A VLSI DESIGN FOR AN EFFICIENT MULTIPROCESSOR CACHE MEMORY

XIAO LUO

A VLSI DESIGN FOR AN EFFICIENT MULTIPROCESSOR CACHE MEMORY

BY

Xiao Luo

A thesis

submitted to the School of Graduate Studies

in partial fulfillment of the requirements for

the degree of Master of Science

Department of Computer Science

Memorial University of Newfoundland

St. John's, Newfoundland, Canada, A1C 5S7

November 28, 1989

ISBN 0-315-61787-X

Abstract

This thesis proposes a cache memory, used for a 32-bit processor system, which consists of four components: the Directory, Line Replacement Unit (LRU), Cache Memory, and Control Unit. An *8-way set-associative* mapping method is employed in the directory. The Line Replacement Unit is based on the *least recently used* line replacement algorithm. The cache memory unit has a capacity of 8k bytes, 32 bytes in each line, and it is directly accessible to 1, 2, 3, or 4 bytes (one word) once by the associated processor. This cache memory is designed for a multiple processor system as well as in single processor system; a *write-through* algorithm and an *updating* algorithm are combined together to keep the information in main memory consistent with that of the cache and to make the multicaches coherent. The hit ratios are predicted to be over 95 percent. A two-phase clock of 40ns is employed to pipeline this cache, and it can turn out a result in 20ns during read operations without line misses. This cache is implemented into a single chip, and is designed so that it is possible to build cache systems of various sizes using these chips, without decreasing the system speed. This cache memory has been laid out as a single integrated circuit using 3 Micron NTCMOS technology, and its electrical and logical behavior has been simulated.

Acknowledgments

First of all, I would like to express my sincere gratitude to my advisor Dr. Paul Gillard for his supervision, guidance, suggestions, encouragement, and patience, which have greatly helped me to complete this thesis.

I would like to thank Dr. Wlodek Zuberek, who has contributed his support and advice toward the development of this thesis, and the other staff members of Department of Computer Science, who have provided me with their technical support. Especially the support and patience of my wife was of great help.

I am very grateful to Department of Computer Science and School of Graduate Studies, Memorial University of Newfoundland, for providing me with an opportunity for graduate studies and financial support in the form of a fellowship and teaching assistantship during my study.

To the Memory of My Father and to My Mother

Contents

List of Figures

List of Tables

1 INTRODUCTION

In 1945, John Von Neumann made proposals for a digital electronic computer structure. In his proposals, the basic logical structure of a digital computer system has the following characteristics:

1. It has an *input* medium, by means of which an essentially unlimited number of operands or instructions may be entered.

2. It has *storage*, from which operands or instructions may be obtained and into which results may be entered, *in any desired order.*

3. It has a *calculating* unit, capable of carrying out arithmetic and logical operations on any operands taken from storage.

4. It has an *output* medium, by means of which an essentially unlimited number of results may be delivered to the users.

5. It has a *control unit,* capable of interpreting instructions obtained from memory or storage, and capable of choosing between alternate courses of action on the basis of computed results.

In general, a computer which meets the criteria defined as the *Von Neumann structure* is organized as shown in Fig. 1. Although the components of the five parts of the basic structure and the technologies used may vary widely, the functions of the parts may be clearly identified in virtually any digital computer.

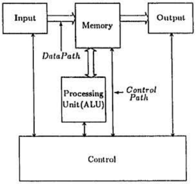


Figure 1: A Von Neumann Computer Organization

Memory is the source of all information, data, and instructions, flowing to or from the four other parts. The data and instructions are stored in the memory cells, each of which is associated with a *location*, or *address*. The cells can be accessed by other parts of the computer by means of these addresses.

The main functions of input and output, as indicated by their names, are to derive information from and to deliver the results and other information to the outside world. They have also two subsidiary functions, buffering and data conversion. The buffering function provides an interface and synchronization between the processing part of the computer and the outside world. The conversion function can convert the data type in the processing unit into forms used outside the computer system.

The processing part of a computer, referred to as the *arithmetic-logic unit*, implements the various arithmetic and logic operations on operands obtained from the memory. The results, after these operations, are typically stored back in the memory.

The control unit obtains instructions from the memory, decodes them, and, depending on their meaning, sends the appropriate control signals to other parts of the computer so that the desired operations will be accomplished. It also makes decisions about what action must be taken after receiving the results of various tests on data made by the *arithmetic-logic unit*. The combination of the arithmetic-logic unit and control unit is known as the *central processing unit*, or processing element in the case of multiple processor systems.

Until the last two decades, almost all the electronic digital computer systems used this *Von Neumann architecture*. Even when the underlying architectures of the computer systems began to contain a limited amount of parallelism (such as in the CDC6600, for example) it was generally concealed from the users. In this period, the demand for higher speed, larger storage, and more reliable computer systems was rapidly increasing because large scale computation applications were visualized. The demand was such that, despite many technological advances in electronics, uniprocessor systems proved to be inadequate for the most highly computationally intensive problems since the point had been reached where communication delays between switching elements or integrated circuits play a dominant role in the speed of the computation. Therefore, new ways had to be found to meet these requirements. The general approach is based on parallelism, implying that computer architectures will have to depart from the strict Von Neumann concept. Parallelism in various forms had already appeared in computers produced during the 1960's, and has proved to be an effective approach. In this context, parallelism does not only mean the replication of logic but also has other meanings. For example, a uniprocessor using a pipelined instruction unit and a pipelined arithmetic unit, as well as the implementation of multiple programs executed "simultaneously", all imply concepts of parallelism. Therefore parallelism in a computer system presently has three meanings:

1. Time interleaving

2. Resource replication

3. Resource sharing

Time interleaving introduces a time factor into the concept of parallelism. That is, several process steps are interleaved in time, each using a part of the same hardware at different times. In this case, it is not necessary to have a replication of hardware to increase the performance of a computer system. Pipelining is an example of time-interleaving.

Resource replication is the replication or addition of hardware units which can operate simultaneously on a problem, thereby attaining computation power through replication of logic, rather than relying solely on fast individual gates and small dimensions to reduce logic delay in order to obtain high speed. Multiple processes using the same hardware in some time-slice order are an example of resource sharing.

Since parallelism was introduced into computer architecture, various parallel computer architectures such as vector processors, pipelines, array processors, as well as multiprocessor architectures, have been developed and used to handle large quantities of data simultaneously and concurrently with high performance. I/O processors have been used for input and output to speed up communication between the processing elements and external storage or users. Thus, in general, parallelism includes not only simultaneity but also concurrency. The former means that two or more events occur at the same time and the latter means that two or more events occur within a given interval of time.

On the other hand, memory has been organized in different ways in order to obtain access speeds compatible with that of processing elements and to have a larger capacity. In general, there are two basic approaches: one is to organize the memory as a memory hierarchy; the other is to decompose the memory into several modules shared by the processors in the system.

These kinds of computer architectures are, more or less, not strictly Von Neumann structures; indeed, the multiple processor systems in particular have quite different characteristics.

2 BASICS OF CACHE MEMORY

Throughout the history of electronic computers, whenever developments have taken place in computer systems which increase processor speed, there is corresponding pressure to have the memory match this speed and, at the same time, increase its capacity. Therefore, performance improvements in computers have been associated with improvements in memory capacity and speed. Although both processors and main memory systems have been improved by steadily developing technologies and novel architectures, there has been a persistent mismatch between the speed of processors and that of main memory. That is, the main memory is slow relative to the processors. The memory system limits the speed at which input data can be delivered to a processor and the results received from the processor. This is the so-called *Von Neumann bottleneck*. Hence there has been a constant need for steady improvements to main memory subsystems for high overall system performance. Approaches of interest toward improving memory speed and capacity have been the following [2, 5, 7]:

1. Memory hierarchies and virtual memory
2. Cache memories
3. Development of larger and faster memory chips
4. Memory interleaving

2.1 Overview of the Memory Hierarchy

In order to improve the performance of computer systems, especially single processor systems, there are two approaches to speed up a memory system with a large capacity. One is to develop a higher speed memory system with a larger capacity, the other is to partition a memory system into an efficient memory hierarchy consisting of several levels of subsystems with various speeds and sizes [2, 3, 5]. The first approach seems more straightforward and simple — to have a fast one-level memory with a large capacity. However, even with improvements in technology, a fast memory system with a large capacity is still very expensive, so that it is necessary to use slower memory at a lower cost to create a memory system with a large enough capacity. In order to give the memory subsystem an adequate effective speed, the memory subsystem can be organized as a *hierarchical memory system*. This kind of memory system can be matched to both the speed and size requirements of the high-speed processor at relatively low cost. A typical hierarchical memory structure is depicted in Fig 2. The top level of the memory hierarchy (near the processor) has the fastest speed but also the highest cost. Therefore, the capacity of this level is made smaller to decrease system costs. For the lower levels, the speed of the subsystem decreases while the capacity increases. At the level on the bottom of the memory hierarchy, the memory subsystem possesses the largest capacity, but slowest speed, with lowest cost per word stored. In this memory hierarchy, each level is directly connected to the immediately higher level. That is, each memory subsystem can directly communicate with the immediately higher or lower

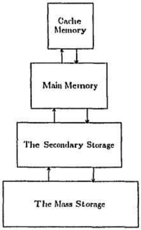


Figure 2: A Typical Memory Hierarchy

subsystem in the hierarchy. For example, the processors can directly communicate with the first-level memory, e.g. register array or cache memory; and similarly the first-level subsystem can communicate with the second-level one, as shown in Fig. 2, and so on. Generally the top-level subsystem, such as cache memory, is used to attempt to bridge the speed gap between the processors and the lower level subsystem, while the lower level memory subsystems are employed to enlarge the capacity of the whole memory system.

2.2 The Concept of Cache Memory

The concept of cache memory was proposed by Wilke [1965] in a brief article in which he described a system that contained two kinds of main memory: one was conventional, and the other was unconventional high-speed memory called at that time *slave memory*, now called *cache memory*. In 1968, the first real cache memory was implemented on the IBM 360/85. Since then, use of cache memory has rapidly increased on a wide range of computer systems, initially on mainframes, then on minicomputers, and today even on microcomputers.

Cache memory, a relatively small, high speed random access memory, is designed for transparently bridging the speed gap between the CPU and main memory, since it typically has a speed compatible with that of the CPU. This means that a cache memory in a cache-based system is invisible and not directly accessible to users or even to system operators. Typically, the speed of cache memory is five to ten times faster than that of main memory. Using this kind of memory hierarchy, the computer may seem to have a one-level memory with the capacity of the slow main memory and the speed of the cache memory [2].

The idea of the cache memory, similar to the primary-secondary virtual memory, is to duplicate the active portions of a lower speed memory in a high speed, but smaller, memory. Only the data most likely to be needed in near future by the CPU reside in the cache, and obsolete data are automatically replaced by the newly requested data. In general, the speed of the cache memory is matched to the maximum data rate of the processor so that the processor can access data in the cache without delay, whenever the data requested by the processor are found in the cache. If the requested data are not in the cache, a cache miss occurs, and a request is made to the main memory for transfer of the requested data to the cache. If the data currently resides in the main memory, it is transferred to the cache immediately. If it is not, but is in the secondary memory, a request is issued to bring the requested data from the backing storage. Therefore, when the required references to the memory can be captured by the cache, speed is not degraded. Otherwise, the performance will be degraded by the time required to transfer data from the main memory to the cache.

The use of cache memories in modern computer systems is based on the locality of memory references — both *spatial* and *temporal* [7, 9]. Spatial locality refers to the property that memory accesses over a short period of time tend to be clustered in space. This type of behavior can be expected based on the common knowledge of typical program behavior: related data items (variables, arrays, etc.) are usually stored together and instructions are mostly executed sequentially. Temporal locality refers to the property that references to a given locality are typically clustered in time. This type of behavior can be expected from program loops in which both data and instructions are reused. Therefore, use of a cache memory in a computer system can minimize the interconnection network traffic between the processor and main memory and speed up the system since the access delay of the memory system and the frequency of references to the slower main memory are highly reduced.

2.3 The Basic Structure of Cache Memory

The capacity of cache memory is far smaller than that of main memory; that is, the address space of cache memory is far smaller than the address space of main memory, therefore cache memory requires an address mapping mechanism to translate the main memory addresses, at a high speed, into the cache memory address where the copies of data in the main memory reside. Also because the most active portions in the main memory are copied in the cache memory, if the cache memory is full and the associated processor needs data not in the cache memory, some of the data in the cache will be replaced with the newly requested data from the main memory. There must exist an algorithm which can predict that the data to be replaced will not be used in near future. Since the speed of the cache memory is the key factor in cache memory design, this kind of algorithm must be implemented in hardware. Hence, the basic structure of a cache memory should have at least three basic hardware components: an address mapping mechanism, a data replacement unit, and storage for the data in the cache.

The basic functions of a cache memory can generally be described as follows: Each reference from the processor to a memory location is presented to the cache memory. The cache first searches the directory of the address mapping mechanism to see if the requested data reside in the cache memory. If the requested data are in the cache, the data are operated on to satisfy the processor immediately without disturbing the main memory. If the data are not resident in the cache, a *cache miss* occurs which will cause the transfer of the new data from the main memory to the cache. Then the requested data can be referenced by the processor. Before transferring a new line to the cache, some data has to be removed from the cache memory to make room for the new. Which old data in the cache will be discarded is determined by the data replacement unit. Therefore, the cache-replacement decision directly affects the performance of the cache. A good replacement algorithm can make the cache have a somewhat higher performance than a bad algorithm.

Since a cache memory has a high speed compatible with that of the associated processor, all the algorithms of a cache memory have to be implemented in hardware. Therefore, the designers of a cache memory have to consider not only how to implement its functions but also how to implement these functions with practical hardware.

Traditionally, a cache system is built with a single cache for both data and instructions. This cache is called as a unified cache, in which case the CPU's components have only one cache unit to refer to for both instructions and data. The associated processor shares the same cache for data and instructions, which makes more efficient use of a limited resource and lowers the average miss ratios. Also a cache system can be split into two separate caches: one for data, and the other for instructions. One of the major advantages to splitting data and instructions into two separate caches is that conflicts between simultaneous instruction fetches and data reads and writes are eliminated [9].

2.4 The Line Size Choice

The performance of most computers depends strongly on the quality of the cache design and the way in which it is implemented. Therefore, cache design is a very significant part of computer system design. In order to design a high-performance cache memory, there are several choices to be made and parameters to be set.

Designers have to make decisions about the algorithms (fetch, placement, etc.), about the best sizes (cache size, line size, etc.), and about the ways of address mapping and maintaining consistency among several caches in a multiprocessor. Designers also have to make tradeoffs in setting these parameters; *e.g.* cache size, line size, the set-associativity, and so on. Each of these parameters affects cache performance; choosing different parameters produces different cache performance.

The cache line size is a very important parameter that strongly affects the cache performance, especially the cache miss ratio [11]. Many surveys of cache memory and/or memory hierarchy performance have been made for high performance systems. In these surveys, the cache line size choice, with the overall cache size, has been shown to strongly affect the cache miss ratio. Smith suggested in [9] the line size giving the minimum miss ratio for a given cache memory capacity. He also indicated that the minimum number of elements per set in order to obtain an acceptable miss ratio is 4 to 8. Beyond 8, the miss ratio is likely to decrease very little. After a great number of simulations, Smith [11] presented practical values for the miss ratio as a function of cache size and line size which are listed in Table 1. The *Design Target Miss Ratios* (DTMR) shown in Table 1 are proposed for unified caches, instruction caches, and data caches, respectively. The DTMR provide designers with a reference to implement a variety of new systems. It can be used to estimate the performance impact of certain design choices. The models of cache memories for the DTMR assume demand fetch, copy-back caches with a LRU replacement algorithm. They also are full-associative for address mapping,

Cache Type:	Miss Ratio					
Unified	Line Size:					
Size	4	8	16	32	64	128
32	0.717	0.556	0.5	0.75		
64	0.686	0.488	0.4	0.48	0.72	
128	0.674	0.467	0.35	0.33	0.428	0.686
256	0.643	0.42	0.3	0.258	0.276	0.386
512	0.596	0.39	0.27	0.216	0.197	0.257
1024	0.473	0.309	0.21	0.162	0.137	0.151
2048	0.405	0.258	0.17	0.124	0.098	0.093
4096	0.329	0.193	0.12	0.082	0.059	0.05
8192	0.232	0.135	0.08	0.05	0.033	0.025
16384	0.182	0.103	0.06	0.036	0.23	0.016
32768	0.124	0.07	0.04	0.024	0.014	0.009
Cache Type: Instructions						
32	0.725	0.478	0.33	0.247		
64	0.674	0.438	0.3	0.222	0.191	
128	0.615	0.397	0.27	0.197	0.164	0.157
256	0.592	0.373	0.25	0.177	0.138	0.129
512	0.562	0.348	0.23	0.159	0.119	0.108
1024	0.504	0.308	0.20	0.134	0.098	0.084
2048	0.391	0.234	0.15	0.098	0.068	0.057
4096	0.271	0.161	0.1	0.063	0.043	0.032
8192	0.172	0.1	0.06	0.037	0.023	0.016
16384	0.148	0.085	0.05	0.029	0.018	0.012
32768	0.091	0.052	0.03	0.017	0.01	0.007
Cache Type: Data						
32	0.731	0.611	0.55	0.715		
64	0.66	0.515	0.45	0.495	0.693	
128	0.561	0.412	0.35	0.351	0.467	0.677
256	0.47	0.337	0.28	0.272	0.326	0.456
512	0.345	0.246	0.2	0.191	0.215	0.282
1024	0.283	0.211	0.16	0.138	0.14	0.161
2048	0.256	0.169	0.12	0.094	0.083	0.089
4096	0.247	0.153	0.1	0.07	0.054	0.048
8192	0.214	0.129	0.08	0.053	0.039	0.032
16384	0.161	0.097	0.06	0.039	0.26	0.019
32768	0.108	0.065	0.04	0.025	0.017	0.012

Table 1: The Design Target Miss Ratios

CACHE TYPE ADJUSTMENTS		
Cache Type	Ratio of Miss Rate to Direct Mapping	Ratio of Miss Rate to Full Associative
direct-mapped	1.00	1.515
two-way set-associative	0.78	1.182
four-way set-associative	0.70	1.061
eight-way set-associative	0.67	1.015
full associative	0.66	1.000

Table 2: The Relevant Cache-mapping-type Ratio

except for those with 4 and 8 byte line sizes, which are 4-way set-associative. The cache miss ratio is also related to the mapping methods used. There are three mapping methods: direct-mapped, S-way set-associative, and fully associative. These are described in the next chapter. Values in Table 2 express the relative ratios of miss rates based on both the direct-mapped and full associative mapping methods. These cache type adjustments originally are from [30]. They are based on the direct-mapped method, and are expanded to be used for those based on the full-associative method. Since the miss ratios shown in Table 1 are based on the full associative model, in order to estimate the actual miss ratio of other systems, the final actual miss ratio can be obtained by multiplying the given miss ratio found in Table 1 by the corresponding relevant cache-mapping-type ratio from column three labeled *Ratio of Miss Rate to Full Associative* of Table 2.

