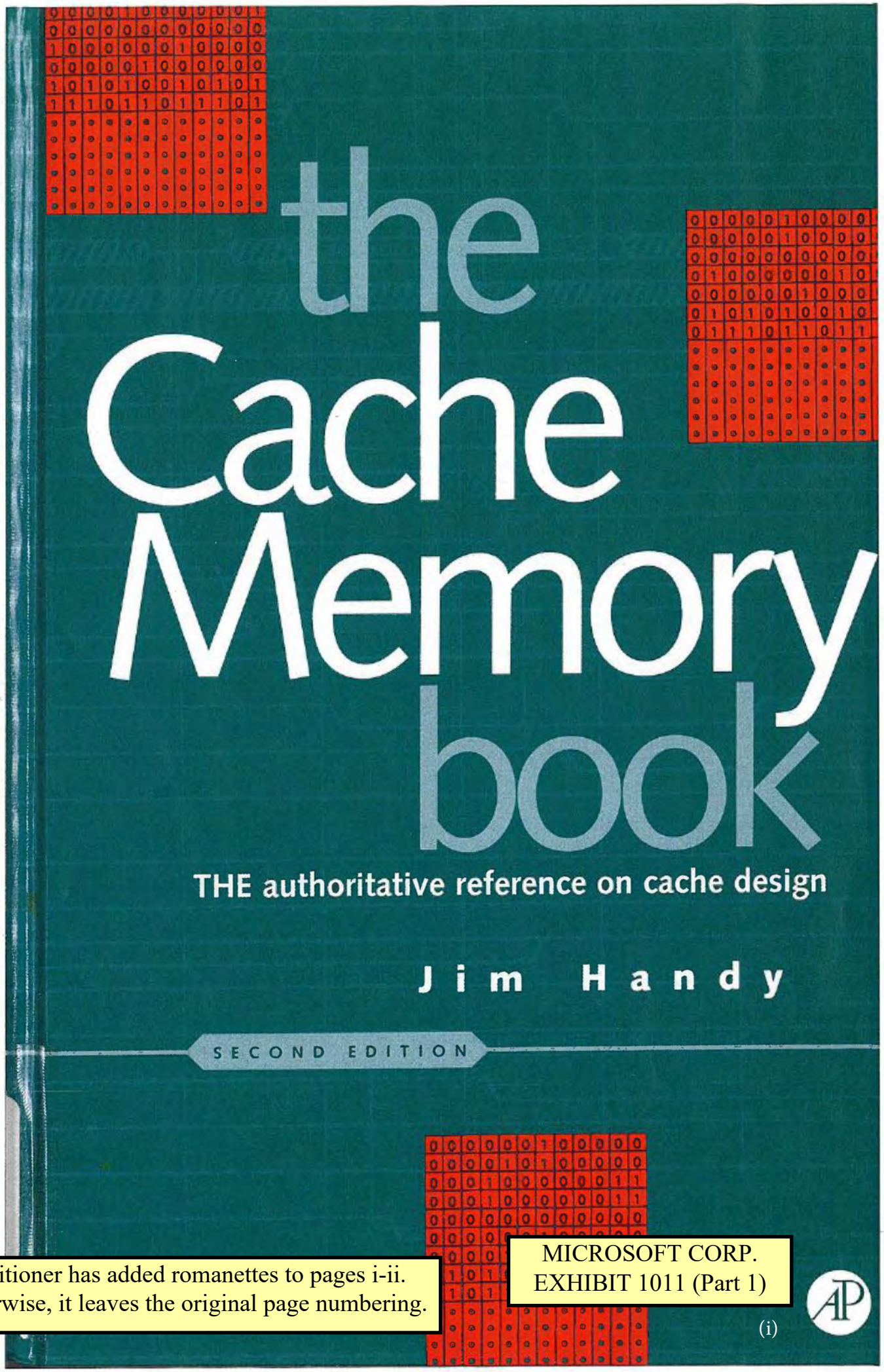


Petitioner has added romanettes to pages i-ii. Otherwise, it leaves the original page numbering.

MICROSOFT CORP.
EXHIBIT 1011 (Part 1)

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

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

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.

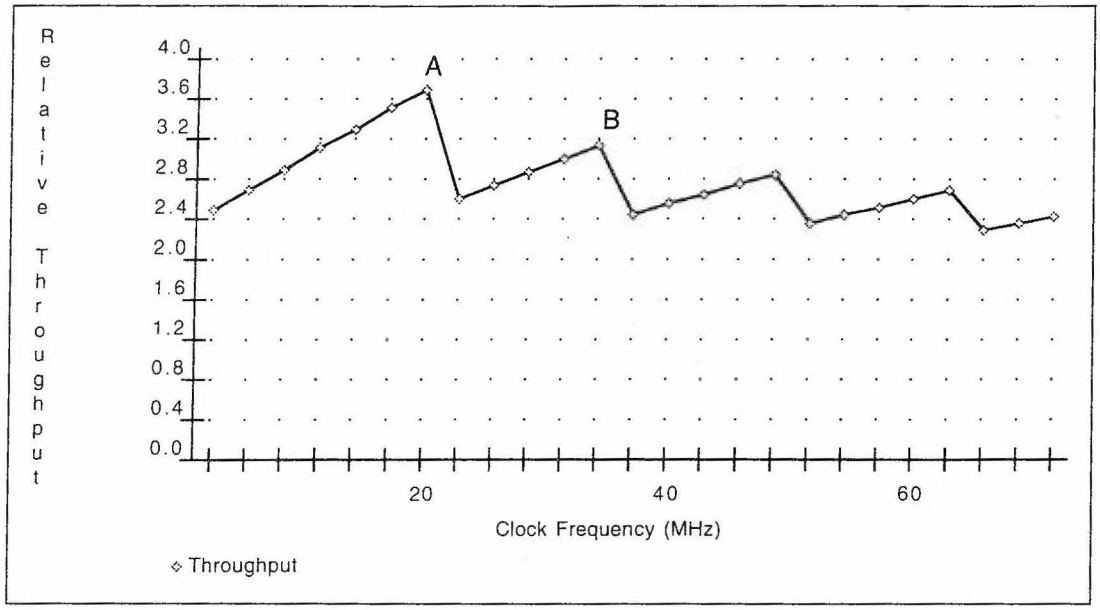


Figure 1.1 Throughput vs. clock frequency of a typical single-processor system.

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

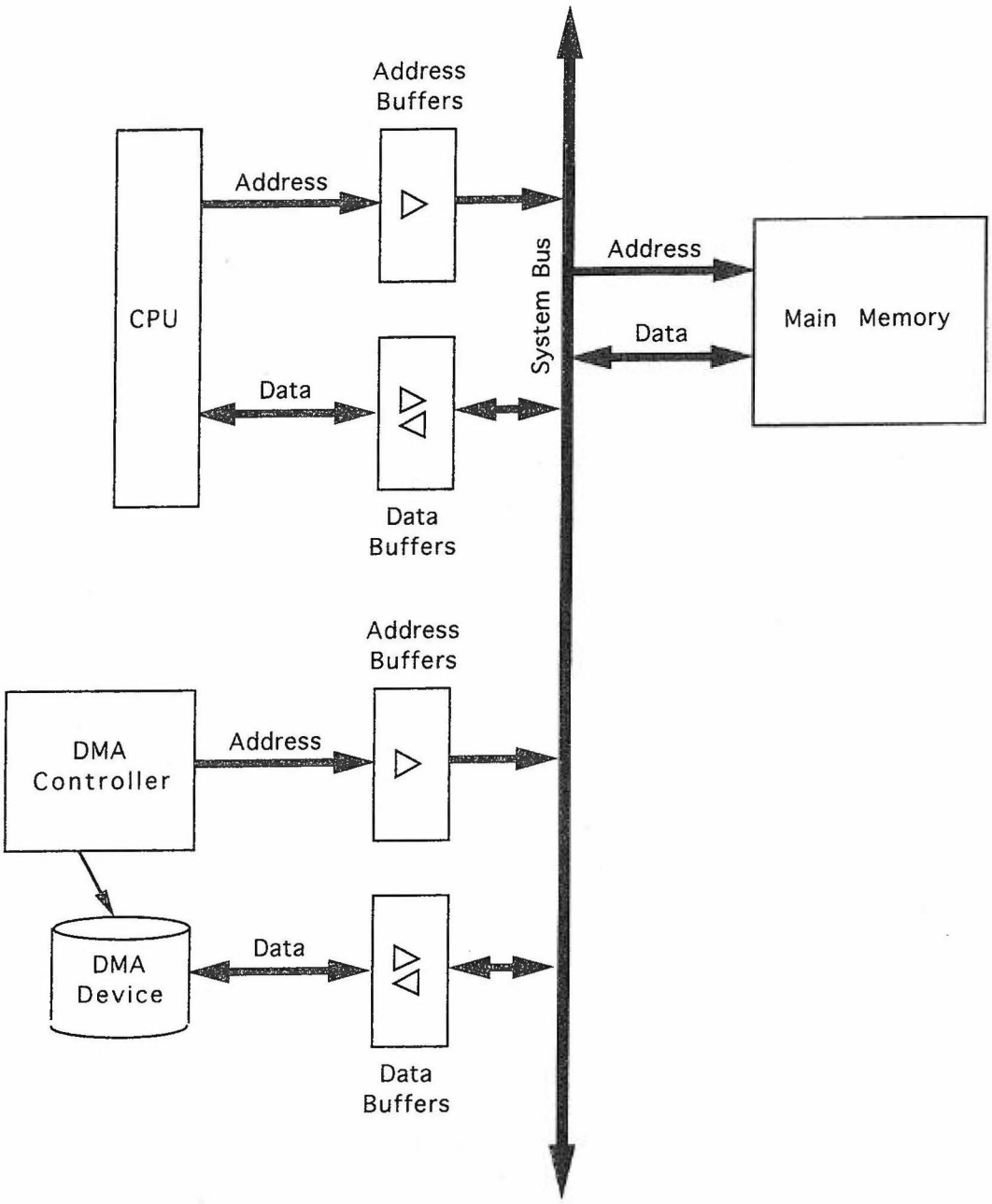
Fast main memory is not inexpensive. As a general rule, the difference between the price of a fast memory chip and the next fastest speed grade can range up to 50% or 100%! All this to gain an extra 20% in speed!

