. The cache's access time is around 3 to 15ns. If the processor has an on-chip cache, any external caches used will usually be implemented with architectures which differ considerably from the architecture of the on-chip cache, both to reduce costs and to make up for some of the deficiencies of the on-chip cache. These design issues will be examined in Chapter 2.

Table 1.1 shows in dollar magnitudes and bandwidth a typical memory hierarchy for a high-performance cached system, whose processor uses an on-chip cache.

Table 1.1. Memory hierarchy, and relative costs and typical sizes of each level.

Memory Type	Typical Size (bytes)	Cost per Megabyte (\$)	Bandwidth (megabyte/sec)
Registers	10^1	10^5	500
On-chip cache	10^4	10^4	300
Off-chip cache	10^5	10^1	100
Main memory	10^7	10^0	50
Hard disk	10^8	10^{-1}	15
Magnetic tape	Infinite	10^{-2}	1

Source: Dataquest 1997.

1.8 REDUCING BUS TRAFFIC

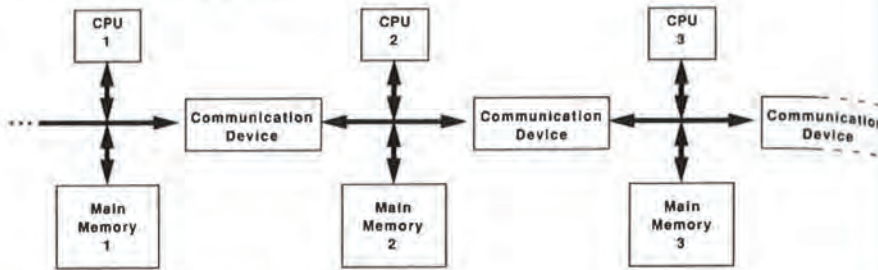
One reason for adding a cache to a system has not yet been mentioned in this chapter: Multiprocessors can take advantage of cache memories to improve the effective **bus bandwidth** into their main memories. Bus bandwidth, also called **bus traffic** or simply **traffic**, is a measure in bytes per second of the maximum speed at which data can be moved around on the bus (bus transactions). For example, a bus with a word width of 16 data bits (2 bytes) which can transfer up to 8 million words per second would have a bandwidth of $2 \cdot 8 = 16$ megabytes per second. The percentage of the bus' bandwidth which is consumed by a processor is called **utilization**. (Similar to main memory bandwidth, the cache to CPU interface has its own bandwidth, which is called the **cache bandwidth**. If the cache bandwidth is lower than that needed by the CPU, the CPU must be slowed down, through either the use of a lower clock frequency or the addition of wait states.)

We will soon see that the use of cache memories to reduce a processor's main memory bandwidth needs is a very important benefit and should become much more important in the near future with the growing acceptance of multiprocessor architectures as a means of dramatically improving system performance. Multiple processors can get into trouble if their utilization approaches 100%, and perform worse than systems with fewer processors once the 100% limit is reached.

There are two kinds of multiprocessor systems, shown in Figure 1.12a and b. The top diagram shows what is called a **loosely coupled** multiprocessor system or a **distributed memory multiprocessor (DMM)**. Loosely coupled systems are actually two or more distinct processors, each of which is capable of operating independently of the other(s). In the broadest terms, any two PCs connected via a modem or network could be called a loosely coupled multiprocessor system; however, the term is more often used for more intimately connected systems, perhaps connected via first-in/first-out buffers (FIFOs), dual-port RAMs, or dedicated serial buses. Most such systems have tasks divided well before run time to allow each processor to perform to its peak ability on a dedicated portion of the task. It is not unusual for the processors to be different types altogether (i.e., a digital signal processor and a general-purpose processor).

The type of multiprocessor system we will discuss in depth in this book is called the **tightly coupled** multiprocessor and is illustrated in Figure 1.12b. In a **tightly coupled** system or **shared memory machine (SMM)**, a single main memory is accessed by two or more processors via the system bus. Task allocation is performed by software at run time, offering complete flexibility in the range of application programs which can be run on the machine. Currently, tightly coupled architectures tend to be used in file servers, su-

a) Loosely Coupled Multiprocessor



b) Tightly Coupled Multiprocessor

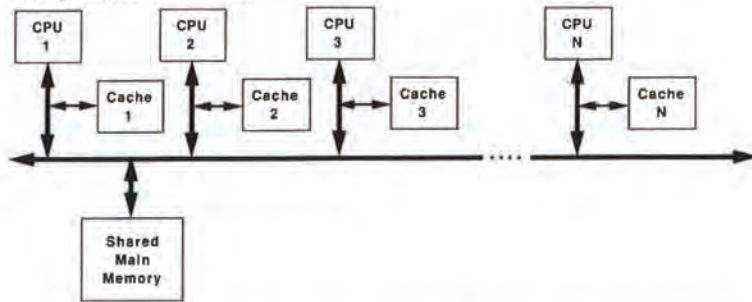


Figure 1.12. (a) A loosely coupled multiprocessing system. Each CPU has its own main memory and system bus. The processors communicate via the communication devices, which are usually accessed through each processor's system bus. (b) A tightly coupled multiprocessing system. All processors (or processor/cache complexes) communicate to a single main memory via a single system bus.

perminicomputers, and mainframes, but current trends indicate that they could become the prevalent architecture of the next century.

The reason that tightly coupled systems need cache is not hard to understand. In a tightly coupled system, each processor has its own cache (Figure 1.12b). The more often the cache is accessed, the less often the processor uses the bus to access main memory. This helps to keep the bus from saturating; that is, utilization is lowered to the point that the bus becomes more available in a timely manner to those processors that need it.

The amount of bus bandwidth required by a cached processor is proportional to the cache's miss rate, so it is not hard to see why designers of multiple-processor systems focus much more attention on miss rates than would the designers of single-processor systems. A designer of a single-processor sys-

tem might hesitate to spend extra money to improve the hit rate of a cache design from 96% to 98%. The effect on benchmarks would be negligible. In a multiple-processor system, however, the difference between a hit rate of 96% and a hit rate of 98% translates to a difference in miss rates of 4% and 2%, or double. In other words, one design would require half the bandwidth of the other, so half the number of processors could be used in the system before the effects of saturation would be noticed.

Let's take a more graphic approach to examine what happens as processors are added in a multiple-processor system. Our ideal is to get an incremental performance increase proportional to the number of processors added to the system. A five-processor system should perform at or near five times the performance of a single-processor system. For the sake of this argument, let's assume that the processor used in the example uses two cycles per instruction, that all instructions require a single memory access (an unrealistically low number), and that main memory has zero-wait performance. With a single processor, half of the main memory's bandwidth is used, and we can achieve a normalized performance level of 100%. If we add another processor, and if the two processors are interleaved, they will consume the entire bus bandwidth and will perform at 200% of the level of the single-processor system. The addition of a third processor in such a system would add absolutely nothing, since it would not have an interleaved slot, and there is no memory bandwidth remaining to serve the third processor. The performance of this system versus the number of processors is shown in Figure 1.13.

This case is drastically oversimplified, but it serves to set the scene for more realistic problems. Typically, complex instruction-set computers' (CISC) instructions consume a variable number of cycles and use more than one main memory access per instruction. This immediately rules out the effectiveness of interleaving, which requires balance and predictability to be useful. In real-life systems, bus arbitration is used to allow all processors access to the bus as needed. The mere use of an arbitration mechanism slows things down, since the bus must be requested by a processor before it is assigned. This means that the main memory access cannot begin until one arbitration delay after the processor attempts to start the access. The effect of arbitration is that a single processor can no longer perform at the 100% level of the example above and that the addition of processors has a more graceful, if less helpful, effect on the system's overall performance. In Figure 1.14a, we see the operation of an arbitrated multiple-processor system as processors are added. The addition of processors increases the probability of simultaneous arbitration requests, so that the performance increase of added processors is not as dramatic as in the first example, and there is a more gradual rolloff when bus saturation is approached. In such a system, after saturation is reached, the addition of processors actually slows down

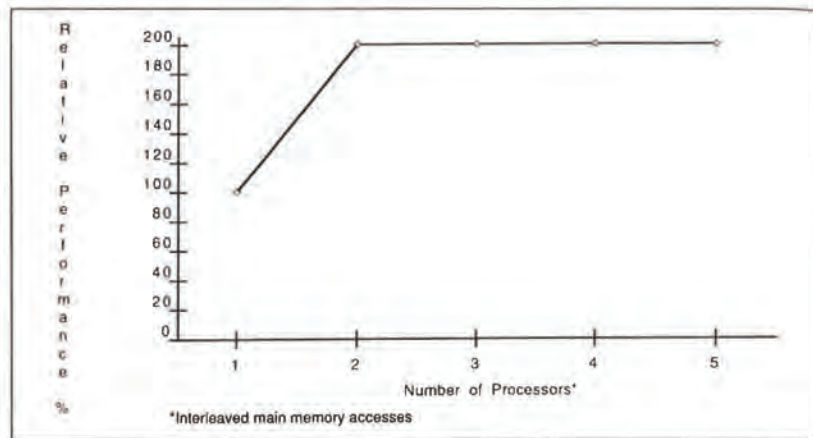
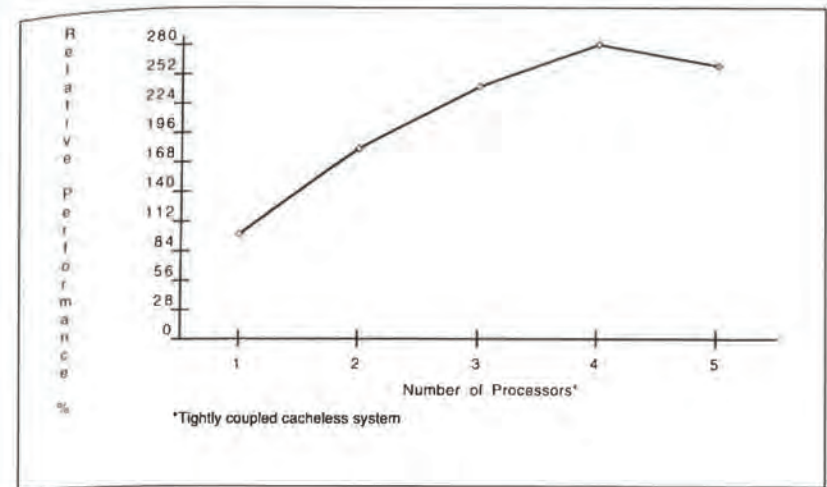


Figure 1.13. Performance vs. number of processors for the interleaved system described in the text. Each processor uses 50% of the available bus bandwidth, and no arbitration is required for either processor. Additional processors will be unable to access the bus, since it will be totally allocated to the two processors.

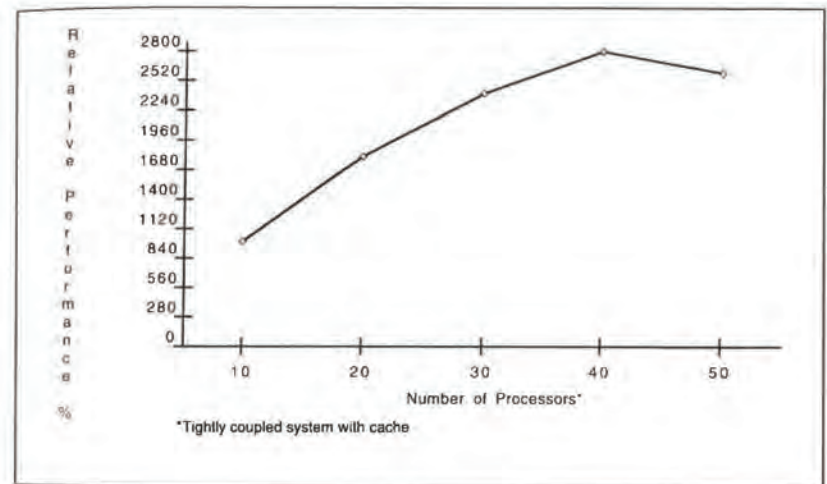
the entire system. One way of looking at this is that if each added processor requires exactly 20% of the available bus bandwidth (an unrealistically low level), then a six-processor system will require 120% of the available bandwidth and will perform at a 100%/120% level. High-performance processors actually require very close to 100% of the bus bandwidth, so the problem is much worse than this example shows.

We are still using a zero-wait main memory in our example. What happens when we add some realism to this dimension? It is easy to see that we can add a wait state to what is normally a single-cycle main memory access and halve the bus bandwidth, so the 100% number used in the preceding paragraph drops to 50%. Each processor can now operate at only 50%/120% of its potential. The normalized throughput of the six-processor system throughput drops to $6 \cdot 50\% / 120\%$ or 2.5 times the throughput of a single processor coupled with a zero-wait main memory system. If the per-processor bus utilization is bumped up to a more realistic 80% level, the six-processor system performs at $6 \cdot 50\% / 480\%$, resulting in a combined system throughput of only 63% of that of a single, zero-wait processor!

So it appears that in the real world, uncached, tightly coupled multiprocessors are not a cost-effective way to improve the throughput of a system. What happens when we add a cache (Figure 1.14b)? Let's go back to the interleaved example. Assume that we have the same zero-wait-state sys-



(a)



(b)

Figure 1.14. (a) Performance vs. number of processors for the arbitrated system. As processors are added, the overall system performance increases with diminishing returns, until the system throughput actually begins to degrade with the addition of a CPU. The curve is based on actual performance data for National Semiconductor's 8060 CPU. (b) Performance of a similar system using cached processors. By reducing each processor's bus utilization to 10% of its uncached figure, ten times as many processors can be added before the bus saturates.

tem as in the first example and that each processor has a cache with a very modest 90% overall hit rate, or a miss rate of 10%. Suddenly, the 50% bus usage requirement drops to 10% of 50%, or 5%. This implies that the bus will not now saturate until 20 processors are installed! What a difference! Of course, this is all in a very ideal environment, where the processors only require 50% of the bus bandwidth, and the main memory has no wait states. We have just seen that wait states throw a whole new bent on the statistics, and most processors require their buses more than 80% of the time.

An unfortunate side effect of adding caches to multiple-processor systems is that the designer must now account for all sorts of potential new problems dictated by the possibility that two caches may contain copies of the same data, and these multiple copies don't automatically match each other. This is such a difficult problem that the whole of Chapter 4 is devoted to it! Suffice it to say that the benefits of adding caches to multiple-processor systems are strong enough to warrant the mind-numbing exercise of understanding multiprocessor cache protocols.

An odd twist that I have seen used to reduce bus traffic in multiple-processor systems involves slow caches built of dynamic RAMs. The larger the cache, the lower the miss rate. If traffic is your problem, then the miss rate is your solution. DRAMs give you four times as much memory per dollar as do SRAMs, but they are considerably slower. The designer of this cache was much, much more interested in bus traffic than in the performance of individual processors and thus chose to make a much larger, albeit slower DRAM cache rather than to make a faster SRAM cache.

1.9 REDUCING POWER CONSUMPTION

A new reason to use cache memories has recently surfaced and is the subject of some debate. Some folks contend that cache memories can be added to a low-power system (like a notebook computer), causing the overall power consumption to go down.

Opponents to this viewpoint immediately point out the high power consumption of the fast memories and logic usually used to implement cache designs. How can these be added to a system without increasing power consumption? In addition, they often quote statistics showing that the major power hogs of a notebook computer are the backlight on the flat-panel display and the hard disk drive, which together usually consume over two-thirds of the system's power.

Proponents argue that the cache cuts down on main memory cycles, often allowing them to be nearly equivalent in number to the minimum number of allowable refresh cycles. This in itself reduces the DRAMs' power

consumption drastically in comparison to the power consumed when they are accessed for every memory cycle, nearly offsetting the power dissipation of an equal number of SRAMs. Further, if the number of DRAMs which are accessed at any one time is larger than the number of SRAMs which are turned on for the cache, the power consumed will necessarily be lower when the processor operates from cache than when it operates from main memory. Lastly, the fewer wait states incurred by the CPU between its shutdown periods, the lower the CPU's overall power consumption will be.

The jury is not yet in on this one. My own suspicion is that the advantage of a cache as a power-saving device is small if it even exists, and is highly software dependent. Imagine a system manufacturer boasting that the battery life of a certain model was typically three hours running Lotus 1-2-3 but up to five hours running Microsoft Word!

In certain hand-held systems, the processor's internal cache can be used in conjunction with hand-tailored software to reduce power consumption. First of all, software is chopped up to fit into the cache in modules, so that an entire module will reside in the cache when it is needed. When the module is being loaded into the cache, the main memory (sometimes a ROM or other nonvolatile memory) runs at normal power. When the module is being executed within the cache, the main memory need not be powered up, so it is either put into a reduced-power mode, or it is turned off completely. In this way, the power consumption can be reduced through the use of the cache, and if the software is properly tuned to minimize cache accesses, power consumption can be minimized.

1.10 AN EXAMPLE CACHE

Figure 1.15 shows an example cache just to assure you that cache designs are not that difficult. The cache is an old one designed for a 68020, a 32-bit Motorola processor with no on-board cache. The cache is designed using discrete logic rather than programmable logic, which is an advantage to us since it makes the design easy to understand.

Figure 1.15a shows two memory arrays, the cache-tag RAM at the left, and the cache data RAM on the right. With more modern parts the entire cache data RAM could be implemented using only one-quarter of a single 32K×32 SRAM chip. The cache-tag RAM has been implemented with integrated resettable cache-tag SRAM chips, devices which will be explained in detail in Chapter 2. Fundamentally there is little difference between these ICs and standard 8K×8s.

Moving to Figure 1.15b, we see all of the logic required to implement the entire cache. This amounts to six storage elements, 16 two-input gates, and

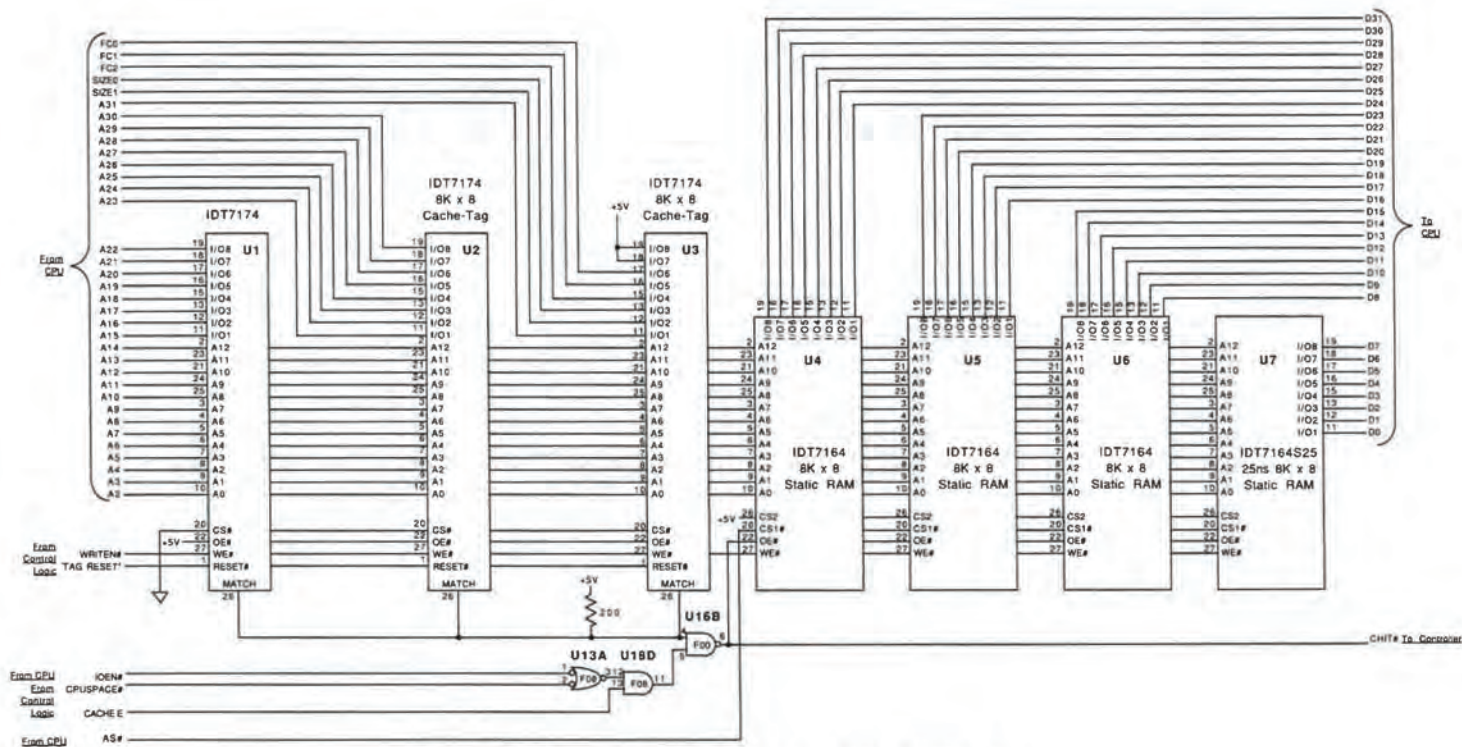


Figure 1.15a. Cache-tag and cache data RAMs for the 68020 cache.

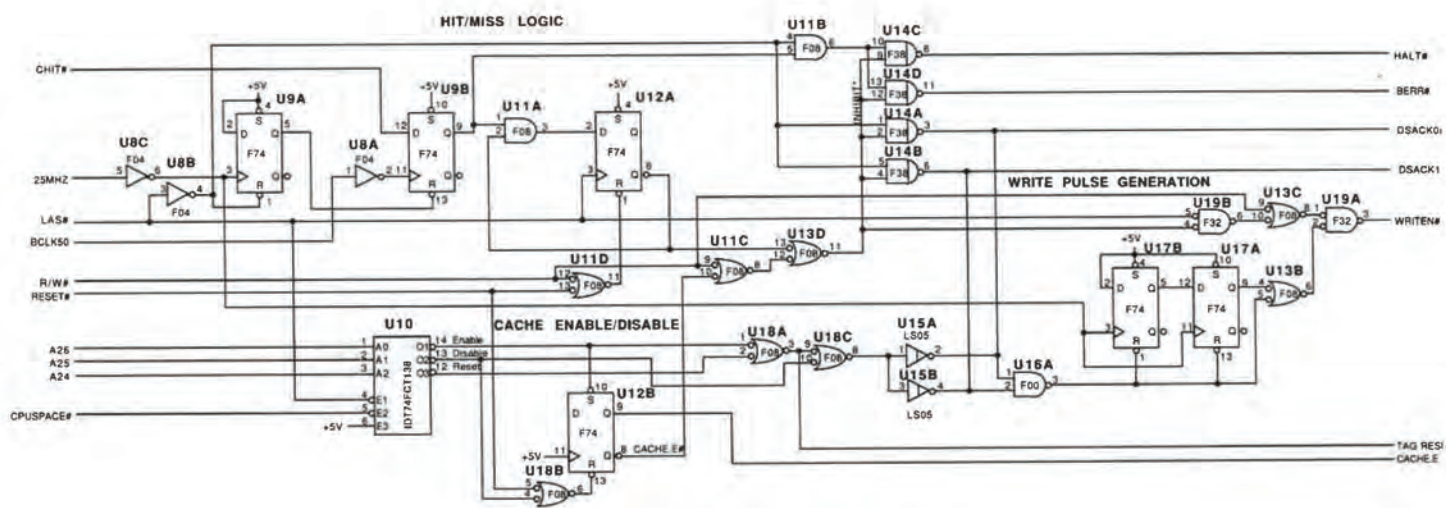


Figure 1.15b. Logic of cache controller for 68020 cache.

five inverters. The address decoder U10 is simply used to map addresses into software cache reset commands. The name of the cache's policies is actually far more intimidating than this handful of logic would lead you to believe. This cache is a direct-mapped write-through unified logical cache. As you can see, there is more to the name than there is to the logic.