2.5 A Survey of Cache Design

Since IBM Corporation introduced the first commercial cache memory in its System 360/85 to bridge the speed gap between the processor and main memory, various cache memories have been employed in different types of computers to achieve higher performance. A number of approaches have been used for developing high-performance cache memories. Although the operation of a typical cache memory seems relatively simple in concept, implementation of a realistic cache memory is quite complex, involving many factors which influence cache performance. These factors involve internal factors such as cache capacity, line size, address mapping strategy, fetch algorithm, placement algorithm, replacement algorithm, as well as the swapping algorithm, and external factors or system factors: processor organization, hierarchical memory organization, as well as the interconnection network, such as the system bus. For supercomputers, synchronization is a more serious problem since at least two or more processors are embedded in the system. Therefore, attempting to evaluate cache performance exactly in a realistic computer system is quite difficult. We can, however, use approximate models for evaluation of cache behavior and performance.

Cache performance can be described with reference to two aspects [9]: cache miss rate and access time. The first aspect is cache access time — the time required for the processor to get information from or store information into the cache. Cache access time depends not only on the design itself but also on the technology used in cache design. Therefore, the effect of design changes on access time is difficult to predict without specifying the circuit technology used. The second aspect is the miss ratio of the cache memory — the fraction of all memory references attempting to access data which are not resident in the cache memory. In general, every cache miss makes the processor wait until the desired data can be received. The miss ratio is related not only to how the cache design affects the number of misses, but also to how the machine design, including hardware and software, affects the number of cache references (main memory references). For example, the cache miss ratio depends on the program locality implied by software and the amount of information (one word, two words etc.) obtained by the processor at a cache reference.

Many computer systems (almost all modern supercomputer and large computer systems) have cache memories of various designs to bridge the speed gap between processor and main memory in order to improve system performance. This section presents a survey of cache memories and their performance in several typical cache-based computer systems.

A high-speed cache memory was employed in the IBM System 370 Model 168. The cache was available in a size of either 8k or 16k bytes. The 8K-byte cache memory had a cycle time of 80 ns (the same as the machine cycle time) for accessing 4-byte data. It was organized into 64 sets as a 4-way set-associative cache. The *write-through* scheme was used for updating the main memory. The average miss ratio was about 7 percent [27], and the miss ratio prediction, according to the DTMR, is 5.3 percent.

The IBM 3033 has a 64k-byte cache memory for both instructions and data with 57 ns cycle time. This large, high-speed cache memory is one of the main reasons for the high performance enhancement of the 3033. This cache is organized into 64 sets as a 16-way associative cache. The line size of the IBM 3033 is 64 bytes. Also the *write-through* policy is employed in the IBM 3033. In this system, the main memory is divided into 8 modules so that main memory can transfer a line by interleaving [5].

The VAX-11/780 is a 32-bit high-performance minicomputer first introduced by DEC in 1978. Its cache has 8k byte capacity organized into 512 sets, two lines per set, and 8 bytes (4 bytes per word) in each line [5]. For the cache memory of the VAX-11/780, a distinction is made between a read and a write miss. If there is a read miss, the required line has to be retrieved from the main memory and written into the data cache. If two lines in the given set are full, some sort of line replacement strategy has to be employed to determine which line is swapped with the new required line. The VAX-11/780 cache memory uses a *random replacement* strategy as its policy for updating the line. If there is a miss caused by a write operation, only the referenced location of the main memory is updated. This data cache uses a buffered *write-through* policy. The miss ratio of VAX-11/780 was measured to be about 13.05 percent [34], and it is also estimated to be 13.5 percent by the DTMR.

Today cache memories have been integrated with their corresponding microprocessors on a single chip, giving so-called *on-chip* cache memories. The Z80000 microprocessor produced by Zilog in 1985 includes a 256 byte on-chip cache memory which is organized into 16 lines, 16 bytes each, as a fully associative cache. The maximum clock frequency for the Z80000 is 25 MHZ, and when the Z80000 fetches from its cache, only one system clock cycle is required [28]. The *least recently used line* (LRU) replacement algorithm is used to choose the line to be replaced by the new one from the main memory in the case of line-miss occurrence. The *write-through* algorithm is used in this cache for its writing strategy. When there is a miss caused by a write operation, only the main memory is updated. This cache has a miss ratio of 25 per cent for a no burst transfer mode and 12 percent for a burst transfer mode [29]. It is predicted to have, as a unified-cache, a miss ratio of 30 per cent using the DTMR.

A cache memory has also been applied to the Balance multiprocessor system introduced by Sequent Computer Systems Inc. in 1988 [37]. This multiprocessor system can pool up to thirty 32-bit processors with a shared main memory. A subsystem in this system is composed of an NS32032 microprocessor, an NS32081 floating-point unit, and an NS32082 paged virtual memory management unit, produced by National Semiconductor. In addition, each subsystem has an 8k-byte two-way set-associative cache memory to achieve a high performance while minimizing bus traffic. In this cache, with a 50 ns cycle time, there are 512 sets, two lines each, and 8 bytes per line. The *write-through* policy is employed to keep all the copies in the system consistent. Whenever there is write request from one processor in the system, this request with the corresponding address is sent to update stale data in the shared memory while it is broadcast to all the caches to see if there are any copies of the data to be updated. If so, the corresponding cache controller invalidates the affected line. The miss ratio of a single-thread cache memory is 15 per cent [37], while the predicted miss ratio from the DTMR is 15.9 per cent.

Since the cache miss ratio is very dependent on the programs that execute on the cache-based systems and the models in [11] are ideal (in general, a real cache memory is more complicated, and there are more factors to be considered), we can see that our design target miss ratios are slightly higher than seen in simulations described above, and close to those from measured results, such as for the VAX-11/780, which lends some credibility to the use of the DTMR as a reasonable estimator of cache performance, as noted in [11]. Thus, the set of design target miss ratios is very useful for design and implementation of a possibly new cache or architecture. Also we can see that the line sizes of the systems discussed above seem too small. A larger line size provides a lower miss ratio under a fixed cache size. It is clear that caches using set-associativity have lower a miss ratio than those using the direct-mapped method. Another problem is that the above systems which use set-associativity have a small set size, which affects the cache miss ratios. In addition, for implementations of existing cache memories, almost all caches are implemented in either multi-chip or on-chip configurations. In the case of chip sets, several chips, including one cache controller and several high-speed static RAM chips, are used to build a cache memory. This kind of cache memory is designed for special processors and has a fixed cache size. They do not have much flexibility; for example, the cache size can not be changed after the cache controller is designed, and they have longer delay time between the cache controller and RAM chips. An on-chip cache does not have a delay penalty due to interconnection between the chips of a multi-chip cache memory, but on-chip caches have the same problem of inflexibility as do multi-chip cache memories. In addition, this kind of cache in general has only a small capacity using today's technology, which leads to a higher miss ratio.

Using VLSI technology, we can make tradeoffs to design a novel cache memory which has a larger cache size, larger line size, and higher set-associativity on a single chip with little delay penalty by eliminating the wire-connection delay between the cache controller and the cache data memory. Multiple uniform cache chips can be used to build cache systems of various sizes, associated with one processor. This cache system can be used as a traditional unified cache for both instructions and data, or as separate instructions or data cache.

3 IMPLEMENTATION OF THE CACHE ALGORITHMS

3.1 Cache Design Parameters

Typically, a cache memory system can capture well over 90 percent of all references to main memory. Optimization of the cache design parameters is very important to decrease the cost/performance ratio for high-performance cache memories.

Optimizing the design of cache memory has four aspects [9]:

1. maximizing the hit ratio
2. minimizing the access time to cache data
3. minimizing delay due to a cache miss
4. minimizing the overhead of updating main memory and maintaining cache coherence

In addition, for cache memories for multiprocessor systems, consideration has to be taken to maximize bus and shared-memory bandwidth and to minimize the bus bandwidth required by each processor in order to maximize the system performance. There are also trade-offs which depend on the technology of implementation for the cache; for example, between hit ratio and access time.

There are many factors to be considered during cache design which affect system performance. Parameters for cache design are classified into intrinsic and extrinsic parameters [5]. Effective memory speed and cost are two intrinsic parameters. Extrinsic parameters, such as hit ratios, control algorithms, etc., are selected based on the results of experimental data and simulation, and are variables which must be considered for the system design,

Of all the considerations which are related to cache memory, the following are mainly considered during design since cache performance is sensitive to choices concerning these aspects:

1. Fetch policies
2. Mapping policies
3. Replacement policies
4. Swapping policies
5. Hit ratio and access time
6. Cache memory capacity
7. Line size
8. Cache data path width
9. Main memory organization

Fetch algorithms are used to determine when the system brings information into the cache memory. In general, the major fetch algorithms are demand-fetch and prefetch. Under the demand fetch algorithm, a line is fetched only if it is needed. The prefetch algorithm, on other hand, gets information before it is needed. Therefore, the prefetch algorithm is based on some kind of prediction about which line will be used next. It must be designed carefully if the machine performance is to be improved rather than degraded [9]. Implementation of a prefetch algorithm is usually more complicated than demand fetch.

Mapping policies are used to translate the logical address space to real address space. Efficient address translation schemes should accomplish address translation in such a way as to minimize the apparent access time. Information generally is obtained from the cache associatively; larger associative memory is more expensive and slower. Hence, there must be some trade-off of associativity during cache design, in terms of the design and technologies that are employed. A mapping such that any of the lines in main memory can be mapped into any line slots in cache memory is called a *full associative mapping.* That is, a line of main memory may be mapped into any location of the cache memory. Typically, length of a line in cache memory is as the same as that of main memory. If the cache memory is full and there is a miss, the requested line can be transferred into any line slot of cache memory from main memory, in a manner depending on the replacement policy employed. Thus this mapping provides the minimum probability for line slot contention problems and the largest hit ratio for a given problem. However, having one comparator per address tag makes it very difficult and costly to implement, especially in a large cache memory.

A direct-mapped cache has only one comparator which is connected to all the address tags in cache memory. Each time only one address tag can be selected to compare with the address from the processor. This mapping is a many-to-one mapping. That is, any given line in main memory can reside logically only in one specified line slot in cache memory. A direct-mapped cache memory mandates a fixed replacement policy; if there is a line miss, both the cache tag and the corresponding line are replaced with the requested main memory address and its line. This mapping has the highest probability of cache memory slot contention since there is a fixed replacement scheme. Furthermore, it generally has a relatively low hit ratio. Unlike the full-associative mapping, it is quite simple and easy to implement.

A third mapping method is an S-way set-associative mapping, which is a hybrid of the direct-mapped and full-associative methods. An S-way set-associative cache has multiple sets which can be selected by direct-mapping, and there are S lines slots in each set which can be simultaneously compared with the address from the processor. In this mapping system, there are S comparators, a comparator for each "way". Set-associative mapping has a reasonable implementation complexity and hit ratio. Increasing the cache size of a set-associative cache gives a greater hit ratio than increasing the depth of a direct-mapped system. On other hand, increasing the number of sets, or ways, of a set-associative cache memory also gives a greater hit ratio. Hence many high-performance cache memories, especially large scale caches, adopt the set-associative mapping mechanism as a compromise between complexity and performance. More details about S-way set-associative mapping are given in the next chapter.

An optimal replacement policy would predict the line which will be used in cache memory (or a given set) furthest in the future and which consequently should be discarded when the cache memory (or a given set in cache memory) is full and a cache miss occurs. This policy would keep data in the cache optimized for the highest hit ratio, and the maximum system throughput. However, this optimal replacement policy can not be implemented since it requires a prediction of the future behavior of the running programs. Therefore, some approximation has to be made. There are three types of practical replacement algorithms commonly used for cache memory systems; first-in first-out (FIFO), random, and least recently used (LRU) line replacement, to approximate this function. The FIFO algorithm is based on the principle that the first line to be referred is predicted to be the line not to be used in cache memory (or in a given set) furthest in the future, and that this line is replaced by the new one from main memory. This algorithm does not really reflect the program locality very well, since the first line may be used frequently, but it is easy to implement. The *random* scheme is based on a random number from a random number generator to create the line number of a line which is replaced by a new one whenever there is a replacement need. A cache memory employing this algorithm typically has a low hit ratio since this algorithm is not able to reflect the program locality. The *least recently used line* replacement algorithm, which looks backward (past), is usually able to reflect the program locality well since it is based on historical line usage. That is, the least used line in the recent past is replaced by the requested line from main memory. Since this algorithm requires more stored information about the past, it is more difficult to implement in hardware, especially in a large scale cache memory. A variation, an approximation of the LRU algorithm, can be used to simplify the hardware implementation. This variation is based on the fact that if a line has not been referenced over a certain time period, it is less likely to be needed next than lines in cache memory (or in a given set) that have been referenced in that period. More details of the least recently used line replacement algorithm are described in the next section. No one best algorithm exists from the practical replacement algorithms [3]. Some algorithm, compared with the other algorithms, is better for particular classes of problems and poorer for other classes. However, in general, the LRU algorithm is clearly the best choice for most applications, since it is based on historical line usage (the recent past appears to be a good estimate of the near future), it works well, and it increases the hit ratio when the number of lines is increased.

Swapping algorithms are designed for transferring a new line from the main memory to the cache when the requested information is not in the cache. Typically, there are two kinds of swapping algorithms: *write-through* and *copy-back*. In the *write-through* scheme, a processor write to cache memory is immediately written through to main memory as well. Therefore, the information in both cache memory and main memory is always consistent. Furthermore in a multiprocessor environment, it can handle multiple-cache coherence in an easy way. Unlike the *write-through* scheme, the *copy-back* scheme (without line miss occurrences) only updates the copies of requested data in cache memory without disturbing main memory. Whenever there is a line-miss, cache memory copies back the line to be overwritten to main memory before transferring the requested line to cache memory. It can reduce traffic between cache memory and main memory. However, it requires more complicated logic; and there is a coherence problem between cache memory and main memory, and potentially between multi-caches in a multiprocessor system. In contrast, the write-through method has higher traffic between cache memory and main memory since write operations vary from 10 percent to 30 percent out of total references, depending on processor architecture and the particular set of applications. The average percentage of write operations in [9] is 16.

The hit ratio for a cache memory is defined as the probability, or the fraction of times, that a memory request is found in cache memory. If we define the probability of all the references to memory as 1, the miss ratio of cache memory is (1-hit ratio). The hit ratio for a cache memory is one of the most important factors for the performance evaluation of cache memory. Other important factors affecting the cache performance are the access time for the cache memory, including time to search the directory, and the cache memory cycle time, which is defined as the time the processor accesses information in cache memory. The access time of a cache memory is affected not only by the architecture, or design (including all the algorithms and parameters selected in cache design and implementation of the algorithms in hardware), but also by the technology adopted (bipolar, CMOS, etc).

The cache capacity is usually dictated by many factors having to do with the system cost and performance. In general, a large cache capacity can produce a higher hit ratio, and in turn a better performance. However, there are some limitations on cache size beyond which cache memory has either a high cost or performance decreases due to the long access time.

The line size of cache memory is one of the most important parameters which sensitively affect cache performance. There are a number of trade-offs for a reasonable line size in terms of architecture and technology. Using VLSI technology, a larger line size is preferred because it achieves a lower miss ratio without much extra cost. But if it is too large, it increases line transfer time and, in turn, decreases system speed even if the hit ratio is increased. It also depends on the data path width between cache and main memory.

The cache data path width must be considered during the design process since it directly determines the time required when a line is transferred from main memory to cache memory. From the point of view of performance, the cache data path should be as wide as possible. It is clear, however, that cache data path is expensive. Doubling the path width means doubling the number of lines in and out of the cache and all the associated circuitry. The path width is critically important to caches implemented using VLSI technology because of the limited number of I/O pins on a chip. Hence, a trade-off of the cache data path width has to be made during the cache design to achieve a reasonable cost/performance.

Although the use of cache memory in computer systems can greatly reduce direct references to main memory, memory traffic is still a very significant performance factor, especially in a multiprocessor system. Memory traffic consists of two components: fetch traffic and write-through or copy-back traffic. The fetch traffic arises from the transfer of data from the main memory to the cache while the write-through or copy-back traffic is from the cache to the main memory. The fetch traffic can be obtained by multiplying the miss ratio by the line size to get traffic in bytes/reference. The write-through traffic can be calculated by multiplying a write ratio (the ratio of writes to total references) by the number of bytes per write operation. Similarly, the copy-back traffic can be determined by multiplying the miss ratio by the line size, since a line miss causes writing of an existing cache line in the cache into the main memory before transferring the requested missing line to the cache. For evaluation of a cache-based multiprocessor system with a single bus, a *bus utilization* can be used to estimate the memory traffic. The bus utilization is defined as the ratio of time spent doing useful work to the total run time of the bus.

Since decreasing memory traffic or transfer time during a line miss can increase the system performance, optimization of the organization of both the main memory and interconnection network is a key factor for high system performance and low cost. For the interconnection network, a wide data path can reduce the transfer time, but the cost is much higher. On the other hand, if the main memory is made up of several modules which can operate independently, traffic can be reduced because more than one module can be busy writing at one time. Furthermore, if modules can transfer different words in a line by interleaving, transferring a line from main memory to cache only takes one main memory cycle. Thus, the main memory bandwidth can be increased while the transfer time is greatly decreased.