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 microprocessor 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



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

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

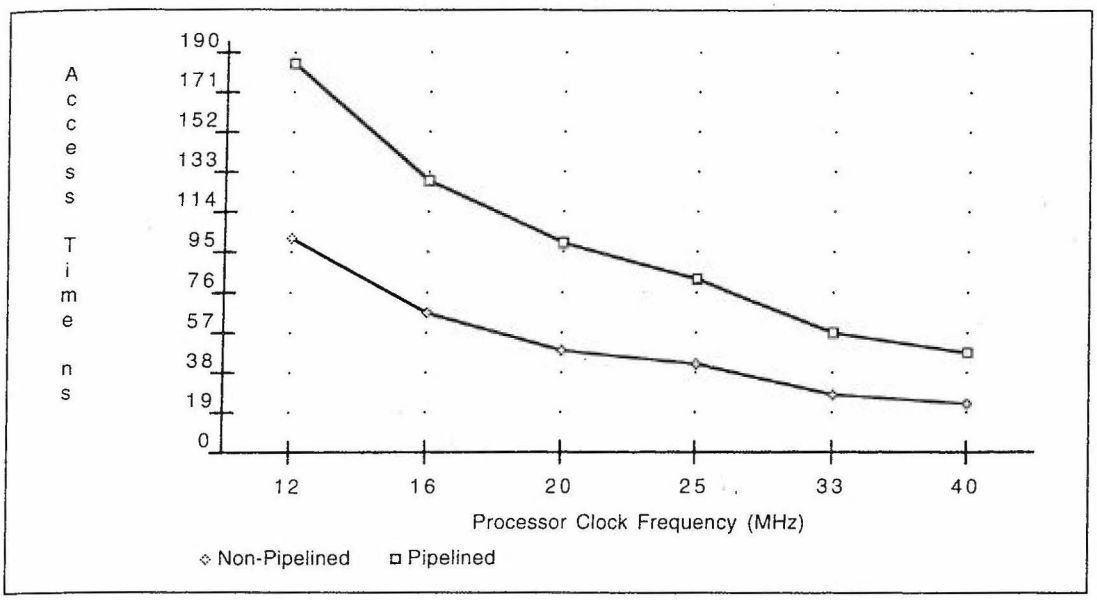


Figure 1.3. This graph shows the access time of a zero-wait-state main memory plotted as a function of the clock cycle time of the CPU for the system of Figure 1.2. The slope of the curve is very steep, showing how the speed of the main memory required in order to support increases in processor speed 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.

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

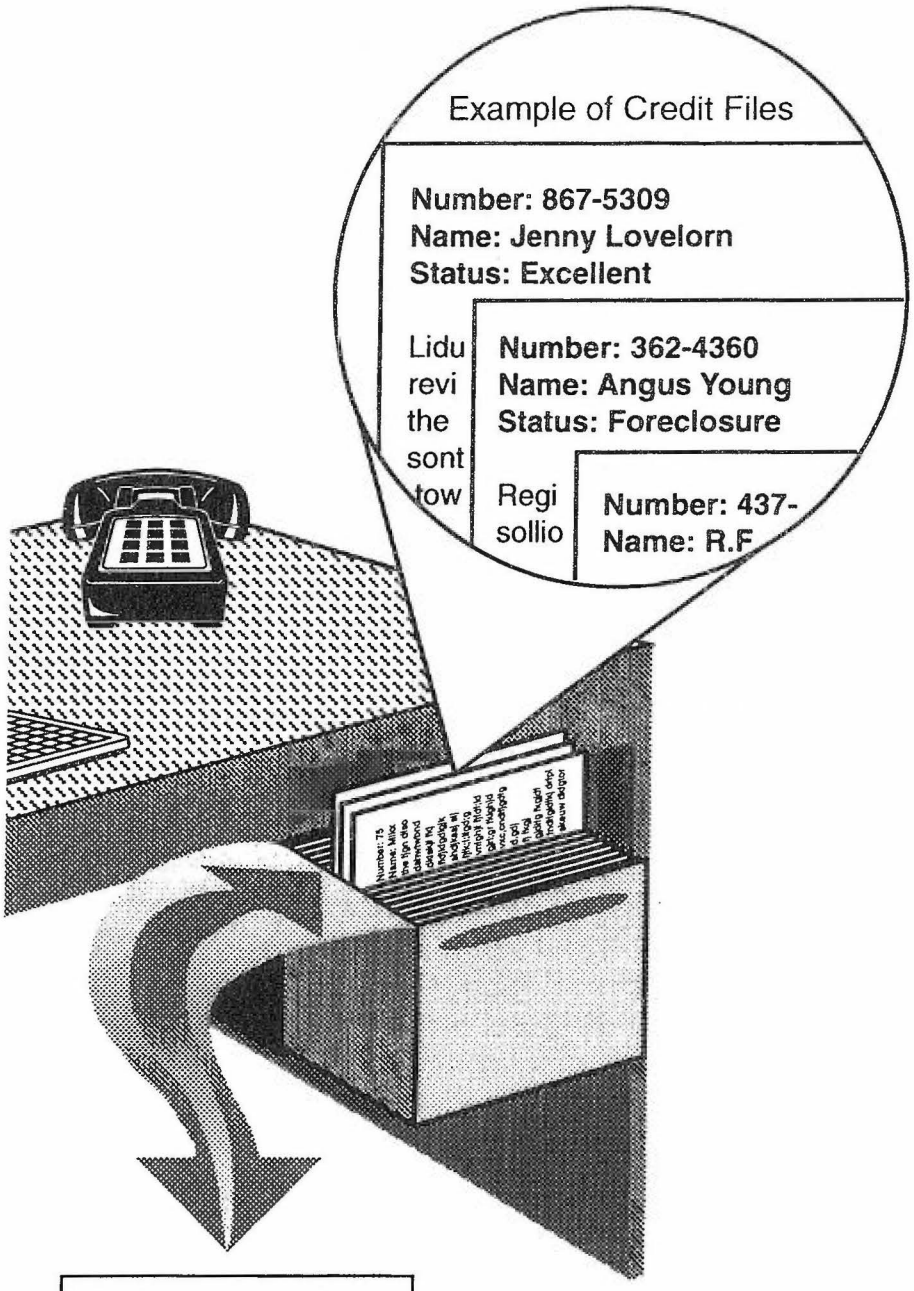
photocopy machine is close to the filing cabinets, so making an extra copy to take back to the desk is no problem. But who knows which files will be requested repeatedly and which will be requested only once? Being unable to predict the future, the clerk will simply make a copy of every document which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