Still, don't be led astray by the logic's simplicity! There are many approaches which at first seem quite simple, and require very little logic to implement, but they involve so many interdependencies with the rest of the system that the problem soon dissolves into a mind-numbing exercise, and many designers fail to get their cache designs right the first time. Chips & Technologies was hurt badly by delays in the M/PAX chipset stemming from cache policies, and Intel had shipped millions of Pentium processors before they had all the bugs worked out of their multiprocessing cache protocol.

CHAPTER 2

HOW ARE CACHES DESIGNED?

In this chapter we will examine some of the methodology or "tricks of the trade" used in cache design. Although no especially complex design techniques are used, a substantial amount of thought has been devoted to unmasking the obvious, and the typical designer will end up having to make mental notes about a lot of cache-specific rules of thumb in order to properly perform cache designs.

2.1 THE CPU-TO-MAIN-MEMORY INTERFACE

A CPU cache actually is inserted between the CPU and the system's main memory, as discussed in Chapter 1. Because of this, the interface between the CPU and main memory must be matched by the cache in order for the cache to deceive the CPU into believing that the cache access is actually a main memory access and nothing more.

2.1.1 Why Main Memory Is Too Slow for Modern CPUs

It seems that all portions of the computer system are speeding up simultaneously, so the system designer might very well come to the conclusion that

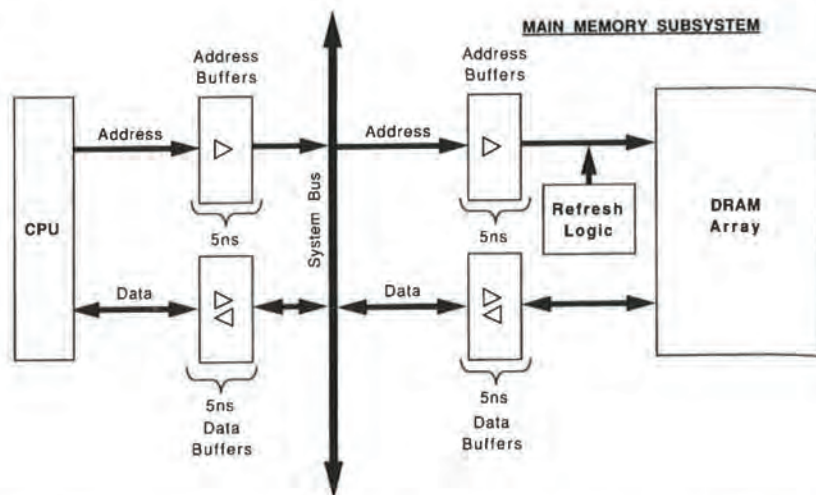


Figure 2.1. A typical computing system, illustrating the delay paths between the CPU and main memory. In this example system, there is a backplane between the main memory and the CPU.

any increase in CPU clock speed will be matched by an increase in memory speed adequate enough that the system throughput can be increased by 50%, for example, simply by combining a 50% faster main memory and a 50% faster CPU. The same upgrades in semiconductor processing technology which made the processor capable of handling a 50% faster clock rate should be able to produce a like increase in memory speed, so there should be no problem.

The error in this line of thought is that it oversimplifies the problem. Quite obviously, from Figure 2.1, we can see that there are logic elements which isolate the CPU from main memory. Examples of such devices would be the buffers required to isolate the CPU from main memory during DMA accesses or DRAM refresh cycles. Buffers are also necessary in most systems simply because CPUs are not designed to support heavy loading on their output pins. Microprocessor timing and output current are usually specified into loads which are far smaller than the collective input capacitance of ten or more other ICs, so the address output buffers become a necessity.

Now each of these buffers adds a delay, which does not scale very well with the speed increases of the memories or CPUs of a similar semiconductor process technology. A 5ns gate in one technology would do well to lose 0.5 to 1ns by a change to the next generation process. Part of the reason for

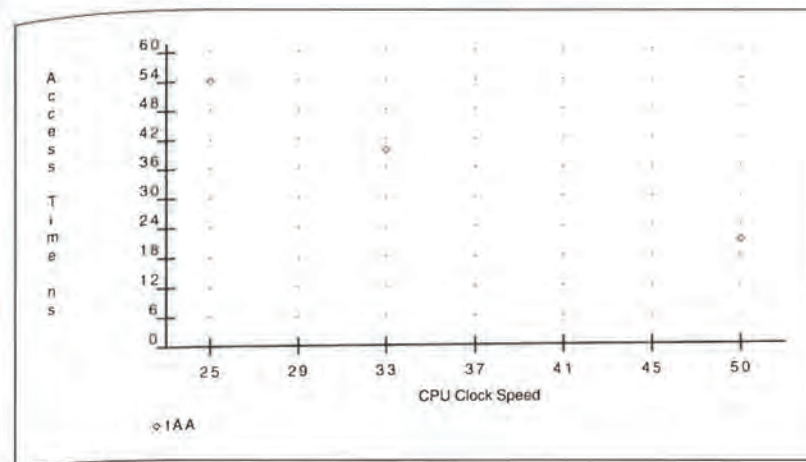


Figure 2.2. Main memory address access times as a function of CPU clock frequency, as measured at the CPU's address and data pins. Data shown is for the 386 microprocessor.

this is that it takes a finite amount of time just to move an output pin up and down. This action takes around 3ns for TTL I/O levels. As the CPU gets faster, buffer delays consume a larger and larger percentage of the main memory's allowed cycle time.

Other timing which does not scale to process and further compounds the problem consists of specifications like data set-up time and data hold time. Although microprocessor manufacturers will often slightly reduce their products' set-up times for faster speed grades, the reduction is anything but proportional to the change in clock frequency and often runs into a brick wall at the faster CPU clock rates, showing no change between the fastest and the next lower speed grades.

Figure 2.2 illustrates the squeeze put onto main memory speeds by these less flexible parameters. Taken to the absurd, a fast enough processor would require a main memory with a negative access time in order for that processor to run at its fastest throughput. So far, dynamic RAMs have not shown any capability to predict the future.

2.1.2 How the CPU Handles System Bus Delays

Present-day microprocessors nearly all follow a similar bus protocol. **Split transactions** are only used in the most advanced processor interfaces. In a split transaction, the processor issues a command, then removes itself from

the bus until a memory or I/O device issues a response. In systems not using a split-transaction protocol, once the address is issued, the processor stays in control of the bus and waits until the transaction is terminated. Should another bus master override this transaction, then the entire process is restarted once the processor regains control of the bus.

Microprocessors use an input signal to determine whether or not the data returned from main memory has settled and is valid. The microprocessor is then allowed to terminate the cycle. The name of this signal varies from processor to processor, but is generally referred to as a Ready signal. We will use the global term "Ready" in this book, even with processors which use a different naming convention.

The Ready input is generated by a device whose response time is set to match the access time of main memory or any other bus device over which the processor has control (i.e., I/O devices). In slower systems, the CPU allows ample time for all devices to respond. This would especially be true of processors running at clock frequencies of 1 to 10 MHz. In systems whose bus devices all respond as fast as the processor requires, the Ready input is often hard-wired to an asserted state, indicating that the data will always be ready in time for the CPU's request. Systems which require the CPU to pace itself to the memory or I/O devices might use monostable multivibrators (one-shots), short shift registers, or counters to generate varying length delays to account for different speed memory or I/O devices (see Figure 2.3). The delay times of these timing generators are triggered by the command from the processor to the memory or I/O device which tells that device to initiate a bus transaction. The timing generator is set to match or exceed the time required by the memory or I/O device to allow the bus transaction to be completed. Highly tuned systems will use one timing generator for each different delay in the system. Less speed-critical applications will use a single timing generator for all response times, and simply set the delay to match the slowest device on the bus.

Should the Ready input respond, when the CPU samples it, that the bus data is not ready, the CPU will wait for a clock cycle, then sample the signal again. All processor output signals are static from the time that the Ready input is expected until it is asserted. Cache designers realize that their systems cannot realistically run without wait states, so they attempt to bring the number of wait states to an absolute minimum for a given processor clock frequency.

2.1.3 The Cache's Interface to the CPU

The cache has to be in control of the CPU. If the cache contains a valid copy of the data requested by the CPU during a CPU read cycle, then the cache

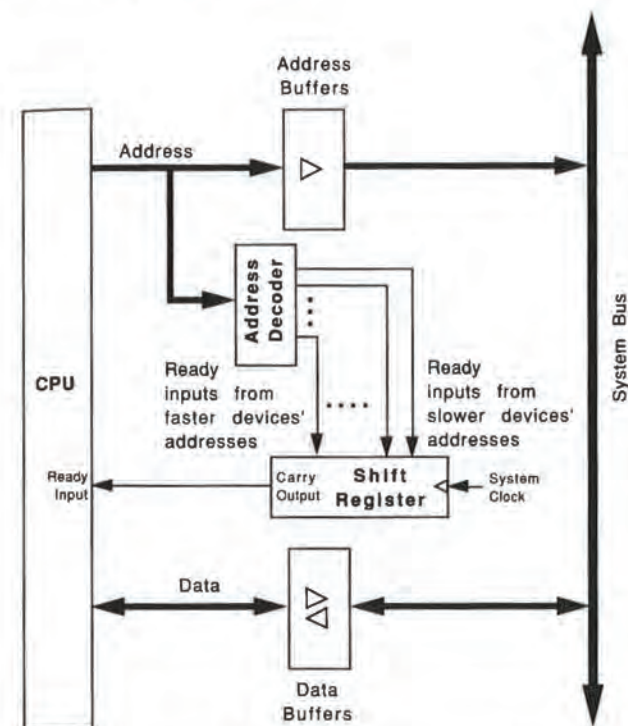


Figure 2.3. Wait state generation in a typical microprocessor system. Decoded address outputs feed the parallel inputs of a shift register. A Ready signal appears at the CPU input pin after an appropriate number of cycles.

allows the CPU to progress at the cache's speed. If the cache does not contain a copy, then the cache initiates a main memory read cycle, copies the data supplied by the main memory (which the main memory has indicated is valid via a Ready output from the main memory to the CPU), and permits the CPU to continue. Data fed to the CPU is routed by the cache controller from either the cache, in the case of a cache hit, or the buffers which isolate the main memory from the cache/CPU bus, in the case of a cache miss.

The cache must intercept all of the CPU's signals, both inputs and outputs, and determine whether these signals are to be routed to/from main memory or whether they should stay local to the cache. In a way, the cache is an insulating layer between the CPU and the outside world.

There are four basic sorts of cache/CPU interaction, all of which are controlled by the cache controller: read hit, read miss, write hit, and write miss. These are illustrated in Figure 2.4. In a cache read hit, the buffers are turned off to isolate the CPU/cache subsystem from the rest of the computer, and the cache controller generates the Ready signal to the CPU. No interaction is required between the CPU/cache subsystem and the main memory bus, and certain systems take advantage of this by allowing DMA or other devices to control the main memory while the CPU operates out of the cache.

Occasionally, a designer will allow the CPU address to propagate through the address buffers to the bus at the beginning of every cycle, whether the cycle turns out to be a read hit or a read miss. The cache is designed in such a way as to initiate a main memory access at the same time as the cache access, rather than to wait for a cache miss before commencing the main memory access. This approach can shorten the main memory access time in the event of a cache miss and will serve to improve the operation of single-processor, single-tasking systems, but can work to the detriment of multi-tasking systems and multiprocessing systems, since the cache will allow the CPU to waste considerable amounts of main memory bus bandwidth. Such caches can be referred to as **look aside** designs. Since the processor has the option of going to either the cache or main memory at all times to request data, the look aside cache can be an add-on to the computer system.

Caches which place themselves right in the middle of things and intervene in all CPU to main memory transactions are called **look through** or **in-line** caches. A look through cache first looks in the cache for the location to be accessed. Only after a miss is detected can the cache trigger the start of a main memory cycle. There is a plus and a minus to this. The plus is that a look through cache dramatically reduces main memory bus traffic. This is really important in systems where other processors must access the main memory bus. The minus is that all main memory accesses due to cache miss cycles are now lengthened by the time it takes to determine that a cache miss has occurred. Fortunately, this doesn't happen very often, and since the cache is designed to respond as quickly as possible, the delay is not large. Still, this is enough of a consideration to have earned its own buzzword, which is **lookup penalty**.

The read miss cycle, where the contents of the directory are found not to match the address being output by the CPU, is handled in just the opposite manner. The cache outputs are turned off, and the address and data buffers are turned on, allowing main memory data to be fed to the CPU. The cache controller's Ready signal is also turned off, and the system's Ready is directed to the CPU instead. As the CPU reads the main memory data, the cache controller commands the cache data RAM to grab a copy, and commands the cache-tag RAM to copy the tag bits of the CPU's address outputs,

thus overwriting any previously existing line of cache data and directory address which previously resided at the same set address. This "read from main memory/write to cache" cycle is called a **line update, update, copy-in, fetch, line fill, or line replacement** cycle. I'll try to stick with just "line fill" to keep the confusion to a minimum. (Although we will not define the word "line" in specific detail until Section 2.2.5, for now we will use the loose definition that an entry in a cache corresponding to a single set address is a cache line.) The new data is now available from the cache, where it most probably will be accessed several more times during subsequent cache hit cycles before it, too, is overwritten. This paragraph describes the major, major concept behind cache memory design, so you don't want to skip ahead if you feel like it didn't soak in.

Just as there are compulsory misses, as defined in Section 1.6, where missed data could not have been expected to have previously resided within the cache, a **compulsory line fill** is a line fill which could not have been avoided by using a better cache strategy.

Write hit cycles are handled in either of two ways. These are complex enough to warrant their own section, so a full explanation of these two strategies is given in Section 2.2.4. In the example in Figure 2.4c, the matching cache line and main memory are both updated with the new data.

Write miss cycles are a free-for-all and are handled in any of a number of different ways. In caches like the one in Figure 2.4d, write misses are ignored by the cache and are passed directly through to the main memory. In others, the write data is used to overwrite a line in cache as it is written to main memory. A third way of handling write misses is to overwrite a line in cache and to disallow the write cycle from being copied into main memory. In Section 2.2.4, I will try to justify these and other approaches and show which fit into which write strategy.

Designers who have wrestled with problems of ill-defined restart sequences have probably already wondered how a cache comes up after a cold start. Obviously, all the data in the cache data RAM, as well as all of the addresses in the cache's directory, are completely random. How do designers avoid the possibility that this random data will be mistaken for good data and will generate false cache hit cycles? There are actually two easy methods of handling this situation. The least expensive method is to disallow the cache from supplying data to the processor until a bootstrap routine (usually in programmable read-only memory [PROM]) has had an opportunity to overwrite all cache data and tag addresses. A one-bit, hardware-resettable flag feeds into the cache controller, signifying that the entire cache is to be ignored. The bootstrap routine reads a memory block of a size equivalent to the size of the cache, from the bootstrap PROM itself, and each of these read cycles will be treated by the cache controller as a read miss cycle, so the

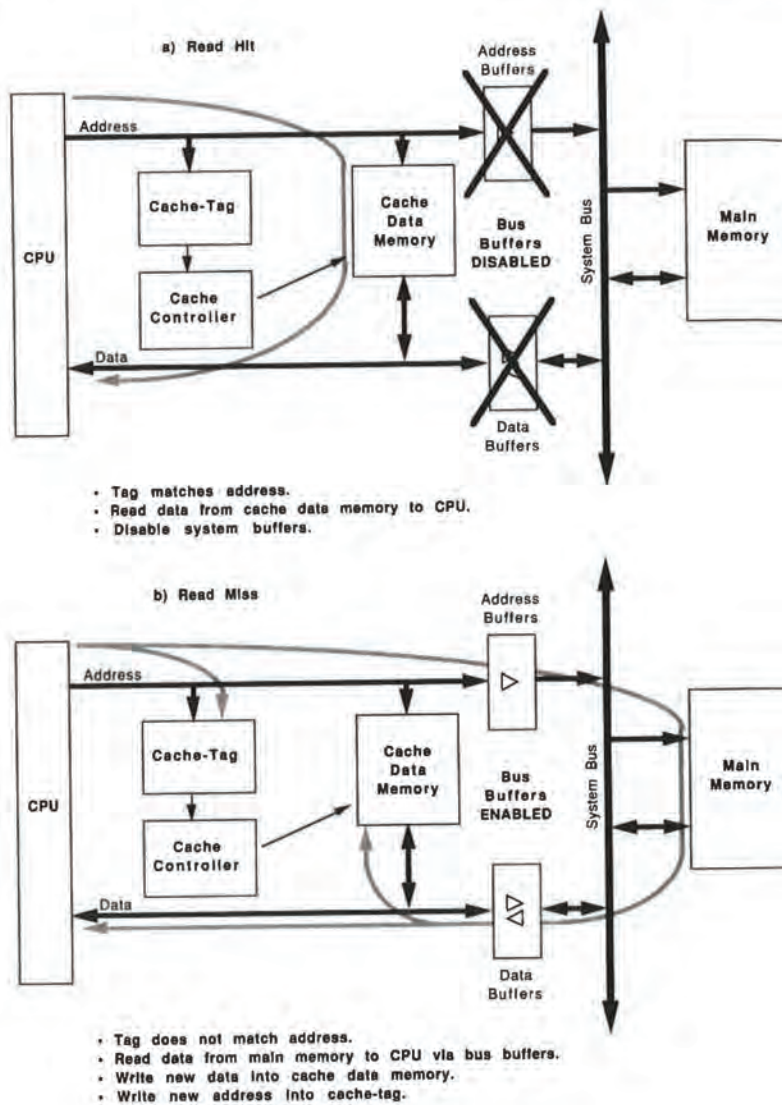


Figure 2.4. Cache cycles: (a) Read hit. (b) Read miss. (c) Write hit. (d) Write miss.

cache controller will write a new copy of the bootstrap into the cache. Once the entire cache has been overwritten with this new data, the one-bit flag is set via software, and the cache controller now allows the cache to supply zero-wait data to the processor.

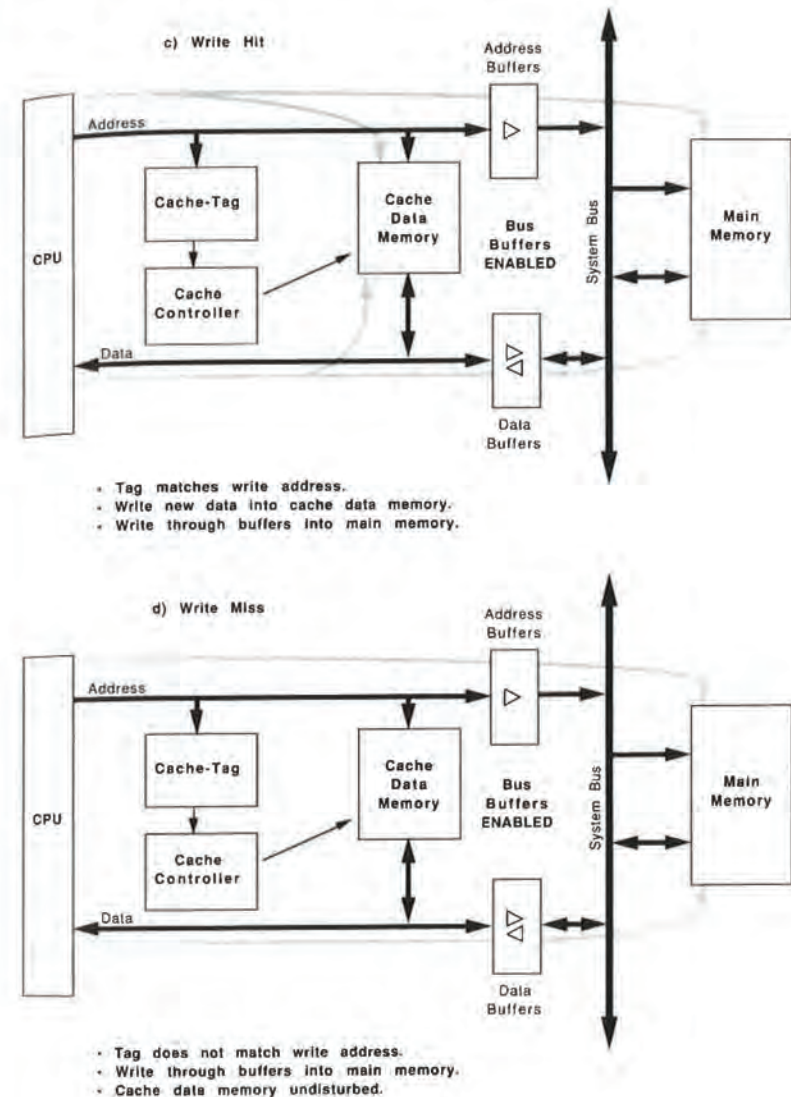


Figure 2.4. (continued)

A more common method of signifying the validity of the contents of the cache is to provide a **Valid** state indicator for every line in the cache. Although validity is usually indicated by a dedicated **Valid bit**, some of the more sophisticated coherency protocols detailed in Chapter 4 allow four or

SET ADDRESS	VALID BIT	CACHE-TAG RAM	CACHE DATA RAM
0	1	Address Tag 0	Valid Data
1	0	N/A	Garbage
2	1	Address Tag 2	Valid Data
3	1	Address Tag 3	Valid Data
4	1	Address Tag 4	Valid Data
5	0	N/A	Garbage
6	1	Address Tag 6	Valid Data
7	1	Address Tag 7	Valid Data

Figure 2.5. Typical contents of a cache data RAM and its matching cache-tag RAM and Valid bit RAMs.

five states to exist for every cache line, and the Valid state is one of many encoded onto two or three state bits. Figure 2.5 shows the memory organization of a cache-tag RAM and the corresponding cache data RAM in one type of system using Valid bits. Alternatives to this particular layout will be detailed in Section 2.2.5.

The problem still exists of how to assure that every line's Valid bit is reset to an Invalid state after a cold start. There are two easy answers to this problem. First, the cache with one Valid bit per line could replicate the cache Invalid flag just described, where the cache would be turned off until all Valid bits had an opportunity to be validated or invalidated. This may seem not to make sense, but we will see in Chapter 4 that such a scheme can prove beneficial if the Valid state is part of a more complex line status scheme. Second, all Valid bits can be reset after a cold start. There are three simple ways to perform this. One is to purchase specialty static RAMs which have a reset feature built in. The application in Figure 1.15 uses this method. Another method is to use a standard static RAM to hold the Valid bits and to use a small state machine to run through all addresses, writing an Invalid state to the Valid bits of all addresses after a system reset, while holding the processor in an idle state. This type of reset sequence is performed internally by certain CPUs. The last method is very similar to the method first described, where all cache locations are set to Valid states before the cache is enabled, but, with this last method, the entire cache is written to Invalid states during the cache's disabled state. The difference between this method and the one described at the beginning of this paragraph is negligible and would probably depend only on whether the bootstrap program was or was not cacheable.