3.2 The Structure of the Cache Memory

During the design of the cache memory described here, algorithms and parameters used have been selected carefully, and a number of trade-offs between them have been made in order to achieve high performance. The cache memory system described here is implemented as a single chip. Furthermore, this implementation allows a cache of variable capacity (larger than the capacity of a single cache chip) by using several of the cache memory chips. The single-chip cache memory described here has a capacity of 8K bytes because of silicon area limitations for the 3 micron CMOS technology. The word size for this cache memory is 32 bits since this cache is designed for a 32-bit computer system. A word is not necessarily the smallest unit that the processor can access. The processor can directly access 1, 2, 3, or 4 bytes from the cache. Therefore, it provides more flexibility to computer systems in which the cache is used. It also allows for the possibility that this cache can be used in 16-bit computer systems, provided control signals for the cache can connect with that of the processor with reasonable additional logic. Two clock phases, $CK1$ and $CK2$, are employed to pipeline this system. Each of the clock phases has a minimum cycle period of 36 nanoseconds (derived from simulation)

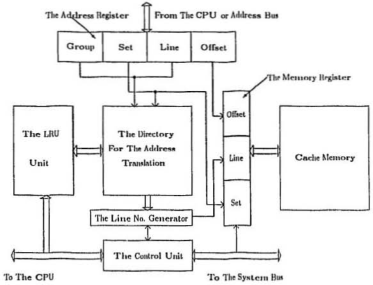


Figure 3: The Basic Cache Memory Structure

in which the associated processor can read an instruction or data from the cache.

Fig. 3 depicts the structure of this cache memory. It is composed of four basic components as follows: the Address Translation Function or Directory, the Line Replacement Unit (LRU), the Cache Memory and the Control Unit.

During $CK1$, the address from the processor is latched in the address register of the cache, and then it is sent to the directory to see if the line containing the requested data is in the cache. If so, the line number generated by the *line number generator*, the set number from the address register, and the word offset in the line are all combined to form a word address for the cache memory and latched into the cache memory register. In addition, the proper byte(s) can be accessed by the processor using both the two least significant bits of the address from the address register and two function bits from the processor, which will be described later. Meanwhile, the LRU unit is updated to indicate that the line referenced is the most recently used one in the given set. During $CK2$, a read/write operation is done. If the requested data is not in the cache, a line miss occurs. The LRU unit is asked to send the least recently used line number in the specified set to the directory, and the directory uses this number to locate its corresponding line slot in the specified set of the directory. Then the contents of this slot are replaced with the group number and the line number in the address register. After replacement, the *line number generator* gives the line number corresponding to this cell to the memory register to transfer the requested line from the main memory to the cache. To obtain the line of information from main memory, a line miss signal is sent to main memory. After the cache receives a "bus use" grant from the system bus controller, the processor is forced to be idle during transfer. There are 8 words (32 bytes) to be transferred from the main memory to the cache memory during a line miss, which would normally take a long time and in turn decreases system performance. In order to reduce the line transfer time, the main memory may be partitioned into several modules, or "interleaved" (in this case, the memory should be partitioned into 8 modules). Whenever there is a line miss, the 8 words of the requested line can then be sent to the cache memory almost simultaneously, with each word coming from a separate module of the main memory. Thus, the transfer time can greatly be decreased. The main memory organization will be discussed later in chapter 5.

3.3 The Address Space Mapping

Since the cache, as the fastest part of the memory hierarchy, is much smaller than the main memory, there has to be a mapping function between the cache address space and that of the main memory. As discussed previously, the direct-mapped method is the simplest to implement, but it has the highest miss ratio of three mapping methods. Therefore, it is not used in this application. Also the fully-associative method requires one comparator per line slot in the directory. This is costly to implement in a large scale cache memory. Also, it may introduce an extra tag-search delay and make the search logic complicated. The set-associative method, a hybrid of the direct-mapped method and the fully-associative method, is used in this mapping mechanism. It involves organizing the cache memory into S sets of N lines per set. When N becomes one, the cache is a fully-associative cache in which there are S sets in total, each consisting of a single line. If S becomes one, the organization of the cache is the direct-mapped cache memory. Since an S-way set-associative cache allows any one of S lines in a referenced set to be replaced on a line miss, this flexibility usually introduces a lower miss ratio without the complexity of a fully-associative cache. Therefore, it is a compromise between complexity and performance.

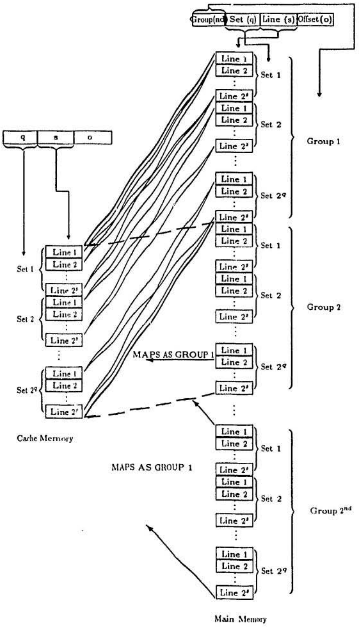


Figure 4: The Set-associative Mapping

3.4 The Set-associative Mapping

The principle of *set-associative* mapping is shown in Fig. 4. The cache memory is divided into 2^q sets with 2^s line slots in each, and the sizes of the sets and lines in the cache memory are the same as those in the main memory. Furthermore the main memory is partitioned into several groups, and the size of each group is equal to the size of the cache memory. Hence, each group contains 2^q sets. Each set slot in the cache memory must be shared by several sets of the main memory. For example, in Fig. 4, the first set in the cache memory is assigned to hold the sets $1, 1+2^q, 1+2 \times 2^q, \cdots$ of the main memory and the second set is assigned to hold the sets $2, 2+2^q, 2+2 \times 2^q, \cdots$ and so forth. Lines within a set of the main memory are associatively mapped into any of the 2^s line slots in the corresponding set of the cache memory. That is, any set in the main memory can only be directly mapped to a specific set of the cache memory and lines in a set are associatively mapped into any of the 2^s line slots in the corresponding set. Sets from different groups can be intermixed within the cache memory; therefore not all the sets of a given group need to be simultaneously resident in the cache memory; similarly lines in the sets from different groups, which are mapped into the same set of the cache memory, can also be intermixed within that set of the cache memory. *E.g.* line 1 of set 1 of group 1 can be assigned to line slot 2 of set 1 in the cache memory and line 2 of set 1 of group 2 can also reside in line slot 1 of set 1 at the same time.

In the Fig. 4, we can see that a memory address is divided into four parts: nd represents the group number, q is the set number, s is the line number and o is the word offset within a line.

3.5 Implementation of the Directory

In general, increasing the degree of associativity of a cache memory can decrease the miss ratio of a cache. In order to obtain high performance, an 8-way set-associative mapping is employed in this directory to achieve a high hit rate without the extra delay penalty while searching the directory. This directory can map the main memory address space (32 bit) to the cache memory address space (13 bit) in a maximum of 16 nanoseconds, including the delay of the LRU unit, determined by simulation. Fig. 5 shows the organization of the address mapping directory with set associative mapping. When the processor requests a read/write operation, the logical address is mapped into the cache memory address by searching the directory. This directory has a tag array of 32 sets with 8 line slots in each set. Each set is represented by a row of the tag array and the 8 line slots in a given set are indicated by columns of the tag array. Therefore, this directory is an 8-way set-associative directory in which each column represents one way.

There is a $MATCH$ signal for each column of slots. If the signal is set, the requested line slot is in this column of slots. Among the 8 $MATCH$ signals, each of which connects to a column of the tag array, there is only one $MATCH$ signal valid at any time since only one slot may be selected. All the $MATCH$ lines are connected to a line number generator. The *line number generator* can translate the *ith* column number, at which the corresponding $MATCH$ signal is

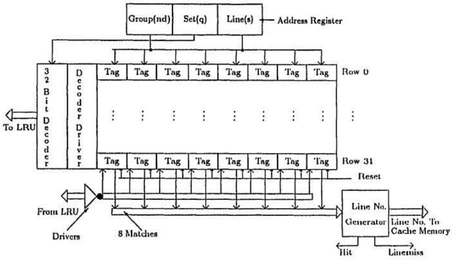


Figure 5: The Directory

valid, into a binary number *i* forming the requested line number for the cache memory. That is, the slot at the position in the *ith* row and the *jth* column can produce the line number *j* of the *jth* line of the *ith* set of the cache memory via the line number generator. In each line slot, the group number is concatenated with the line number (*nd+s*) of a logical address, which indicates that the specified line of the main memory resides in the cache line indicated by the line slot. Whenever a requested set is selected through the 32-bit directory decoder after an address is latched into the address register, the contents of the 8 line slots in the selected set are simultaneously compared with *nd+s* of the address register. If the contents of any one of the 8 line slots are the same as *nd+s*, then the requested data are in the line of the selected set, and the corresponding $MATCH$ signal should become valid to make the line number generator produce the requested line number for the cache memory. A HIT flag is also generated by the line number generator to indicate that the requested data are in the cache. The line number is combined with the set number and the word offset in the line from the address register to form the required word address of the cache memory. Meanwhile, the HIT flag informs the LRU unit to update the records in the selected set. Otherwise, a $MISS$ flag is set to indicate that the requested data are not in the cache, at which time there are three tasks to be done:

- Invoke the LRU replacement unit to find the least recently used line in the selected set of the cache. This line will be replaced by a new one when the requested data are available from the main memory.

- Inform the CPU that it must be idle during the line replacement.
- Request the main memory to transfer the required line to the cache.

Fig. 6 shows a simulation for the tag array of the directory for the case where a line miss occurs during a read/write operation, and the address residing in slot i of row j is replaced with that in the *address register*. Signals B_0 to B_{21} represent the *nd+s* from the address register. SEL_j is a signal from the decoder to select row j of the directory. A $WORD_i$ is a signal from the LRU unit to update the line slot in row j and column i of the tag array during a line miss; and a HIT_i is the *ith* $MATCH$ line of the directory in Fig. 5, indicating whether or not the corresponding column is matched. After the tag array is reset by the RES, a miss (a low HIT_i) is produced since the *jth* row of tags selected by SEL_j is empty. The signal $WORD_i$ from the LRU unit updates the slot in row j and column i of the directory. After a delay of 6 nanoseconds, the HIT_i signal becomes high to indicate that the updating has been finished. When the signals on B_0—B_{21} are changed to a new address and the new address is not found in the directory, then the HIT_i signal becomes low after a 3ns delay.

3.5.1 The Line Slot of the Directory

As described before, the directory is composed of line slots. Each of the line slots is used to store the group number and line number (*nd+s*) of a given main memory address, which indicates that the corresponding line from the main memory is in the cache memory. For each line slot, there is a 22-bit built-in comparator to be

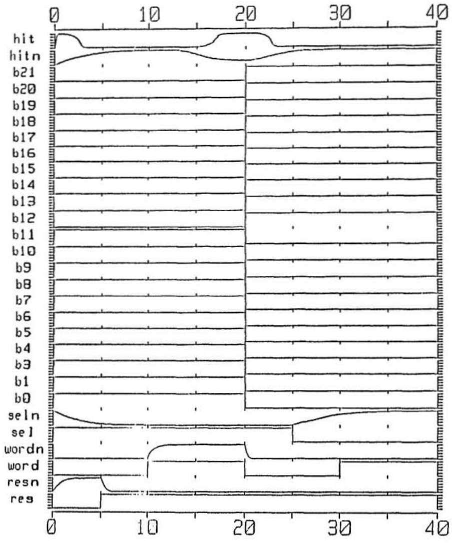


Figure 6: Simulation of the Tag Array

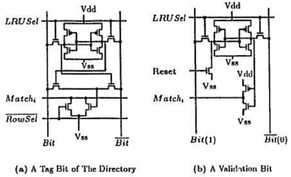


Figure 7: The Directory Tag

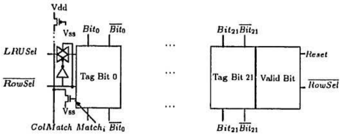


Figure 8: The Tag Bit and Valid Bit of the Directory

used for parallel comparison of its contents with the *nd+s* bits of a main memory address. The organization of a line slot (or line tag) is shown in Fig. 7. There are 22 bits for *nd+s* (19 bits for *nd* and 3 bits for *s*) and one *valid* bit. If the valid bit is reset, the content of this slot cannot be compared with *nd+s* and the directory simply sets a line miss. Otherwise, if $\overline{ROWSEL}$ from the inverted driver connected to the *32-bit decoder* is at logical 0, the contents of the line slot are compared with *nd+s* to see if the requested data reside in the cache. If all the bits of a line slot match the *nd+s* on the lines $(BIT_0, \overline{BIT_0}) - (BIT_{21}, \overline{BIT_{21}})$, the $MATCH_i$ signal for the line slot becomes high to turn on the N type device so that the $\overline{COLMATCH}$ signal is pulled low. If any bit of the line slot does not match the corresponding bit of *nd+s*, the $MATCH_i$ signal for the line slot remains low so that $\overline{COLMATCH}$ is high. If there is a line miss (there is no line slot matching with *nd+s* in a selected set) and that line slot in Fig. 7 is chosen by the LRU unit as the least recently used line slot, the LRU selection signal $LRUSEL$ becomes high. Since at this time $\overline{ROWSEL}$ turns on the pass gate, this line slot is replaced with *nd+s* so that the missing line will be transferred from the main memory into this cache line.

Fig. 8 (a) shows the circuit for one line slot bit. If the $\overline{ROWSEL}$ line which is connected to one output of the 32-bit decoder through an inverted driver is at logical 1 (meaning this line slot is not selected by the decoder), the $MATCH_i$ line remains low so that no comparison can be done in any case (it can be considered to mean "not-match"). If $\overline{ROWSEL}$ becomes low, this cell becomes a normal content addressable memory cell. For a comparison operation, $DATA$ is applied on the BIT line and $\overline{DATA}$ on the $\overline{BIT}$ line. If the datum matches the value in the bit, then the match transistor A remains turned off so that $MATCH_i$ is high for this bit. In a line slot, all the $MATCH_i$ lines are cascaded together. If any bit of the line slot does not match the value on the bit input, the match transistor for this bit pulls down the $MATCH_i$ line of this line slot, previously precharged by the validation bit, indicating that this slot does not match the *nd+s*. For a replacement operation, if there is no slot match at all in the selected set, a line miss has occurred. The LRU unit determines which line slot is to be updated with *nd+s* by asserting the $LRUSEL$ line for the corresponding line slot in the set selected by $\overline{ROWSEL}$. As shown in Fig. 8 (a), when the $LRUSEL$ line is asserted, the values on BIT and $\overline{BIT}$ change the state of this bit. After a change of the contents of this line slot, the contents of the line slot always match *nd+s*. Thus, the $COLMATCH$ becomes low to cause the *line number generator* to create a miss line number to transfer the requested line from the main memory to this cache line. Note that all the $COLMATCH$ lines in one column of the directory are wire-ORed. This means that there is only one $MATCH$ line for each column of the directory. If any one of the $COLMATCH$ signals in a column switches to a low value, the $MATCH$ line of this column becomes low to indicate that one of the line slots in this column matches the *nd+s*. The exact position of the line slot in this column is located by the *32-bit directory decoder*. For this 8-way set associative directory, there are 8 $MATCH$ lines connecting to the *line number generator* which produces in turn the corresponding line number for the cache memory.

Fig. 8 (b) demonstrates the function of the valid bit. The signal BIT is set to logical 1 while the signal $\overline{BIT}$ is set to logical 0. After applying a *reset* signal to the $RESET$ line, the input of inverter B becomes high to discharge the $MATCH_i$ line so that the line slot can not be compared with the *nd+s*. If the $LRUSEL$ line is asserted, the input of inverter B becomes low so that the $MATCH_i$ line is charged to make this slot active for comparison when $\overline{ROWSEL}$ is low. (This means that the set to which this slot belongs is selected). In this case, the line slot is valid for searching.

3.5.2 The Address Register

The address register is used to latch addresses from either the associated processor or the system address bus in the case of a multiprocessor system. Fig. 9 shows one bit of the address register. During the ALE (Address Latch Enable) period for the processor, (the period when the address becomes stable), the address from the processor, imposed on inputs AB of the register, is latched in this register. When the cache receives a search interrupt $SEARCHINT$ from other caches, the *clock pulse generator* of the control unit produces a pulse $CK1'$. During $CK1'$, the address imposed on the AB lines from the other cache through the address system bus is latched in this register for an update operation. After being updated, the cache returns the address before the update operation to the address register during a pulse $CK2'$ generated by the *clock pulse generator*. At initialization, the $\overline{reset}$

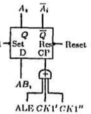
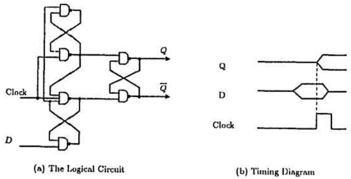


Figure 9: One Bit of the Address Register



(a) The Logical Circuit

(b) Timing Diagram

Figure 10: The D-type Rising-edge-triggered Flip-flop

signal from the processor resets the register. This register is composed of 32 D-type rising edge triggered flip-flops. Fig. 10 (a) shows the logic circuit for this D-type flip-flop. Note that the delay until an address is valid is merely the signal propagation time in the D flip-flop. Fig. 10 (b) is the timing diagram for the D-type flip-flop.

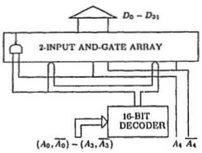


Figure 11: The 32-bit Directory Decoder

3.5.3 The 32-bit Decoder for Set Selection

An address decoder is an essential component for set selection in the set-associative directory. A decoder has n inputs and 2^n outputs. One and only one output will have a value of logical 1 for each combination of input values. In principle, a one-level decoder could be any number of inputs using 2^n gates with n inputs. Unfortunately, in practice, the fan-in limitations and propagation delay require that a large decoder be organized into a multilevel network.

This 32-bit decoder is used to decode 5 bits of the set number from the address register. The inputs of the decoder are the 5-bit set number and its complement directly from the address register. The outputs have 32 bits and at any time only one bit has a value of logical 1. The decoder consists of a 16-bit decoder and 32 2-input NOR-gates as shown in Fig. 11. A 2-input NOR-gate is preferred for the last stage in the multi-level network to allow fast rise time [20]. Address bits $A_0 - A_3$ and their complements $\overline{A_0} - \overline{A_3}$ are imposed on the input lines of the 16-bit

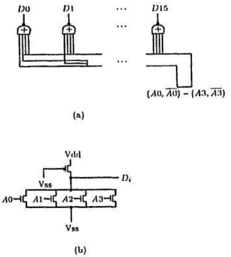


Figure 12: The 16-bit Decoder

decoder while address bits A_4 and $\overline{A_4}$ are sent to the 16 2-input NOR-gates at the second stage of the decoder, respectively. Outputs of the *32-bit decoder* DA_0 to DA_{31} are sent to the directory to select the corresponding set. The 16-bit decoder is illustrated in Fig. 12 (a) where the 4-input NOR-gates are employed. The 4-input NOR-gate is implemented with pseudo-nMOS logic as shown in Fig. 12 (b). There is only a single p-type transistor in the circuit, with the gate connected to V_{ss}. Use of pseudo-nMOS technology can decrease the area and delay time of the decoder since the number of the slow cascaded p-devices is reduced from four to one. The transistor sizes are ratioed carefully to ensure correct logic function and high speed. A simulation for the *32-bit decoder* is shown in Fig. 13, in which the time delay is approximately one nanosecond.