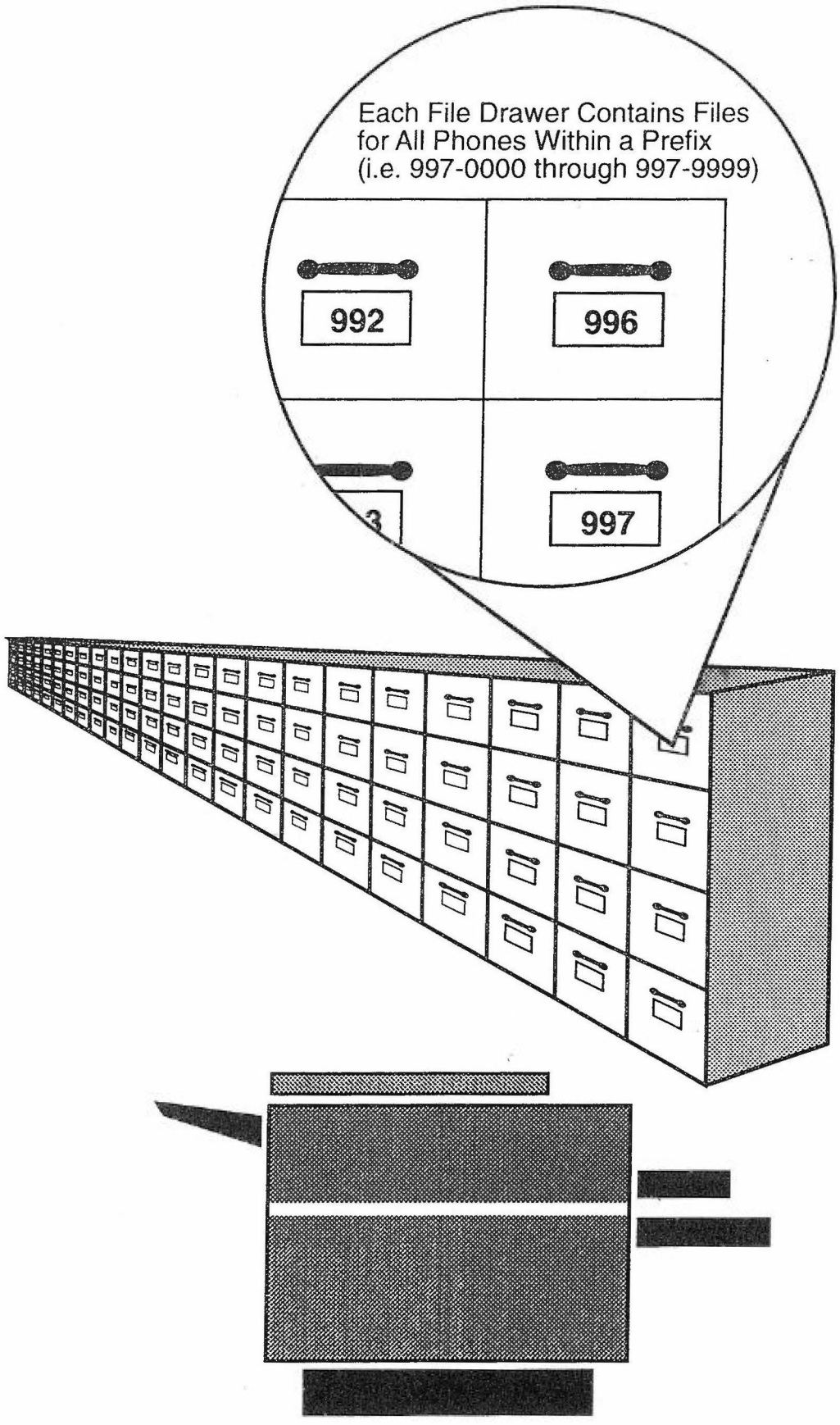


DIRECTORY

Slot	Phone	Access
0	867-5309	10/29
1	362-4360	6/3
2	437-8000	12/30
3	365-1854	1/5
•	•	•
•	•	•
•	•	•
•	•	•
•	•	•
9998	437-8471	11/18
9999	437-8228	2/17

Each entry on the **Directory** corresponds to a document filed in the drawer.

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.


Each File Drawer Contains Files
for All Phones Within a Prefix
(i.e. 997-0000 through 997-9999)
992
996
997

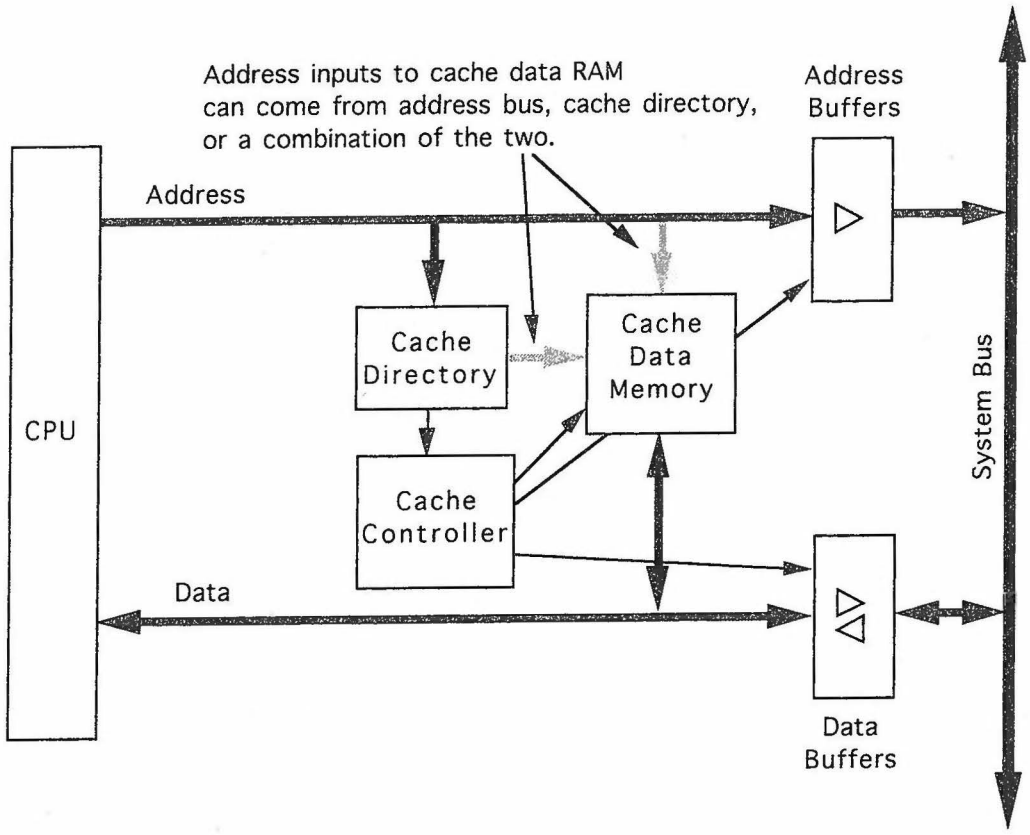


Figure 1.6. The cache memory consists of a directory, a cache data memory, buffers (to isolate the cache subsystem from the system bus), and the cache controller.

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 buss 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 cacheing 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

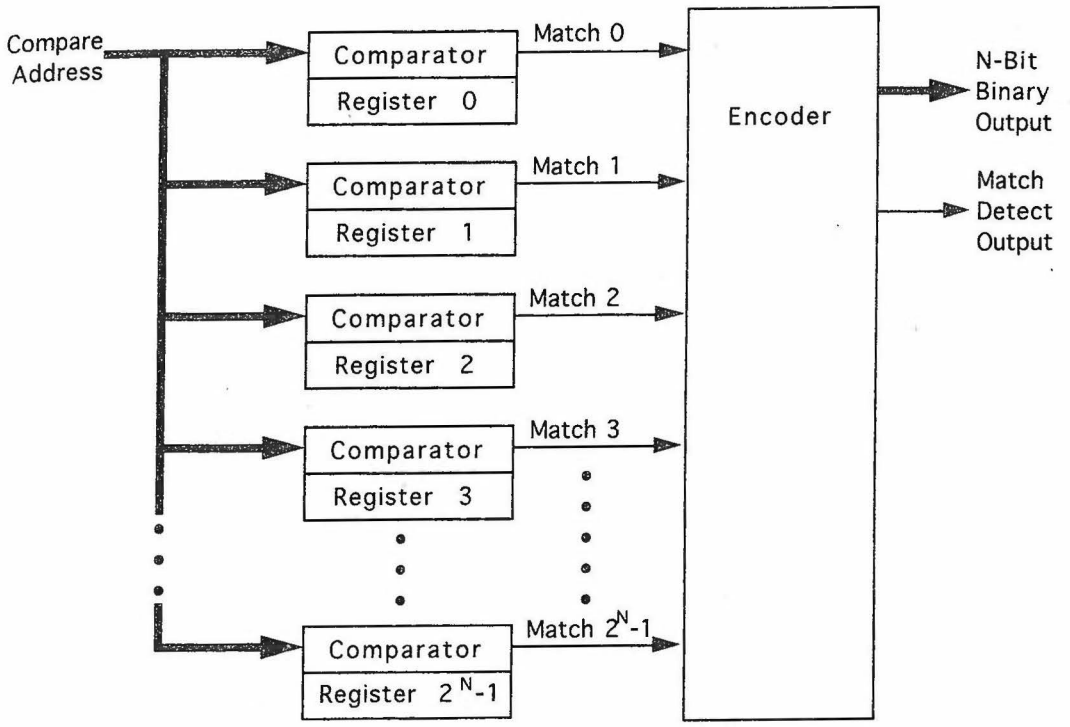


Figure 1.7. In a content addressable memory (CAM), an address presented to the compare address bus is compared simultaneously with the contents of every memory location. If there is a match at any location, the address of that location is sent to the *N*-bit binary output pins.