There is an extremely simple method of generating Valid bits which is used in many commercially available cache designs. Cache-tag RAMs are constructed using a comparator and a standard static RAM. Certain inte-

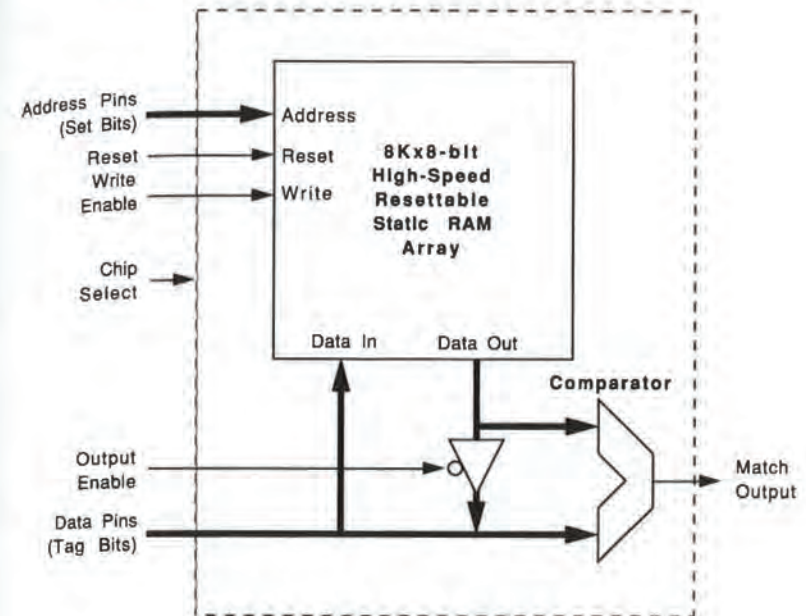


Figure 2.6. Block diagram of the internal workings of a typical cache-tag RAM.

grated cache-tag RAMs contain a comparator and a resettable static RAM (Figure 2.6). The reset pin on these RAMs clears every bit in the entire memory array to zeros. The trick is to tie any spare data bit inputs on these devices to a logic high level, so that, on a compare cycle, these inputs will either be compared to a zero, indicating that no tag address has been written into the RAM since it was reset, or a one, which means that a write cycle has indeed occurred, and the high level on these data inputs has been written into the selected RAM set address.

2.2 CHOOSING CACHE POLICIES

There are numerous cache policies, only the most common of which will be dealt with in this book. Cache policies are the rules of operation of the cache. Which cycles will be read from the cache instead of main memory? Where does the cache fit into the system? How associative is the cache? What happens during write cycles? All of these questions must be answered for all cases before the design is even started.

Cache policies are chosen for a single motive. The designer wants to get the most performance for the lowest cost. Two variables play into this equation: 1) Which is more important, to save engineering time or to save overall system parts cost? 2) Is the cache to be integrated or constructed from discrete components?

After reading this chapter, you should have gotten the impression that certain cache designs are pretty trivial and can be designed in little time. On the other hand, the last drop of performance can be eked out of a cache if the designer spends a near-infinite amount of time on its development. Another point I will be making where applicable is that certain architectures don't combine well with standard RAM organizations. In a few cases, this problem has been addressed by static RAM manufacturers, but, for the most part, more sophisticated architectures can best be handled in a dedicated single-chip cache design. This avenue has been explored by the designers of the internal caches on processor chips.

One problem in trying to produce the optimum cache design is the exercise of optimizing an equation which has about 20 variables. We will explore these variables in this chapter.

Cache policies may be chosen in a number of ways, depending on the generality of the system and the amount of resources available to improve the design. In the best case, the hardware and software of the system are being designed together, and the software contains few enough cases that the hardware can be optimized to a very good degree, based on a large amount of empirical research on the effects of different cache policies on the software's performance. In the worst case, the hardware designer is asked to design a cache without any knowledge of the software to run on the system, no empirical data, nor any chance to develop any, and with very little education about the trade-offs of various cache policies. Real-life scenarios tend to follow the financial strength of the company producing the system and the openness of the system. Closed systems designed within profitable firms will usually follow the best case scenario, while less profitable firms designing systems to compete in an open systems market will have to suffer with a scenario very similar to the worst case.

The following examples should help those designers who cannot afford much in the way of empirical studies and will serve to illustrate the trade-offs of various policies to those just learning the art of cache design.

2.2.1 Logical vs. Physical

In systems using virtual addressing, caches can reside either upstream (on the CPU side) or downstream (on the main memory side) of the processor's

memory management unit (MMU). Both configurations are illustrated in Figure 2.7. The addresses upstream of the MMU are all logical or virtual addresses, and the addresses downstream are physical addresses. If the cache is placed upstream of the MMU, it is called a **logical cache** or **virtual cache**, and if the cache is downstream of the MMU, it is called a **physical cache**. Either placement has pros and cons.

Since the logical cache is upstream of a delay-causing device (the MMU), a logical design can run faster than a physical design. A logical cache for the Motorola 68020 CPU was shown in Figure 1.15. The 68020 used a separate chip for its MMU, and Figure 1.15's cache was placed on the processor side of the MMU chip.

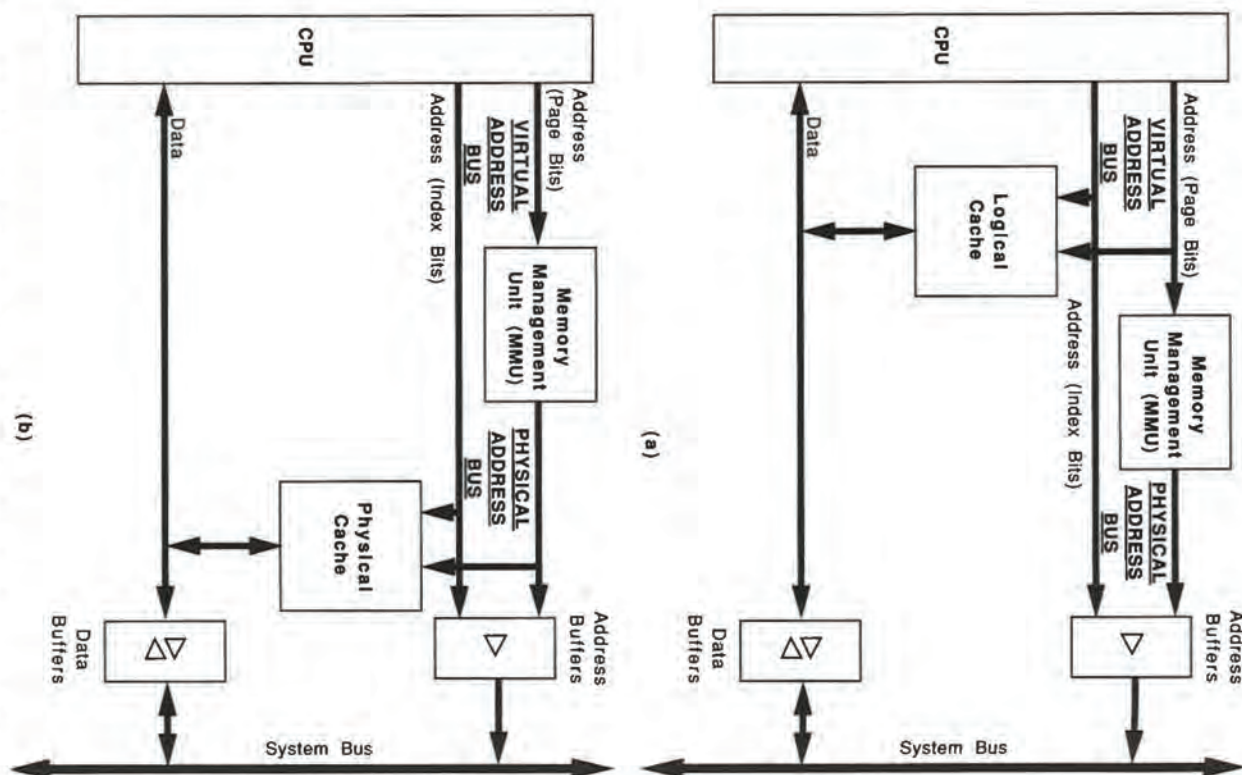
Problems arise in logical caches with a phenomenon called **address aliasing**, simply **aliasing**, or **synonyms**. In virtual memory systems, the same physical address may be mapped to two or more completely different logical addresses. Say that both of these logical addresses were cached and that a write cycle occurred to one of them. The cache would update on cached copy of the physical address as well as the main memory itself (policies permitting), but the cached copy of the other logical address would remain unchanged and would subsequently contain erroneous data. Several solutions for this problem exist, and one will be documented thoroughly in Section 4.2.2, but these solutions are not trivial.

2.2.2 Associativity

In Section 1.5, we saw the difference between a set-associative and a fully associative cache. In a fully associative cache, there was one comparator per line in the directory, and all lines were examined at once for a match. In the set-associative cache, only one comparator was used, and the directory was divided into set bits and tag bits, the set bits determining exactly where within the cache a piece of data would reside. Since there was such a limitation, no two cache lines in the set-associative design could use the same lower address or set bits. This could lead to thrashing, a process where two addresses were continually displacing each other in the cache at the expense of throughput. You may recall that thrashing occurs not only because of instructions stepping on each other's toes, but also when something gets pushed into a stack location, or data is read or written, and the set bits happen to match the set bits of a valid cached location.

A middle ground lies between the two extremes of a fully associative design and a set-associative design. By adding comparators to a simple set-associative cache design, higher degrees of associativity can be achieved. By adding a single degree of **associativity** to the design of a cache, there will be

Figure 2.7. Logical and physical caches. (a) A logical cache is found upstream of the memory management unit. (b) Physical caches are on the physical address bus, downstream of the MMU.



WAY "A"				WAY "B"			
SET ADDRESS	VALID BIT A	CACHE-TAG RAM A	CACHE DATA RAM A	VALID BIT B	CACHE-TAG RAM B	CACHE DATA RAM B	
0	1	Address Tag 0	Valid Data	1	Address Tag 0	Valid Data	
1	0	N/A	Garbage	1	Address Tag 1	Valid Data	
2	1	Address Tag 2	Valid Data	1	Address Tag 2	Valid Data	
3	1	Address Tag 3	Valid Data	0	N/A	Garbage	
4	1	Address Tag 4	Valid Data	1	Address Tag 4	Valid Data	
5	0	N/A	Garbage	0	N/A	Garbage	
6	1	Address Tag 6	Valid Data	0	N/A	Garbage	
7	1	Address Tag 7	Valid Data	1	Address Tag 7	Valid Data	

Figure 2.8. Contents of the cache-tag RAM and matching cache data entries in the two Ways of a two-Way set-associative cache. Note that these are simply two implementations of the simple direct-mapped cache of Figure 2.5.

another location into which main memory addresses with shared set bits can be mapped. If two locations would ordinarily thrash with each other in a single-comparator design, they will no longer need to thrash in a design with a second degree of associativity. The cache of Section 1.6 used a single comparator, a single cache-tag RAM, and a single cache data RAM, giving it a single degree of associativity. This is the simplest, most common way to design a cache and is called a **direct-mapped** implementation.

In smaller caches, a significant hit rate improvement can be realized through the implementation of greater degrees of associativity or more **Ways** (sometimes called **banks**) in the cache. Each Way in a cache is pretty much another cache which is a carbon copy of the others (Figure 2.8). Where a direct-mapped cache consists of a cache data RAM, a cache-tag RAM (built out of a RAM and a comparator), a cache controller, and isolation buffers, an **N-Way set-associative** cache uses N cache data RAMs and N cache-tag RAMs (built out of N RAMs and N comparators), a cache controller, and isolation buffers. In a set-associative cache, a main memory address can be mapped into as many different places in the cache as there are Ways. (One-Way caches are always called direct-mapped caches.)

Here's an example: A 16K-byte, direct-mapped cache for a 16-bit microprocessor could be designed using two $8K \times 8$ SRAMs for the cache data RAM, and possibly one more $8K \times 8$ -bit SRAM and comparator for the cache-tag RAM (depending on the number of address bits used in the system, 20 for this example). To build a two-Way, 16K-byte cache for the same system, four $4K \times 8$ -bit data RAMs would be needed, and two $4K \times 9$ -bit RAMs and comparators would be required to implement the tag.

The reason that there are more tag bits in the two-Way design is that each cache acts independently, so they must each behave as if they were the only 8K-byte cache in the system. You can piece together a trend from this simple example. For a given cache size, increasing the associativity decreases the depth of the cache, thus decreasing the number of set bits. The displaced bits must in turn be made into new tag bits, so the cache-tag RAMs and comparators must be made wider. Given the number of address bits we have assumed (20), a 16K-byte cache would require the use of 13 set bits, seven tag bits, and a Valid bit, whereas an 8K-byte cache would need to have 12 set bits, eight tag bits, plus a Valid bit. In a similar manner, a four-Way, 16K-byte cache would require 11 set bits and nine tag bits. If you take this to the extreme, you come right back around to the content addressable memory of Figure 1.7. Each cache data RAM has been reduced down to a single location, and there are 2^N cache data RAMs in the fully associative cache. Likewise, there are 2^N separate cache-tag RAMs (and comparators), each of which stores only a single address. The number of set bits has been reduced to zero, and the address now consists entirely of tag bits. In a fully associative cache, every cache line is a different Way.

In an N -Way cache, all of the set bits are sent simultaneously to every Way's cache-tag RAMs and cache data RAMs, so the final decision of which Way to use comes down to enabling the appropriate Way's cache data RAM's data output pins when that Way's cache-tag RAM indicates that a hit has occurred.

One unfortunate casualty of the conflicts of terminology in this field is the word **page**. To an MMU, a page is the section of main memory referred to by the translated bits. This is the terminology we used in the preceding section. To certain cache designers, the word "page" refers to a Way in an associative cache, and to others, the word "page" means a single entry in the cache. In this book, page is only used in the MMU sense.

Back in Section 1.4, I briefly mentioned Intel's analogy that a cache was like a refrigerator which could be much more readily accessed from home than could a grocery store for a limited subset of the contents of the grocery store. In the same analogy, Intel cleverly likens associativity to shelves in the refrigerator, which neither increase nor decrease the space in the refrigerator itself, but reduce the competition for places to put the contents of the refrigerator, thus increasing the likelihood that you can get more inside. If we look back to the filing system analogy, higher associativity could be likened to adding file drawers to the desk.

As the cache's size increases, the hit rate improvement realized by increased associativity rapidly decreases (see Figure 2.9). In this example, once a cache transcends a certain size (around 4K bytes), doubling the size

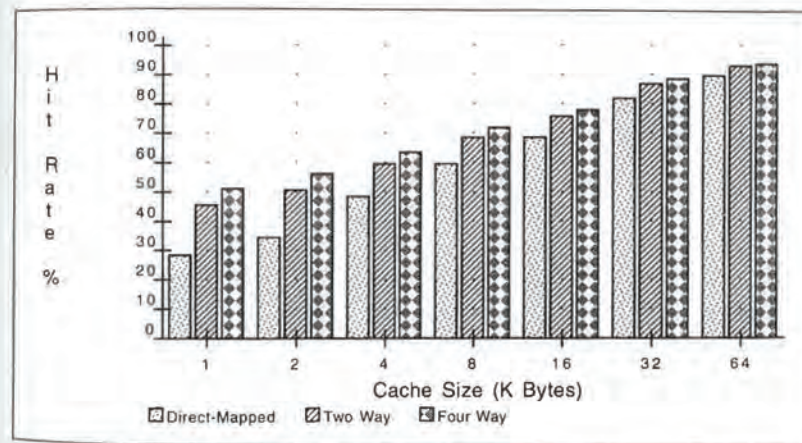


Figure 2.9. One set of statistics showing the relative performance of caches of varying sizes and associativities. Note that, for this example, the hit rate for a direct-mapped, 8K-byte cache is equivalent to that of a 4K-byte, two-Way cache. Above these sizes, a direct-mapped cache of twice the size outperforms all increased associativities.

of the cache improves the hit rate more than does increasing the associativity of a given size cache. Although a slightly better hit rate may be obtained for a 32K-byte cache through the use of higher degrees of associativity, the component count needed to implement such a cache would increase in proportion to the number of degrees of associativity and probably not be justifiable against going to a 64K-byte implementation. This is because each Way of a cache needs a separate cache-tag RAM as well as a separate cache data RAM. This is a key point in discrete cache designs, but is far less of an issue in integrated designs, since integrated caches aren't concerned with chip count, but are rather concerned with the maximum hit rate attainable within a certain die size.

You may want to keep in mind some commonly quoted statistics, however far they may actually be from the way your own system performs. The first is that a doubling in associativity decreases the miss rate by about 20%. This comes from research done by M. D. Hill in his thesis, "Aspects of Cache Memory and Instruction Buffer Performance," done in 1987 at the University of California, Berkeley. The second rule of thumb is that doubling the size of a cache decreases the miss rate by about 69%. This is from research done by Anant Agarwal at Stanford University. Looking at Figure 2.9, or at any similar chart, it is difficult to rationalize these numbers. Still, it does indeed appear that, above certain cache sizes, it is much better to increase the size of the cache rather than the associativity when the opportunity presents itself. As always, I will simply say that your cache must be designed for your system and your software. You just can't take advantage of anybody else's statistics for anything more than simple guidance.

There is another key difficulty in designing discrete multiway caches. Most caches are designed to support the highest possible processor clock rates, and consequently they tax the capabilities of even the fastest static RAMs and cache-tag RAMs. In a direct-mapped cache, the critical path for the cache-tag RAM runs through the static RAM and the address comparator, to the cache control logic, and eventually back to the processor's Ready input pin (Figure 2.10). The cache data RAM in direct-mapped designs is usually enabled onto the processor's data input pins as soon as the processor has indicated that a read cycle has begun, and is only disabled upon the detection of a cache miss, so the timing on the data RAM's output enable pin is not tight. In a discrete implementation of a multiway cache (Figure 2.11), neither (or none) of the different Ways' data RAM outputs can be enabled until *after* a cache hit has been detected. This means that the critical timing path now flows through the cache-tag RAM, then the comparator and cache controller, as before, and then through the cache data RAM's output enable pin, adding the "output enable to valid" data delay to the critical timing path. This is usually significant (around 5ns at the writing of

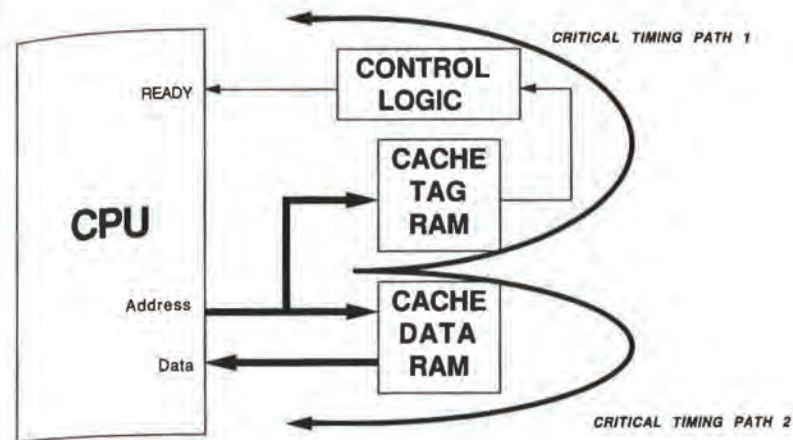


Figure 2.10. The critical timing paths in a direct-mapped cache are from CPU address output to CPU data input, through the cache data RAM alone, and from CPU address out to CPU Ready in, which runs through the cache-tag RAM, the comparator, and the cache controller logic. The Ready path is usually the trickiest from a timing standpoint.

this book) and steers designers away from multiway implementations of discrete caches at the onset of the design.

Some mention needs to be made about the methods used to select which Way in a multiway cache should be replaced when a line needs to be updated. This policy is called the **replacement algorithm**, although some simply call it **placement**. A replacement algorithm selects which Way is to be updated when a new line is to be brought into the cache. (Recall that any main memory address can be put into as many locations as there are Ways in the cache.) Ideally, any stale piece of cached data which is no longer needed by the processor would be chosen to be overwritten. Some cache controllers watch over the accesses into the cache, and categorize the order in which each Way was accessed, making a note of which Way's line was accessed least recently. This is called a **least recently used (LRU)** algorithm. LRU statistics are maintained separately for each cache line.

In a two-Way system, a single bit per line can be used to implement an LRU (see Figure 2.12). When a cache hit occurs, the Way which was hit makes sure that the LRU bit is written to point to the other Way. When a line is to be replaced, that line's LRU pointer already points to the Way where the replacement should happen, and the pointer is rewritten to point at the other Way. The power-up state of these bits is insignificant, since everything in the cache is unusable.

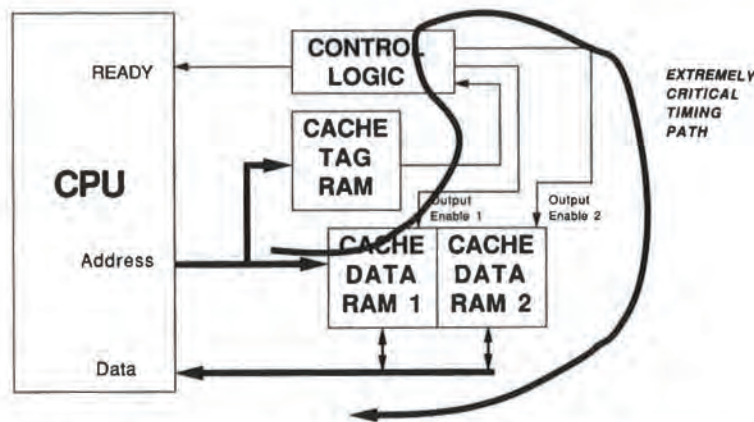


Figure 2.11. In a multiway cache, the outputs of all cache data RAMs are disabled until a hit has been detected. The most critical timing path then runs from the CPU, through the cache-tag RAM and controller logic, then through the output enable path of the cache data RAM before data can be returned to the CPU.

One interesting point is that true LRU replacement can quickly gobble up lots of bits of memory. It can be easily observed that there are $N!$ orders in which N letters of the alphabet can be ordered. A four-Way cache must have five LRU bits for each line to represent the 24 ($4!$) following possible states of use of the cache contents (order in which Ways A, B, C, and D were used):

ABCD	ABDC	ACBD	ADBC	ACDB	ADCB
BACD	BADC	CABD	DABC	CADB	DACB
BCAD	BDAC	CBAD	DBAC	CDAB	DCAB
BCDA	BDCA	CBDA	DBCA	CDBA	DCBA

since these 24 states require five bits ($2^5 = 32 > 24$) to encode.