3.5.4 The Line Number Generator

The circuit for the *line number generator* is illustrated in Fig. 14. The function of the *line number generator* is to translate the line numbers of the cache memory from "one-hot code" (or hot code), composed of 8 MATCH lines from the directory, into binary code. In a "hot code" number, there is at most one bit at logical 1; Table 3 is the truth table for the 8-bit "hot code" and its corresponding binary code. Each bit of the complementary "hot code" corresponds to one of 8 match lines $\overline{ColMatch_0}$–$\overline{ColMatch_7}$ from the directory. If all 8 bits of the "hot code" are zeros, then none of the match lines is at logical 0. In this case, the *line number generator* produces a $LINEMISS$. If any one of the $\overline{ColMatch}$ signals is logical 0 after comparison, the generator turns out the corresponding line number of the cache memory in 3-bit binary code on the output lines $LINE_0$ to $LINE_2$; meanwhile, HIT is asserted. This binary line number is latched into the line register of the memory register during ALE', immediately following. The layout of this circuit is shown in Fig. 15. A simulation for this circuit is shown in Fig. 16 in which the time delay for a valid signal is seen to be about 2 nanoseconds.

From the simulations, the total delay before turning out a valid cache address by the directory for both the miss and hit situations is shown in the Table 4. (The abbreviation LNG stands for the Line Number Generator.) If the requested data reside in the cache, the required delay of the directory is about 14ns (Decoder +

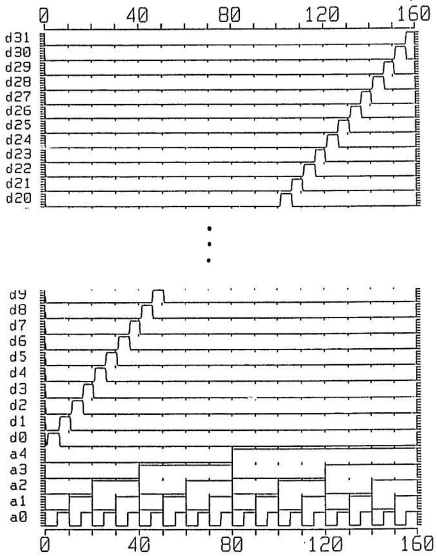


Figure 13: Simulation of the 32-bit Decoder

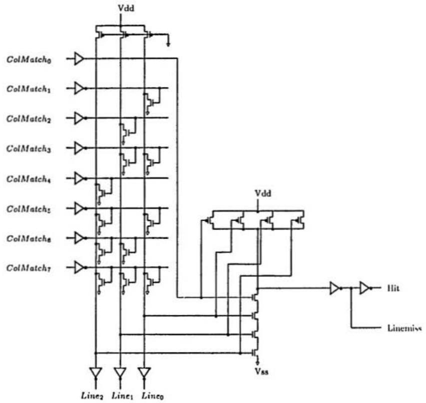


Figure 14: The Line Number Generator

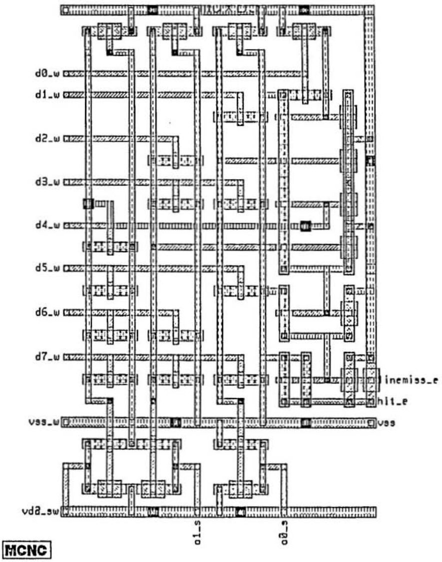


Figure 15: Layout of the Line Number Generator

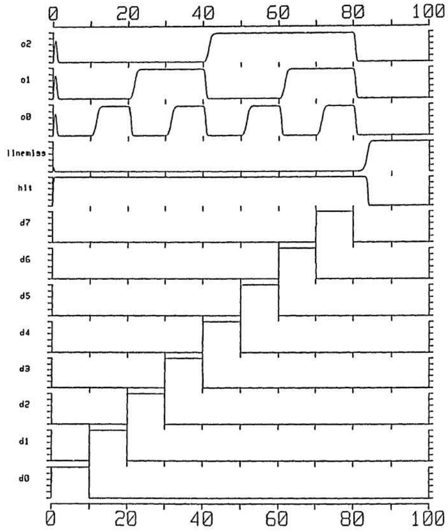


Figure 16: Simulation of the Line Number Generator

Hot Code								Binary Code		
0	0	0	0	0	0	0	0	Line		Miss
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

Table 3: The Truth Table Relating the "Hot Code" and the Binary Code

Operations For A Line Hit			Operations For A Line Miss		
Component Name		Delay Time(ns)	Component Name		Delay Time(ns)
Decoder		1	Decoder		1
Tag Array	comp.	3	Tag Array	comp.	3
	update			update	6
LRU	sendback		LRU	sendback	2
	update	8		update	
LNG		2	LNG		4
Total		14	Total		16

Table 4: Time Delay for the Directory

Tag Array[comp.] + LNG + LRU [update]). Since the updating operations of the LRU unit can overlap with the directory search for next *read/write* operation (note that the cache is pipelined), the actual delay can be less than calculated above. If there is a line miss, the total delay is at least 16ns (Decoder + Tag Array[comp.] + 2LNG + LRU[sendback] + Tag Array[update]).

3.6 The Line Replacement Unit

In a cache system, one of the related problems is to predict which sets of addresses already buffered in the cache memory will be needed furthest in the future because it would then be possible to determine the optimum line to be replaced by a new one from the main memory. Since this algorithm is based on future knowledge of the program's behavior, it cannot be realized in a practical cache memory. Therefore, some approximation must be made to this ideal. In the cache system described here, the *least recently used line replacement* (LRU) algorithm is employed. Under this strategy, the line to which any memory references were made the longest time ago is replaced by a new one. This algorithm is based on the assumption that the line which was referenced the longest time ago is the most likely not to be used in the near future; it relies on the temporal locality of reference characteristic of most programs.

The unit shown in Fig. 17 consists of 32 LRU cells which can be selected individually by the *32-bit directory decoder.* It is organized into four rows, with 8 LRU cells per row. During initialization, the $\overline{RES}$ signal resets all the LRU cells. In general, both $\overline{Delay}$ and $\overline{WRITETHRU}$ are high so that all the N-devices connecting to these signals are closed while all the P-devices are open. Under this condition, the HIT signal can be propagated through the N-devices directly to affect all the $HIT(d)$'s. If there is a signal HIT from the *line number generator* after searching the directory during a read/write operation (the requested data reside in the cache), one of the 32 LRU cells selected by the *32-bit decoder* is updated by

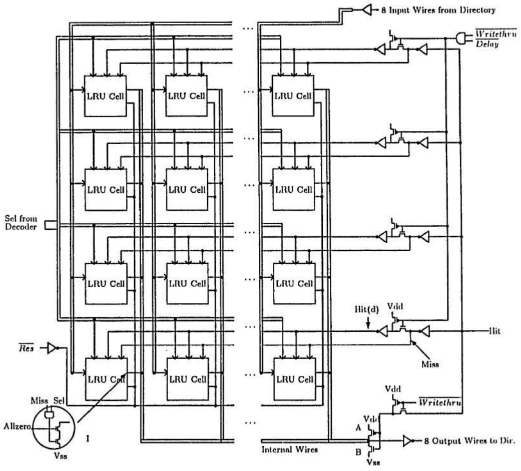


Figure 17: Structure of the LRU Unit

8 signals from the directory, each one corresponding to one line number slot of a given set. (The outputs of this LRU cell are locked by a low $MISS$.) Meanwhile, the HIT signal also passes the nMOS pass-transistor to gates of both devices A and B since $\overline{WRITETHRU}$ is high. The internal wires connecting the outputs of 8 pairs of devices A and B to the outputs of the LRU cells remain low regardless of the status of $ALLZERO$'s shown in subfigure I of Fig. 17 since the N-devices B are closed to discharge the wires, making all 8 output wires to the directory at high level after the inverters. Thus, the directory can be prevented from updating. If HIT becomes low after the directory is searched (that means a line miss occurs), the $MISS$ signal is high so that the LRU cell selected by the *directory decoder* can send the least recently used line slot number of a given set to the directory while the corresponding $HIT(d)$ signal is low to prevent the LRU cell from updating. Meanwhile, the low signal HIT switches devices A on and B off through the N-pass transistor controlled by $\overline{WRITETHRU}$. The 8 internal wires are charged by devices A immediately. These wires are connected to one LRU cell selected by SEL from the decoder. As shown in subfigure I, the transistors controlled by the result of ANDing the signals $MISS$ and SEL are now closed so that the 8 internal wires can be used to evaluate values on the 8 $ALLZERO$'s of the given LRU cell. Only one of 8 $\overline{ALLZERO}$'s in the LRU cell is low to indicate that the corresponding line is the least recently used line in the selected set of the cache. So, the corresponding wire is charged high while other wires are held low by passing charge to ground through the two cascaded nMOS pass-transistors in subfigure I. The active (high)

internal wire is inverted to update the associated line slot, and the remaining keep their line slots in the given set of the directory unchanged. Note that transistors A and B are balanced with those shown in subfigure I so that the operations are reliably completed in minimum time.

In Fig. 17, it is seen that if $\overline{DELAY}$ is low, the unit is prevented from updating by cutting off the HIT signal. Also if $\overline{WRITETHRU}$ becomes low, the unit is prevented from both updating itself and changing line slots in the directory by holding the outputs to the directory high until the signal $\overline{WRITETHRU}$ rises.

$\overline{DELAY}$ is designed to handle the situation that, during a read/write operation, the LRU unit may otherwise be updated incorrectly before the directory turns out a valid result for HIT. This is because at the beginning of an operation the directory decoder selects both a given LRU cell and a corresponding row of the tag array. During the period that the tag array searchs for a line slot in the specified row in which the requested address resides (the *line number generator* has not turned out a valid HIT for this operation and the signal HIT stands at a high level at this time), the status of the LRU cell may be changed by the invalid signal HIT. For instance, assuming that, before ALE is asserted, the output of the *32-bit decoder* is 8 and HIT is logical 1. When ALE is asserted, a new address whose set number is 10 is latched in the address register and broadcast to the decoder immediately. The output of the decoder (here it is 10) is sent simultaneously to both the tag array, to see if the requested data reside in the cache, and the LRU unit, either to update the corresponding LRU cell if the requested data are in the corresponding set, or to send the least recently used line slot number to the directory if the requested data are missing. Because searching the directory takes more time than propagating the signal from the *32-bit decoder* to the LRU unit, the LRU cell corresponding to set 10 is updated before the *line number generator* turns out a new result for HIT since the old HIT still remains effectively at logical 1. This results in an error! The signal $\overline{DELAY}$ is used to prevent the LRU unit from being updated.

As shown in Fig. 17, when $\overline{DELAY}$ is high, all the N-pass transistors connected to $\overline{DELAY}$ through an AND-gate are closed, while the P-pass transistors are open, to maintain all the $HIT(d)$'s the same as the signal HIT from the *line number generator*. Once $\overline{DELAY}$ becomes low, the N-type devices are open and the pathes to $HIT(d)$'s are cut off while $MISS$'s remain as $\overline{HIT}$. Meanwhile, the P-type devices are closed to charge the inverters so that $HIT(d)$'s are discharged via the inverters to prevent the LRU cells from being updated. After a period during which the *line number generator* produces a valid result, $\overline{DELAY}$ changes to logical 1 so that the LRU unit can correctly operate depending on the valid signal HIT. Thus, correctly updating of the LRU unit during a read/write operation is ensured. The valid period of the signal $\overline{DELAY}$ is an inverted 4-nanosecond pulse, which is long enough for a search of the directory and a valid result of the HIT signal to become stable.

The $\overline{DELAY}$ signal can be produced by an *one-shot circuit*, as shown in Fig. 18, which can produce a narrow pulse from a wide pulse. When the input V_i of the circuit is imposed logical 0, the inverter precharges capacitor C through resistor

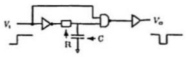


Figure 18: The One-shot Circuit

R. Meanwhile, because the 2-input NAND-gate is locked by V_i, the output of the NAND-gate remains high so that the output V_o is high. Whenever V_i is changed from 0 to 1, since the NAND-gate input connecting to the capacitor still remains high, the output of the NAND gate causes V_o to become low immediately. After the capacitor C is discharged below the threshold of the NAND-gate, the output of the NAND-gate becomes high regardless of V_i so that the output V_o is pulled up to logical 1 by the buffer. The period of the inverted pulse of the circuit output is determined by the RC (resistance-capacitance) product. Resistor R is implemented by a polysilicon wire while capacitor C is formed by an N-device gate. In order to produce the $\overline{DELAY}$ signal, ALE is imposed on V_i, and the output of this circuit is a 4-nanosecond pulse of $\overline{DELAY}$.

The $\overline{WRITETHRU}$ signal is used to ensure that the status of the LRU unit remains unchanged during an update operation required by other caches in a multiprocessor environment. That is, whenever there is an update request from other caches, the cache must not change the contents of both the directory (if the data to be updated are not found in the cache) and the LRU unit (if the data are in the cache). For an update operation, if the requested data are found in the cache, the cache only updates the data without changing records in the LRU unit; otherwise, the cache does nothing for this request. Hence, a signal $\overline{WRITETHRU}$ from the *miss circuit* is used to handle this situation. Whenever there is an update request, the *miss circuit* produces the $\overline{WRITETHRU}$ signal. The signal $\overline{WRITETHRU}$ locks the HIT signal to prevent records in the LRU unit from modification. This operation is similar to that of $\overline{DELAY}$. Meanwhile, it is also used to lock the path to the directory to prevent the directory from changing if the requested data are not found in the cache. That is, during an update request, the $\overline{WRITETHRU}$ signal becomes low to lock the nMOS pass transistor and turn on the P-type device to charge the gates of transistors A and B. Thus, P-devices A are opened while N-devices B are closed to discharge the 8 internal wires so that the outputs to the directory remain high to prevent the directory from being modified during $\overline{WRITETHRU}$.

The output structure of the LRU unit is organized in this way because of the output delay time. Since the internal wire propagation delays caused by the distributed resistance-capacitance product are large and the capacitive load on each wire is heavy (each wire is connected to all the 32 LRU cells), simulations show that the use of standard CMOS design techniques can not obtain high speed.

3.6.1 One LRU Cell

Fig. 19 shows one of 32 LRU cells which is selected by the *32-bit decoder*. Each cell corresponds to a set of the cache. An LRU cell is an 8×8 binary matrix in which there are no bits on the diagonal. Both the row number and the column number represent the line number (indicated by I_1 to I_7 from the directory in Fig. 19) in the set selected by the *32-bit decoder*. Each matrix corresponds to a set in the directory. Changes in the state of a matrix are controlled by the *update control circuit* of the LRU cell. When $HIT(d)$ from the directory is high, the requested line resides in the set selected by SEL from the decoder. The control circuit simultaneously updates the states of the row and the column indicated by the line number. Assuming that the *ith* line in the selected set is requested and the line is in the cache, all the bits of the *ith* row of the corresponding matrix are changed to 1 to record the fact that the corresponding line was the last one used while all the bits in the *ith* column of the matrix are reset to zeros to decrease the number of 1's of other rows: The number of 1's in a row represents the used times for the corresponding line. The largest number, all-one's, indicates that the corresponding line is the most recently used while the smallest number, all-zero's, is the least recently used. In each row, 7 bits are compared. If and only if all the bits in a row are reset to logical 0, the corresponding $ROWMATCH$ line is high. In turn it makes the corresponding signal $\overline{ALLZERO}$ of this row low, when both SEL from the *32-bit decoder* and $MISS$ from the *line number generator* are logical 1, while 7 other $\overline{ALLZERO}$'s remain high.

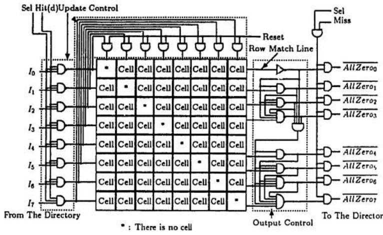


Figure 19: Structure of an LRU Cell

For example, Fig. 20 shows an LRU cell which is a 4 × 4 matrix. Both the rows and the columns are numbered from 0 to 3, which represent the line numbers. The initial status is shown in Fig. 20 (a); line 1 is the most recently used one since the number (here it is 3) of 1's in row 1 of the binary matrix is the largest while line 0 is the least recently used one because the first row is all-zeroes. After reference to line 2, the matrix is updated as shown in Fig. 20 (b), line 2 becomes the most recently used one by setting row 2 full of ones while line 0 still remains at the last position in order after resetting the column 2 full of zeroes. Similarly in Fig. 20 (c), after reference to line 0, the order of the lines changes to 0, 2, 1 and 3. Now line 0 becomes the most recently used one after row 0 is set full of ones while line 3

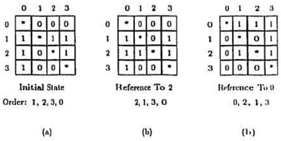


Figure 20: An Example of the LRU Algorithm

becomes all-zeroes after column 0 is written full of zeroes to decrease the numbers of all the rows except row 0.