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

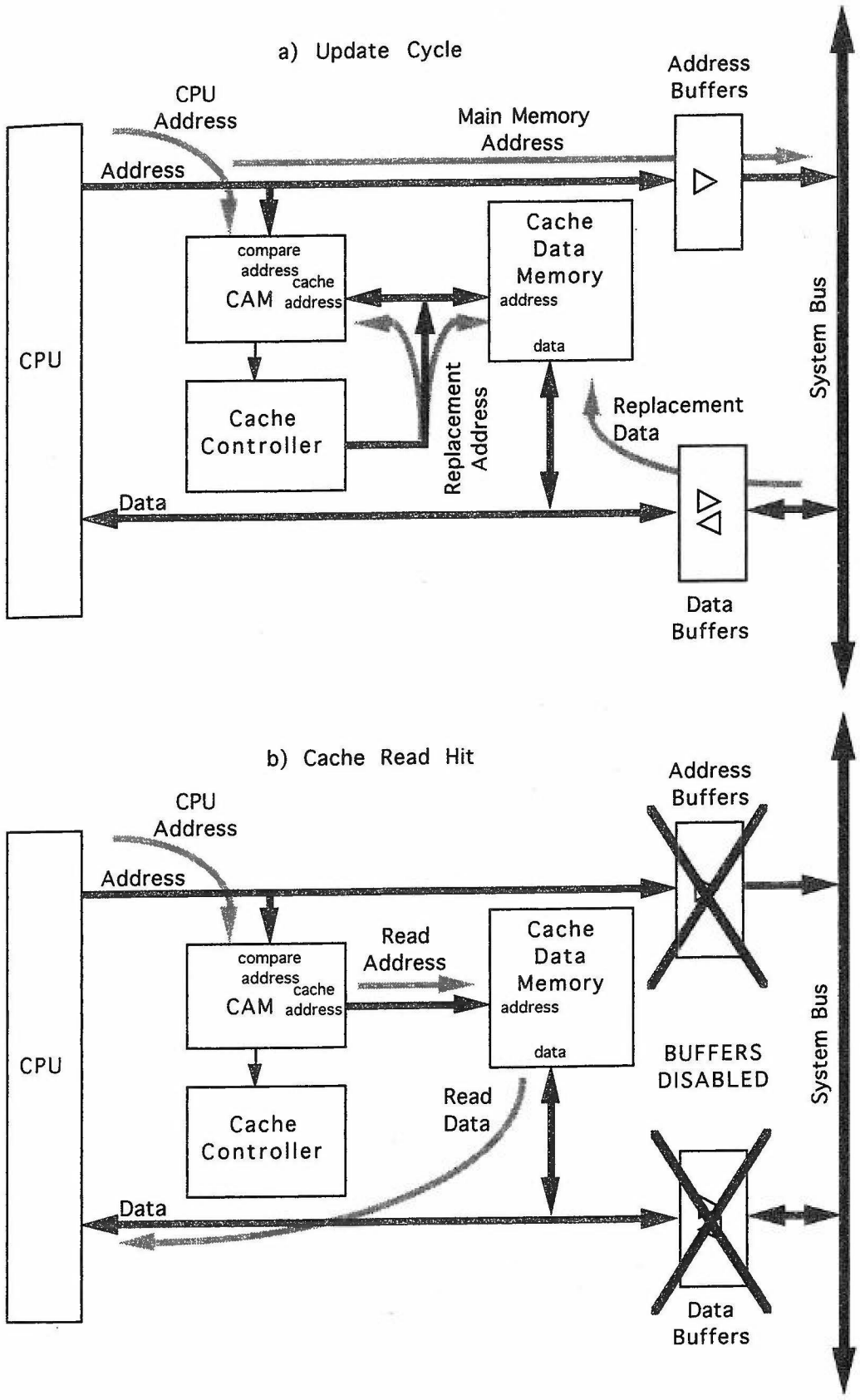


Figure 1.8. During the update of an entry in a fully associative cache (a), the cache controller feeds the address which is to be replaced into both the CAM and the cache data memory. When there is a cache hit (b), the CAM outputs this address to the cache data memory, causing its data to be fed into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

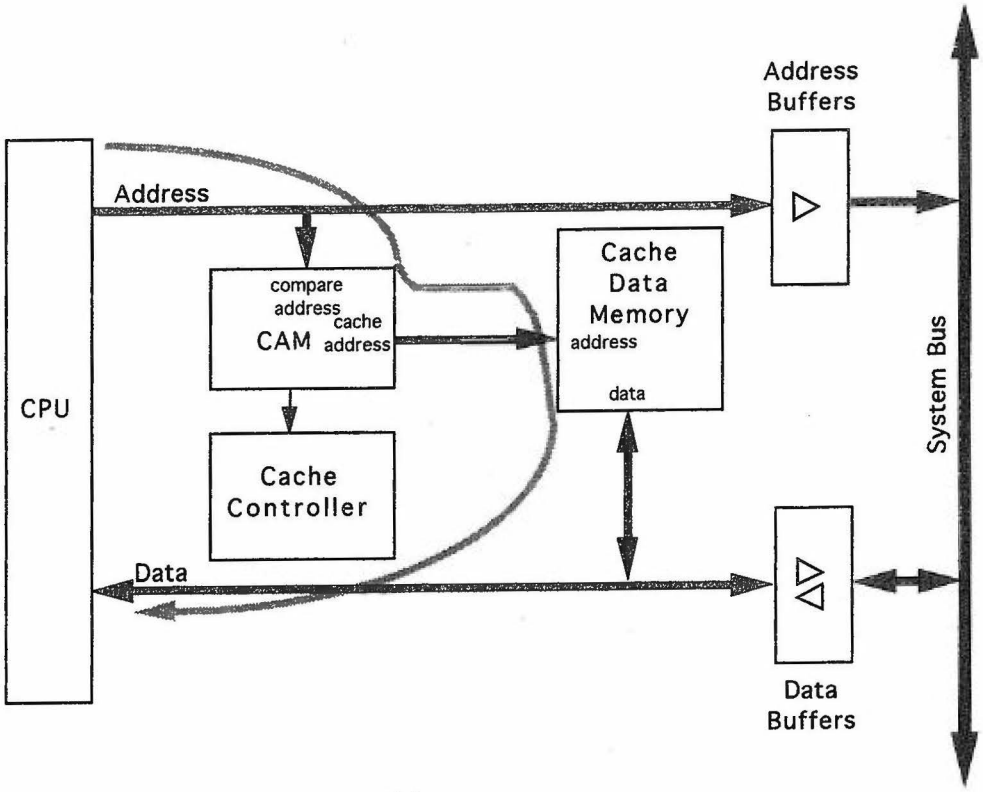
With this new method, the clerk can begin to look into the desk drawer while at the same time looking up the entry in the directory. There is no longer a need to cross-reference the directory to determine which desk drawer location contains the file. If the prefix (867) at the correct set address (5309) 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 subroutine 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-

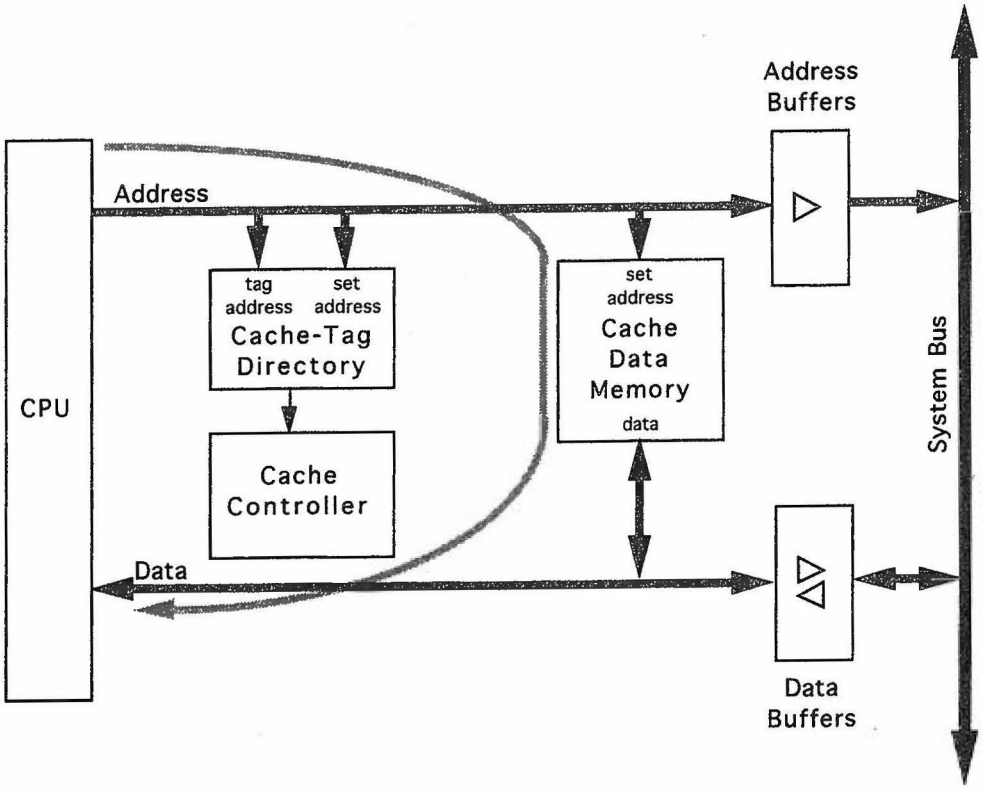
trated, the set-associative cache does not allow a main memory location to be mapped into any cache address, but instead the cache is constrained so that the lower address bits of the cache location must match the lower address bits of the matching main memory address. Since the processor will be running 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



(a)



(b)