Similarly, an eight-Way cache's LRU would require adequate bits to represent the $8!$ states of the use of the eight Ways of the cache. This amounts to 40,320 states, requiring 16 bits of LRU information per cache line. A 16-Way cache would require 45 bits per cache line, and a fully associative cache with 256 lines (which is essentially a 256-Way set-associative memory) would require more LRU bits than my pocket calculator is capable of representing!

Another problem facing designers of true LRU systems is that if the LRU algorithm is to be updated to show the order in which the four last Ways

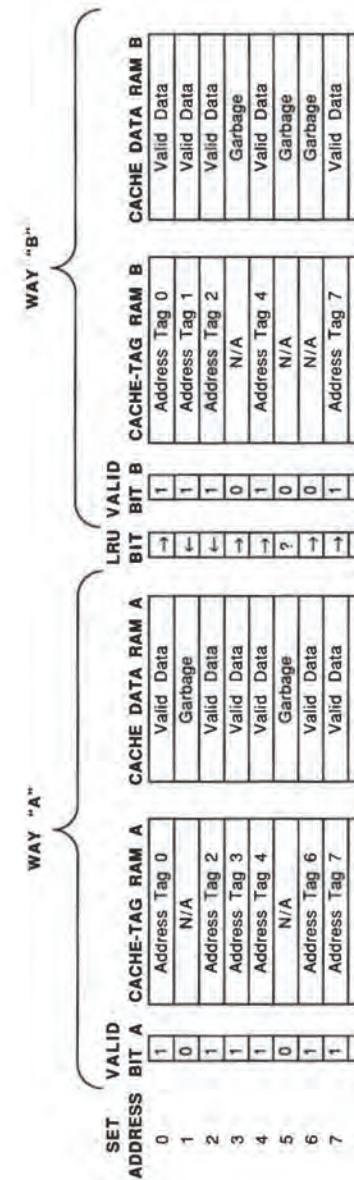


Figure 2.12. The two-Way cache of Figure 2.8 with an additional bit per line to mark which Way was least recently used for any particular line.

were accessed, then both a read cycle and a write cycle must be performed on the LRU bits during each processor cycle. Another way to see it is that if the LRU currently states the order as ABCD, and then D is accessed, the order must be changed to DABC. The ending order is inextricably tied to the initial order. The last sections of this chapter are devoted to timing problems in cache designs, and we will see that performing a simple read cycle is taxing enough in a cache, so a read/write pair could easily become impossible. Many alternatives to a true LRU algorithm have been tried, and the following paragraphs will briefly describe a few of them.

One alternative is that used by Intel in the company's IntelArchitecture microprocessors. Intel calls their method a "pseudo-LRU." Three bits are used per cache line, as shown in Figure 2.13. The top bit in the tree (AB/CD) is set when there is a hit in A or B, and is cleared upon a hit to C or D. Only one of the bits in the second level of the diagram will be set or cleared on a cache hit. If A is hit, the A/B bit will be set, and nothing will happen to the C/D bit. If B is hit, the A/B bit will be cleared, and once again nothing will happen to the C/D bit. The same scenario occurs for the C/D bit upon hits to the C or D Ways. The scheme can be implemented as write only; a simple write cycle can be used to update the pseudo-LRU bits during cache hits, instead of a time-consuming read/modify/write cycle. This can essentially double the speed with which the cache can be accessed over the speed of a true LRU algorithm. The only time that the pseudo-LRU bits must be read is during a line replacement, which will be a slower cycle simply since it involves an off-chip access.

The difference between this method and a true LRU is small and can be simply illustrated. Say that a certain line contains valid data in all Ways, A, B, C, and D. If the CPU executes a loop in which that line is continually ac-

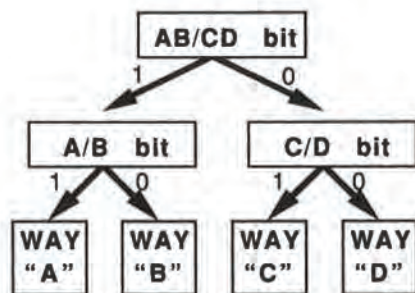


Figure 2.13. The pseudo-LRU scheme used in certain Intel processors to get away from timing and housekeeping difficulties posed by true LRU implementations for highly associative caches.

cessed in the order A, B, C, A, B, C, then the D Way would obviously be the least recently used. After the C access, though, the AB/CD bit would point to AB, and the A/B bit would point to A, forcing the update data to overwrite the A Way, rather than the D Way. The likelihood of this occurring is very software dependent, but is apt to be statistically small, and certainly fades in comparison to the possibility that the entire processor's clock speed might have to be slowed to accommodate a true LRU algorithm. Intel's pseudo-LRU also consumes half the memory bits, thus half the die space of a true LRU, die space which was probably devoted to issues more capable of improving the processor's overall throughput.

One interesting point is that Intel's embodiment of this three-bit scheme requires the use of a dual-port SRAM. No matter which Way is hit, two of the bits will be made to point away from the Way that was most recently used, and the third bit will stay where it was left by a prior operation. Intel's approach to keeping the third bit intact was to perform a read/modify/write on all three bits, which seems to bring the complexity back to near that of a more straightforward scheme. Rather than the write-per-bit memory that one would assume designers would have chosen, a technique which makes each bit behave as if it resides in a different memory chip, each with its own write control, Intel's PC processors read/modify/write the three LRU bits, causing the unmodified bit to get written back to the value it held at the start of the operation. The dual-port SRAM has one port dedicated to reading the LRU bits early in the cycle, and the other port dedicated to writing the modified LRU bits back into the SRAM late in the cycle.

An alternative, which is called the **least frequently used (FRQ)** method, is less often used in cache designs than in memory management units. This method is used to control replacement of the contents of the MMU's CAM. It is still useful in a cache architecture and should be illustrated here. Each cache line has its own pointer, which points to a random Way. In a four-Way cache, there would be a two-bit pointer at each line which would point to Way A, Way B, Way C, or Way D. When a line is accessed in this scheme, the pointer is also accessed. If the pointer is currently pointed at the Way which generates the cache hit, the cache controller would increment the pointer so it would finish by pointing to the next Way. On successive cache hit cycles to that line, the pointer is bumped every time it is found to be pointed at the Way being accessed, until it comes to rest upon a Way which no longer generates any cache hits. When this line is accessed during a cache miss cycle, the pointer will therefore be pointing to the most reasonable candidate Way for replacement. During the replacement cycle, the pointer is again bumped so that the most recently updated data is not immediately overwritten.

Although the least frequently used method is simple and elegant, it once again requires the use of both a read and a write cycle during every cache

hit. This slows things down every bit as much as the true LRU. This means that the only advantage to using this method is that the overall number of bits required to implement it is significantly less than the number we worked out earlier for implementing a true LRU.

One of the most popular alternatives to a true LRU is the **random replacement** algorithm. It doesn't take any explaining for the reader to understand that the basis for this algorithm is that the Way to be replaced is chosen at random. Implementation is simple, especially if the designer is not worried about exactly how random the replacement really is. Somewhat random numbers can be drawn at almost no cost from a number of places. Naturally, in a four-Way random replacement cache, the cache controller runs a one-in-four chance of overwriting the most recently used Way, but this is still better than the 100% chance of overwriting a thrashed location in a direct-mapped cache. A big advantage to the random replacement algorithm is that, while true LRUs for a four-Way cache consume six bits per line, and Intel's method consumes three bits, and the pointer method consumes a mere two, random replacement requires no bits to implement.

Between the least frequently used method and random replacement lies a method called **not last used (NLU)**. Like FRQ, NLU also uses a pointer, but the pointer points at the most recently used Way and is simply a stored copy of the number of the Way which was last hit for any particular set address. The Match output of each cache-tag RAM is simply captured in a RAM which is as deep as the cache-tag RAMs. This can be done in a write-only cycle at a zero timing penalty. The idea behind the NLU replacement algorithm is that random replacement is fine, but it's a tad better if you can avoid the inherent possibility that you'll randomly step on the last-used Way for any set address. Since there is no penalty for using true LRU for a two-Way cache, this method would only be used for caches of more than two Ways, and then there would only be one chance out of $(N-1)$ (in an N -Way cache) of the random replacement algorithm stepping on the second more recently used Way of an address. NLU is probably a hair better than strict random replacement, but it's hard to imagine that it offers noteworthy improvement in cache performance.

There is nothing magic that requires a design to use a number of Ways which ends up being a power of two. Although the most common degrees of associativity are one (direct-mapped), two, and four, you can add or subtract associativity in the exact manner which best fits your system. Sun Microsystems has used a five-Way internal instruction cache plus a four-Way internal data cache in one processor design. But we're getting ahead of ourselves. Split instruction/data caches will be discussed in the next section.

A piece of code like the example in Figure 1.4 might do best in a four-Way design simply because it is using the same set address for the data table, the

stack, the calling routine, and the subroutine. Feel free to draw yourself an imaginary trace of the program in Figure 1.4, then see how it would behave in direct-mapped two-Way and four-Way caches. I decided against illustrating it step by step, showing what would get written into cache when, since it would consume several times the space of the example shown in Figure 1.11.

2.2.3 Unified vs. Split Caches

One way to realize some of the benefits of a two-Way cache, without incurring the problems, is to split the cache into two parts, one for instructions, called the **instruction cache**, and one for data, called the **data cache**. This is called a **split** cache architecture, and the alternative is, not surprisingly, dubbed a **unified** or **unified instruction-data** cache. All discussions so far have assumed that the cache is a unified design. The advantage of the split cache is immediately obvious if you look at the piece of code given as an example in Figure 1.4. The code thrashes at times when data accesses conflict with code accesses and vice versa. If a split cache were used in the example of Figure 1.11, the stack push/pop in steps 5 and 7 would not conflict with any of the other steps shown, and the copy of the program counter would still reside within the cache at the Return instruction at the end of the subroutine. In a split cache, accesses to the data space, as well as to the stack, would occur through the data cache, and the instructions would be accessed through the instruction cache, reducing the thrashing about the same degree as it would be reduced by using a two-Way architecture. The disadvantage is that one of the caches might fill up faster than the other, and there is no way that the full cache could gain access to unused lines in the opposite cache. Worst case would be that the instruction cache would thrash significantly while the data cache would rarely be accessed, or the other way around. Either case is extremely software dependent.

Probably the nicest attribute of split caches is their inherent simplicity. Split caches are simpler to construct than two-Way caches for several reasons. First, each side can be designed as a simple direct-mapped cache, completely independently of any considerations of the other half of the cache. Second, most processors indicate whether the current read cycle is an instruction or a data fetch via a pin which becomes valid as soon as do the address outputs. This means that the outputs of the appropriate data RAM can be enabled as soon as the cycle begins, much the same as in a direct-mapped unified design, loosening the timing constraints not only on the cache-tag RAM, but also on the cache data RAM and the control logic. Some split-cache architectures simply treat the instruction/data output from the processor as the most significant set address bit. Third, in a split cache, there

is no decision about which Way to replace a line in. Instructions are automatically put into the instruction cache, and data is automatically placed into the data cache. Fourth, the caches do not have to match each other in size. Instruction caches should be larger than data caches in most general computing applications, but when real number crunching happens in such applications as array processing, small instruction caches will be adequate, while large data caches are a requirement. Another unusual benefit is that altogether different policies can be used in the two caches. There is no reason why the data cache cannot be a highly associative design, while the instruction cache might be direct-mapped. Should you choose to implement a split-cache architecture, look through this book once to help decide on a set of policies for the instruction cache, then do it again for the data cache. Your design might look unorthodox, but will be more likely to outperform your competitors'.

You can look into different ways to split the cache. Most processors not only indicate whether the current access is for an instruction or data, but many will also disclose user/supervisor mode and whether or not the request is for a stack operation. You just might be able to realize a lot of the advantages of a multiway cache without incurring the speed headaches simply by using any or all of these status signals to enable different split caches.

2.2.4 Write-through vs. Copy-back

Way back in Section 2.1.3, I deferred explaining what action the cache took during a write cycle. This was because there are several means of handling write cycles in caches, or **write strategies**, and the decision impacts the cost as well as the complexity of the cache. Even in this section, I will be deferring until Chapter 4 the explanations of some of the more specific problems involving write cycles.

There are two fundamental strategies defining the behavior of the cache during a write cycle. One is called the **write-through**, or **store through (ST)** cache, and the other is a **copy-back**, **write back**, **deferred write**, **nonwrite-through**, or **store-in** cache (**SIC**). Given all these options, this book will exclusively use the terms write-through and copy-back, since they are both common and are not easily confused with one another. Naturally, once a write strategy is implemented in hardware, it becomes a **write policy**.

In a write-through cache design, either of two courses of action is taken. This is key in designs which can process some write cycles in no-wait states, some of which will be described in Section 2.2.6. In many designs, the cache is updated if a hit occurs, and the cache ignores the write cycle if a miss occurs. In others, the line is automatically invalidated (**write invalidate**). This

approach is used to get over certain speed limitations of the hardware. With a third alternative, a line in the cache is written to whether the write cycle is a hit or a miss. This last course of action is called a **write update** and is usually taken to remove the need to check for a cache hit before writing to a line in a direct-mapped design. The cache controller can begin the write cycle just as soon as the processor indicates that a write cycle has begun. The cache example of Figure 1.15 uses a write update policy. Naturally, in a multiway cache, updating cache lines on miss cycles would not work, since the controller would not know which Way to update until a cache hit or miss had been detected. Imagine the problem of having an updated and an old copy of the same address in the same line of two different Ways of the cache!

The difference in throughput between updating the cache on a miss versus not updating on a miss appears to not yet have been deeply explored. Intuitively, it seems that most programs would read data before writing it, with the exception of data being pushed onto the stack, and with the initialization of memory-based pointers, where an immediate value would be written to a main memory location before being read, then rewritten a number of times. If the cache designer has the option of choosing either of these two approaches, it would be wise to discuss it with the programmers. No matter which approach is used, the write-through cache *always* updates main memory during all write cycles.

The copy-back cache does *not* always update main memory, but reduces main memory accesses by writing data to the cache alone at a much higher speed than that data could be written to main memory. This has three major advantages. First, write cycles become much faster than they would be if a main memory cycle were the penalty for every CPU write; second, some write cycles, like loop counters and stack entries, will only be written to the main memory a fraction of the number of times that the CPU tries to write them. Third, the processor is on the main memory bus a lower percentage of the time. This last factor is a concern in tightly coupled multiprocessing systems, where the speed of the entire system depends on the main memory bus traffic (we'll handle this in depth in Section 4.2).

These advantages come at a cost, however. Housekeeping in copy-back caches requires a lot of thought, especially in multiprocessing systems. The most basic concern is how to handle the data which has been written to the cache, but not to the main memory. At some point, the main memory will need to be updated with the fresher data which is in the cache. An opportunity for this usually occurs when the updated line is to be removed from the cache. Obviously, if this data were simply overwritten, as it would be in a write-through cache, new data would be destroyed, and the data integrity of the entire program would suffer. Therefore, a method must be implemented for the updated line to be transferred to main memory once it is to be re-

SET VALID DIRTY			CACHE-TAG RAM	CACHE DATA RAM
ADDR	BIT	BIT		
0	1	1	Address Tag 0	Dirty Data
1	0	?	N/A	Garbage
2	1	0	Address Tag 2	Valid Data
3	1	0	Address Tag 3	Valid Data
4	1	1	Address Tag 4	Dirty Data
5	0	?	N/A	Garbage
6	1	1	Address Tag 6	Dirty Data
7	1	0	Address Tag 7	Valid Data

Figure 2.14. The Dirty bit signifies whether a cache line contains data which is more up to date than the matching address in the main memory.

moved from the cache. The process of writing data back to main memory when it is being replaced in the cache is called **eviction** or **deallocation**. (The evicted line is called the **victim**.) Some less-often-used terms for eviction cycles are copy-back, write-back, **copy-out**, and **victim write**. I'll stick with the word "eviction" so that there will be no misunderstanding about whether I'm talking about the cache's write policy or the cycle being processed.

One very simple method of implementing a copy-back cache would be to write every Valid line which was being replaced back into main memory, whether or not it had actually been written to by the processor. This would make the cache waste a considerable amount of bus bandwidth with unnecessary main memory write cycles, due to all the evictions of lines which had not been written to by the CPU. Another problem with this method is that all line replacements would take twice as long as would line replacements in a write-through cache, since write-through line replacements require only a main memory read cycle. To avoid this burden, the cache is usually implemented with a means to signify whether a line in cache is more current than the main memory location it represents. The simplest method is to use another bit for each line in the cache, and this bit is called the **Dirty bit**. Data which has been written in the cache, but has not been updated in the main memory is tagged as **Dirty** by the cache controller's setting this bit. Like the Valid bit, there is usually a Dirty bit for every line in the cache (Figure 2.14). During a cache miss cycle, the line to be replaced is examined, and, if its Dirty bit is set, the current contents of that cache line are evicted back into main memory.

The Dirty bit is not the only means of signifying the Dirty status of a cache line. More complex methods are in wide use and will be examined in Chapter 4. In general, these other methods take advantage of the fact that more

than three states can be encoded by the two bits normally used for the Valid and Dirty bits. The non-Valid state can exist with the Dirty bit either set or cleared, and only one of these states is really necessary to the protocol. The leftover state can be used to signify another cache line state.

Just as a point of interest, some split-cache designs use a copy-back data cache, but don't even have logic to accommodate write cycles in the instruction cache, since such cycles just don't happen.

2.2.5 Line Size

In Section 2.1.3, we deferred the true definition of a cache line to this section. A cache **line** is the smallest portion of the cache which has a unique address tag. Some researchers call this unit a **block**, others an **entry**. Those who use the term "block" use the words "**block fill**" to describe the line fill defined in Section 2.1.3.

In the caches in our illustrations so far, all lines have been one word long. Another way to describe them is to state that each word in the cache has its own address tag.

There are two reasons to use **line sizes** longer than a single word. First, if the line is two or four words in length, the cache-tag RAM need only be one-half or one-quarter of the depth of the cache data RAM. Although this is not a big money saver in discrete caches since static RAMs of different densities don't differ too widely in price, the reader can appreciate the savings in die size if a longer line is chosen for an integrated cache design where the die area consumed by a static RAM is directly proportional to the size of the RAM array, and where die cost is more than proportional to die size (see Figure 2.15).

The second reason to choose a longer line size is because multiword transfers from main memory (**burst** or **burst refill** transfers) can be designed to perform much faster than the number of individual transfers which would be required to fill the same number of cache words. If the whole cached system is designed around multiple-line transfers from main memory to the cache, the design can make superior use of available bus bandwidth. This is easy to see and is illustrated in Figure 2.16. With individual transfers, the processor/cache subsystem must output an address for every transaction at the beginning of that transaction. After a bus latency time, the data for that address is placed upon the bus, and the processor subsequently can change the address to request the next word. This is shown in Figure 2.16a. Figure 2.16b shows a burst transfer cycle. The address of the first word of the burst transfer is output, and the data for this word is transferred with the same latency as before, but the main memory realizes that a burst trans-

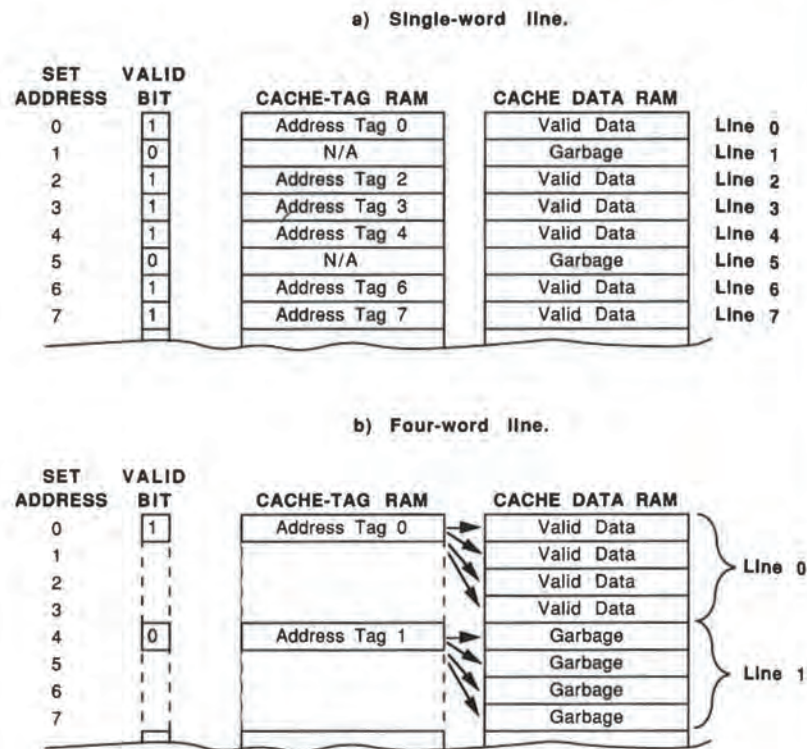


Figure 2.15. The difference between single-word and multiple-word lines. (a) In a cache with a single-word line, every word stored in cache has a corresponding cache-tag entry and Valid bit. (b) In a multiword line cache, there is only one Valid bit and only one tag entry for every N words stored in the cache.

fer is occurring and outputs the second, third, and fourth (or more) words a single CPU clock cycle apart from each other to allow that line to be refilled at the maximum possible rate. For a four-cycle refill, this is dubbed a 2:1:1:1 refill, since the first data is returned after the second cycle, and every following cycle presents data to the cache without wait states (if there were a wait state on each cycle, the refill would be dubbed 3:2:2:2).