It is clear that the least recently used line in the selected set of the cache is the one for which the row is entirely equal to 0 and the column is entirely equal to 1. Therefore, if a line miss occurs in that set, the LRU cell specified by both SEL and $MISS$ sends the least recently used line slot number to the directory. In some cases more than one row can be all-zeroes, for example, initially all the rows are zeros after $RESET$. This means that not all the lines in the selected set of the cache memory are full. In this situation, the ***output control*** which is composed of NAND gates in Fig. 19 is used to pass the line whose number is the smallest of all the unused lines in the set as the least recently used line by pulling down its output $\overline{ALLZERO}$ and to lock other unused lines by maintaining their corresponding $\overline{ALLZERO}$'s high. Whenever an output is low (meaning that the line slot number implied by this bit is the least recently used line slot number in the given set), it locks all other outputs which follow by NANDing all the $ROWMATCH$ signals logically behind it. Fig. 21 shows the layout of an LRU cell.

3.6.2 One Bit of the LRU Cell

The logic of one bit of the LRU cell is illustrated in Fig. 22. The bit cell used in the LRU cell is a variation of a static memory bit. If a signal imposed on $ROWSEL$ is high, inverter B is pulled down to 0 while inverter A is brought up to 1. Thus, the $MATCH$ signal becomes low by closing the MATCH transistor. On the other hand, if the $COLSEL$ line is asserted, the A inverter becomes low and B is high. The MATCH transistor is open so that the signal $MATCH$ of this bit is high. Note that the signals $ROWSEL$ and $COLSEL$ must not be asserted at the same time, to prevent this bit from being placed in an indeterminate state. In an LRU cell, all the $MATCH$ lines and all the $ROWSEL$ lines of the LRU bits in a row are connected in series, respectively, and all the $COLSEL$ lines of the bits in a column are connected in series. A row can be selected by I_i from the directory when both HIT and SEL are high. If all the $MATCH$ lines of the bits in a row are high, implying that all the bits in the row are at logical 0, the $ROWMATCH$ line of this row becomes high to indicate that this row has been set to all-zeros. The cache line corresponding to this row may be the least recently used line (the least recently used line in a given set is indicated by a low $\overline{ALLZERO}$ signal of the given LRU cell), depending on whether any of the rows logically before this one have high $ROWMATCH$es.

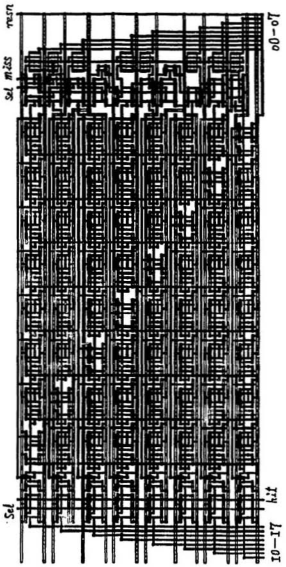


Figure 21: Layout of the LRU Cell

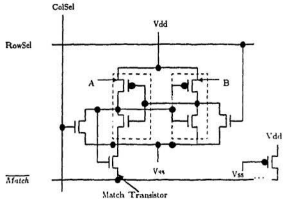


Figure 22: One Bit of the LRU Cell

Fig. 23 shows a simulation result of the LRU unit. The signals from I_0 to I_7 are 8 input lines from the directory and O_0 to O_7 are 8 output lines to the directory. After the signal $RESN$ is valid (low), the unit is reset. It can be seen in the simulation in Fig. 23 that all the LRU unit outputs O_1 to O_7 are high except O_0. This means the cache line implied by the low O_0 output is the least recently used line after initialization, although at this time all the rows in a given cell are zeros, since the first row indicates the smallest line number in the given set. When HIT is logical 1 (note that the low HIT signal implies that $MISS$ is high) and I_0 is high, the first row of a given LRU cell is updated to all-ones. At this time, there are no outputs on the output lines (from O_0 to O_7) of the LRU unit, which indicates that a line miss occurs. During $MISS$, the outputs of the LRU unit are valid. In

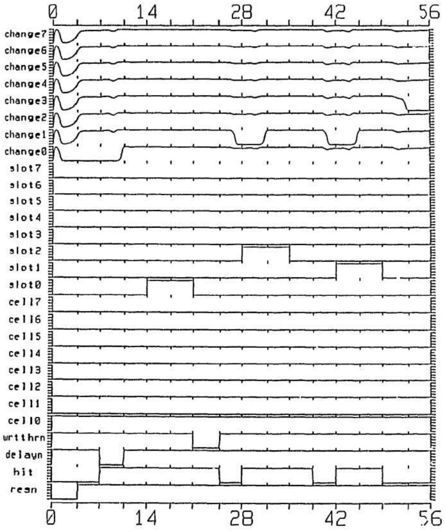


Figure 23: Simulation of the LRU Unit

this case, O_1 becomes low, which indicates the first row has been updated and the line indicated by the second row in the given cell is the least recently used line. After the 3rd row is updated to all-ones during I_2, when $MISS$ is asserted again, the least recently used line is still the line indicated by the second row. It can be seen that at the second $MISS$ the LRU unit produces a low output at O_1. Then after updating the second row by asserting I_1, the LRU unit produces the least recently used line, during the next $MISS$, implied by the 4th row whose output is O_3. From the simulation we can see that the delay time for a valid output is about 3 nanoseconds. Note that the signal $DELAYN$ in the simulation is used as the result of ANDing $\overline{DELAY}$ and $\overline{WRITETHRU}$ to prevent the LRU unit from being updated.

In this chapter, cache algorithms are surveyed. CMOS implementations of algorithms selected for this cache design such as the 8-way set-associative mapping and the least recently used algorithms are discussed. A bit-matrix method is used to simplify the implementation of the LRU algorithm in CMOS. The functions of the Directory and the LRU Unit have been verified by circuit simulation.

4 THE MEMORY AND CONTROL UNITS

One of the most important parts in a cache is the cache buffer, or data storage, which is used to store the most up-to-date data. Its main function is similar to that of a small, high speed *random access memory*. Another is the control unit which determines the internal and external timing of the cache, controls the functions implemented in the previous chapter, and provides the communication functions required for a multiprocessor or uniprocessor environment. This chapter discusses the design and implementation of both the cache buffer and the control unit used in this project.

4.1 Structure of the Memory

Fig. 24 shows the structure of the cache memory in which a row represents one line, with 8 words per line and 32 bits in each word. The row number is in the range 0 to 255. An 8-word line size was chosen for this cache memory, assuming that the associated main memory is partitioned into 8 modules which can transfer a requested line to the cache by interleaving. This main memory organization is more suitable for a multiprocessor system.

A cache memory address is divided into two parts; one part containing the set and line numbers is used to select the specific line in the cache memory through the memory decoder while another part (the offset) in the *register/counter* is used to determine which word or byte(s) in the selected line is (are) accessed.

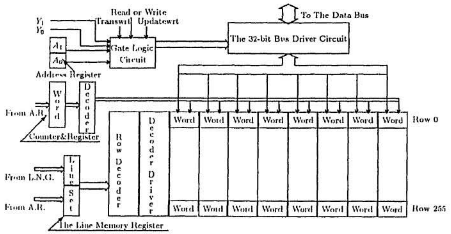


Figure 24: Structure of the Cache Memory

In this system, the smallest element the processor can access is not a word but a byte. There are 4 sizes of data which can directly be accessed by the processor — one, two, three or four byte blocks. The size of the data to be accessed by the processor is determined by a combination of the two least significant bits of the requested data address, A_0 and A_1, and the two function bits, Y_1 and Y_2, which come from the processor. During a processor read/write operation, when the requested data reside in the cache memory, the cache will either send the requested data to the data bus or store the data on the data bus into the cache, depending on the specific operation of the processor. In this case, the *register/counter* performs like a register. The word offset of the address register is latched in this special register, and the requested word is selected via the column decoder of the memory. Otherwise, the *miss flag* is set to indicate that the requested data are not in the

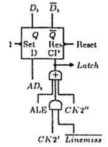


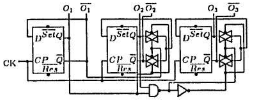
Figure 25: One Bit of the Memory Register

cache. During a line miss, the least recently used line in a given set of the cache memory is overwritten with the requested line from the main memory. In this case, the *register/counter* becomes a counter, controlled by the $TRANSFER$ signal from the main memory, to choose each of the 8 possible word offsets in the selected line in the increasing order (from 0 to 7). For each time at which a word offset is chosen, one of the 8 words in the requested line from the main memory is written to the corresponding location. The main memory sends 8 words of the requested line, by interleaving, for a miss request. Each word being transferred is accompanied by a valid pulse of the $TRANSWRT$ signal, which is used to increment the counter and to require the *bus control circuit* to pass a word on the data bus to the cache memory during transfer.

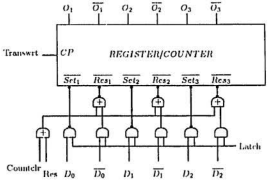
4.1.1 The Cache Memory Register

The *cache memory register* is composed of two parts: one is the *line memory register* and the other is the *counter/register*. Fig. 25 illustrates one bit of the *line* *memory register* which is used for both the line number and the set number of the cache memory. A D-type flip-flop is employed. The signal for latching a cache memory address into the *memory register*, including both the *cache line register* and the *register/counter*, is also described in this figure. Latching a cache memory address in the *memory register* occurs under two conditions. The first condition is that during a read/write operation, the corresponding cache memory address has to be latched in this register after searching the directory. If this condition is satisfied, the ALE' signal produced by the *clock pulse generator* is asserted. The second condition is that when there is an update request from other caches and the requested data are found in the cache, the *clock pulse generator* produces two pulses $CK2'$ and $CK2''$ for this request. The $CK2'$ signal is used to latch the cache address corresponding to the requested data during the update operation. After the operation, $CK2''$ returns the address residing in the memory register before the operation. Whether or not the requested data reside in the cache is indicated by the $\overline{LINEMISS}$ signal from the *line number generator*, after searching the directory during the update request. Thus $CK2'$ has to be ORed with $\overline{LINEMISS}$ to form the latching signal for the update operation. Note that $\overline{LINEMISS}$ is produced by the *line number generator*. It differs from the $MISS$ signal from the *miss flag* which is used to inform the main memory that the data requested by the associated processor are not in the cache.

The logic circuit for the *counter/register* is illustrated in Fig. 26. It consists of 3 D-type falling-edge-triggered flip-flops which are organized as a synchronous



(a) Structure of The Synchronous Counter


O_1 $\overline{O_1}$ O_2 $\overline{O_2}$ O_3 $\overline{O_3}$
Transwrt
CP
$REGISTER/COUNTER$
$\overline{Set_1}$ $\overline{Res_1}$ $\overline{Set_2}$ $\overline{Res_2}$ $\overline{Set_3}$ $\overline{Res_3}$
Latch
Countclr
Res
D_0 $\overline{D_0}$ D_1 $\overline{D_1}$ D_2 $\overline{D_2}$

(b) Functional Block Diagram of The Counter/Register

Figure 26: The Register/Counter Logical Circuit

counter shown in Fig. 26 (a), but it can operate as both a counter and a register in different situations. The logic function block for this circuit is depicted in Fig. 26 (b). The initializing signal, system $RESET$ or $COUNTCLR$ from the *transfer decomposer*, resets this circuit. The word offset of an address on circuit inputs $D_0 - D_2$ and $\overline{D_0} - \overline{D_2}$ can be latched from SET's and RES's of the D-type flip-flops, respectively, when the $LATCH$ signal is active; $LATCH$ is created when there is either a hit, during a read/write operation, or an update request from other caches. The 3-bit outputs $O_0 - O_2$ and $\overline{O_0} - \overline{O_2}$ are sent to an 8-bit memory column decoder implemented with eight 3-input NOR-gates, in a way similar to the 16-bit decoder described previously. When there is a miss, the low $LATCH$ signal locks the data inputs of this circuit and $TRANSWRT$, from the *transfer decomposer*, is imposed on CK of the counter after $COUNTCLR$ resets the *counter/register*. At this time, the *counter/register* behaves like a counter. When each pulse of the $TRANSWRT$ signal is imposed on CK of the counter, the outputs of the counter are not changed until the falling edge of the pulse. Thus, when a word is transferred into the memory during each pulse of the $TRANSWRT$ signal, the corresponding word offset selected by the *memory column decoder* cannot be changed, which guarantees 8 words of the requested line are transferred into correct places in the memory. The layout of the *memory column control*, including the *counter/register* and *8-bit column decoder*, is illustrated in Fig. 27 and its simulation is shown in Fig. 28.

The circuit of the *transfer decomposer* is shown in Fig. 29 (a). Since the number

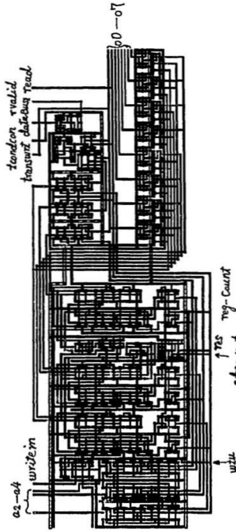


Figure 27: Layout of the Memory Column Control

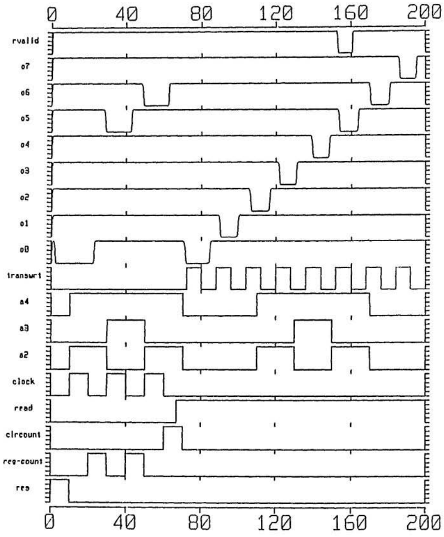
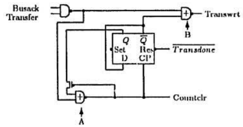
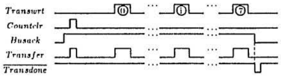


Figure 28: Simulation of the Memory Column Control



(a) The Circuit of The Line Transfer For Miss



(b) The Timing Diagram of The Line Transfer For Miss

Figure 29: Circuit of the Transfer Decomposer

of pins on this chip is limited, the $TRANSFER$ signal from the main memory consists of two parts: the first narrow pulse is used to clear the counter, and the following 8 wide pulses are used to write a line on the data bus into the cache memory, one word per pulse. The decomposer divides the $TRANSFER$ signal into $COUNTCLR$ which clears the counter and $TRANSWRT$ which drives the counter from 0 to 7 and the the *bus driving circuit* during a bus grant indicated by the $BUSACK$ from the system bus controller. Initially, the D-type falling-edge-triggered flip-flop is reset so that its output $\overline{Q}$ is logical 1. $\overline{Q}$ is fed back to the D input of the flip-flop. When the first pulse of the $TRANSFER$ signal (used for clearing the counter) enters the circuit, it is gated through NOR-gate A to create a

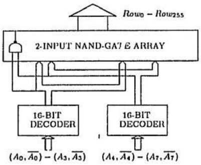

$Row_0 - Row_{255}$
2-INPUT NAND-GATE ARRAY
16-BIT DECODER
16-BIT DECODER
$(A_0,\overline{A_0}) - (A_3,\overline{A_3})$
$(A_4,\overline{A_4}) - (A_7,\overline{A_7})$

Figure 30: The 256-bit Memory Decoder

valid $COUNTCLR$ signal since the other input of the gate is logical 0 at this time, while $TRANSWRT$ is invalid (low) because the 2-input NOR gate B is locked by $\overline{Q}$. When the falling edge of the first pulse of $TRANSFER$ passes gate A, the flip-flop changes its state. A transition from 0 to 1 of its output Q locks NOR-gate A; meanwhile, NOR-gate B becomes unlocked to allow the following pulses of the $TRANSFER$ signal to get through NOR-gate B to the output $TRANSWRT$. After transfer of a requested line from the main memory to the cache memory, the cache control unit produces an inverted pulse $\overline{TRANSDONE}$ to reset the D flip-flop at the transition of $\overline{BUSACK}$ from 0 to 1. Fig. 29 (b) is the timing diagram for operations of the *transfer decomposer* during the transfer of a requested line from main memory.

4.1.2 The 256-bit Row Decoder

This 256-bit decoder is used to decode 8 bits of both the set number and line number from the *memory register* simultaneously, 5 bits for the set number and 3 bits for the line number. It can produce 256-bit outputs, but only one bit of all the outputs is logical 0 at any given time, to be used to select one out of 256 rows of the cache memory. The outputs of the decoder are connected to the inverted row drivers of the memory. The decoder consists of two 16-bit decoders discussed previously and 256 2-input NAND-gates as shown in Fig. 30. The simulation for the 256-bit decoder in Fig. 31 only shows the first 32 outputs of this decoder. From the simulation, there is about 2 nanosecond delay for the decoder stage.

4.1.3 The Cache Memory

A fast static memory is used in the cache memory unit rather than smaller but slower dynamic memory, which also needs to be refreshed. Fig. 32 depicts the organization of the memory. It is split into four memory arrays, two arrays in the upper row and two in the lower row. Both the upper row and lower row arrays are connected to outputs, $\overline{ROWSEL}$'s, of the *256-bit row-decoder* through the inverted row drivers. Intermediate buffers are used between two memory arrays at the same row. When there is a $\overline{ROWSEL}$ signal active, two arrays in the same row are selected simultaneously. Memory organized in this way can reduce delay time and in turn increase the memory access speed. Fig. 33 shows any four adjoining memory bits in an array along with their column selection circuits. The circuit for

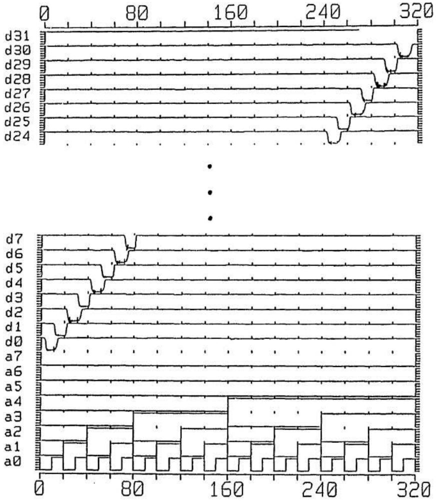


Figure 31: Simulation of the 256-bit Row Decoder

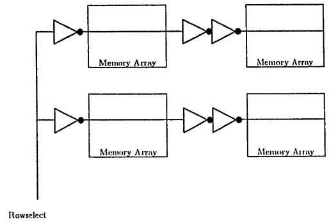


Figure 32: The Memory Arrays

column selection, one for each column of the memory, is quite simple; only two pass transistors are connected to the BIT and $\overline{BIT}$ lines of memory cells in that column, respectively. Note that 32 column selections (one word) are driven simultaneously by one bit of the memory column decoder during access to one word. Therefore, a total of 256 column selections are formed as eight groups by connecting them to 8 output bits of the memory column decoder, respectively. Consequently, one memory array has two groups, each containing one word. Only the cells selected by both row and column selections can be accessed. Fig. 33 shows four static CMOS RAM cells. Each RAM cell consists of two inverters wired together to make a flip-flop; they are connected by two nMOS pass transistors to the BIT and $\overline{BIT}$ lines, respectively. During a read operation, the conducting side of the flip-flop pulls the precharged data line (BIT or $\overline{BIT}$) toward ground through the pass transistors

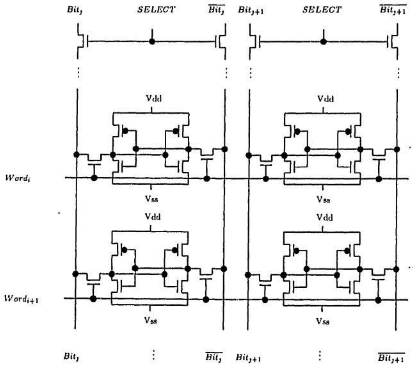


Figure 33: Four Bits of the Memory

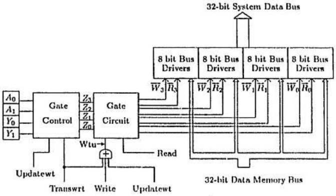


Figure 34: The Data Bus Control Circuit

while the other side remains high. Writing is accomplished by forcing the value in the cell to be the same as that on the data lines.