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

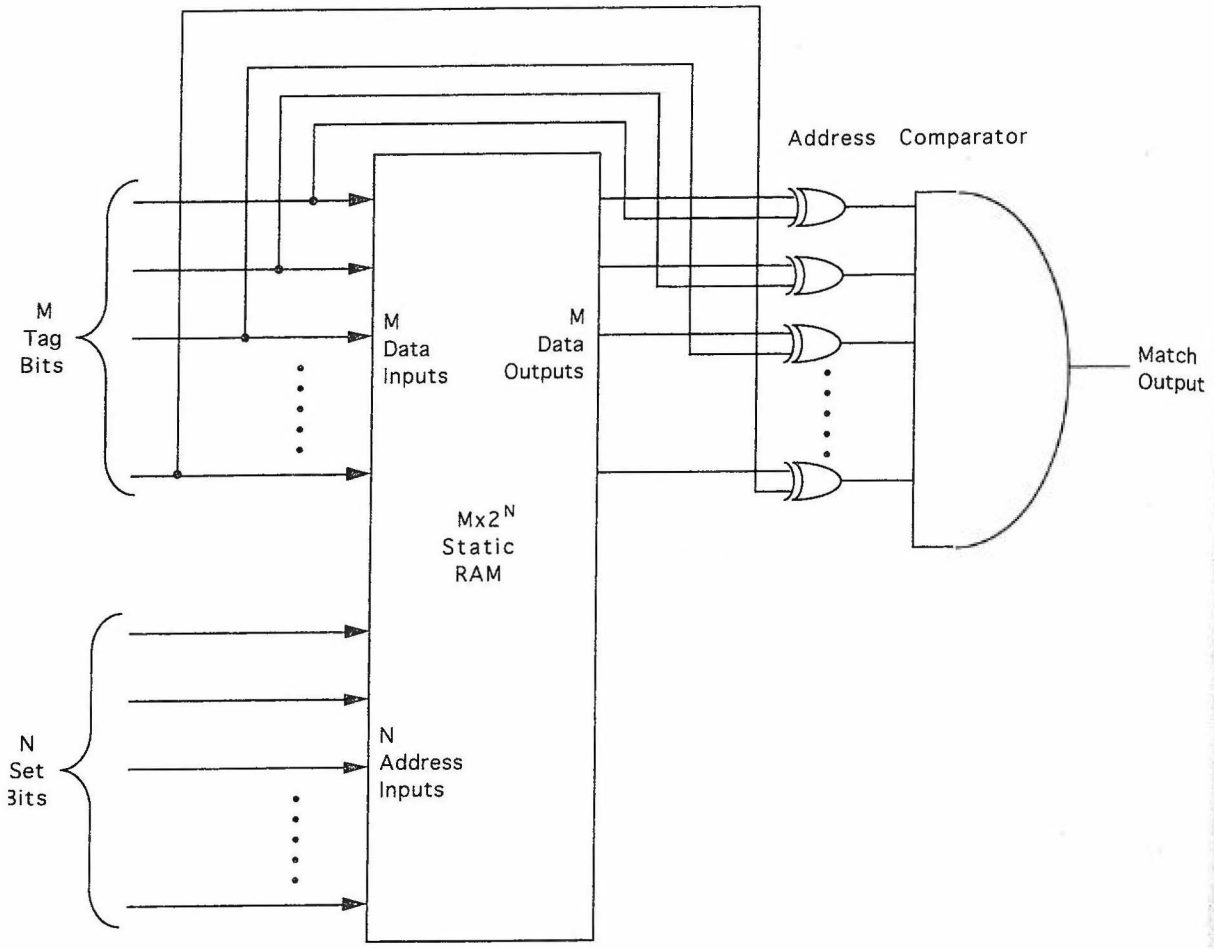


Figure 1.10. In a cache-tag RAM, the stored data at a certain RAM address (addressed by the set bits of the address bus) is compared against the upper address bits, or tag bits of the address bus. This comparison is performed with a simple address comparator consisting of exclusive OR gates feeding into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

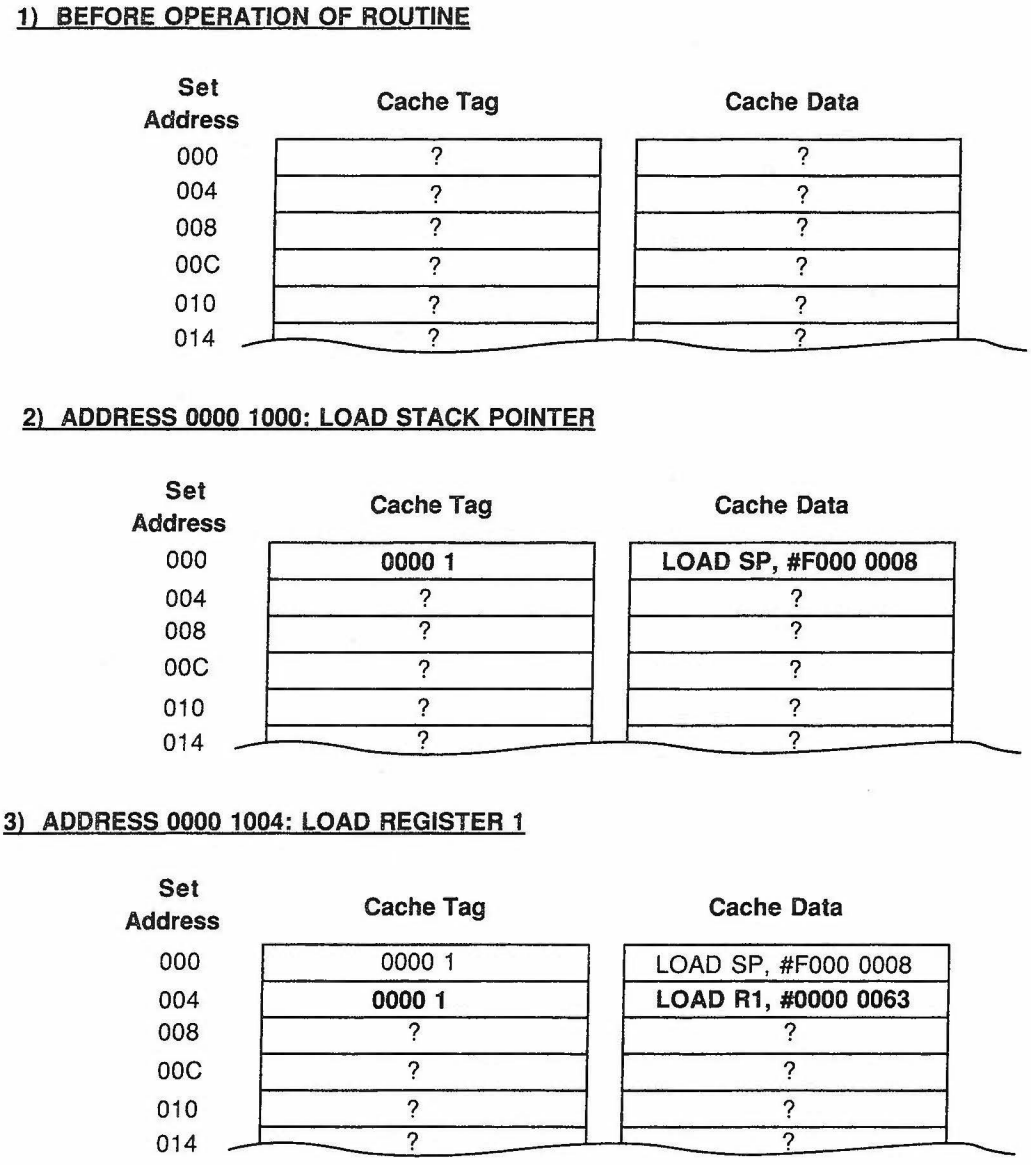


Figure 1.11. The contents of a set-associative cache during execution of the code shown in Figure 1.4. Note that location 008 never retains the cache data long enough to offer any speed advantage.