This approach obviously takes advantage of the principle of locality, and it is easy to illustrate that there is a point after which it stops helping the system and becomes a burden. The subject of exactly where it becomes a burden is of quite some debate. I will argue my point from two extreme cases. First, look at the cache with single-word lines. As just mentioned, it takes an

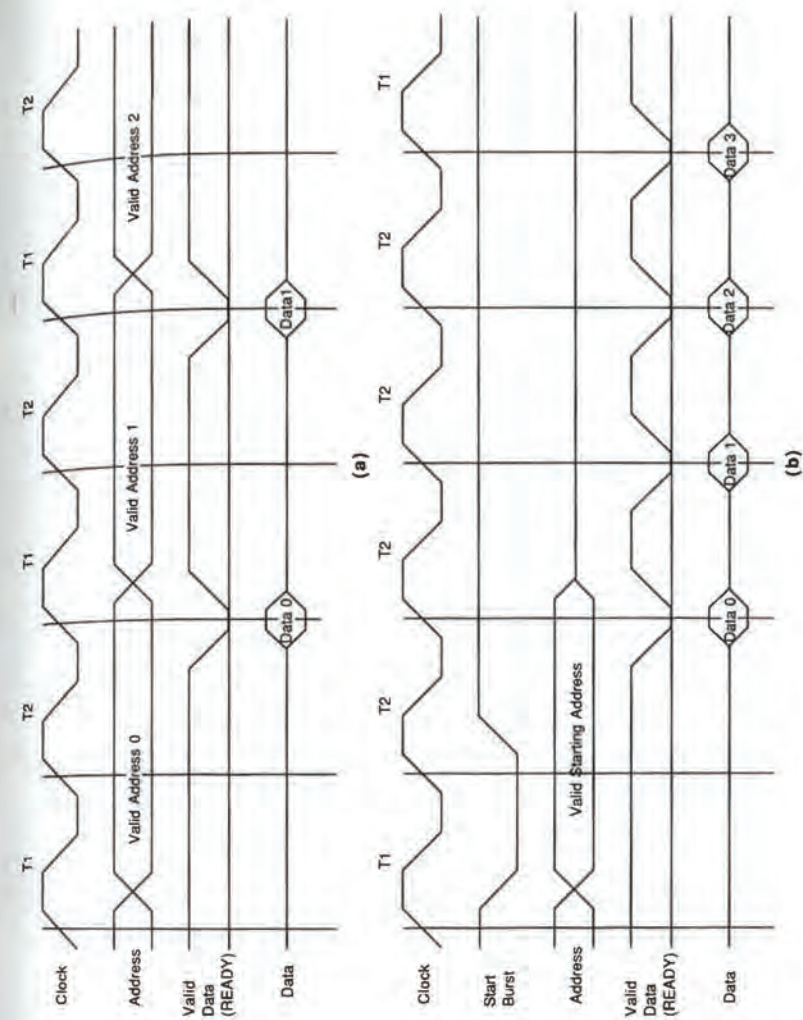


Figure 2.16. Individual word transfers versus burst transfers. In individual read cycles (a), each access consumes two cycles: one for address output, and the other for data input to the CPU. Burst transfers (b) output the address only once, expecting data from a number of adjacent addresses to be returned as a result.

address output and a data input to update any single word in the cache. After this, the CPU can start to run again, perhaps without incurring another cache miss. However, there is a latency cycle for each and every word updated within the cache. On the other extreme, let's look at an example where the entire cache is a single line, no matter if the cache is 512K bytes deep! A single address represents the entire cache, and either the entire cache is a hit, or the whole thing has to be replaced. A couple of mechanisms will make the system with this cache perform worse than any uncached system. During a read miss cycle, the CPU will be held up as the entire line is being replaced. For a four-word line, the 512K-byte cache example would require 131,072 (128K) bus cycles to complete, which is probably longer than most loops which the processor would execute. The other problem is easy to see if you look at a typical piece of code, like the one shown in Figure 1.4. Most programs execute simultaneously out of several spaces. This is what makes associativity work. If there is only one line in the entire cache, any stack access would require a line refill, then the code pointed to by the stack would require a refill, and so forth *ad nauseum*. Although a certain degree of thrashing will occur in a one-word-per-line cache, it is pretty intuitive that anything we might do to reduce the number of distinct addresses contained in the tag RAM would produce more thrashing. It's as if the different spaces, which seemed to occasionally step upon each other, were given bigger shoes in order to step upon each other even more. Unfortunately, there is no globally optimum line size. According to some measurements, a line size of two words is optimum. Other researchers will swear by an eight-word line size. Some researchers point out that the miss rate can continue to decrease with increasing line length even though the performance of the system stops improving.

One very nice approach which some designers use to increase the speed of line replacement is to make the cache-to-main-memory interface use a wider word than the CPU-to-cache interface. As an example, assume that the processor uses a 32-bit word and that the cache's line size is four words. The cache can be designed to actually be 128 bits wide (4×32), but will only present 32 bits to the CPU through some sort of multiplexer (Figure 2.17). Every time there is a cache miss, the cache will conduct a 128-bit interaction with a 128-bit main memory in a single latency. This quickly solves the problem of the CPU's waiting for burst cycles to complete, but does not solve the problem of increased thrashing. Most designers assume that thrashing is the lesser of the two evils. Mitsubishi semiconductor has taken this same approach with a chip they call the "Cache DRAM." This IC is a 4-megabit DRAM with an additional $4K \times 4$ SRAM element. On a cache miss, a 64-bit cache line is moved from DRAM to SRAM and vice versa within a single clock cycle. Since the ex-

ternal data path is 4 bits, a system with a 32-bit bus would use eight devices, causing the line replacement to be a total of sixteen 32-bit words, all in a single clock cycle! This makes sense mostly because the array inside any SRAM or DRAM is often square and subsequently has a very wide internal word, which it internally narrows for the outside world (like the multiplexer shown in Figure 2.17), and also because within a single chip, the penalties of wide words are not nearly as severe as the board space and chip count penalties implemented using industry-standard discrete devices.

At the time of this writing, there are two different camps in microprocessor internal cache line burst refill sequences. The first is the one used by IBM and Motorola for the internal caches on the Power PC microprocessors, where the lower two word-address bits are used to specify which word is being replaced within a four-word line. The bits count upward, starting with the exact address of the cache miss. The other camp consists of the Intel PC processors. These caches will either count up or down, depending on whether the missed address is an odd or an even number. Both counters wrap around the two lower bits of the missed address and don't carry any increment into the higher order bits. At first flush, the Motorola algorithm might appear to be more practical, since code executes in a forward order.

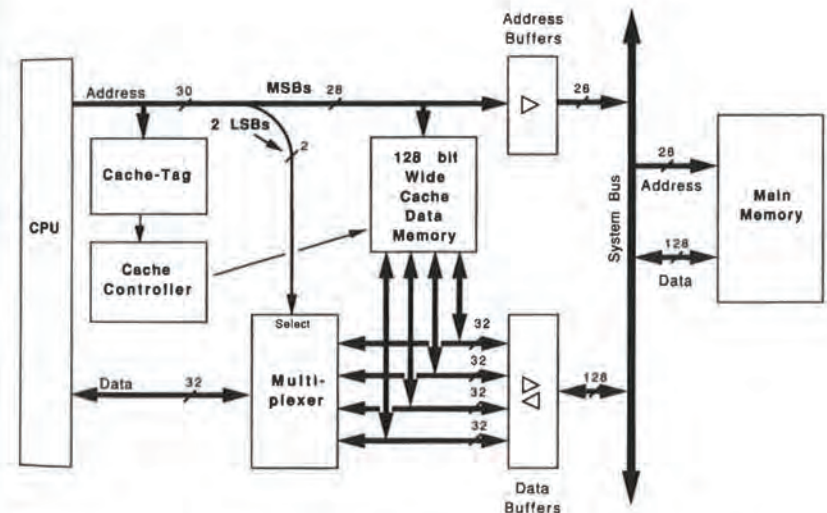


Figure 2.17. Using a wide bus between main memory and cache to allow single-cycle, multiple-word line replacements.

This viewpoint neglects the fact that stacks, which are also stored in the cache, count backward when being read, as do certain other data structures. Another less obvious point is the fact that since the counter wraps around after overflowing, any cache miss will necessarily fill the cache line with the same four words, no matter which sequence is used. Although I am sure that someone will perform a study at some time in the future which will show a marginal difference between the two sequences, I honestly doubt that line refill sequences make much difference to the performance of the cache.

Long lines make a lot of sense in older systems designed using ECL CAMs. Recall that high-speed CAMs only come in a 4×4 density, forcing any 32-bit address system to use eight of these devices simply to decode four lines' worth of addresses. If such a system had 2K lines of cache, the cache directory would have to use over four thousand parts! There is an overwhelming incentive in such a system to use a smaller cache and to make the line size as long as possible. A cache with a 16-word line could automatically use one less CAM per four lines. A 256-word line would use two less CAMs, for a total of six CAMs per four lines. Actually, most processors use only the lower two address bits for byte addressing, so the latter case would even work with line sizes as short as 64 words.

Write policy is also a factor in line size decisions. If a write update rather than a write invalidate policy is chosen (in which a write miss overwrites an existing line in cache), and if the write is shorter than the line length, how does the cache signify that only a portion of the line is valid? There are several methods of handling this, and we'll explore two here.

The first method is **write allocation**. When a write miss occurs, the balance of the cache line being written is fetched from main memory, and the write data is merged into the line as that line is written into the cache. In more discrete steps, once a write miss has been detected, the cache controller starts a line replacement cycle, possibly evicting a Dirty line from the cache, performing a main memory read of the data which will be copied into the replaced line. Data from the address which is being written to is either not transferred from main memory or is immediately overwritten by the CPU's output data. At the end of the cycle, the cache line is filled with main memory data with the written word updated. Some designers also call this function a **merge**. Figure 2.18 shows write allocation graphically. In this example, the CPU encounters a write miss while attempting to write a byte to an address which is not represented in the cache. The line is updated from main memory, then the appropriate byte is overwritten in the cache. Whether or not the write data is sent to the main memory depends upon the write policy of the cache, but, in any case, the first write miss to any cache line will incur a significant penalty, since an entire line must be fetched before the processor can proceed. One way some designers have gotten

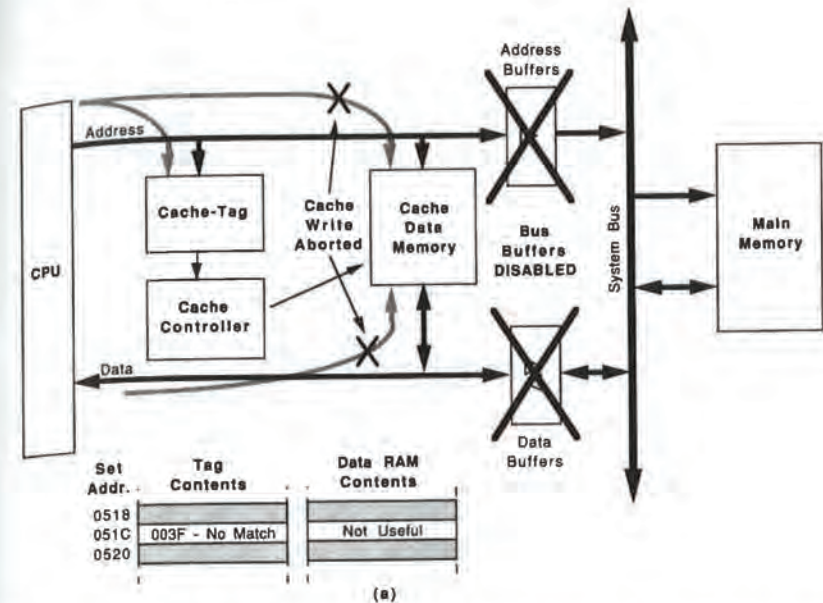


Figure 2.18. Write allocation in a write-through cache. When a cache write miss is detected, the write cycle is aborted (a), and the processor is put into a wait state. A line is then read from main memory into the cache data RAM (b), and the corresponding tag is updated. The processor is then allowed to proceed with the write cycle, writing simultaneously to the updated cache line and to main memory (c).

around this problem is to post writes to the cache, but this can cause the cache hardware to become inordinately complex.

The second method involves **sectoring**. Those who call lines blocks call sectors **sub-blocks**. In the preceding example we assumed that all cache to main memory transactions involved entire lines. When the CPU encountered a miss while trying to write to a portion of a line, the line was replaced by a matching line via write allocation. That line used a single Valid bit to indicate its authenticity. A sectoried cache design allows the smallest writeable data unit (usually a word) to have its own Valid bit, so that each cache line contains several Valid bits, one representing each sector or sub-block. This is illustrated in Figure 2.19. Readers will note a strong similarity between this diagram and Figure 2.5, where the line is a single word long. The cache data and the Valid bits for every word have been kept, but the cache-tag RAM has been economized to have only one-fourth as many entries. This approach is often used to save silicon on integrated cache controllers.

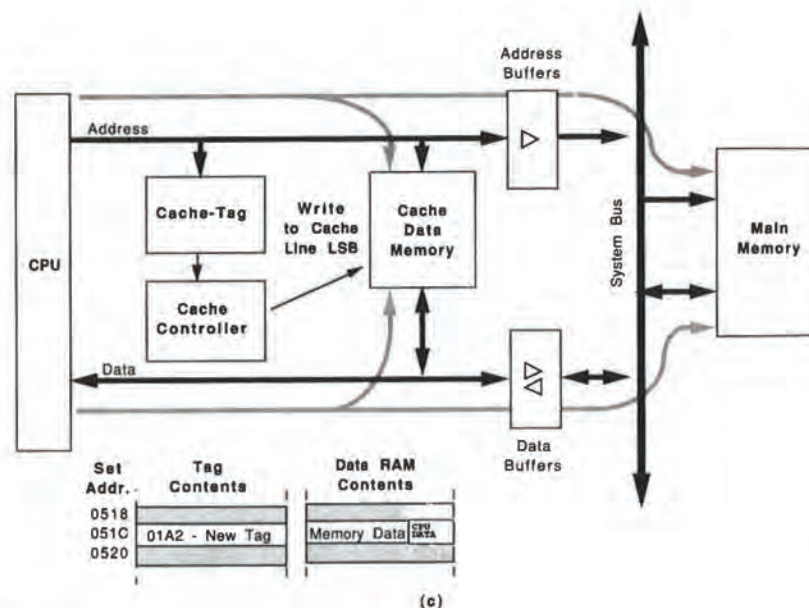
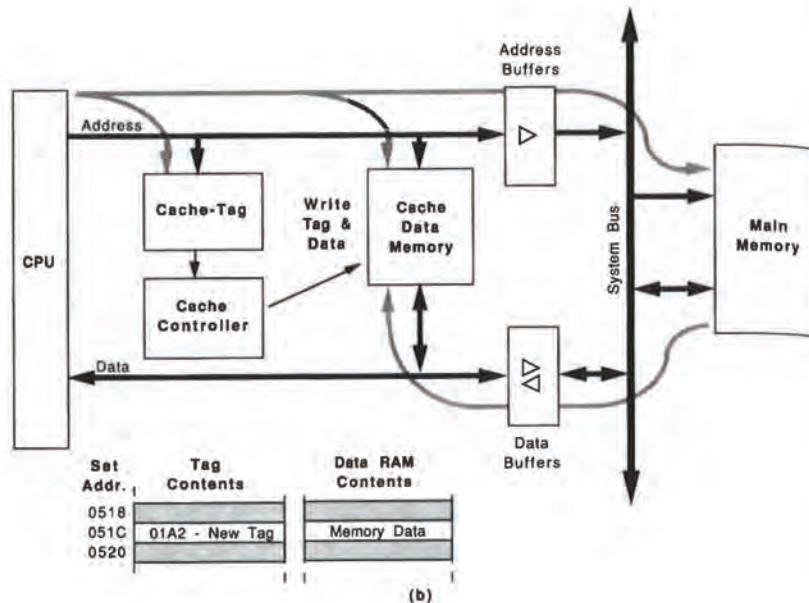


Figure 2.18. (continued)

When a cache read miss occurs, the target line is invalidated, and the requested word is brought in and added to the cache. Only the Valid bit for that word is set. By spatial locality, we would be drawn to conclude that some nearby word would be requested by the CPU soon. When this word was indeed requested, if it fit within the same line, the cache controller would update the cache and set the Valid bit for that word within the line, so that now two Valid bits would be set. Only those words that the CPU actually requested would be brought in from main memory; however, each word would require an entire bus cycle, rather than the abbreviated cycle afforded by burst refills. In our example of a write miss, all Valid bits for the line being replaced would be cleared, and the Valid bit would be set only for the word being written into the cache (Figure 2.20). This allows cache write miss updates to occur at the same speed as the write cycle would if the cache were not updated. This beats having to wait for the multiword read cycle required in a wire allocated system. If a write buffer is used, or if this method is used with a copy-back policy, the write cycle can happen as fast as the CPU can operate. An added benefit in systems using sectoring is that the main-memory-to-cache interface can now be handled on a word-by-word basis, which is important if the cache is added to an older bus architecture which cannot support burst read cycles.

Looking back on the last few paragraphs, we see that a lot of trouble arises from trying to cache write miss cycles. Of course, the simplest way to handle the entire problem is to disallow line replacements on write misses in the first place. Write-through caches are unlikely to realize much benefit from caching write misses, since data is usually read first, then written to the same location unless it is the first stack push to a given address. Although a policy of disallowing line replacements on write misses could cause an in-

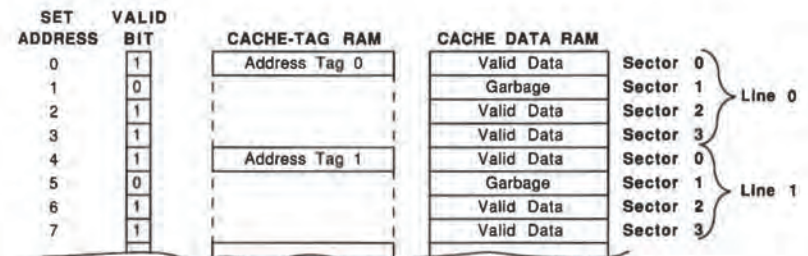


Figure 2.19. The organization of the cache-tag RAM, Valid bit RAM, and the corresponding cache data RAM in a sectored cache design. Each line has a cache-tag entry, and each sector has a Valid bit. This is similar to the long line cache in Figure 2.15b but adds a Valid bit for each sector.

(a) Before Miss cycle

SET ADDRESS	VALID BIT	CACHE-TAG RAM	CACHE DATA RAM
0	1	Non-Matching Tag 0	Valid Data
1	0		Garbage
2	1		Valid Data
3	1		Valid Data
4	1	Address Tag 1	Valid Data
5	1		Valid Data
6	0		Garbage
7	1		Valid Data

(b) End of Miss cycle.
New Line in cache.
Only one sector valid.

SET ADDRESS	VALID BIT	CACHE-TAG RAM	CACHE DATA RAM
0	0	New Tag 0	Garbage
1	1		Missed Sector
2	0		Garbage
3	0		Garbage
4	1	Address Tag 1	Valid Data
5	1		Valid Data
6	0		Garbage
7	1		Valid Data

(c) Second sector read from
same address added to new
cache line.

SET ADDRESS	VALID BIT	CACHE-TAG RAM	CACHE DATA RAM
0	0	Address Tag 0	Garbage
1	1		Valid Data
2	1		New Sector
3	0		Garbage
4	1	Address Tag 1	Valid Data
5	1		Valid Data
6	0		Garbage
7	1		Valid Data

Figure 2.20. Line replacement in a sectored cache. Before the miss (a), the tag bits for set addresses 0–3 do not match those output by the CPU. The CPU's output address should be mapped into the cache's set address 0, causing the replacement of the tag and data for that address and the invalidation of all the other sectors in that line. When a second sector (c) is read into the same cache line, the data and Valid bits are updated, but the matching tag is left alone.

crease in bus traffic in copy-back caches, even here the penalty would be slight.

2.2.6 Write Buffers and Line Buffers

A simple write-through cache design works with a microprocessor to reduce the effective main memory read cycle time; however, it has no effect upon main memory write cycle time. Effective main memory write cycle time can be improved through the addition of a **posted write buffer** or, more simply, a **write buffer**, to perform (quite naturally) a **buffered write**.

In a write-through cache designed without a posted write buffer, the microprocessor must complete a main memory bus transaction every time it performs a write cycle. This will cause it to suffer the associated system bus delays. However, in a system using a write buffer, the microprocessor writes the data to the cache and writes the data, address, and pertinent status signals to the write buffer (but not to the system bus) during a write cycle. The microprocessor then continues to access the cache, while the cache controller simultaneously downloads the contents of the write buffer to main memory. This reduces the effective write memory cycle time from the time required for a main memory cycle to the cycle time of the high-speed cache.

The use of write buffers can nearly remove the performance differences between write-through and copy-back caches. Just as studies have been performed to determine the most cost-effective cache size and associativity, similar studies have focused upon the appropriate depth of write buffers. For economy's sake, many designs use a single level of depth. Some semiconductor companies have produced four-level write buffers and claim that this configuration will allow zero-wait write cycles 99.5% of the time, but it is patently obvious that the appropriate depth of write buffers, like most other cache design trade-offs, depends greatly upon such phenomena as main memory access time and the write cycle activity of the program being run.

Write buffers cause their own set of problems. Let's examine the case of a stack push followed shortly by a pop, where the push was a write miss which was not written into the cache. Naturally, the pop will also suffer a read miss cycle. Even if the write buffer is only a single level deep, there is the possibility that the data from the push will not have made it into main memory before the pop is executed. Unless care is taken, the pop will read main memory before main memory is updated with the pushed data. A simple way around this is to disallow the cache from performing a line update until the write buffer's contents have been loaded into main memory. A forced write from the write buffer to main memory is called a **drain** of the write buffer. Another answer is to always update the missed line on a write miss.

The problem gets far trickier when a multilevel write buffer is used. Either the write buffer must be entirely depleted before the cache can continue with the line update, or the write buffer must satisfy the request for data. Certain commercially available write buffers allow this last approach and behave as if they were fully associative caches for any data they contain which has not been written to main memory. This is sometimes referred to as a **victim cache** and is touted as a one- or two-line fully associative cache used to reduce thrashing one or two lines at a time. No matter where the data is in the queue, if a read cycle requests that data, the write buffer will supply the data back to the CPU rather than the cache or main memory. A version of this approach is also known as a **pollution control cache**.

Byte gathering is a nice feature which is offered on certain multilevel write buffers. In programs which perform text manipulation, and in some older revisions of programs which might have been written for an architecture which previously supported only narrow data, the processor might find itself writing twice or four times to the same word address in order to update individual bytes or pairs of bytes within that address. As an example (Figure 2.21), let's say that a four-letter word was being written, one byte at a time, to address 09AF 45ED. In this example, the processor outputs the same address four times and outputs a separate byte write command each time. If our write buffer had four levels, all four would quickly become filled during this operation. A byte-gathering write buffer notices the similarities between the addresses and keeps updating bytes within the word, which has yet to be written to main memory. The hardware for doing this is the same as is used by the write buffer to satisfy read requests with pending write buffer data, since either one is enabled by address matches. In a more severe case, a string might be written to two locations, with the same data being written alternatively to two addresses. Byte 0 goes to 0000 0000, then to FFFF FFFF; then byte 1 is written to 0000 0000, then FFFF FFFF, and so forth. In a byte-gathering write buffer, two buffer locations would collect the data to these two locations, no matter what the order in which the data were presented. In caches which use multiword cache lines a similar mechanism called **write combining** combines separate word writes into a single cache line write.