4.1.4 The Data Bus Control Circuit

The *data bus control circuit* is shown in Fig. 34. The data bus driving circuit has 32-bit dual-port input/output drivers. It is split into 4 components, each of which can control access to one byte. The operation of each component is controlled by a pair of read and write control signals: $(\overline{R_i}, \overline{W_i})$. There are four pairs of control signals, $(\overline{R_0}, \overline{W_0})$, $(\overline{R_1}, \overline{W_1})$, $(\overline{R_2}, \overline{W_2})$, and $(\overline{R_3}, \overline{W_3})$ generated by the *gate circuit*.

As mentioned previously, data in the memory can be accessed as one, two, three or four bytes, respectively, using combinations of Y_0, Y_1, A_0 and A_1 (A_0 and A_1 are in the two least significant bits, bit 0 and bit 1, of the address register). During normal read/write operations of the processor, both $TRANSWRT$ from

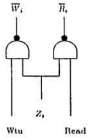


Figure 35: One Bit of the Gate Circuit

INPUT				OUTPUT			
Y_1	Y_0	A_1	A_0	Z_3	Z_2	Z_1	Z_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	1
0	1	0	1	0	0	1	0
0	1	1	0	0	1	0	0
0	1	0	1	1	0	0	0
1	0	0	0	0	0	1	1
1	0	0	1	0	1	1	0
1	0	1	0	1	1	0	0
1	0	1	1	0	0	0	0
1	1	0	0	1	1	1	1
1	1	0	1	0	1	1	1
1	1	1	0	1	1	1	0
1	1	1	1	0	0	0	0

Table 5: The Gate Control Functions

the *transfer decomposer* and $UPDATEWRT$ produced by the cache control unit for updating data requested by other caches are not asserted (both of the signals $TRANSWRT$ and $UPDATEWRT$ are logical 0). Therefore, signals $Z_0 - Z_3$ produced by the *gate control* circuit dominate the read/write operations of the data bus control components via the *gate circuit*. There are four subcircuits producing $\overline{W_i}$ and $\overline{R_i}$ in the *gate circuit*. Fig. 35 shows one subcircuit of the *gate circuit*. A memory write operation is determined by the condition that there is either a write operation required by the associated processor, a transfer operation during a line miss, or an update operation caused by an update request from other caches. If any of these operations is needed, the WTU signal from the 3-input OR-gate is asserted so that if any Z_i is logical 1, and the corresponding write control bit $\overline{W_i}$ is gated out to control an 8-bit data bus driver to allow writing the data on the external data bus into the cache memory. If there is a read request from the processor, $\overline{R_i}$ is valid, causing the corresponding bus drivers to pass the requested data from the cache memory to the outside data bus. Note that the operations of write and read are exclusive, so there is no case that $\overline{W_i}$ and $\overline{R_i}$ are asserted at the same time. The truth table for the signals Z_0 to Z_3 is shown in Table 5.

After simplifying the functions specified in Table 5, we have the output functions, $Z_0 - Z_3$, as follows:

$$Z_0 = \overline{A_0A_1}Y_0 + \overline{A_0A_1}Y_1 + \overline{A_1}Y_0Y_1 \quad (1)$$

$$Z_1 = \overline{A_1}Y_1 + A_0\overline{A_1}Y_0 + \overline{A_0}Y_0Y_1 \quad (2)$$

$$Z_2 = \overline{A_1}Y_0Y_1 + A_0\overline{A_1}Y_1 + \overline{A_0}A_1Y_0 + \overline{A_0}A_1Y_1 \quad (3)$$

$$Z_3 = \overline{A_0}Y_0Y_1 + \overline{A_0}A_1Y_1 + A_0A_1Y_0\overline{Y_1} \quad (4)$$

These functions can be implemented efficiently by PLA. The logic circuit for the *gate control* functions implemented in PLA is illustrated in Fig. 36. The inputs to the circuit are the two least significant bits from the address register and two function bits, and their complements if needed. The outputs of the PLA circuit, $\overline{Z_0}$ to $\overline{Z_3}$, are NANDed with the result of NORing $TRANSWRT$ for transfer-write of a requested line and $UPDATEWRT$ for update-write, to produce the outputs Z_0 to Z_3. During a write operation both $TRANSWRT$ and $UPDATEWRT$ are at logical 0 so that the values of Z_0 to Z_3 are determined by the outputs of the PLA circuit. If either $TRANSWRT$ or $UPDATEWRT$ is asserted, Z_0 to Z_3 are set high to force the bus drivers to overwrite one word (32 bits) on the system data bus into the cache memory. The layout of the *gate control* and *gate circuit* is shown in Fig. 37, and Fig. 38 shows the simulation for these circuits.

The data bus driving circuit is partitioned into four components, each for controlling 8 bits. A component is used to control 8 bits of the data to be written into or read out of the memory. Fig. 39 illustrates one bit of a component. There is a write logic block and a read logic block for each bit of the data bus driver. The write logic is shown in Fig. 39 (a). When $\overline{W_i}$ is low and W_i is high, the value on the $DATA$ line is gated onto the BIT line and the $\overline{BIT}$ lines. Note that sizes of the transistors in this circuit are large enough to write data into the memory at high speed. On the other hand, the read logic shown in Fig. 39 (b) is more complicated

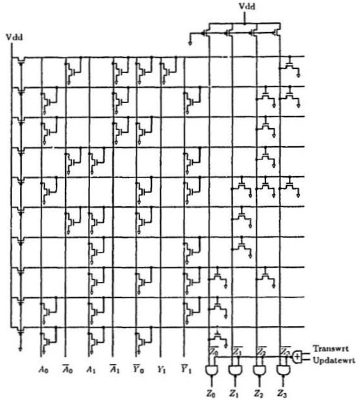


Figure 36: The Gate Control Logic

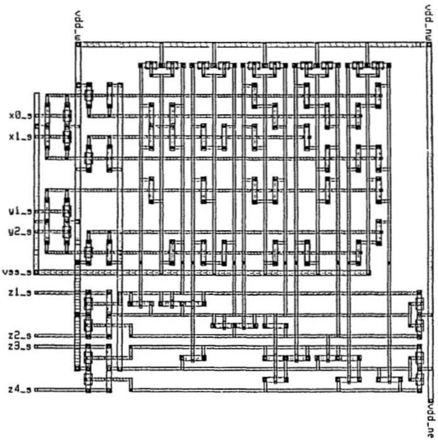


Figure 37: Layout of the Gate Logic

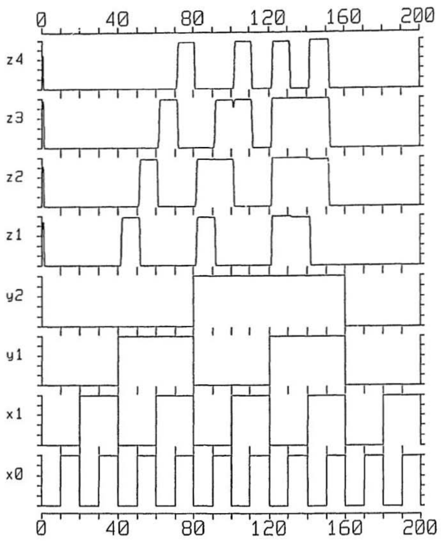
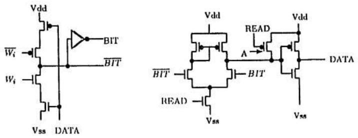


Figure 38: Simulation of the Gate Logic



(a) Write Operation Control (b) Read Operation Control

Figure 39: The Bus Write/Read Operation Control

since the signals from memory cells are usually weak. Therefore, there have to be amplifiers to increase the signal strength for the memory cells. This read circuit is a two-stage amplifier which detects the state of a memory cell. The first stage of the amplifier is a differential sense amplifier which can sense small differences between voltage levels on the BIT and $\overline{BIT}$ lines from the memory and amplify this to provide very fast sensing. The second stage is an inverter which provides further driving capacity and makes the rise time and fall time of the $DATA$ signal shorter. Note that the differential sense amplifier is evaluated while the $READ$ signal is active. In order to obtain a correct output rapidly, the sense amplifier is precharged through transistor A to eliminate the *charge-sharing* effect during no read operations. Simulations for both the write and read logic control of a component are shown in Fig. 40 and Fig. 41. In Fig. 40, when $WRITE$ is high and $READ$ is low, data on the $DATA$ lines are gated to the BIT lines. The delay time is about 3 nanoseconds. In Fig. 41, when $WRITE$ is low and $READ$ is high, data on the BIT lines are quickly placed onto the $DATA$ lines.

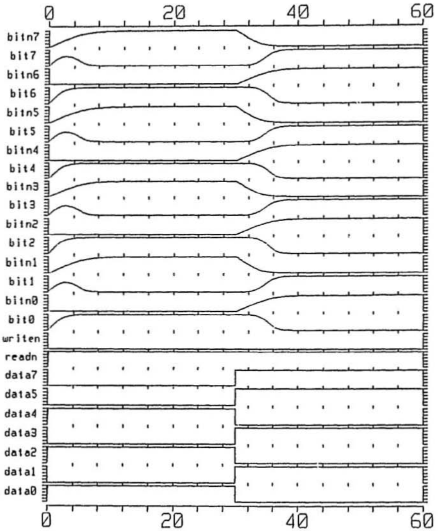


Figure 40: Simulation of the Write Control

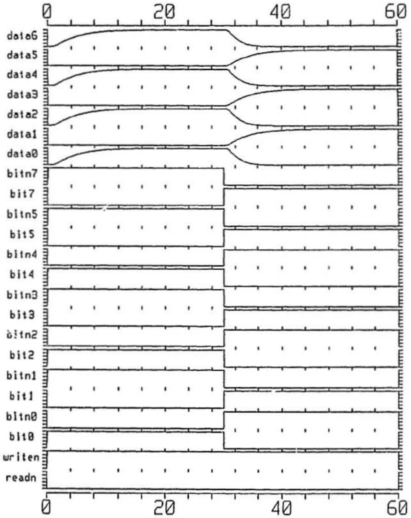


Figure 41: Simulation of the Read Control

4.2 The System Control Unit

The system control unit is used to control all the communications among cache memory and the associated processor, the main memory, as well as the multiple caches in a multiprocessor system. Therefore it is the most important part of this cache memory system. In terms of the communication operations, this unit is logically partitioned into three parts: the normal read/write operation, the update operation, and the miss operation. In this section, more details about the functions of these three parts will be described.

4.2.1 The Regular Read/Write Operations

In order to control the different operations, the clocks and ALE have to produce several clock pulses. A *clock generator* is used for this purpose. First, let us discuss the circuit in Fig. 42 (a), the *single pulse producer*. This circuit produces one complete pulse and its complement on the circuit outputs Q and $\overline{Q}$ from a sequence of pulses on IN after receiving an inverted pulse on $\overline{CP}$. The behavior of this circuit is as follows: $\overline{Q}$ is initially at a high level since the 2-input NAND-gate input connecting to the flip-flop through nMOS pass-transistor A is low regardless of IN. When there is an inverted pulse on $\overline{CP}$, the output of the flip-flop connected to pass transistor A is set to logical 1 and it in turn enables the 2-input NAND-gate since the pass transistor is closed by the high $\overline{Q}$ signal at this time. If IN has a transition from 0 to 1, $\overline{Q}$ becomes low, and it in turn both locks pass transistor A and resets the flip-flop. Although the flip-flop is reset, the NAND-gate input

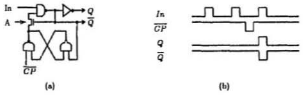


Figure 42: The Single Following Pulse Producer

connected to the pass transistor remains at logical 1 because the path to the input of the NAND-gate is blocked by pass-transistor A. Therefore, the signal at IN passes the NAND-gate to Q and $\overline{Q}$. When the IN signal makes a transition from 1 to 0, the $\overline{Q}$ signal changes to logical 1 from logical 0, which releases the pass transistor A so that a low output from the flip-flop is imposed on the NAND-gate input via transistor A. Thus the 2-input NAND-gate is locked so that the following pulses on IN cannot be propagated to outputs Q and $\overline{Q}$. The timing diagram in Fig. 42 (b) depicts the operation of this circuit.

Whenever there is an update request from other caches via the system bus, the *update circuit* produces an $UPDATE$ signal to inform the cache to update with the data on the system bus. In this case, the *clock generator* produces several pulses to accomplish the update operations. The circuit in Fig. 43 (a) is used to produce a pair of pulses $CK1'$ and $\overline{CK1'}$ from $CK1$. At the beginning of $CK1$, the circuit always checks if there is an $UPDATE$ signal. If the $UPDATE$ signal is found to be low after this check, $CK1'$ is set low while $\overline{CK1'}$ is set high. Otherwise, the high signal $UPDATE$ is locked by a P-type pass-transistor on the input of the NAND-

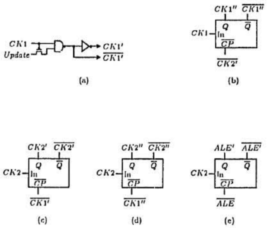


Figure 43: The Clock Pulse Generator

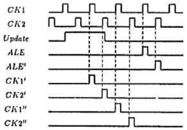


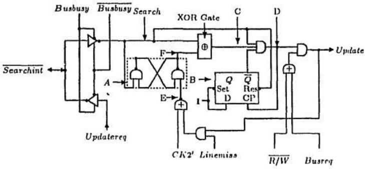
Figure 44: A Timing Diagram for the Clock Pulse Generator

gate so that during $CK1$ any change of the $UPDATE$ signal cannot affect the result of the circuit, and $CK1$ is gated out to create both $CK1'$ and $\overline{CK1'}$. Hence, even if $UPDATE$ is changed during $CK1$, the pulse from $CK1$ still passes the 2-input NAND-gate to make $CK1'$ and $\overline{CK1'}$ be complete pulses. Furthermore, the *pulse generator* also produces other control pulses for an update operation using the *pulse producers*. Three *pulse producers* are employed to produce pairs of signals $CK1''$ and $\overline{CK1}''$, $CK2'$ and $\overline{CK2}'$, as well as $CK2''$ and $\overline{CK2}''$ by imposing different signals on inputs of the circuits, as shown in Fig. 43 (b), (c), and (d), respectively. $CK1'$ is used to latch addresses from the system bus into the *address register* during an update request; $CK2'$ is to latch the corresponding cache addresses from the directory into the *memory register* to update the requested data. $CK1''$ and $CK2''$ are used to return the addresses which reside in both the *address register* and *memory register* before the update operation into the respective registers after the update. The *pulse producer* also generates an ALE' pulse, as shown in Fig: 43 (e), to latch a valid cache address from the directory into the memory register during a normal processor access. In order to produce ALE', $\overline{ALE}$ is imposed on $\overline{CP}$ of the *pulse producer* while $CK2$ is present at IN. Fig. 44 shows a timing diagram for the *pulse generator* in which we can see the relations among these signals.

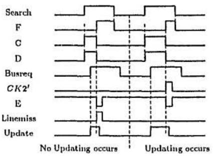
Note that all the inputs from the processor are controlled by the chip select $\overline{CS}$ signal from the associated processor. The cache memory can be accessed only when the cache is selected by the processor with $\overline{CS}$. The update operations, however, are not controlled by $\overline{CS}$. This makes the cache memory used in a cache-based computer system more flexible.