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-

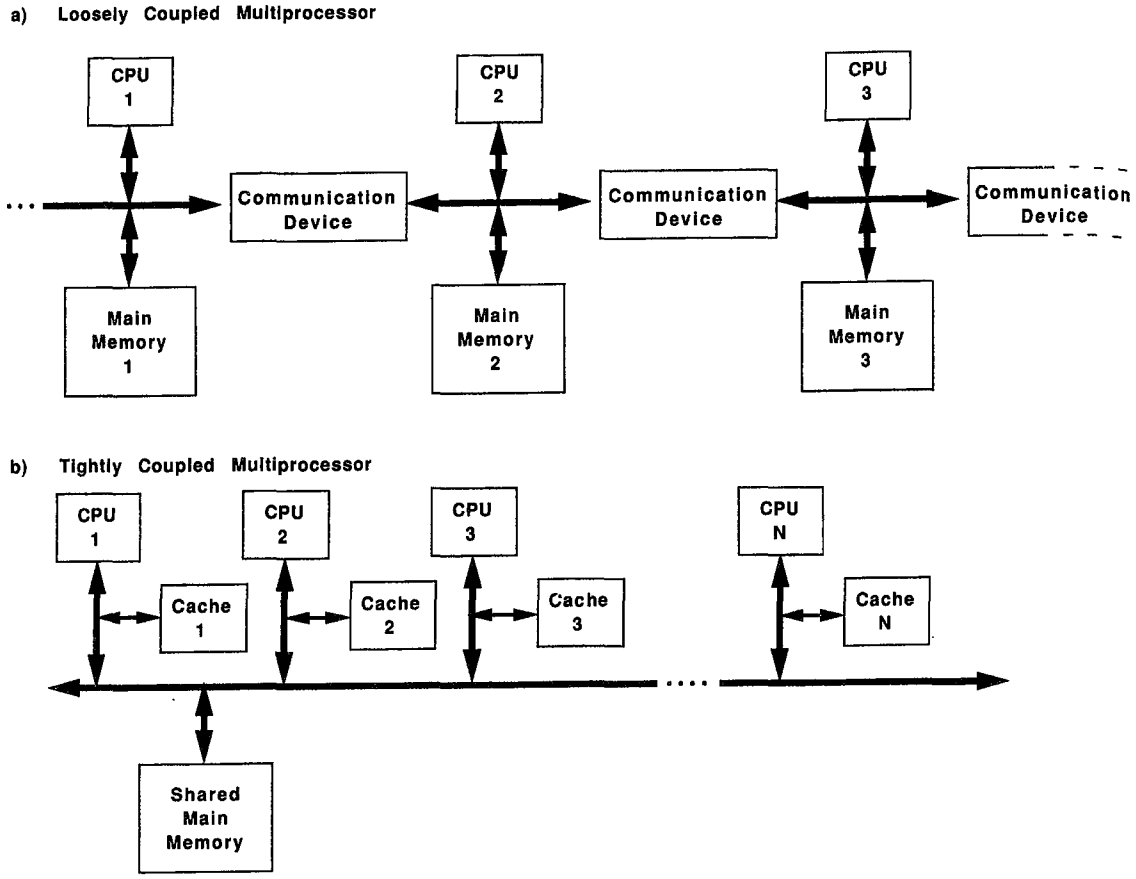


Figure 1.12. (a) A loosely coupled multiprocessing system. Each CPU has its own main memory and system bus. The processors communicate via the communication devices, which are usually accessed through each processor's system bus. (b) A tightly coupled multiprocessing system. All processors (or processor/cache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

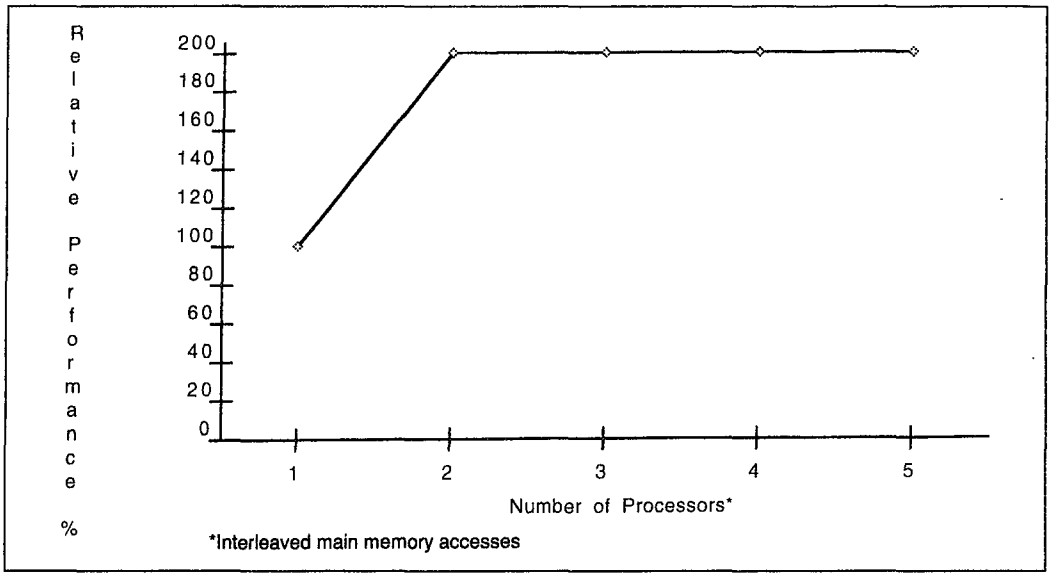
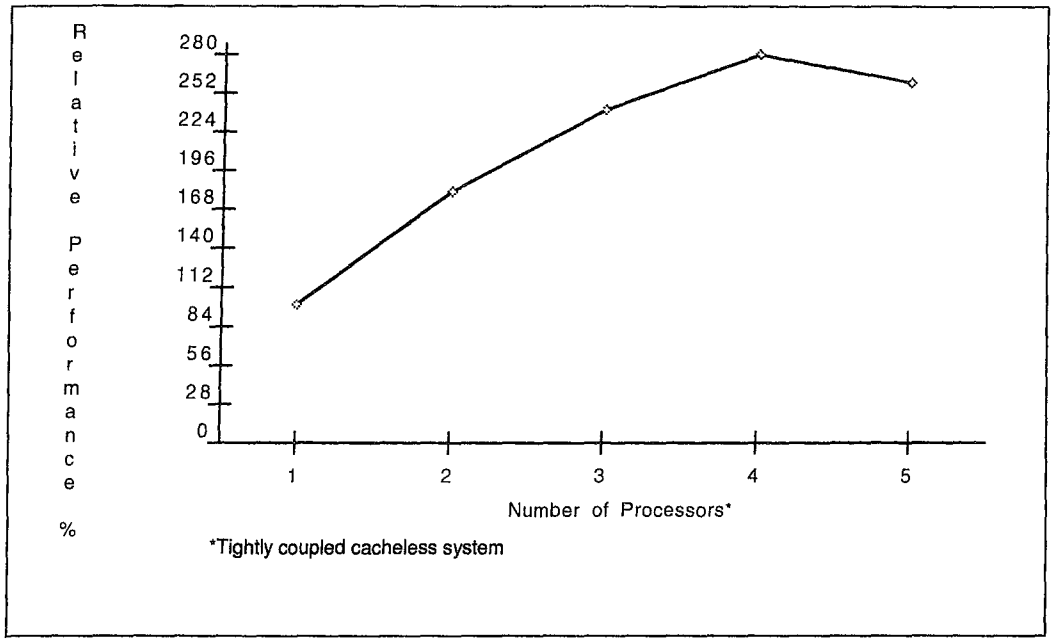


Figure 1.13. Performance vs. number of processors for the interleaved system described in the text. Each processor uses 50% of the available bus bandwidth, and no arbitration is required for either processor. Additional processors will be unable to access the bus, since it will be totally allocated to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

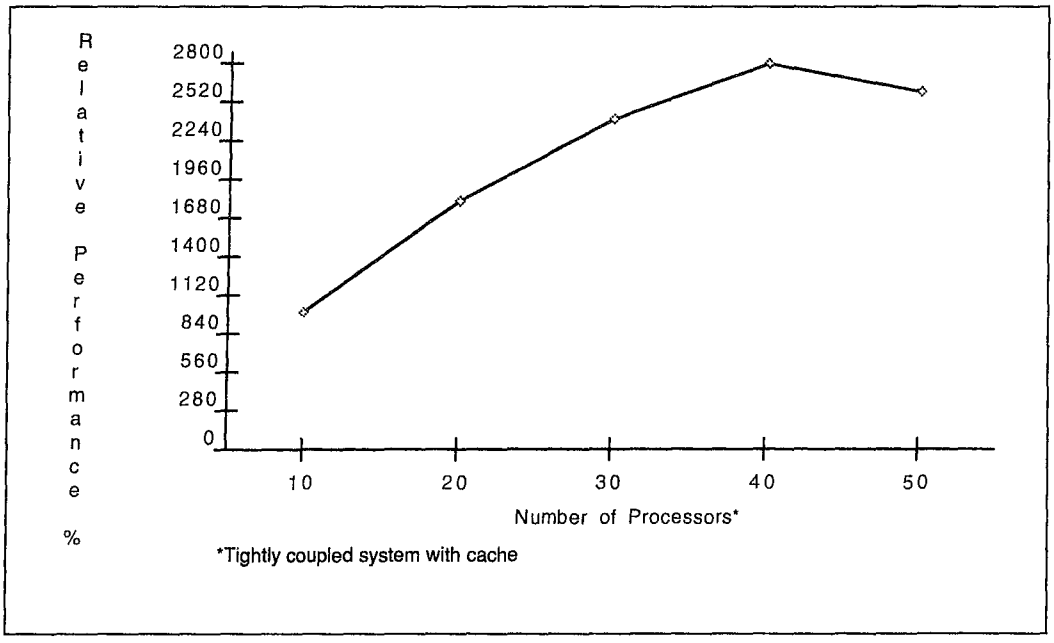
the entire system. One way of looking at this is that if each added processor requires exactly 20% of the available bus bandwidth (an unrealistically low level), then a six-processor system will require 120% of the available bandwidth and will perform at a 100%/120% level. High-performance processors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