One outcome of all of this is that the traffic on the main memory bus doesn't look a thing like the traffic on the CPU's pins. A cache miss which is unfulfilled by the write buffer data can place a read request on the main memory bus before the preceding write cycles have been downloaded from the write buffer. A succession of alternating byte writes like those just illustrated would slim down to two simple four-byte word writes. To put it another way, a string of eight single-byte writes, followed by a read cycle, might appear on the main memory bus as a read cycle followed by two four-byte word writes. This greatly disturbs the order in which events occur on the sys-

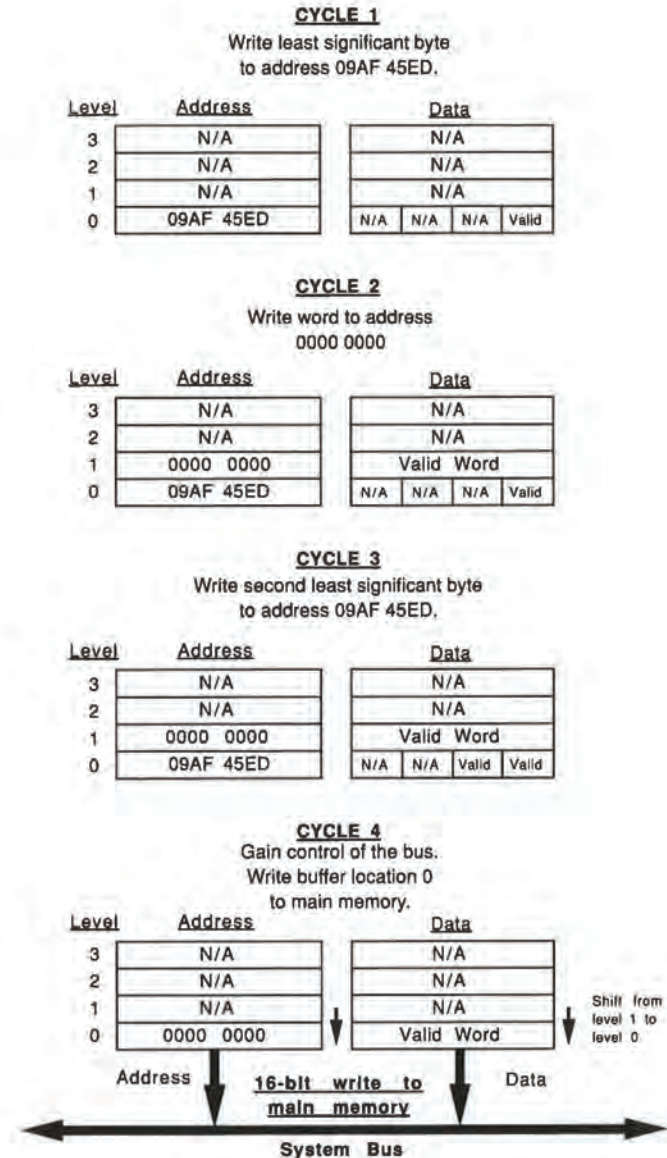


Figure 2.21. Operation of a byte-gathering write buffer during interspersed multiple-byte writes to the same address.

tem bus (as if the randomness of the main memory read cycles was not bad enough due to the cache's absorbing most of the program's locality). The degree to which the write and read cycles are out of order is referred to as **write ordering**, **read ordering**, **sequential consistency**, or just **consistency**. Write ordering is called **strong** when write cycles and read cycles occur on the system bus in a sequence close to that followed by the CPU. The strongest ordering is called **processor ordering**, indicating that cycles appear in exactly the same order on the bus, as on the processor. If the sequence is completely jumbled, the write ordering is referred to as **weak**. Without covering this in too much depth, let's simply observe that write ordering can become important in I/O or multiple-processor transactions where a location is read, and, according to its value, a correction factor is written to a different address. A subsequent read, to determine the effect of the preceding write, could conceivably be placed upon the system bus before the preceding write had made it through the write buffer. Naturally, there are few systems where this will actually occur, but in real-time systems, this could cause some hard-to-find instabilities.

Write buffers are not used only at the CPU-to-main-memory interface. Certain processors have such unforgiving write cycle timing that their caches cannot possibly accept write cycles at zero wait states. Some designers solve this problem by putting a single-level write buffer between the CPU and the cache. In other designs, the apparent speed of main memory can be accelerated through the addition of a write buffer to the main memory itself. Although write cycles are still delayed by the system bus latency, the main memory write cycle time is hidden from the processor.

Write buffers can also be used profitably in a copy-back architecture in what is known as **concurrent line write-back**, **fly-by write-back**, or **background write-back**. Concurrent line write-back is a method of hiding eviction cycles from the processor. They are hard enough to explain that the diagram in Figure 2.22 will be used to help. In a typical read miss eviction cycle, the evicted line is copied back to main memory before the read cycle for the replacement line is allowed to begin. This amounts to a doubling in the effective main memory access time, which is a pretty bad deal. In a concurrent line write-back, the read cycle is the first to occur. This can be accomplished in either of two ways. In the first, which is called **buffered line fill**, the replacement line is read into a **line buffer** (something like an incoming write buffer, writing main memory data into the cache) and is used to satisfy the CPU's immediate needs. Later on, when the CPU is off doing something else, the evicted line is written into main memory, and the line buffer is written into the cache. This can involve some pretty tricky timing, especially since the cache is nearly never left alone by the CPU.

The second approach to performing a concurrent line write-back is simpler, but can slow down line replacements if the main memory is signifi-

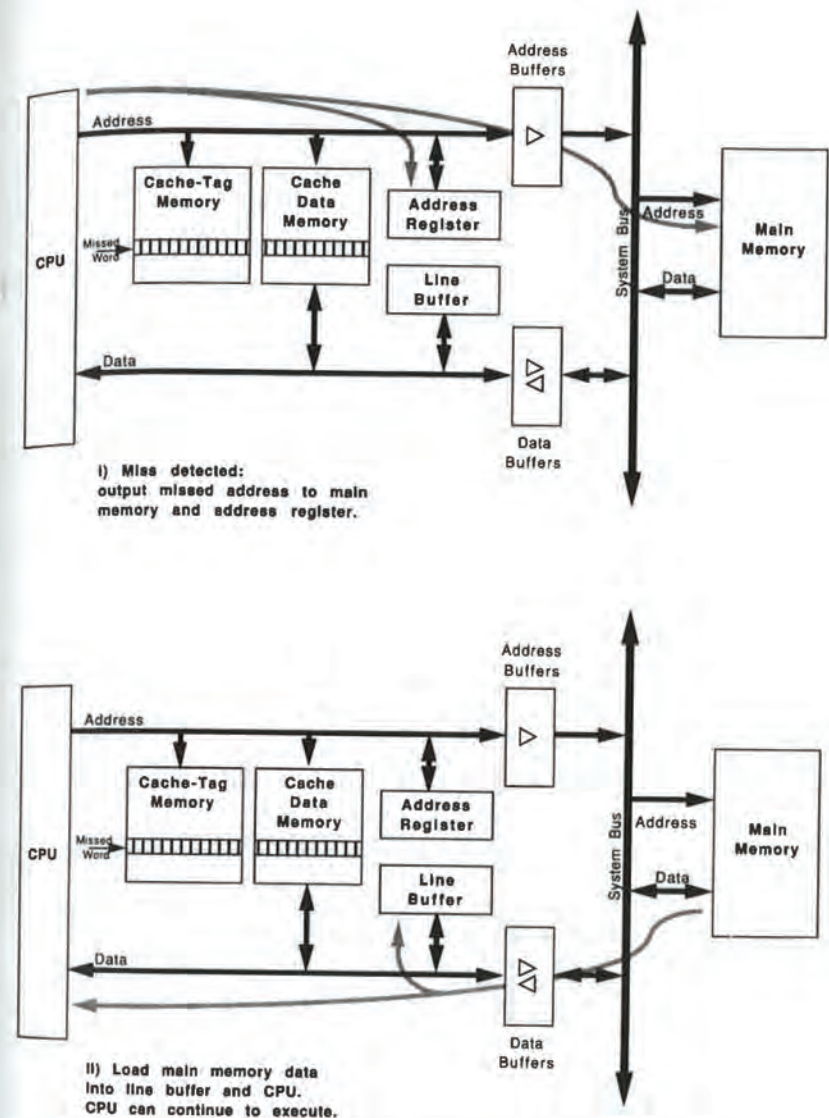
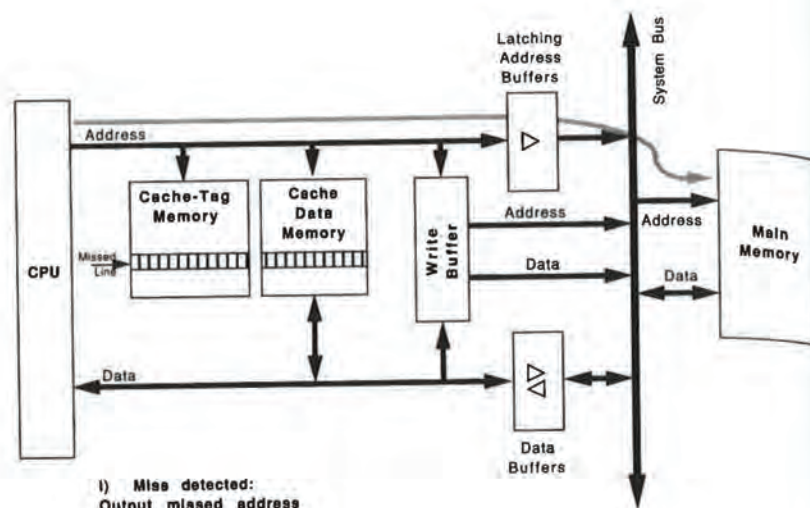
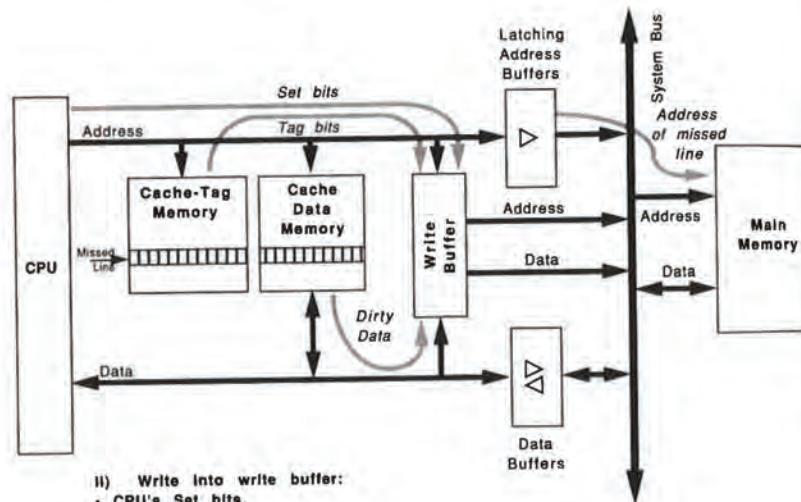


Figure 2.22. Concurrent line write-back implementations using line buffer for read data (a), and write buffer for evicted line (b).

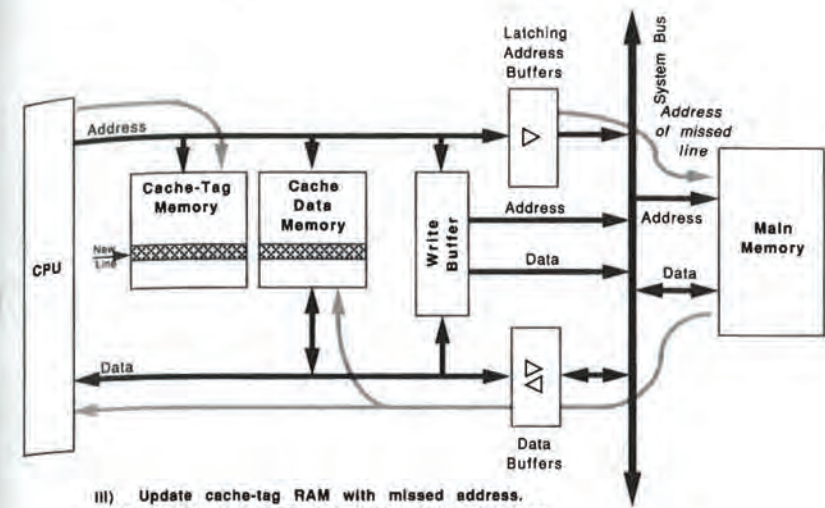


I) Miss detected:
Output missed address
to main memory.

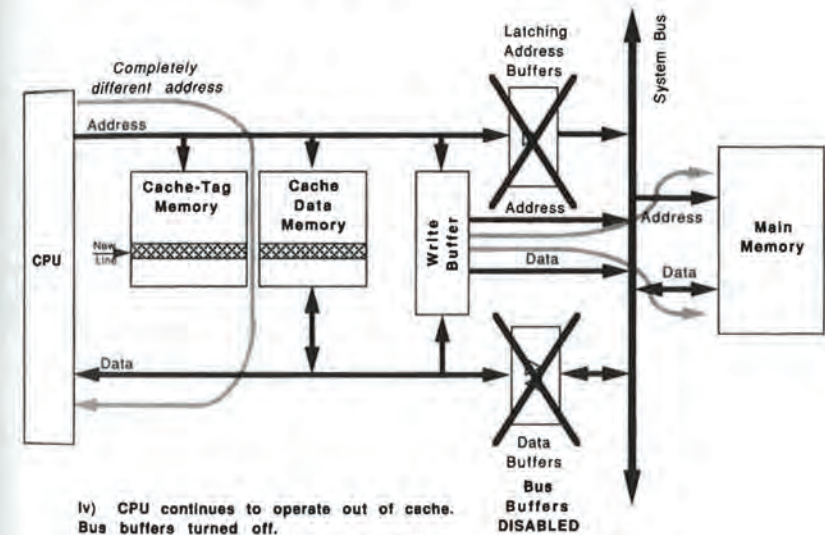


II) Write into write buffer:
• CPU's Set bits,
• Cache-tag RAM's tag bits,
• Dirty line from cache data RAM.

Figure 2.22. (continued)



III) Update cache-tag RAM with missed address.
Update cache data RAM with main memory response.
Feed data to CPU.



IV) CPU continues to operate out of cache.
Bus buffers turned off.
Write buffer updates main memory with
stored Dirty line.

Figure 2.22. (continued)

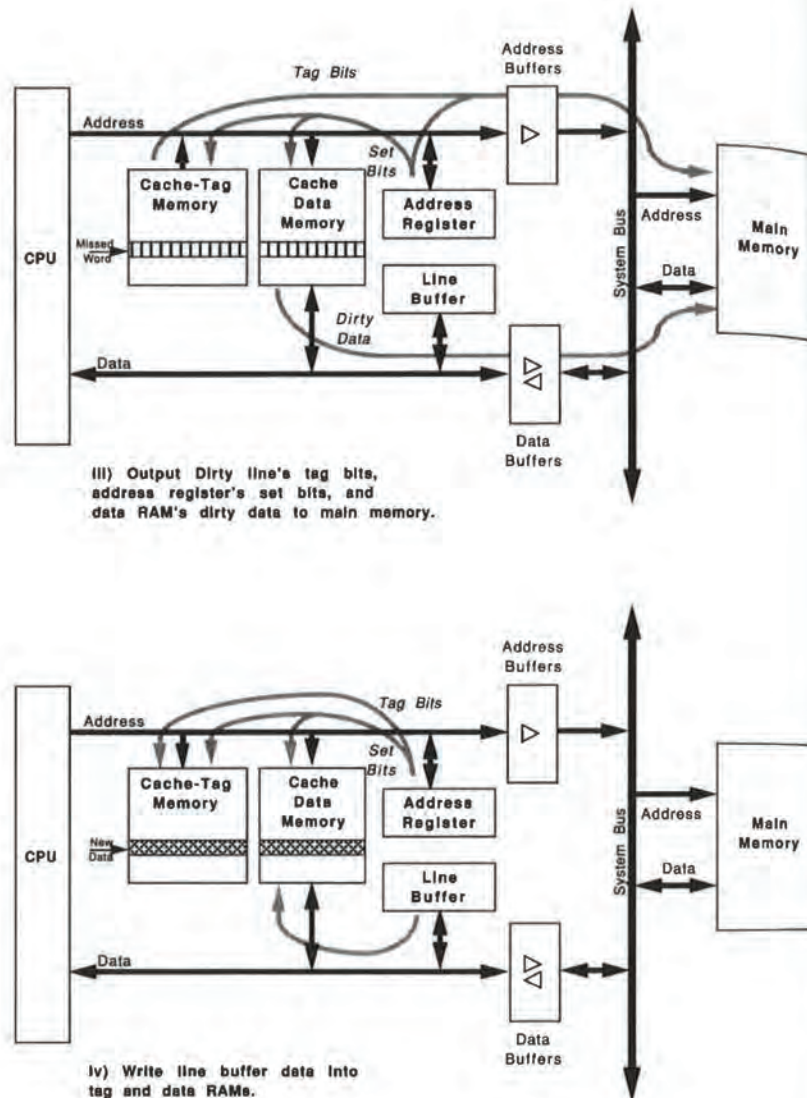


Figure 2.22. (continued)

cantly fast and the cache line is long enough. With this method, once the read miss is detected, the main memory read cycle is initiated. At the same time that the cache controller is waiting for a response from the main memory, the evicted line is being loaded into an outgoing write buffer. Hopefully, the write buffer takes less time to fill than the main memory's access time, so that the main memory data is not languishing, waiting for the cache controller to finish moving the evicted line into the write buffer. As soon as both the main memory data is available and the evicted line is completely copied into the write buffer, the main memory data is presented to the CPU and copied into the cache as a line update. Once the line update is completed, the CPU can continue to operate out of the cache, while the write buffer copies its contents into main memory as a background task.

While either of these methods is pretty complex, the advantage of halving the effective main memory access time is well worth the effort. For this reason, the approach of allowing two things to happen at once is called **concurrency** or **bus concurrency** and is a sought-after feature in cache designs. Just like write ordering, concurrency is called strong if a high number of events can overlap and weak if few events overlap. If you think about concurrency a little, it will become clear that one means of increasing concurrency, namely, write buffers, serves to mess up the write ordering, or consistency, of the cache. In other words, systems with weak concurrency will exhibit strong consistency, while systems with strong concurrency will exhibit weak consistency. Good grief!

Line buffers are not only helpful in mastering concurrency, they tend to speed up line replacements in any sort of cache which supports multiword lines. The designer has two options when deciding on a line refill strategy. An option which does not require a line buffer is to hold the processor in a wait state until the entire line has been refilled. After the complete refill, the processor is allowed to continue. This stands to reason, since the data path into the processor's internal cache is tied up with the cache line refill and cannot be easily made to service the needs of the CPU as the data is being fed in from outside. It is not uncommon for such systems to use a **line fill order** or sequence of lines burst into the cache of **data requested last** (also called **desired word last** and **critical word last**). The line fill starts with a different address than that required by the cache miss, and ends with the missed word.

Line refill strategies based upon line buffers can be called **streaming** caches. The missed word is fed to both the line buffer and the CPU simultaneously, then the CPU is allowed to continue, perhaps to request the next word in the line, which might be the next word being updated in the line buffer at that exact moment, or even to read from a completely different cache line as the rest of the missed line is being read into the line buffer.

Other streaming cache designs which are based on sectored caches without line buffers allow simultaneous execution and cache line update at the expense of the line update possibly being interrupted by CPU-to-cache interaction at a location away from the address of the line update, resulting in the cache only updating a portion of its line. This is dubbed an **aborting** line fill, since the CPU has the capability of discontinuing the line fill in order to service a miss at a different address. Of course, this can also be done in a non-sectored cache design if the designer doesn't mind invalidating the entire line on the miss which aborts the line fill. On the other hand, if the line fill is allowed to continue, and the CPU is caused to wait for the second miss to be serviced, the line fill is called **nonblocking** (the CPU cannot block an ongoing line fill). Most caches with line buffers use a nonblocking strategy. It's anything but intuitive why one strategy would be any better than the other. Streaming caches go by a variety of names (like everything else in this book) such as **bypass**, **load forwarding**, **early continuation**, or **early restart** designs. Examples of streaming caches are found in most modern processors.

In general, if a line buffer is used to accelerate a cache line replacement, a **wrapping fetch** is used (the miss may happen at the middle of the line, in which case the burst wraps around until the entire line is fetched). The line fill order of wrapping fetches is called **desired word first**, **data requested first**, or **critical word first**.

Certain caches perform **prefetches** in anticipation of the next line to be missed by the cache. In general, the line following the last line fetched is prefetched and stored in the line buffer, with the assumption made that all misses are compulsory; that is, the fetched line has not previously resided in the cache. Caches which prefetch in anticipation of the next miss are called **fetch always** or **Class 2** caches. Conversely, caches which only fetch missed lines are called **Class 1**, **fetch on demand**, or **fetch on fault** caches.

After you've read through this section, look it over and count all of the new buzzwords shown in bold print. And then decide whether you would rather be a cache designer or a Tibetan monk!

2.2.7 Noncacheable Spaces

Certain spaces in main memory should not be cached. The most obvious example is portions of the main memory address range which are used as device inputs. This is especially important in processors which don't differentiate between memory and I/O addresses, like Motorola's 680x0 series. Should the processor be looping while awaiting a change in the status of a switch panel, for example, no change will be noticed if the first reading from the switch panel was cached. Some processors also use a portion of the

main memory address range for communication with coprocessors. Once again, the data cannot be assumed to be static, so a cached copy might become out of date. Lastly, multiprocessor systems often communicate via flags which are set and cleared in dedicated main memory locations. No communication can occur if the main memory flag that one processor sets cannot be read by another since the other is reading a cached copy.

All of these examples are reasons for accommodating **uncached addresses** or **noncacheable addresses (NCAs)** within the cache. The method of implementing this is very simple. An address decoder signals to the cache that the current memory request is within a noncacheable space. The cache controller, in turn, disallows the selected line from being updated within the cache. In a way, the cache controller treats the cycle as neither a hit nor a miss, but causes the cache to behave as if it did not exist during this particular cycle.

The exact implementation depends on the freedom of the system designer to change the system architecture. In some systems, all of either high or low memory can be declared noncacheable, and the cache will ignore addresses with the top bit either set or cleared. In other systems, each add-in I/O board will signal which portion of its own address space is noncacheable through a dedicated signal on the backplane. The most difficult case is the addition of a cache to an architecture which previously existed without caches. In such systems, the usual approach is to implement an address decoder as a part of the cache, right next to the CPU, and to feed all possible noncacheable addresses to the cache controller, even for devices which might not be installed in the system. Naturally, this will impact cache performance if the decoder disallows the caching of several spaces where the I/O device in question does not exist.

2.2.8 Read-only Spaces

Along the same lines as the noncacheable space is the **read-only** space. This is a cacheable address where the cached line is updated during cache miss reads only, and not on write hits. Read that last sentence again because it doesn't seem to make sense. Why would a cache want to contain the last data read from a main memory address and not to update that data when the same main memory address was written?

This peculiar concept owes its existence to the fact that certain I/O board designers overlay the main memory write addresses of their I/O devices' output registers with a PROM containing the I/O device's driver. There are actually two memory spaces residing at the same address: a read-only space and a write-only space. The write-only space is the output registers of the

I/O device. The read-only space is the I/O driver PROM. Naturally, the PROM would do well to be cached, but since the PROM's contents don't get revised as the CPU writes to the I/O device's registers, the cached version of the PROM's contents cannot be changed when this write cycle occurs. This problem, sometimes called **write side effects**, tends to occur largely in PCs, whose main memory space is limited, so designers use the PROM-I/O overlap to save room.

The simplest method of handling this problem is to mark all addresses within a certain range with a read-only or **write protect** bit as they are cached. Once again, an address decoder is used, but this time the decoder's output is sent directly into a separate bit, similar to another Valid bit, as the cache line is being updated. This bit automatically disallows any change to the contents of the cache data RAM during any write hit cycle at the read-only address.

This approach is used in certain secondary cache controllers. The write protect bit is used to allow an address to be cached, but not overwritten, in the secondary cache. The secondary cache controllers further use this bit to disallow the processor's internal primary cache from even containing a copy of the location. This prevents the inevitable updating of the primary cache's copy during a CPU write cycle, since the CPU's internal primary cache usually does not have a read-only bit (see Section 2.2.10).