4.2.2 The Update Operations

When the cache is used in a multiprocessor system, there must be a "bus watcher" to watch the system bus to see if there are any other caches requesting to update copies of data. The circuit in Fig. 45 is designed for this purpose. If there is an update request on the system bus, the *bus update watcher* must interrupt the cache to update the data in the cache at the proper time. The proper time should be when either an access by the processor to the cache is finished or the cache is waiting for the system bus, in the cases of a write operation or a line miss. The finish of an access can be detected by a high $\overline{W/R}$ signal which can be obtained by NORing W and R from the processor. A request from the cache for use of the system bus can be made by $BUSREQ$ from the *bus control generator*. Initially, flip-flop A and flip-flop B are reset when $SEARCH$ is at logical 0, and $UPDATE$ from the *bus update watcher* is at logical 0. (That means the circuit does not work as long as $SEARCH$ is at logical 0.) Flip-flop A is a R-S flip-flop while flip-flop B is a falling-edge-triggered D-type flip-flop. Whenever there is an update request from other caches, the $\overline{SEARCHINT}$ signal is gated in so that $SEARCH$ becomes high when $BUSBUSY$ is low which indicates that the cache memory is not using the system bus. The output of the exclusive-or (XOR) gate has a level transition from 0 to 1 since the value of the XOR-gate input F is logical 0. The output of the XOR-gate passes through the 3-input AND-gate to both CP of flip-



(a)



(b)

Figure 45: The Circuit of the Bus Update Watcher

flop B and the 2-input AND-gate since the other inputs of the 3-input AND-gate ($SEARCH$ and $\overline{Q}$ of flip-flop B) are at logical 1. Since flip-flop B is falling-edge-triggered, the state of the flip-flop is not changed at this time so that CP from flip-flop B is high. The result of the 3-input AND-gate is gated through the 2-input AND-gate to $UPDATE$ if either a cache access is finished or the cache needs to use the system bus (that is, either $\overline{R/W}$ or $BUSREQ$ is logical 1). Then, the $UPDATE$ signal becomes valid. This mea.is the update operations can only be done when either the cache memory is not used by its associated processor or the cache is waiting for use of the system bus. The $UPDATE$ signal may be fed back to clear the R-S flip-flop in the case that the data to be updated are not found in the cache during the update operation, in which case $LINEMISS$ from the *line number generator* becomes valid. In turn, a change of the XOR-gate from 1 to 0 triggers the D flip-flop since the D of flip-flop B is always logical 1 ($\overline{Q}$ transits to logical 0); and this causes the $UPDATE$ signal to become logical 0. In this case, the cache does nothing since the data to be updated do not reside in the cache. If the data to be updated are in the cache ($LINEMISS$ is low), flip-flop A is not reset until $CK2'$ is asserted, then $UPDATE$ is pulled low. When the $\overline{SEARCHINT}$ signal changes from 0 to 1, both flip-flops are reset simultaneously, which means one update operation is finished. Fig. 45 (a) also depicts a dual-direction switch which can either gate in an update request on $\overline{SEARCHINT}$ from other caches, or gate out an update request on $UPDATEREQ$ from the *bus control generator* to other caches via the system bus. The dual-direction switch consists of two transmission gates controlled by $BUSBUSY$ and $\overline{BUSBUSY}$ from the *bus control generator.* When $BUSBUSY$ is high, $UPDATEREQ$ passes one of transmission gates onto $\overline{SEARCHINT}$ while the other transmission gate locks the path between $\overline{SEARCHINT}$ and $SEARCH$. When $BUSBUSY$ is low, the signal on $\overline{SEARCHINT}$ from the system bus passes the gate to $SEARCH$ while the path to $UPDATEREQ$ is locked by the $BUSBUSY$ lines. Fig. 45 (b) shows a timing diagram for the *bus update watcher.* From the diagram we can see the operation of this circuit for two cases: one when no updating occurs and the other when updating occurs. In the first case, the data to be updated are not in the cache. In this case, the $UPDATE$ signal has a shorter valid period and is reset to logical 0 by $LINEMISS$ as shown in the timing diagram. In the second case, the data reside in the cache memory. In this case, the period of the $UPDATE$ signal is longer and $UPDATE$ is cleared by $CK2'$. The longer $UPDATE$ signal is used to produce an $UPDATEWRT$ signal for writing the data on the system bus into the cache memory.

The circuit in Fig. 46 (a) is used to produce the $UPDATEWRT$ signal which overwrites the data to be updated if the data reside in the cache. $CK2'$ latches the $UPDATE$ signal into the rising edge-triggered flip-flop if the $UPDATE$ signal has a long period. The $UPDATEWRT$ signal is locked for the duration of $CK2'$ by an inverter clocked by $CK2'$, since the updating write has to wait until the 256-bit memory decoder finishes its operation. Meanwhile, the $DATACONTROL$ signal is sent off the chip to control the path to the system bus. $\overline{CK2''}$ is used to reset

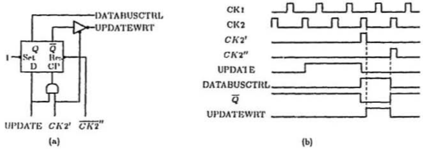


Figure 46: The Update-Write Generator

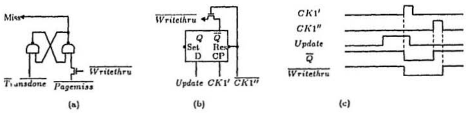


Figure 47: The Miss Circuit

the flip-flop and in turn $UPDATEWRT$. The timing diagram of this circuit in Fig. 46 (b) shows the operations described above.

4.2.3 The Miss Operations

As discussed previously, there is a circuit which create. a $MISS$ signal when there is a line miss during a read/write operation. The $MISS$ signal is produced by the *miss circuit* shown in Fig. 47. In Fig. 47 (a), initially, a high $\overline{WRITETHRU}$ signal forces the pass transistor to be closed so that the high-level $\overline{LINEMISS}$ signal from the *line number generator* (which means HIT is high) can be imposed on an input of the flip-flop while another input of the flip-flop connected to the *transfer clear circuit* remains high. At this time, the output $MISS$ of the circuit is low. If there is a HIT caused by a regular operation from the associated processor and there are no update requests from other caches (in this case, $\overline{WRITETHRU}$ is high), the state of the flip-flop remains unchanged. If there is a line miss, the $\overline{LINEMISS}$ line is low so that the flip-flop sets its output $MISS$ high. The $MISS$ signal is used to make a request that the use of the system bus be granted to transfer a missing line from the main memory, and in turn the main memory responds to this signal by sending the requested line, along with the $TRANSFER$ signal, to the cache after the system bus arbiter grants the cache the use of the system bus by setting $\overline{BUSACK}$ of that cache valid (logical 0). At this time $\overline{TRANSDONE}$ from the *tranfer clear circuit* still remains high until $\overline{BUSACK}$ makes a transition from 0 to 1 which indicates that the line transfer is finished. Then the *transfer clear circuit* produces the $\overline{TRANSDONE}$ signal, a narrow pulse of about 4 nanoseconds, to reset the flip-flop so that the $MISS$ signal changes from 1 to 0.

As we have seen, the $MISS$ signal is used to inform the main memory of a line miss. Note that not only a read/write operation can cause the *line number generator* to produce a $LINEMISS$ signal if the requested data are not found in the cache but also an update request from other caches in the multiprocessor system requires that a $LINEMISS$ signal be generated. These two kinds of line misses must be handled in different ways. The way the first situation is handled has been discussed previously. For the second situation, absence of the data to be updated in the cache should not cause any changes in the cache. Therefore, in this case, the state of the *miss flag* should not be changed as a result of searching the directory even though the search indicates the requested data do not reside in the cache. On other hand, if the data to be updated reside in the cache, the cache only updates the requested data without changing the status of the cache. Thus, th re has to be a circuit to ensure the cache contents are not changed during an update operation except for replacing the data to be updated in the cache if the requested data are found. In order to handle this case, the circuit shown in Fig. 47 (b) is employed to produce the $\overline{WRITETHRU}$ signal which prevents the *miss flag* and other components of the cache from changing during an update operation. For the *miss flag*, the $\overline{WRITETHRU}$ signal is used to lock the path between the flip-flop input and $\overline{LINEMISS}$ to prevent the miss flag from changing the $MISS$ signal by the $\overline{LINEMISS}$ signal during an update operation. As discussed previously, $CK1'$ and $CK2''$ are produced for an update operation. $CK1'$ latches the address for the update into the address register and $CK1''$ returns the address before updating into the address register. Hence, the $\overline{WRITETHRU}$ signal must be at logical 0 to guarantee that the *miss flag* is not changed during the period between the beginning of $CK1'$ and end of $CK1''$. This circuit employs a rising-edge-triggered D flip-flop. The flip-flop is triggered by the rising edge of the $CK1'$ while $UPDATE$ is high. $\overline{CK1''}$ is used to clear the flip-flop before the next update request and to lock the pass transistor to guarantee that the $\overline{WRITETHRU}$ signal remains

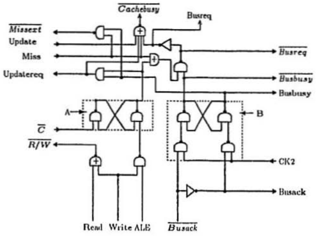
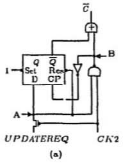


Figure 48: The Circuit for the Bus Control Signal Generator

unchanged until the end of $CK1''$ even though the flip-flop is reset by $\overline{CK1''}$. The operation of this circuit is shown in the timing diagram of Fig. 47 (c). Note that the $\overline{WRITETHRU}$ signal becomes low at the beginning of $CK1'$ and returns to the high level at the end of $CK1''$.

The *bus control generator* in Fig. 48 generates a number of signals used for communication with the processor and system bus controller. In Fig. 48, R and W come from the processor and are used to access to the cache memory. $\overline{R/W}$ is sent to the *bus update watcher* to check the update request on the system bus. Only when both W and R are low is the signal $\overline{R/W}$ set high. When ALE from the processor is asserted during a write operation, the W signal from the processor is latched into flip-flop A (called the ***write flag***). A request $\overline{BUSREQ}$ for use of the system bus can only be made by either a write operation (known by the state of the ***write flag***) or a line miss (indicated by the *miss flag* as long as flip-flop B, the ***bus busy flag***, is reset). In the case of a line miss, $\overline{BUSBUSY}$ is high, while $BUSBUSY$ is low, so that $\overline{BUSREQ}$ is low. The signal $\overline{BUSREQ}$ is sent out to request the use of the system bus. Note that at this time $\overline{BUSBUSY}$ is invalid (logical 1) since there is not a valid $\overline{BUSACK}$ signal from the *system bus controller*. After the cache sends the signal $\overline{BUSREQ}$ to the system bus controller, the bus controller responds to the cache request for the use of the system bus by generating a low $\overline{BUSACK}$ signal to inform the cache that it can use the system bus as soon as the system bus arbiter makes a grant to this cache. Once a valid $\overline{BUSACK}$ signal (low) from the ***bus controller*** is received by the cache during $CK2$, the ***bus busy flag*** latches $\overline{BUSACK}$ and in turn makes $\overline{BUSBUSY}$ low to reply to the ***bus controller*** that the cache memory is using the system bus. Meanwhile $\overline{BUSBUSY}$ also eliminates the bus request made before by pulling up $\overline{BUSREQ}$. The high $BUSBUSY$ signal can gate out the $MISS$ signal from the *miss flag* to form a new signal $\overline{MISSEXT}$ and/or the signal from the *write flag* to transmit $UPDATEREQ$ to the system bus. If only the miss flag is set, it means the cache memory is doing a miss operation caused by a read operation. If only the write flag is set, the cache is doing a write-through operation for a write operation. If both the miss flag and the write flag are set, the cache is working on a miss operation caused by a write operation. When the miss flag rises, the valid $\overline{MISSEXT}$ signal is sent



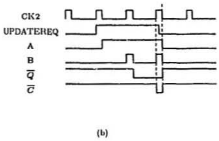


Figure 49: The Update Request Clear Circuit

valid $\overline{MISSEXT}$ signal is sent out to inform the system bus to satisfy the miss operation during $BUSBUSY$; and when the write flag is held, the $UPDATEREQ$ signal causes a $\overline{SEARCHINT}$ signal to ask the main memory and all other caches to update the data on the system data bus. Furthermore, there is another signal $\overline{CACHEBUSY}$ to be sent out to inform the processor to be idle under certain conditions. The conditions which make $\overline{CACHEBUSY}$ valid are the following:

1. to update other caches ($UPDATEREQ$),

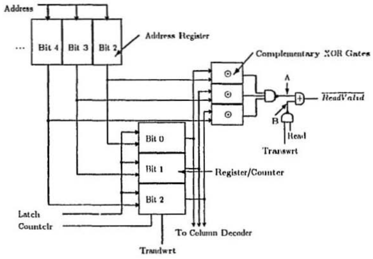
2. to be updated by other caches ($UPDATE$),

3. to have a line-miss operation ($MISS$),

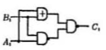
4. to request the use of the system bus ($\overline{BUSREQ}$).

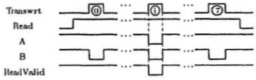
If the write flag is set by W, it has to be reset by $\overline{C}$ for next operation after a certain period, during which the cache does a write operation.

The circuit in Fig. 49 (a) is employed to produce a clear signal $\overline{C}$ to reset the write flag. Initially the output $\overline{Q}$ of the falling edge-triggered D flip-flop is at logical 1 to lock the 2-input OR-gate so that the output $\overline{C}$ is high. If there is a high $UPDATEREQ$ signal before the beginning of a pulse of $CK2$ (note that $UPDATEREQ$ can only begin to be high during $CK2$, see Fig. 48), the $UPDATEREQ$ signal remains on the D input of the flip-flop until the end of the $CK2$ pulse even though the $UPDATEREQ$ signal changes during the $CK2$ pulse. The first pulse of $CK2$ passes the 2-input NAND-gate to produce an inverted pulse on one input of the OR-gate while it is also imposed on the input CP of the flip-flop. During this pulse, $\overline{C}$ is not changed since $\overline{Q}$ locks the OR-gate. The falling edge of the first $CK2$ pulse latches the high $UPDATEREQ$ signal into the flip-flop which makes $\overline{Q}$ switch from 1 to 0. Thus, after the first $CK2$ pulse, $\overline{Q}$ is low so that the 2-input OR-gate now becomes active. During the second pulse from $CK2$, the inverted pulse from $CK2$ passes the 2-input OR-gate to make the clear signal $\overline{C}$ valid while $CK2$ is imposed on CP. The $\overline{C}$ signal resets the *write flag* to stop the $UPDATEREQ$ signal. The $UPDATEREQ$ signal is changed from 1 to 0 to end the update request on the system bus. Note that the low $UPDATEREQ$ signal is not reflected on the D input of the flip-flop since it does not get through the pass transistor until the end of the second $CK2$ pulse so that the $\overline{C}$ signal is complete. At the end of the $CK2$ pulse, the flip-flop is reset by the low $UPDATEREQ$ signal to make $\overline{Q}$ logical 1 which locks the $\overline{C}$ signal again. The operation of this circuit is shown in the timing diagram of Fig. 49 (b).



(a) The Circuit of Read Valid





(b) The Complementary XNOR Gate

(c) The Timing Diagram of The Read Valid

Figure 50: The Read Valid Circuit for Read Miss

Fig. 50 (a) depicts a special circuit called *read valid circuit* which is used to create a signal $READVALID$ to inform the processor that the data on the data bus are the data requested during transferring a missing line to the cache. The processor can receive the data from the bus without reading the data from the cache memory after transfer of the missing line. Thus the delay time for transferring a missing line for a read operation is decreased. During the transfer of a missing line, the *counter/ register* operates as a counter. For each pulse of the $TRANSFWRT$ signal from the *transfer decomposer*, the counter increases by 1. The 3 bits of the counter are compared with the corresponding bits, bit 2 to bit 4 inclusive, of the address register simultaneously. If all 3 pairs of the comparator inputs are matched, the output of the 3-input NAND-gate is logical 0 at point A. If the line miss is caused by a read operation ($READ$ is high), the inverted $TRANSFWRT$ signal arrives at point B. In this case, the $\overline{READVALID}$ signal is valid (low level). This $\overline{READVALID}$ is sent to the associated processor, and when the processor receives the $\overline{READVALID}$, it reads the data on the data bus immediately. The comparator consists of three complementary XOR-gates. Fig. 50 (b) shows the logic circuit for one complementary XOR-gate. The output becomes high only if two inputs of the circuit have the same values. The operation of the *read valid circuit* is illustrated in Fig. 50 (c).

In this chapter, the memory and control unit are designed, implemented and simulated. Eliminating both cache rewriting during a write miss and data bypass to processors during a read miss will reduce line-transfer time.

5 USE IN A MULTIPROCESSOR SYSTEM

In a large modern computer system where there are often several independent processors with a shared memory, competition between interconnected processors for access to the shared memory may become a serious problem since several of the high speed processing elements may try to reference the shared main memory at the same time. The performance of such multiprocessor systems is limited by the speed and bandwidth of the bus and the main memory. A key to efficient operation is to reduce both network traffic and direct references to the main memory. The long memory reference latency caused by the network can be greatly reduced by the local memory for each processing element since the majority of references to the main memory can be captured by a local memory such as a cache memory [9, 10]. Fig. 51 illustrates a typical cache-based multiprocessor system with a shared memory, in which each processor has an attached cache memory. Although the use of caches in a multiprocessor system can greatly reduce the bus traffic and speed up the system, such a system can cause a coherence problem because multiple copies of data in the shared main memory will likely reside in several different caches at the same time.

5.1 The Coherence Solution Strategy

Since the use of multiple private cache memories can cause a cache coherence problem, a reliable strategy must be found to keep data in the system coherent.

Figure 51: A Typical Cache-based Multiprocessor System

Many different solutions have been proposed for this problem [5, 8, 10, 12, 14]. A memory system is coherent if the value returned from a read in the system reflects exactly the last value written in the referenced address by any processing element. There are two kinds of data incoherence [7]:

1. After the data in caches are updated by the processing elements, they are not consistent with those in the main memory.

2. Multiple copies of a given line of data can exist in several caches; updating any copy of this line by a processing element will cause the values in caches associated with other processing elements to be obsolete.

To eliminate the first case, a *write-through* policy is chosen in this system to keep the information between the main memory and caches consistent. Whenever there is a write request for a given address, the copies of the requested data in both the cache and the main memory are updated simultaneously with the new value. This scheme has some advantages [5]: first, it can be implemented without complicated logic. Second, constant updating of both the cache and the main memory at every write request keeps the information in the main memory always consistent with that in the caches. Hence, if there is a write request for an address that is not in the cache, the system can simply transfer the requested line from the main memory to the cache to satisfy the request of the processing element using a replacement policy without *writing-back* the old line before it is replaced with the new one since the data in main memory are always *clean*. Therefore, it is an effective way to handle this type of coherence problem in a multicache system with a shared main memory.

In the second case, an *updating* algorithm is employed rather than *invalidation*. That is, whenever there is a write request to a cache from the processor, this request will be broadcast to all the caches in the system to cause each one to search for any copies of the requested data. If there are any, they are updated while the copy in the main memory is rewritten. Otherwise, nothing is done in the caches. The major drawback is that it does not tend to minimize communication network and main memory traffic caused by write operations and forces all the caches to do update operations, even copies of the data to be updated do not reside in most of the caches.

In this system, the *write-through* policy and *updating* are combined as an algorithm *write-through with updating* to handle both coherence situations. Whenever there is a write request from a processor, this request is broadcast to all the caches to inform the caches to update the data being written, if applicable; meanwhile, the main memory will receive the correct value for the data. The *write-through with updating* is based on the expectation that, if the data are actively shared, the caches that have copies of updated data will use the copies before they are purged.