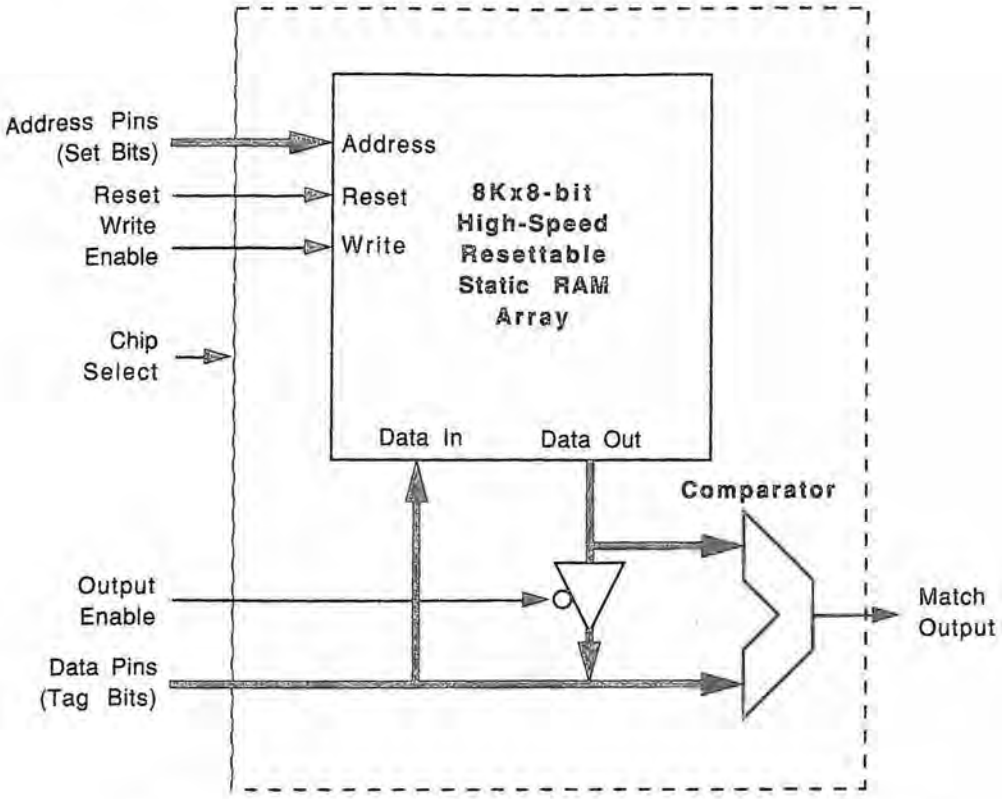


Figure 2.6. Block diagram of the internal workings of a typical cache-tag RAM.

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

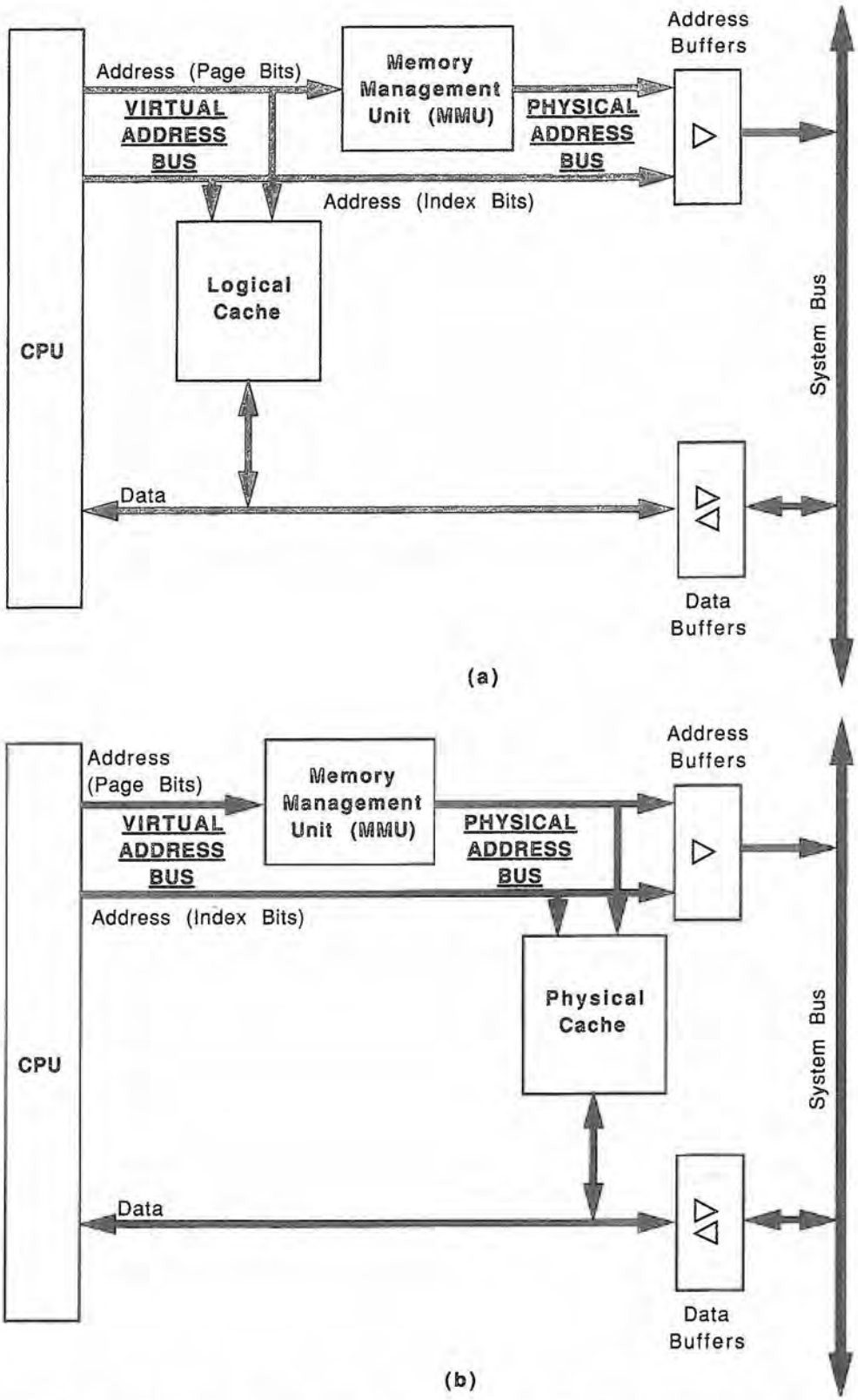


Figure 2.7. Logical and physical caches. (a) A logical cache is found upstream of the memory management unit. (b) Physical caches are on the physical address bus, downstream of the MMU.

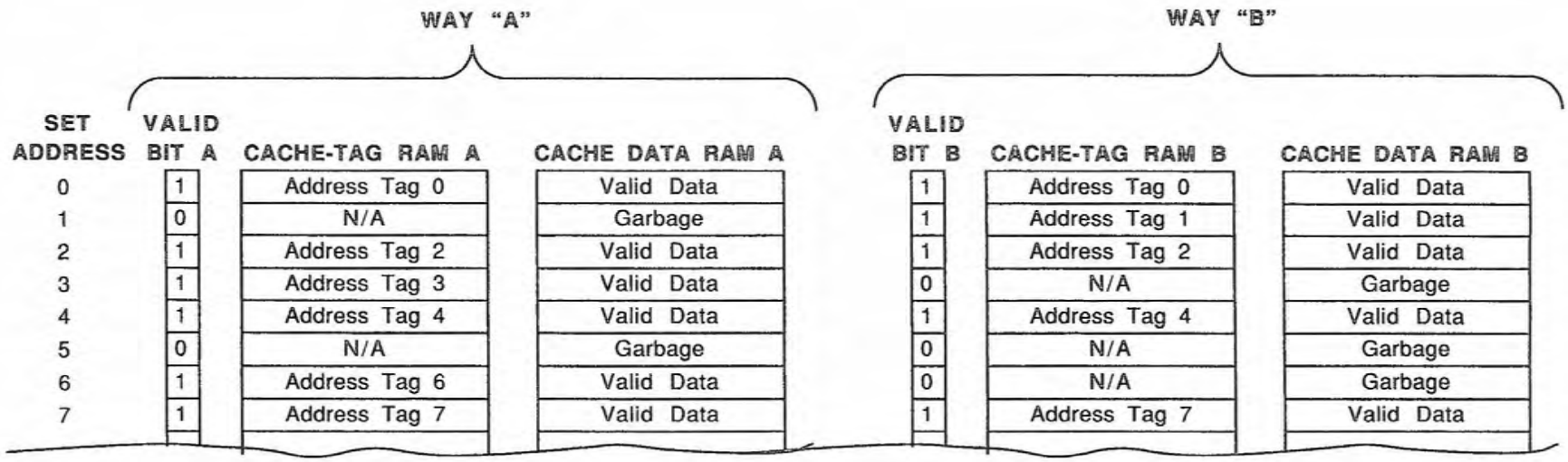


Figure 2.8. Contents of the cache-tag RAM and matching cache data entries in the two Ways of a two-Way set-associative cache. Note that these are simply two implementations of the simple direct-mapped cache of Figure 2.5.