2.2.9 Other Status Bits

All of the housekeeping bits described in the preceding sections can be called **status bits**. This category includes Valid bits, Dirty bits, read-only bits, LRU bits, and anything else which is stored along with every other cache line, but is not data or an address tag.

If you do a little math, you will find that you can reduce your cache's overall bit count through judicious use of status bits. A good example is the use of sectorized lines, or longer lines, to get rid of tag entries. While these approaches might not help out much with discrete cache designs, on-chip caches can really take advantage of any opportunity to reduce transistor count. This makes it even more important to measure the actual performance of all of your design options before committing to a specific cache design. Another way to save memory bits is to disregard more significant tag bits if these represent address bits which are not used by the system.

There are other reasons to add status bits to a design. **Control domain identifier** bits can be used to identify the user level of a cache entry. Sometimes these are used to determine which cache entries should be invalidated during a task switch in a multitasking operating system. In some cache de-

signs, task numbers are stored to counter aliasing problems which might otherwise crop up in a logical cache design. The logical cache in Figure 1.15 uses this approach with the processor's function code bits FC0-FC2.

Lock bits are used to force an entry not to be removed from the cache. This can help speed up interrupt response in time-critical software applications, such as real-time operating systems. Lock bits make most sense in a multiway cache, but can also find their way into direct-mapped designs. A **reader's/writer's lock** is a bit which can keep unauthorized processors or processes from overwriting the contents of a cache line.

No doubt you will run into other caches which use status bits in some unique manner. They are easiest to understand if you just think of them as markers for the individual lines of the cache. We'll see designs in Chapter 4 where cache line states are encoded to reduce the number of status bits used to store a line's state.

Now let's move on to some approaches which do not interplay with the number of status bits in each cache line.

2.2.10 Primary, Secondary, and Tertiary Caches

You can't have gotten this far in the book without having determined that caches do a good job during their hit cycles, but don't really speed up miss cycles at all. The question then becomes one of a cost/performance trade-off, pitting the optimum cache size and speed against the practicalities of building such a system. Microprocessor designers are confronted with the trade-off of die size versus cache size for a processor with an on-chip cache, since an on-chip cache can naturally be made to run significantly faster than anything that involves signals being routed off of and onto the processor chip itself.

Let's work with a hypothetical case where the processor has been designed with a small, but very good cache, and slow DRAM is used to implement main memory in order to keep the memory array's cost down. How can such a system be made to operate faster? The designer is going to try to minimize the access time of the main memory during on-chip cache miss cycles, and although tricks like interleaving main memory help, they are effective only to a certain point. A very valid approach to this problem is to build a larger but slower cache outside of the chip and to use this cache to accelerate the apparent access time of main memory during an on-chip cache miss cycle. In a system where two caches are cascaded in this way (see Figure 2.23), the cache more intimately tied to the processor is called the **first-level** or **primary** cache, and the cache which is placed between the CPU/primary cache subsystem and the main memory is called the **sec-**

ondary or **second-level** cache. More levels may be added, of course, and these would be called **third-level** or **tertiary** for the cache between the secondary cache and main memory, and so on. Some architects use the terms **Level 1** or **L1** for the primary cache, **Level 2** or **L2** for the secondary cache, **Level 3** or **L3** for the tertiary cache, etc. Whichever terminology is used, the cache which is closer to the processor than the one being discussed is called the **upstream** or **predecessor** cache, and the cache closer to the main memory is called the **downstream** or **successor** cache.

Rest assured that the world has not become all that complex yet. Although the author has seen a number of systems with secondary caches, systems which use more than two cache levels are extremely rare at the time of this writing, although some multiprocessor systems use a shared tertiary cache for processors which each have their own primary and secondary caches.

There are other reasons to use a multilevel cacheing scheme than the one just mentioned, such as saving die size on the CPU chip. Certain processor architectures have designed-in cache controllers which automatically limit the size and policies of the primary cache. Another reason is that the size or type of cache which is desired to fit between the CPU and the main memory might not be implementable with state-of-the-art static RAMs. Large RAMs tend to be slower than small ones, so a designer might be caught compromising between a small, fast cache and a large, slow one. The problem can be broken down so that both a small, fast cache and a large, slow one are used together as the upstream and downstream caches in the system, thereby reducing the level of compromise which must be made. This same approach has been discussed as an approach to be used within a single chip, where the processor chip contains the CPU plus a small primary and a large secondary cache. Wait a minute, you say. Won't both on-chip caches operate at the same speed? Not really. The larger a RAM grows, the more levels of logic required to implement its address decoder. Followers of the RISC argument, where the cycle time of a CPU is reduced by minimization of the critical path logic delay elements (Section 3.1), will observe that by reducing the cache size the critical path can be made that much faster. In this sort of chip, the cache and CPU are designed closely together, with all efforts going to the reduction of cycle time. We'll show some examples in Section 5.2.

One last reason, and probably the most common one used to rationalize the use of a secondary cache, is that the system designer is chartered to use an industry-standard CPU which has an internal cache which was not designed with the current system in mind. A good example is that of a multiple-processor system designed around any of the many currently available processors featuring write-through caches. Caches in multiple-processor systems are usually designed to minimize system bus traffic far more than is

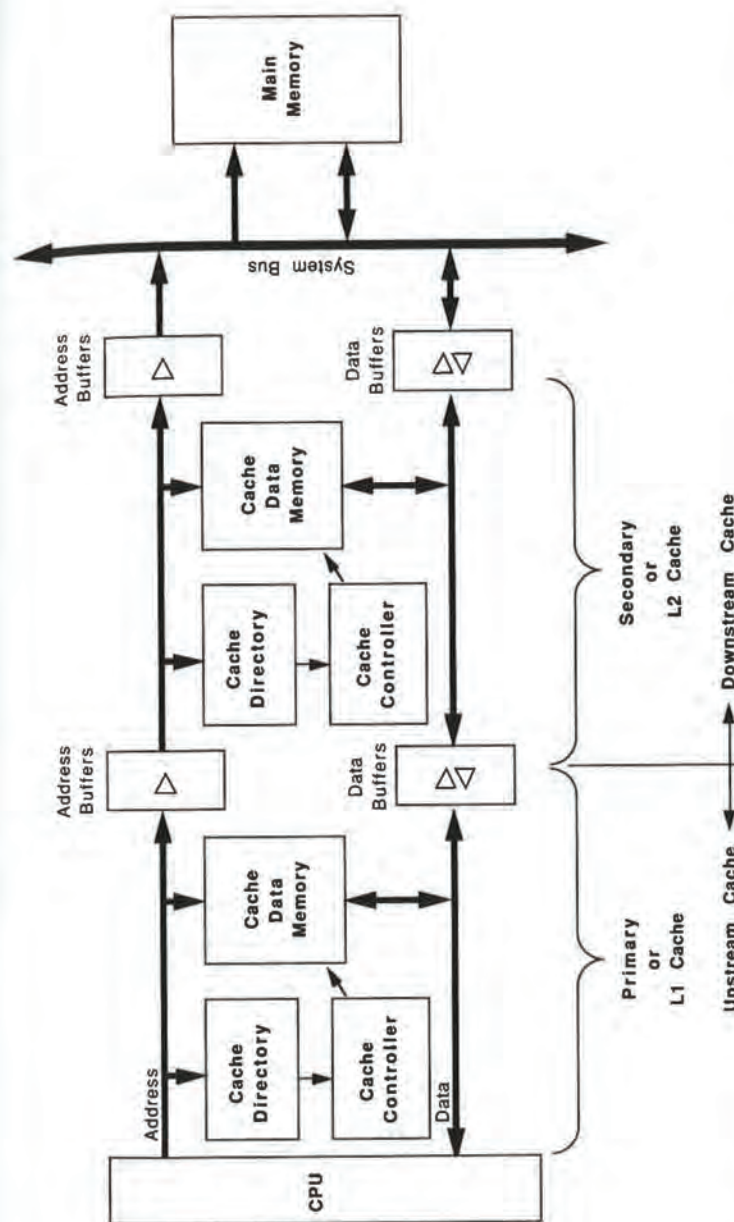


Figure 2.23. Primary and secondary caches. Note how little difference there is between any two levels of cache from the perspective of so simple a block diagram. Any number of levels of caches can be implemented.

possible using a write-through cache, so designers of such systems often choose to supplement the on-chip write-through cache with a secondary copy-back cache.

Naturally, the design of the secondary cache depends greatly upon the kind of traffic expected out of the primary cache. It is safe to say that the activity demanded of the second cache will look nothing at all like that required of a primary cache. A secondary cache is generally idle, while a primary cache is almost never idle. The balance of read to write cycles can be completely reversed between a primary and a secondary cache, the hit rate will probably be significantly lower, and the sequence of requests will probably exhibit a much more random behavior, with both temporal and spatial locality playing a smaller role. How can it all be so different? We'll examine this one piece at a time.

First, let's assume that the processor normally would perform one write cycle for every ten read cycles (this once again is a function of the code being run on the machine) and that the primary cache is a write-through design which satisfies a flat 90% of all read cycles. This is a pretty typical scenario. Since the design is write-through, the primary cache will not intercept any of the write cycles, so that 10% of the processor's cycles automatically get passed through to the secondary cache as writes. How many read cycles does the secondary cache see? Well, if the other 90% of the processor's cycles are reads, and 90% of these are satisfied by the primary cache, then 10% of them make it to the secondary cache, or $10\% \cdot 90\% = 9\%$ of the overall CPU cycles. This means that the secondary cache will see more writes than reads, and, if the primary cache has a higher hit rate than 90%, the balance will be even more skewed toward write cycles. This flipflop would not occur in a system using a copy-back primary cache; however, the writes from the primary to the secondary cache would only occur upon cache evictions, so there would be no bearing for temporal locality as far as write cycles were concerned. The eviction might occur either a very short or a very long period after the write cycle actually had occurred.

Downstream caches never exhibit anywhere near the hit rates of the primary cache, and with good reason. As we saw in Figure 2.9, the cache's hit rate tapers off with increasing cache size. The improvement in hit rate can be viewed as a differential, $\Delta H / \Delta S$, where H is the hit rate and S is the size of the cache. If the primary cache offers a hit rate of 90%, and the secondary cache brings the hit rate up to 95% (Figure 2.24), then the secondary cache is only cutting the miss rate in half, thus exhibiting a hit rate of 50%. This brings up questions about the value of secondary caches. Why pay for so little an improvement? If we look at the problem from the bottom up, we see that the miss rate of the single-level cache system is 10%, and the miss rate of the two-level cache system is 5%, meaning that the bus traffic in the two-

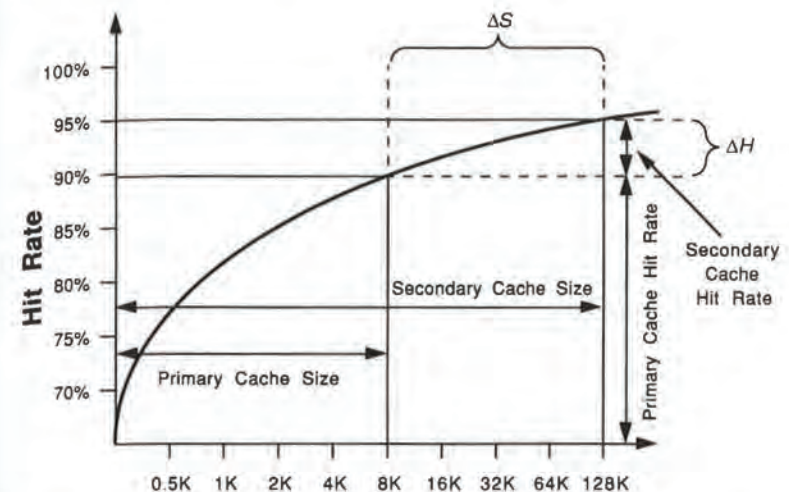


Figure 2.24. Incremental hit rate of a secondary cache. A very large secondary cache can exhibit a really small hit rate in comparison to the hit rate of the smaller primary cache. Most designers focus on miss rate improvements in order to rationalize using multiple levels of caches.

level system is half that of the traffic in the single-level system. This is a key improvement in multiprocessing systems, as was discussed in Section 1.8, and is also very important in systems which have long main memory latencies. (By the way, Figure 2.24 is fictitious data, so don't try to use it to prove any points or to design your own cache.)

If the downstream cache has the same line size as the upstream cache, then it stands to reason that the only hits the downstream cache will have are from the upstream cache misses which are caused by conflicts. This is simple to see. The secondary cache will only be filled with the same data as is requested by the primary cache at the same time that it is being loaded into the primary cache. The only way that this data will stay in the secondary cache but will be removed from the primary cache is if the primary cache has a conflict which causes its copy of that line to be overwritten, yet the same conflict does not occur in the secondary cache, due to either a larger line size or a larger address space. For this reason, the downstream cache should either be considerably larger than the upstream cache or should use a line size much larger than its upstream neighbor.

Some designers choose to assure that the contents of a primary cache are excluded from being contained in the secondary cache. Why hold redundant data in both of these places? This is a more efficient use of the two

caches, but sometimes this added efficiency gets in the way of the implementation of other good cache policies. We'll examine one called "inclusion" in Section 4.1.1. Sometimes, too, the exclusion of primary cache data from the secondary cache becomes extremely difficult, and the alternative not only shortens the design cycle, but also might reduce the complexity and chip count of the cache control logic.

Secondary caches can also be used to implement good screening mechanisms in multiple-processor systems, a topic which will be described in depth in Section 4.1.3. In a nutshell, the less often a system intrudes on the CPU/primary cache subsystem, the faster that subsystem will operate. A secondary cache can filter system intrusions from the primary cache and only allow interaction by potentially meaningful transactions.

2.3 STATISTICAL PREMISES

Figure 2.9 was used to illustrate both the diminishing returns of increasing cache size and the hit rate improvements offered by increasing the associativity of a cache of a given size. Although the figure gives a good impression of the potential of cached systems, it is not at all accurate. True cache statistics are highly dependent upon the exact target system, both hardware and software, and there is no substitution for determining the exact characteristics of your own system. Unfortunately, most designers don't have the resources available to test the various combinations of caches which they might implement, so a graph like this one might be the fallback for certain design decisions.

In general, the curves are a reasonable estimate of the performance to be expected of a cached system. Several papers have been written in which actual measurements are provided, and they usually don't follow curves as smooth as those in the graph. Many exhibit lumps which may be attributable to the maximum size loop which a programmer is comfortable writing, or limits to the size data set being used, and some performance curves actually cross each other, showing a cache using certain policies performing better than another for small cache sizes, but worse for larger caches!

2.3.1 The Value of Choosing Policies Empirically

If we are to be purists, the cache must be designed only after significant statistics have been collected to indicate the performance trade-offs of certain policies and cache sizes against all others. More probably, the designer will go on from this text to read a few of the available papers, then make decisions based on statistics which were measured upon an altogether different system running software completely unlike the programs the target system is to use.

This can lead to bad decisions, since the statistics in these papers have been collected on systems which are so completely different than the current design that some of the more subtle trade-offs which were determined to make the paper's system perform better might actually impede the operation of the newer cache. Real-time multiprogrammed systems tend to do better with higher associativity levels than are required in simple single-tasking operating systems like MS-DOS. The main memory bus protocol can have a significant impact on the choice of replacement algorithms.

The following section should be discussed with the system programmers in the hopes that they can help you out by providing tools to help measure the cache's true performance.

2.3.1.1 Simple Methods of Measuring Performance Software modeling is an inexpensive way to test possible cache strategies against each other. Usually, the target processor is used in an uncached system, with a simulator actually intercepting all memory accesses. The simulator is made to emulate different sorts of caches, and the model is made to run programs which will be run on the target system. A great disadvantage to software modeling is that it runs more than an order of magnitude (and quite possibly two orders of magnitude) more slowly than the same program would run on the uncached system without using the model. This is pretty easy to understand, since each instruction of the target code must be hand-fed via the model software to the processor, and statistics must then be stored for each reference made. The large addition to the execution time means that the exercise of trying several different cacheing strategies on a large software base can consume quite a bit of time. Nonetheless, the low cost and ease of implementation of this method make it a popular alternative.

A less popular and considerably more expensive alternative is hardware modeling. A typical hardware cache simulator consists of the most full-blown cache the designer could hope to use, with the addition of switches to change or remove one variable or another (examples might be cache size, write policy, or associativity). The target programs are then run with the various features enabled or disabled, and the resulting execution times are used to decide upon a system's implementation. Naturally, such a system can take a considerable effort to debug. To simplify the designer's life, such systems are sometimes run at a scaled speed, like half or one-quarter of the final system clock speed. As long as all parameters are scaled equally (CPU clock, main memory latency, bus clock, etc.), valid measurements can be obtained, and good decisions can be made regarding the optimum cache strategy.

Two other simple methods involve the use of traces of example executions. There are two sorts of traces: hardware and software. Hardware traces usually are performed by placing an apparatus upon the bus which will measure real-life address activity in such a way that statistics can be compiled to

reveal an optimum cache strategy. One method is to make a hardware model of the example cache-tag RAM for a possible alternative and to measure the miss rate it would have. The cache-tag RAM only has to run as fast as main memory, so making one which allows the simulation of several different cache architectures is a simple task. Although it would be reasonably easy to design another sort of trace machine which counted simple statistics on the number of main memory accesses within certain address ranges (possibly by using nothing more than a logic analyzer), these statistics reveal nothing about the locality of the program being run, and end up not being very useful.

Software traces, often called code profiles, are even simpler to implement: however, they lack somewhat in resolution. Those used in most cache research are implemented in microcode, but can only be run in systems which allow the microcode to be modified. The simplest sort of software trace is driven by the nonmaskable interrupt in a real-time system. If the system being used for traces doesn't have an interrupt of this sort, it is often not too difficult to set one up. At the beginning of the interrupt routine the program counter is pushed onto the stack. The trace routine is layered into the normal interrupt service routine, and looks at the stack to keep statistics on the location of the program counter. The program counter is not the only address generator which can be traced by software tracing. All other pointers can be examined as well, once again at the expense of latency in the interrupt service routine.

As with the simpler hardware method just mentioned, care will have to be taken to assure that the interrupts are not so widely spaced that issues of locality are lost. On the other hand, causing too many interrupts might slow the system down to the point where it would operate as slowly as a software model.

One good reason to use traces is if the scenario in which the target system is to run cannot be simulated adequately at a much lower speed and if the cost of a hardware model cannot be justified. Examples include a single node in a network of other computers or a widely used multiuser system, either of which depend greatly upon timely interaction, with uncontrollable outside sources to be measured realistically.

Certain compilers support profiling by inserting extra instructions into the compiled code when a profiling option is invoked. This is a great way to gather information about the program counter and what the code is doing, but does not often do a lot to help you see what's happening in the data space. If your main thrust is to optimize code accesses, this is a workable solution. At least one programmer I know of profiles his code by measuring the length of loops using a ruler on an assembler listing (generated either by a compiler's output, or from disassembled object code) and figuring the aver-

age number of times each loop is executed. He can use this to determine the percent of time the program counter resides within any single location.

2.3.2 Using Hunches to Determine Policies

Of course, now that you have an understanding about how you can measure your own cache statistics, that doesn't mean that you are going to do it. Maybe you don't have the time, or, quite possibly, your marketing department has told you the only features which a customer will buy (sadly enough, hardware specifications are usually more deeply scrutinized by potential customers than are benchmarks). This is where you must depend upon your (or somebody else's) intuition.

For those who haven't the time, I hope that the intuitive approach taken in this book is found to help. As an example, it should be pretty clear from what has already been discussed that a design which falls downstream of a write-through cache should focus more attention on write cycle performance than on read cycles, or that an increase in the associativity or even size of a write-through cache often pales in comparison to the benefits (or complexity) of converting the design to a copy-back cache, or that wait states are so bad that there is no cache strategy which is worth incurring one. I should also hope that the reader has the time to read some of the more academic studies on cache performance. Although this book does not try to categorize which papers are of value, there are plenty of them out there, and it is a simple task to find one paper you feel good about and to get copies of every paper in its bibliography. Still, remember that the papers were written about a different system than yours, running different software on a different CPU across a different bus structure. In cache design, the only absolute truths are the ones describing how your cache works with your software on your system. Nothing else really matters. One step to be sure of is that you argue every decision with your peers. Nothing helps work out the bugs in advance more than having to defend your assumptions.

For those of you who have to design a cache to some arbitrary specification because somebody told you to, don't despair. Look a little deeper. Maybe there is something you learned in this book which can be coupled with the restrictions you have been placed under which can turn a less optimal strategy into a near-stellar one. You are probably learning that there are so many variables in cache design that there must be one or two options you have so far neglected to take advantage of. One of the nicer points is that many of these options require little or no additional hardware and can be had for the simple price of a little more effort in the design/debug cycle.

2.4 SOFTWARE PROBLEMS AND SOLUTIONS

So far, we have assumed that the cache designer is burdened with solving the cache design problem without any help from the software side of the house. This usually happens to be the case, but in those instances where the programmers and designers are able to work together to solve the problem, there are certain simple rules which can be used to drastically simplify the design of the cache.

Most often, the real challenges encountered are due to the fact that the cache is being designed to accelerate the performance of a system which has been around for several years. The software is already written in a certain way, and the cache might need to maintain compatibility with existing hardware which has anomalies which must be accommodated. In altogether new designs, however, the cache designer has the luxury of working with the design team to help define the overall system specification.

2.4.1 Trade-offs in Software and Hardware Interaction

It is easier to understand hardware/software cache design trade-offs with some examples. One good example is the problem of cache validity, as described in Section 2.1.3. The software solution is to disallow the cache from satisfying any CPU bus cycles until a bootstrap routine has made certain that all cache lines have been overwritten with valid data, after which the same program sets a flag which enables the cache to respond. The hardware solution is to maintain a Valid bit for every line and to reset all Valid bits through a hardware reset mechanism before allowing the cache to respond. Obviously, the hardware method will be more costly to reproduce.

This sort of valid/invalid criterion is not only a problem at startup, but also occurs during direct memory access (DMA) activity, where an outside device modifies the contents of main memory, but not necessarily the corresponding contents of the cache memory. The brute-force hardware method of handling this problem is to invalidate the entire cache whenever the bus signals supporting DMA write cycles occur (several more elegant methods will be detailed in Chapter 4). The software method of handling the problem is to have the operating system invalidate either the entire cache or the appropriate section of the cache whenever a context switch is invoked.

Another example is a bit tougher to explain here, mostly because it involves multiprocessors, the subject of Chapter 4. In a multiprocessing environment, processors often communicate via locked read/modify/write instructions. One processor writes to a main memory mailbox location to tell another processor what it is doing. This problem is similar to the problem of I/O addresses mentioned in Section 2.1.3. If the processor which is to

read the mailbox location refers to a copy of the location in its cache instead of reading the actual main memory, the main memory will be updated by the writing processor without the reading processor ever noticing. The software solution to this problem is always to map the mailbox locations to the same physical memory address. It will then be simple for the cache designer to disallow that address range from being cacheable. In multiprocessing systems which are not in control of the mailbox locations, some extremely sophisticated intercache communications protocols must be worked out. These are sufficiently complex that I won't go into them here, but will wait until we arrive at Chapter 4.