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 × 8 SRAMs for the cache data RAM, and possibly one more 8K × 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 × 8-bit data RAMs would be needed, and two 4K × 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

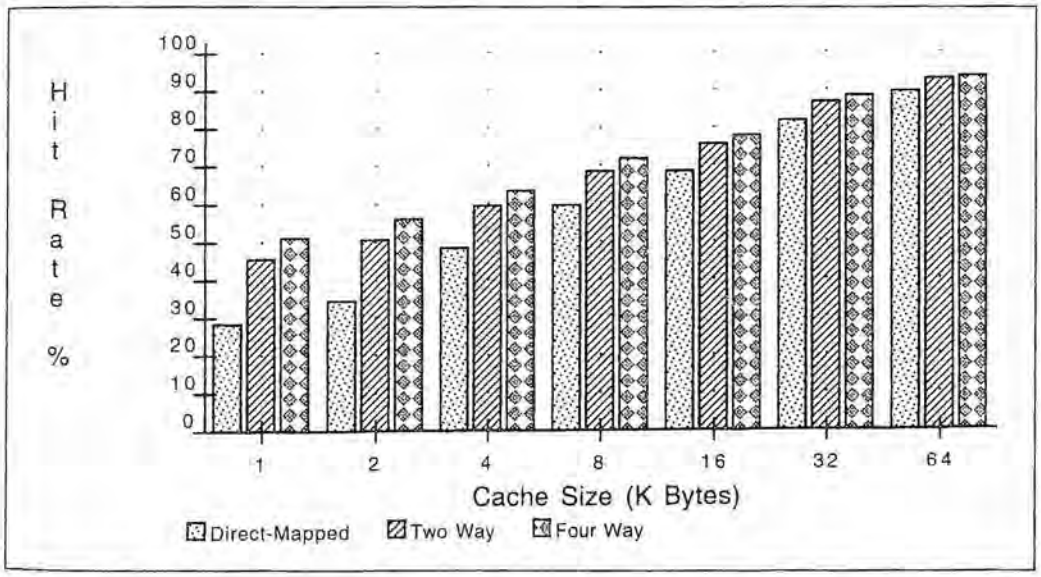


Figure 2.9. One set of statistics showing the relative performance of caches of varying sizes and associativities. Note that, for this example, the hit rate for a direct-mapped, 8K-byte cache is equivalent to that of a 4K-byte, two-Way cache. Above these sizes, a direct-mapped cache of twice the size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

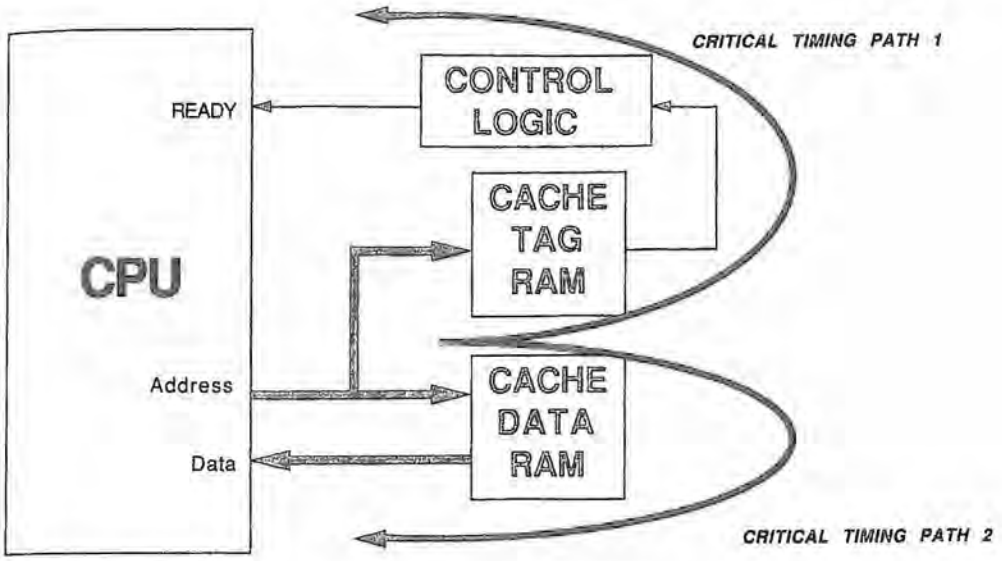


Figure 2.10. The critical timing paths in a direct-mapped cache are from CPU address output to CPU data input, through the cache data RAM alone, and from CPU address out to CPU Ready in, which runs through the cache-tag RAM, the comparator, and the cache controller logic. The Ready path is usually the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

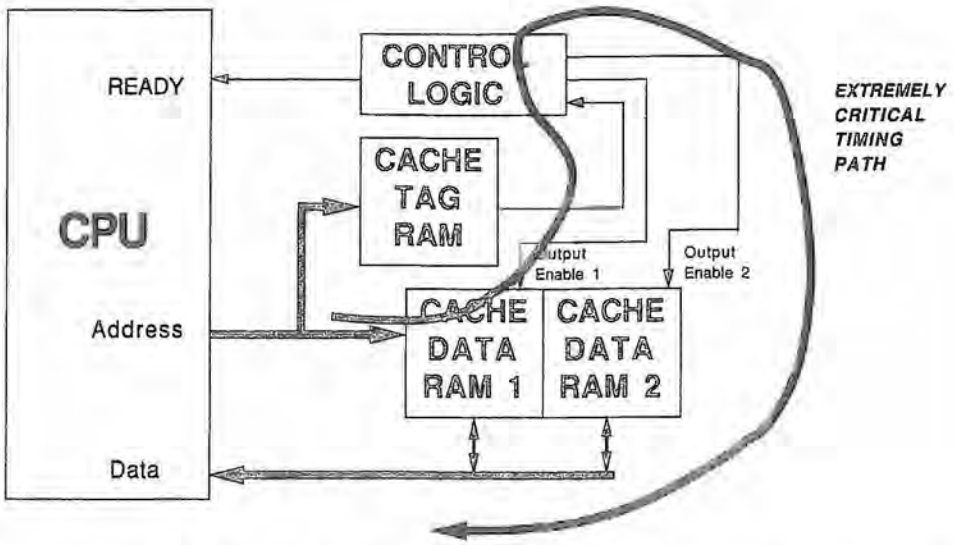


Figure 2.11. In a multiway cache, the outputs of all cache data RAMs are disabled until a hit has been detected. The most critical timing path then runs from the CPU, through the cache-tag RAM and controller logic, then through the output enable path of the cache data RAM before data can be returned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

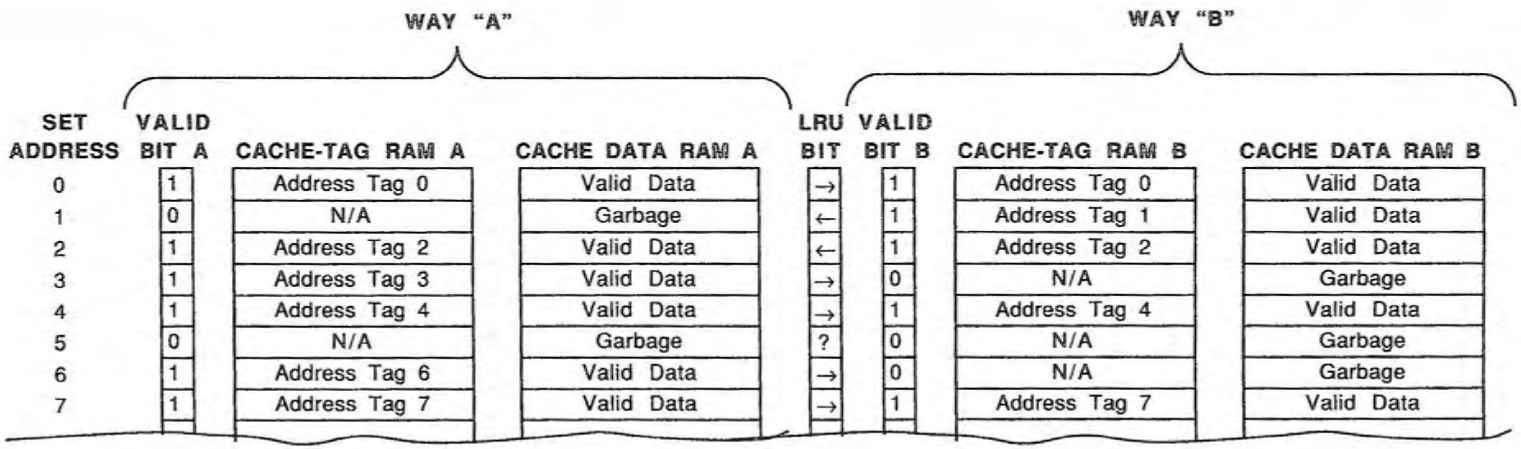


Figure 2.12. The two-Way cache of Figure 2.8 with an additional bit per line to mark which Way was least recently used for any particular line.

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-

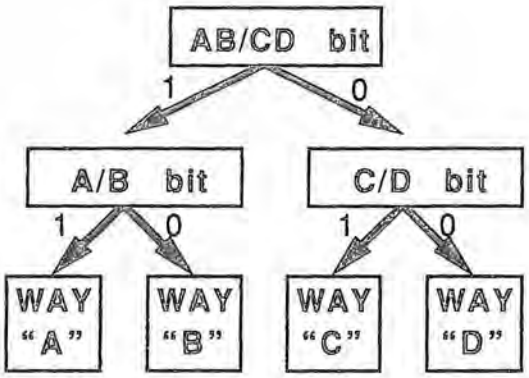


Figure 2.13. The pseudo-LRU scheme used in certain Intel processors to get away from timing and housekeeping difficulties posed by true LRU implementations for highly associative caches.

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