2.4.2 Maintaining Compatibility with Existing Software

If you review several cache designs, you will start to notice the bizarre twists which sometimes must be added to account for ways in which existing software has been written.

Probably the biggest headache for many cache designers is the software timing loop. The timing loop runs at exactly the right speed as long as the processor is clocked at exactly such and such a frequency, and might even depend on a certain main memory latency. This type of programming pretty well does away with the possibility that the hardware can undergo improvements. The only really clean way to deal with such a problem is to nail down the location of the timing loops for the program(s) under question, and to disallow these from being cached, meanwhile assuring that the processor clock is never improved. A simpler alternative is to allow the user two operating modes. It has become the terminology in the IBM PC world to dub the two operating modes in systems so equipped as normal mode and turbo mode.

In certain real-time systems, interrupt latency must be exactly the same, no matter when the interrupt occurred. This is not easily accomplished in cached systems. Say the interrupt service routine were written to count the number of interrupts and to respond to the interrupt with a high or low output, depending on the previous interrupt activity. Naturally, the interrupt routine and the record of previous interrupt activity will be stored in main memory. The routine runs the possibility of thrashing on certain interrupt cycles and not thrashing on others, meaning that the interrupt latency will be variable. Add to this the chance that the program being run between the interrupts will at some times overwrite some of the interrupt routine and at other times will leave it alone. Two simple approaches to this problem are to disallow the cache from working during the entire interrupt response or to disallow the caching of those locations involved in the interrupt service routine. The latter solution might be a little too drastic, since the interrupt

service routine, at a minimum, involves not only the service routine's code space and an associated data space, but also the stack. Disallowing the cacheing of the stack will have a negative impact on the performance of the entire program!

For those who have chosen to use a logical cache implementation, care must be taken to understand any tricks which programmers may have played with the logical to physical address mapping. Address aliases can be used very artfully to improve the communication between routines, but they are really tough to account for in a logical cache design. Once again, a heavy-handed, but useful, approach is to simply disallow the cacheing of any page which might be used in this manner at some time.

2.5 REAL-WORLD PROBLEMS

Compared to the rest of the book, this is a really mundane section. Here, we will not cover the tricks and techniques of artful cache design, but will tackle the tougher problem of getting the hardware to run, and to run reliably in such a way that it can be mass produced.

It's pretty reasonable to assume that the cache designer is going to use the fastest processor available. After all, caches are a way to make up for the disparity between true main memory access time and the access time required by the CPU to run at maximum throughput. This disparity does not become a problem until the designer uses the fastest CPU speeds. Also, the cache is usually a more expensive means of increasing throughput than is the use of a faster CPU (although this might not be the case, as was shown in Figure 1.1).

At the fastest CPU bus speeds available, the CPU/cache subsystem cannot be examined only as a digital system with the problem of logic bugs, but the timing must be scrutinized and derated, circuit boards must be designed to accommodate high frequencies, and the designer must be resigned to a lot of sleepless nights.

Most high-speed system designers allow themselves time to get the system working thoroughly at a scaled-down speed, like 10% of the maximum CPU speed, then to gradually increase the clock frequency to full operation until all timing bugs have been found and removed. Although this sounds slow, less methodical methods seldom can be used more profitably.

2.5.1 Parasitic Capacitance and Bus Loading

Designers who are working for the first time with cycle times of under 50ns are often surprised at the care and attention required by high-speed sys-

tems' bus signals. At lower speeds, the few nanoseconds consumed by capacitive loading might have been mentioned in a thorough timing analysis (if one were performed), but could easily be accommodated by using a slightly faster, slightly more expensive part. At 20ns cycle times, once the processor's output delay and set-up times are accounted for, the designer finds that the cache will require the use of leading-edge component speeds even before derating calculations are made. Inaccurate derating can hurt in two ways. Overoptimistic derating can cause the cache simply not to function, whereas overconservative derating might keep the cache from ever being designed, possibly allowing a competitor to beat you to the punch.

High-speed processors tend to be specified into small (50pF) loads for three reasons. First, processor manufacturers don't want to lose yield simply because they tested their parts under overly stringent loading, possibly on a pin which nobody uses anyway. This is a way for them to cut themselves some slack. Second, to drive large output loads at high speeds, high-current output drivers must be used. The consequences of designing these high-current drivers into a processor chip are many. All ICs are designed within strict power budgets. Fast circuits require more current, so the more of the budget's current which is diverted to the output drivers, the less there is available for the faster parts of the processor; hence, the processor must be made to run more slowly. That's not a good deal for a processor which is being pushed to the highest speeds. High-current output drivers also generate a lot of noise on the processor's internal ground, which can confuse some of the internal thresholds, leading to slower performance or even bit errors and lockups. The last reason for the CPU manufacturer to specify outputs into a light load is that the test equipment tends to come from the tester manufacturer looking like a light load, and the addition of heavier loading is somewhat of a burden and a puzzle to the CPU manufacturer. If additional loading is to be put onto the tester, what exactly should it look like? It seems that no two system designers can agree on what the "typical" output load looks like to a CPU chip.

A generally accepted method of derating is to add 1ns of propagation delay to a signal for every 20pF of loading above the specified output loading of the processor or other driving device. This works for light drivers like CPUs and memories, but some logic outputs can drive either more or less, and the device's data sheets should be consulted if that data is given. Although I've never heard a consistent number to use for the capacitive effects for printed circuit card traces, one number is 1pF per inch. This adds up really fast, so careful layout is a must.

As an example, let's derate a CPU output which is driving the address inputs of four 128K $\times$ 8 cache data RAMs and three 128K $\times$ 8 cache-tag RAMs, plus an address buffer along a 30-inch trace (this sounds huge, but is about

Table 2.1. Capacitive derating on the address bus of a typical cached system.

Load	Capacitance (pF)	Notes
Tag RAMs	21	Three 7pF inputs
Data RAMs	28	Four 7pF inputs
Address buffer	7	
PC board traces	+30	30 in. · 1ns/in.
Total capacitance	86	
Less: CPU rated output drive	-50	
Derating capacitance	36	
Times: Derating factor	*0.05ns/pF	1ns for every 20pF.
Derating	1.8ns	All tags, etc., must be 1.8ns faster than figured in timing calculations without derating.

as short as you can run such a trace on a two-sided PC board). The address buffer and all the RAMs will be given 7pF input capacitances, and the processor is specified into 50pF. Table 2.1 shows how the typical derating equation would look for the lower address bits in this system.

Why does this only apply for the *lower* address bits? Remember that these are the set bits, so each set address bit must be routed to the address inputs of all the tag RAMs plus all the data RAMs. The tag bits only must go to one tag RAM and the address buffer (unless a discrete comparator is being used, which will add a load). Remember that no tag bits are used in accessing the cache data RAM. Also, the loading will be different for the address bits which are lower than the set bits in a system using multiword lines or sectoring. Further, bidirectional buses must have these equations performed for each driver on the bus. This may sound as if it only pertains to the data bits, until you realize that the addresses in a copy-back cache are obtained from the tag RAMs during line evictions. We will see in Chapter 4 that there can also be episodes in which the CPU/cache subsystem's address inputs will be driven by the address buffer. There will be an awful lot of these little calculations in a high-speed system!

One method to get around some of these derating issues is to use **multi-chip modules (MCMs)**. Multichip modules are a hot topic of conversation at the writing of this book, but have not been put into widespread use. The Intel Pentium Pro is one of few widely available MCMs. In an MCM, the CPU, the cache, and all buffers to the outside world are mounted onto a substrate, preferably without previously having been packaged. The advantages to this technique are that the input capacitance is lower for devices which are not packaged, and the module substrate is capable of carrying signals using shorter connections at a lower capacitance per inch. Further, if all of

the signals which are not required to be placed on a module output pin are allowed to use smaller I/O swings, the charge/discharge cycle of the node capacitance becomes smaller, allowing the module's chips to be run at a higher speed. The downside to MCMs is that the manufacture of modules is costly and tends to require the use of sole-sourced RAMs and logic. Add to this the cost of rework should a single chip upon the module fail, and the costs go through the roof! Testability is also a big problem. In the near term, it appears that MCMs will remain the domain of vertically integrated firms' highest performance and highest cost systems. A reasonable compromise is to use a carefully designed module made of packaged parts on a standard circuit card, as is being done by Intel with their standard processors.

Another nagging real-world concern is clock skew. Clock skew can result from mechanisms as obvious as the use of different buffer outputs to drive two different devices' clock inputs, usually motivated by heavy loading on the clock lines. The worst case of this is if two different buffers in two different packages are used to drive two versions of the same clock signal. One device may be particularly fast and be mounted in a cooler part of the board, while the other device is allowed to get warm and, even at cooler temperatures, runs at the slow end of the devices shipped by the component manufacturer. Heat slows silicon down. Less obvious are the times that clock skew comes from mismatched loading on two identically buffered but separate clock lines, or from deviations in trace length between two clock paths. Sometimes, the skew which causes the problem comes from two different taps on the same signal line, where these taps are simply too distant from one another. The methods often used to prevent clock skew from becoming a problem are to drive carefully placed underloaded clock lines with hefty signals all coming out of the same package. Some designers simply use the same octal buffer for all clock outputs (pin-to-pin skew is not tested or guaranteed on these devices), but there are devices available which are designed and tested specifically for this function, which have much stronger current drive than the octal buffer, and are tested and guaranteed to maximum pin-to-pin skew. Certain clock driver circuits go so far as to use a phase-locked loop to synchronize one of the clock driver's output pins with the reference input.

Resistive loads can also help systems avoid pitfalls due to delay-line effects. Any conductor, unless properly terminated, will reflect power back in the direction from which it came. This is fundamental transmission line theory. If both ends of a PC board trace are improperly terminated, the waveform can adopt a severe ringing which may even pass back and forth across an input threshold, causing false triggers of registers, erroneous write cycles into RAMs, or even a reading as the opposite logic state of the value represented on a data line. Once again, there are many schools of thought on this subject, all of which will be left to better qualified sources to explain. PC

board layout techniques are neither the forte of the author nor the subject of this book. Still, don't overlook this important issue.

Designers whose products are aimed at the U.S. market must worry about complying with Federal Communications Commission (FCC) radiofrequency emissions regulations. This ties in with the clock skew and termination issues just discussed and is a challenge which will be noted but not discussed here. Let's just say that it is one more in the list of concerns facing the high-speed cache designer.

2.5.2 Critical Timing Paths

Figure 2.10 shows the critical timing path of the cache-tag RAM in a direct-mapped cache. For most systems, this is the real hot spot and will be the most difficult timing problem to solve. The use of integrated cache-tag RAMs and cache controllers is often determined by the speed required of the cache-tag RAM. In some architectures, the cache controller is a single chip which accepts address inputs, and outputs a Ready signal back to the CPU. Although this is a fast approach, it is usually implemented using ASICs, so the cache-tag RAM is never as large or as fast as the state of the art in static RAM design. Other implementations involve putting all of the cache control logic except the cache-tag RAM within the CPU chip itself to reduce chip transitions, or to include the tag comparator into either the cache controller chip or the cache-tag RAM chip. Only the slowest designs have the luxury of allowing the use of a discrete cache-tag RAM, followed by a discrete comparator, followed by the cache control logic.

One observation which pops out when considering the problem of designing a cache-tag RAM and downstream logic with sufficient response time is that the problem might be difficult for direct-mapped caches, but with multiway caches it becomes really nasty.

Back in Section 2.2.2, we saw that the tag on a multiway cache is used to gate the output of the data RAM, whereas the data RAM in a direct-mapped system starts the cycle in an enabled state. An approach used by at least one cache controller is to implement the two-Way architecture using a **most recently used (MRU)** bit to signify the Way most recently accessed for the line being read. The MRU bit is just an inverted version of the LRU bit discussed in Section 2.2.2. Using this approach, the cache data RAM starts the cycle with the most recently used Way's outputs enabled, just as a direct-mapped cache starts a cycle. This allows the cache-tag RAMs to be as slow in this sort of two-Way design as they would be in a direct-mapped architecture. Time is only lost on a Way miss where the Way which was initially enabled is the wrong one, and the CPU is delayed until the proper Way can be selected. It

is easy to conclude that the highest miss rate which would be likely to occur in such a cache would be about 50%, and, with the operation of this particular design, that Way miss only costs a single cycle. Intel's designers seem to believe that the Way miss rate is actually significantly lower than 50%. The actual performance benefit of a two-Way architecture over a direct-mapped version would need to overwhelm the detriment to overall cache bandwidth of these added wait states in order for this concept to improve system performance. It is not intuitively obvious that the MRU-based, two-Way cache would outperform an equivalent-size direct-mapped cache. Only measured statistics could prove or disprove this argument.

Other vexing timing difficulties can occur in the data loop. Figure 2.10 shows the path of the addresses out of the CPU, through the cache data RAM, and into the processor's data input pins. If the processor has no internal cache, it will tend to consume a few CPU clock cycles every time it requests memory data, and the cache data RAM's timing might be tight, but not impossible. Processors which have internal caches tend to use multiword lines and fetch the entire line at one time, using the burst cycle described in Section 2.2.5. Naturally, there is a speed benefit of allowing the burst refill to run just as quickly as possible, so the designer will attempt to implement a zero-wait data RAM in the design. The first cycle is easy to match. The address is output over a full CPU cycle time before the data is required. However, the address stays valid until after the first word is fetched, skewing the timing on the other cycles to the point that the access time of the remaining cycles is the CPU clock cycle time less the CPU address output propagation delay and the CPU data input set-up time.

Some designers use interleaving to solve this timing problem, where the cache or main memory is four times as wide as the CPU's data bus, and multiplexing is used to perform the burst sequence automatically, based on the CPU clock rather than on the CPU address outputs. (A DRAM which is interleaved this way will usually have a couple of wait states only at the beginning of the cycle and will thus be called 4:1:1:1.) Other designers place outboard burst counters between the CPU and the RAM, so that they can reduce the clock to burst count propagation delay. Today's most popular cache data RAMs have included the counter on-chip.

Back in Section 2.2.6, we spent a lot of time and buzzwords on describing the use of line buffers and early continuation (where the CPU is allowed to continue to operate as a multiword line is being updated in the cache). It might save you some time and effort to find out whether or not the internal cache of the processor you're designing with uses early continuation. As a case in point, one acquaintance of the author went to lengths to design a secondary cache around the Motorola 68030, which supported the 68030's burst cache line refill mechanism. Oddly enough, the burst design *under-*

performed a similar design without burst support. Why? Because the 68030's internal cache does not support early continuation. To better understand the problem, look at Figure 2.25. If all instructions execute in the minimum amount of time, and if the instructions all execute in line without intervening accesses to other addresses, as is shown in the figure, the burst will work no faster than the discrete cycles. In a lot of instances (probably 75%, since the line is four words long), however, only a portion of the line will be used before the processor needs to access a completely different address. With the 68030's nonblocking design, the burst must be completed before the CPU's need to look elsewhere can be accommodated, thus stopping the CPU from performing any work during this contention.

Another point of difficulty in the design of high-speed caches is the production of well-timed write pulses, especially for cache write cycles which are allowed to occur at zero wait-states, either because the cache adopts a copy-back policy or because a write-through design employs a write buffer. There are two sources of problems with high-speed write cycles: skew and noise. Write pulse skew is caused by the use of separate silicon to time the write pulse and the data and address inputs to a RAM. If the write pulse is generated by the same clock on the same piece of silicon as the data and address, all of these signals move together over temperature, voltage variations, and IC manufacturing process variations. No matter what the environment, the timing is accurate to a fare-thee-well. In systems which buffer the address inputs or the data I/O to the cache, or systems which use different silicon to generate the cache data RAM's write pulse than that used to generate the RAM's address and data inputs, the designer must assume the possibility that these different ICs will come from different manufacturers and will exhibit different propagation delays, different temperature tracking (not to mention that each chip will probably be at a different temperature at all times), and all other worst case possibilities which will cause the data, address, and write pulse inputs to arrive at wildly fluctuating intervals with respect to each other. A common way to get around this problem is to use synchronous static RAMs in the cache design, since the synchronous SRAM uses no write pulse, but instead uses a separate clock input to sample address, data, and write enable inputs all at once. We won't go into synchronous SRAMs in depth here, but the timing waveform of a write cycle of a synchronous SRAM is shown in Figure 2.26, and it is easy to see that the write cycle can have a lot of margin and still be performed at very high speeds.

Synchronous SRAMs also neatly solve the noise aspect of high-speed write pulses. Assume that you have succeeded in collecting the appropriate logic to precisely time the write pulse to the address and data inputs to the cache data RAM. At 50 MHz clock frequencies, the transmission line effects of a printed circuit card become significant, and fastidious board layouts be-

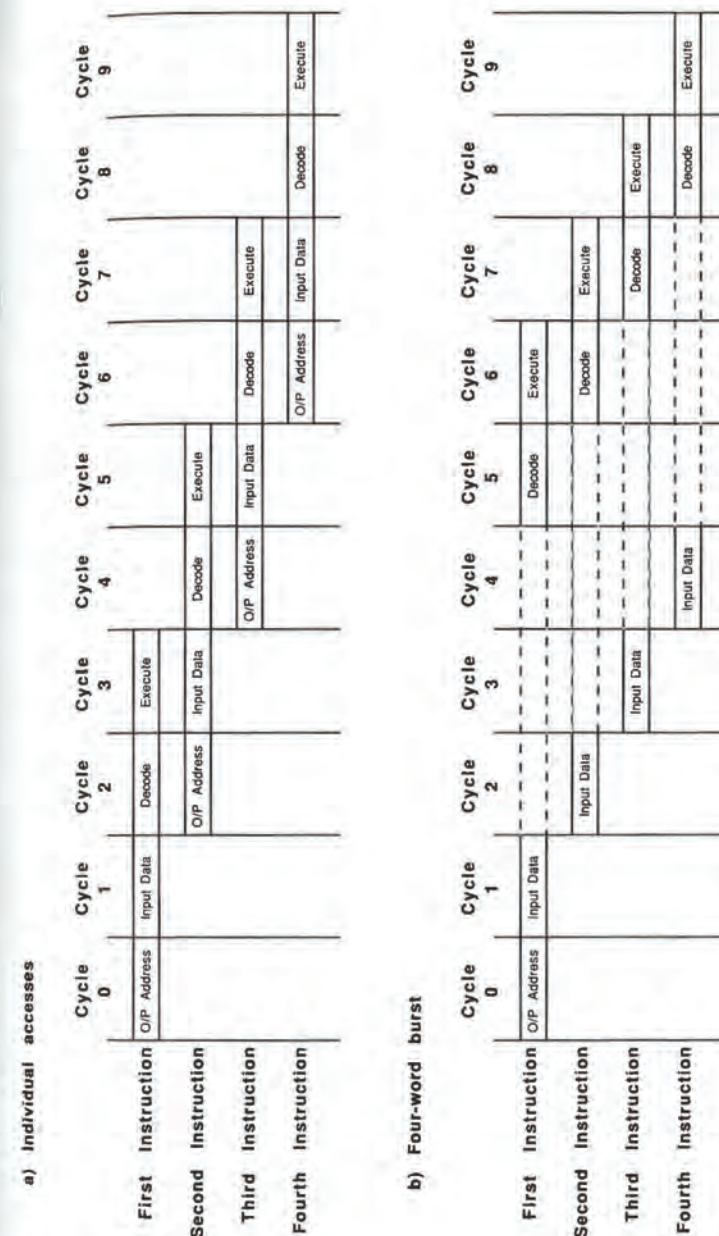


Figure 2.25. Why bursting in nonblocking caches is no faster than individual zero-wait accesses. Note that the first instruction in the burst sequence in (b) is not even decoded until cycle 5, resulting in the timing of the fourth instruction to be the same for burst (b) and nonburst (a) approaches. If the processor doesn't need all four burst addresses, then the instructions which were requested will not be decoded in such a cache until all of the unnecessary addresses have been fetched.

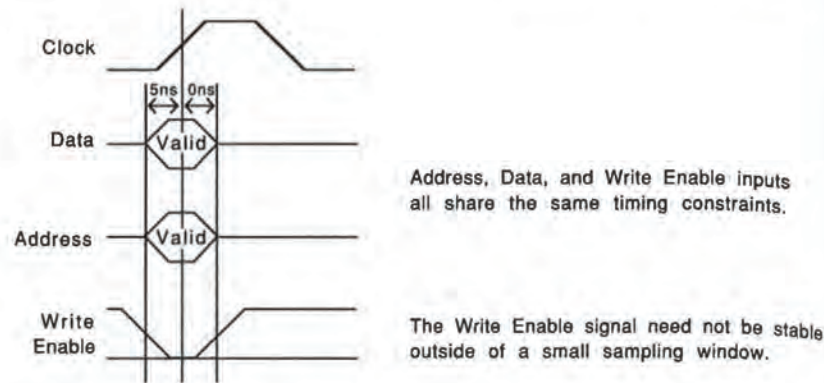


Figure 2.26. A synchronous write cycle. The write enable input is not sensitive to random edges or ringing which occur outside of the sample window. Also, write cycles can be decided upon much later in the memory cycle with synchronous parts.

come a must, as well as proper termination with impedance-matched loads at both ends of the write pulse line. Still, reflections will occur at every change in the trace due to a solder joint, a device pin, or even a corner of the trace, and when these reflections are added to the original signal, they might cause the write pulse to appear to ring, not being stable as a high or low level at the onset of the write cycle, and hanging around or recurring after the cycle is over. This causes headaches because the write enable must be a precisely timed pulse, rather than a voltage level. Synchronous SRAMs look at the write enable input as a level, which can go through countless transitions before the sampling window and which can bounce up and down without consequence after the window has passed.

2.5.3 Bus Turnaround

There are three main reasons that a cache design would have a potential for bus contention. The first is if the cache design satisfies a burst cycle using interleaved RAMs. The second is the case where a multiway architecture is used, and the cache controller must decide, in the minimum amount of time, which of two RAM banks is to be used to satisfy a data request. The third is in caches which interleave data and instruction caches to simulate a Harvard architecture. In all of these examples, the problem is to turn one RAM off and the other on in the minimum amount of time. Overlap of the outputs of two or more RAM banks is to be avoided at all costs, since the result-

ing contention causes high power dissipation, ground noise, radiofrequency interference (RFI), stress on the RAMs' output pins, and other concerns.

One problem with attempting to interleave RAMs this way is that the specifications for static RAMs are not adequately tight to ensure that bus contention will not occur. Some manufacturers are addressing this issue by guaranteeing minimum and maximum turn-on and turn-off times, but this is the exception, rather than the norm. Because of this, designers are often forced to assume that the turn-on and turn-off times will lie anywhere between zero and the maximum specified for the part. More venturesome designers assume that the parts cannot respond in 0ns, and will find some number which makes them comfortable, but the margin in the maximum and minimum numbers still slows down the interleaved system unless the designer chooses to ignore the effects of bus contention (at least until the system needs to be debugged).

Certain CPUs use separate data input and output pins, at quite a cost in package pin count, just to get around the problems of contention caused by bus turnaround, when the CPU stops inputting data and starts to write. This approach helps to solve the problems of synchronizing the outputs of the CPU to the outputs of the cache data RAM.