Data can be classified as *shared* and *unshared* as well as readable and writable [10, 15]. The data are defined as *shared*, including readable or writable variables, if they currently reside in more than one cache, while the term *unshared data* means the data can only reside in one cache at any time. Therefore, there are four kinds of data:

1. Shared read/write data are the data which can be either read or written by several processors at the same time, such as shared read/write variables.

2. Shared read-only data are those which can only be read by several processors, such as shared only-read variables and instructions (assuming that programs are not self-modifying).

3. Unshared read/write data, meaning the data can only be read or written by one processor at any time.

4. Unshared only-read data are defined as those which can only reside in one cache at any time.

This policy is more appropriate for the case where much of the shared data (a number of caches share the same data) is to be processed concurrently among processors. When a processor rewrites the shared data in its cache, the copies in all other caches are updated immediately so that other associated processors do not need to transfer the updated data from the main memory when they have to use the updated copies. An example to illustrate that this policy is efficient is management of the common shared queue. It is assumed that this queue with a *semaphore* exists in each of several caches at the same time. When one of the processors intends to update the queue, it first checks the *semaphore* to see if the queue is being used by the other processor. If the queue is not used, the processor sets the *semaphore* in the corresponding cache. Meanwhile, this updated *semaphore* is broadcast to update the *semaphore* in all the caches, to prevent the queue from being used by any other caches at this time. After updating the queue, the processor resets the *semaphore*. The reset *semaphore* is also broadcast to update those copies in other caches. If the queue is used by any of other caches, the processor must wait until the *semaphore* in its cache is reset. Another example is the calculation of the sum of products of two sequences of numbers: $SUM = A_1B_1 + A_2B_2 + \cdots + A_NB_N$, and the corresponding program is as follows:

```
$SUM := 0;$
for $i =: 1$ to $N$ do
    $SUM := SUM + A(i) * B(i);$
```

Assume that there are N processors for this calculation, the variable SUM is shared and there is a *semaphore* initialized. To execute this program concurrently with N processors, first, all the processors compute products of two numbers (that is, processor i calculates $A(i) * B(i)$, $1 \leq i \leq N$), and the results are stored in their corresponding local variables. Then the intermediate results are added together by serialization. Any one of the processors intending to add its intermediate result into the global-shared variable SUM must check the *semaphore* to see if it has been set by another processor. If so, the processor must wait until SUM is released by the operating processor by reseting the *semaphore*. Otherwise, the processor sets the *semaphore* to prevent SUM from being updated by other requesting processors at this time, and then adds its local intermediate result into the SUM variable. The result of the addition is broadcast (caused by the write operation for the SUM variable) to all other processors, updating their copies of SUM. Then the processor resets the *semaphore* for the next sum operation to be done by one of the other requesting processors. Finally, in all the caches, there are consistent copies of SUM

which may be used for the next parallel calculations. Unlike this *updating* policy, the *invalidating* policy simply invalidates all the copies of SUM in other caches as the operating processor writes the partial result into the copy of SUM in its cache. Thus, when any other processors want to continue the following operation of the sum, they have to transfer the correct partial result of SUM from the main memory before doing the sum operation. Hence, in this case, the *updating* policy is more efficient than the *invalidating* policy.

For *write-through with updating*, it seems that low miss ratios will increase the probability that each of the updated data in the caches can be used before they are purged since low miss ratios indicates fewer purges of cache lines. Therefore, a larger size and a higher set associativity of the cache are preferred for this policy . On the other hand, this policy incurs the cost of updating all the caches for each write operation, and only a few updated copies may be used by respective caches before the lines containing the copies are removed for requested lines. The worst case for this policy is that no updates are useful for other caches; this happens, for example, when all the processors execute independently their own processes without use of shared data.

5.1.1 The Protocols between the Bus and the Cache

In this cache system, there is a mechanism for the cache to communicate with the system bus; an asynchronous single system bus is assumed. Generally, the cache has to communicate with the system bus in three cases; the first is a write

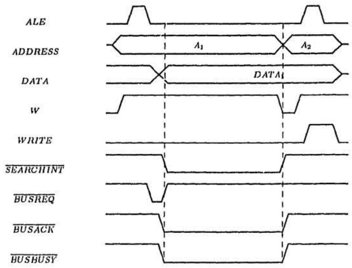


Figure 52: Communication between the Cache and Bus for a Write Operation

operation in which the cache has to send the data to be written onto the system bus to update both other caches and the main memory, the second is the transfer of a missing line in which the missing line is transferred to the cache via the system bus, and the third is an update request from another cache.

Fig. 52 shows a timing diagram for a write operation. In the diagram, all the control signals are active low except those from the processor, like W, $WRITE$, and ALE. When W is asserted, the processor is doing a write operation on the cache, along with a valid address on the address bus of the cache. ALE latchs the address into the *address register* of the cache memory. After searching its directory, the cache system makes a bus request $\overline{BUSREQ}$ for the use of the system bus to the system bus controller. Meanwhile, the cache checks $SEARCHINT$ to see if there is an update request from any of other caches during $\overline{BUSREQ}$. If the system bus grants the system bus to the cache via the bus arbiter, $\overline{BUSACK}$ is set low which removes the request $\overline{BUSREQ}$. Now the cache sends out the bus busy signal $\overline{BUSBUSY}$, along with the address and data on the system bus, to reply to the bus controller that the bus is being used. Since this cache combines the *write-through* policy and an *updating* algorithm to simplify the control, it also signals an update request $\overline{SEARCHINT}$ onto the system bus to have all other caches do an update operation. After a two-cycle period, the cache removes the request $\overline{SEARCHINT}$, which make the bus controller invalidate $\overline{BUSACK}$. Invalidation of $\overline{BUSACK}$ clears the signal $\overline{BUSBUSY}$, and the cache informs the processor that the write operation has finished, which will make the processor remove W for next operation. As soon as the bus controller receives a $\overline{BUSBUSY}$ signal, it selects one bus request from a bus request queue by sending a valid $\overline{BUSACK}$ to the selected cache.

Communication between the cache and the system bus for a line miss is more complicated than that for a write operation. Fig. 12 illustrates the communication operation between the cache and the system bus for a line miss caused by a write operation. When the cache receives a write request from the processor, it makes a bus use request $\overline{BUSREQ}$ to the bus controller since the data do not reside in the cache. After the cache detects $\overline{BUSACK}$, it removes $\overline{BUSREQ}$ and sends out $\overline{BUSBUSY}$ to the bus controller, requesting use of the system bus. Meanwhile,

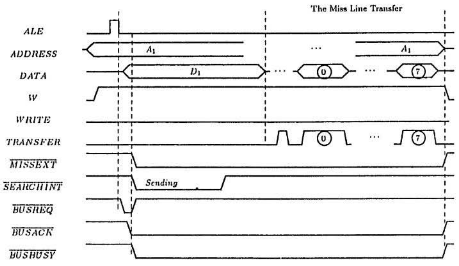


Figure 53: Communication between the Cache and Bus for a Line Miss

it also gates out the updating request $\overline{SEARCHINT}$ to all other caches and the main memory and the transfer request $\overline{MISSEXT}$ to the main memory. After the update operation for a write request, the main memory issues the requested line, accompanied by the $TRANSFER$ signal, to the cache. As soon as the transfer operation is finished, the main memory informs the bus controller so that the bus arbiter clears $\overline{BUSACK}$. The high $\overline{BUSACK}$ signal in turn removes the requests $\overline{BUSBUSY}$ and $\overline{MISSEXT}$. The cache will inform the processor to terminate the write operation by making $\overline{CACHEBUSY}$ high. Note that there is no need to update the requested data in the cache memory since the line, being transferred from the main memory, contains that data updated.

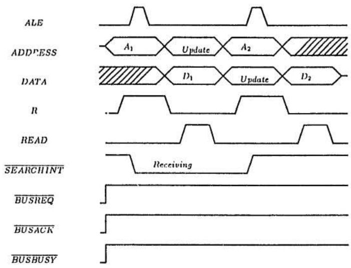


Figure 54: Communication for an Update Operation

If a line miss is caused by a read request, no $\overline{SEARCHINT}$ signal is required, since no update operation is necessary. Only the line transfer operation is done, as shown in Fig. 53. Note that the shared main memory typically consists of interleaved modules so that a requested line can be transferred in a short time.

Communication between the cache and the bus for an update operation is shown in Fig. 54. When the processor sends a read request R to the cache, along with the requested address A_1 on the address bus, the cache does the read operation. After the read operation is finished (indicated by change of R from 1 to 0), the cache allows the updating address on the system bus to reach the cache address bus and latches this address into the cache address register for the directory search.

At the same time, the cache reads out the data D_1 (Corresponding to A_1) during the first pulse of the $READ$ signal, since the cache is pipelined. During this time, the cache informs the processor to wait for one cycle. In this cycle, the cache can determine if the data to be updated are in the cache or not, and at the end of the cycle the directory has finished and is ready for the next request. Therefore, in the next cycle, the processor sends the second read request, and the directory is searched for the second request while the cache memory unit is updating the data requested for the update operation on the cache data bus if the updating data reside in the cache. In the following cycle, the cache sends the data to satisfy the second read request of the processor. Since the cache does not intend to use the system for read operations, $\overline{BUSREQ}$, $\overline{BUSACK}$, as well as $\overline{BUSBUSY}$ remain high during the update operation. Note that the address and the associated data for the update operation are placed onto the cache address bus and data bus from the system address bus and data bus, respectively, under the control of four signals which will be discussed in the following section.

5.1.2 The Protocols between the Processor and the Cache

In order to communicate with the processor in some special cases, the cache has a signal $\overline{CACHEBUSY}$ to inform the processor, which is directly connected with the cache, to be idle. The conditions for a valid signal are:

1. Occurrence of a line miss or an update request to other caches.

2. Updating of the cache.

3. Waiting for use of the system bus (the system bus is being used by another cache).

Whenever any of these three conditions are true, $\overline{CACHEBUSY}$ is valid, which makes the processor remain idle until $\overline{CACHEBUSY}$ is invalid. How long the processor is idle depends on the particular condition; for example, updating the cache needs at most two cycles.

Because of the limited number of pins on a chip, this system has only 32 pins for addresses and 32 pins for data. These pins are used by both the processor and the system bus. Therefore, the addresses both from the processor for access operations and from the bus for updating of the request data have to be latched in the address register of the cache at different times. This is realized by two bi-directional switch arrays, BSA1 and BSA2, as shown in Fig. 14. Each array has two parts, part 1 for the data bus and part 2 for the address bus. BSA1 is used to control the path from the cache bus (both the address bus and data bus) to the processor; BSA2 controls the path from the cache bus to the system bus. The control signals are from the cache, and they are based on different conditions.

Usually BSA1 is on and BSA2 is off since most of the time the cache communicates with its processor. During a write operation to the cache, the data being written are broadcast on the bus to update other caches and rewrite the main

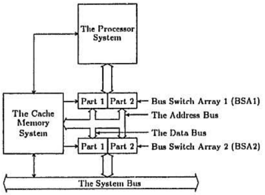


Figure 55: The Processor Subsystem

memory after the cache system receives the $BUSACK$ signal from the bus controller, while $SEARCHINT$ is sent onto the bus to inform other caches and the main memory. BSA2 switches on at this time to gate the data and address to the system bus. After the write operation, BSA2 returns to the off state. When there is an update request on the bus, the system has to do the update operation. BSA1 switches off to cut off the path to the processor while BSA2 turns on to connect the path to the bus for the update operation. After updating, the arrays return to their original states.

If there is a line miss caused by a read operation and if the data is in the main memory, the requested line is transferred from the main memory to the cache, one word at a time by interleaving. In this case, BSA2 is on while BSA1 is off. Furthermore, once the transferring word on the bus is the word needed by the processor, the signal $READVALID$ is generated by the cache to inform the processor to take the data on the bus to satisfy the processor instead of reading the requested data from the cache after the full line has been transferred. If the line miss is caused by a write operation and the cache is granted use of the bus, first BSA2 switches on while BSA1 remains on, to gate the data directly onto the bus. Both the other caches and the main memory are updated. Then BSA1 switches off to cut off the path to the processor so that the updated line is only transferred to the cache via BSA2, without updating the requested data in the cache after transfer. Thus the delay for transferring a new line during a line miss can be decreased for both read and write operations.

The four signals, $ADDBUS1$, $ADDBUS2$, $DATABUS1$, and $DATABUS2$, control operations of the BSA1 and BSA2 described above. The $ADDBUS1$ signal controls operations of the address bus of BSA1 while $ADDBUS2$ determines operations of the address bus of BSA2. Also $DATABUS1$ is used to control the data bus of BSA1, and the $DATABUS2$ signal is used to control the data bus of BSA2. The timing diagram for operation of the signals for communication between the cache and the processor are shown in Figures 57, 58, and 59 in the next section.

5.2 External Interface

To be used in a cache-based computer system, the cache needs to interface to other components in the computer system, including the associated processor, system bus, main memory, etc. This section contains a brief description of the cache I/O signals and timing.

5.2.1 The Interface Signals

The cache's external interface has 86 signals as shown in Fig. 56. A summary of the pin functions is given below:

A_0 — A_{31}, Address bus lines (input). During execution of the write/read operation, these inputs are the address from the associated processor via part two of the Bus Switch Array 1 (BSA1). During execution of an update operation, they contain the address from other caches in the multiprocessor system through part two of the Bus Switch Array 2 (BSA2).

D_0 — D_{31}, Data bus lines (3-state, bidirectional). These signals provide the data path between the cache and the processor as well as the system bus. The data bus can transmit and accept data using the dynamic bus sizing capabilities of the cache memory; the dynamic data size may be one, two, three or four bytes, depending on the data requirement. During execution of the write/read operation, these inputs/outputs are the data from/to the associated processor via part 1 of BSA1. During execution of an update operation, they contain the data to be updated by this cache from other caches through part 1 of BSA2.

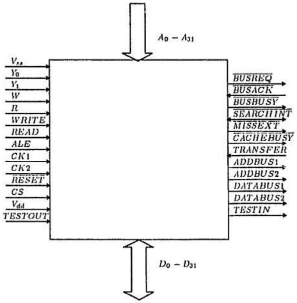


Figure 56: Pin Functions

Y_0, Y_1, Accessed data size (input). These inputs from the processor indicate number of bytes of the data being accessed in one processor access cycle (See the previous section).

W, Write operation (input, active high). This signal indicates to the cache that the operation is a write operation.

R, Read operation (input, active high). It is used to indicate a read operation.

Write, Write strobe (input, active high). This signal is used to write the data on the data bus into the cache. If the write operation causes a line miss, this signal does not appear.

Read, Read strobe (input, active high). This signal is used to read the data requested by the processor from the cache. If the read operation causes a line miss, this signal does not appear.

ALE, Address latch enable (input, active high). It indicates that the address on the address bus is valid, and is used to latch the address into the address register of the cache.

$CK1$, Clock phase 1 (input, active high). It is used to generate cache control signals and pipeline the cache.

$CK2$, Clock phase 2 (input, active high). It is used to generate cache control signals and pipeline the cache.

CS, Chip selection (input, active high). This signal is used to indicate if this cache is selected during processor operations. It is very useful for multiple cache chips used in a computer subsystem.

$\overline{RESET}$, System reset (input, active low). It clears the internal logic of the cache memory.

V_{DD}, System power (input). It is a +5 volt power supply.

V_{SS}, System ground (input).

$\overline{BUSREQ}$, Bus request (output, active low). This output is asserted to indicate that the cache requests use of the system bus.

$\overline{BUSACK}$, Bus grant acknowledge (input, active low). This signal indicates that the system bus now is granted for use by the cache.

$\overline{BUSBUSY}$, Bus busy (output, active low). This output indicates to the system bus controller that the cache is using the system bus.

$\overline{SEARCHINT}$, Search interrupt (bidirectional, active low). If there is a write operation, this signal is an output which informs the main memory and other caches to update the data on the system bus. Otherwise, it is an input which is checked by the cache to see if there is an update request from other caches in the multiprocessor system.

$\overline{MISSEXT}$, Line miss (output, active low). It indicates that the data requested by the processor are not found in the cache and asks the main memory to transfer the missing line to the cache.

$TRANSFER$, Transfer of a missing line (input, active high). This signal from the main memory responds to the request for transfer of a missing line to the cache. It is used to write the missing line into the cache.

$\overline{CACHEBUSY}$, Cache busy (output, active low). This signal is used to inform the processor to be idle when it is valid.

*ADDBUS*1, Address bus control 1 (output, active high). This signal is used to control part 2 (for the address bus) of BSA1. BSA1 is employed to control the cache bus path to the processor (See the previous section).

*ADDBUS*2, Address bus control 2 (output, active high). It is used to control part 2 (for the address bus) of BSA2. BSA2 is employed to control the cache bus path to the system bus.

*DATABUS*1, Data bus control 1 (output, active high). This signal is used to control part 1 (for the data bus) of BSA1.

*DATABUS*2, Data bus control 2 (output, active high). This signal is used to control part 1 (for the data bus) of BSA2.

TESTIN, Test data input (input, active high). It is used to shift out the cache memory addresses for testing only. 8 pulses are input for each address.

TESTOUT, Test data output (output, active high). It is used to shift out the cache memory addresses for testing only. The outputs are an 8-bit sequence of a cache line address from bit 0 to bit 7. After one testing pulse input from *TESTIN*, one bit of the cache line address can be observed on *TESTOUT*.

5.2.2 The Timing Operations

As described previously, operations of the cache can be divided into five types: normal read, normal write, read-miss, write-miss, and update operations. The operations of these types can be depicted by Figures 57, 58, and 59. Fig. 57 shows

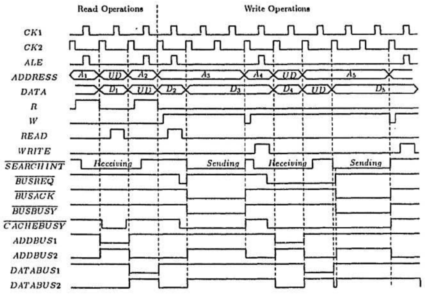


Figure 57: A Timing Diagram for Read/Write Operations

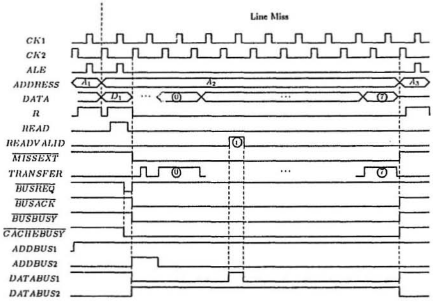


Figure 58: A Timing Diagram for Read Operations with a Line Miss

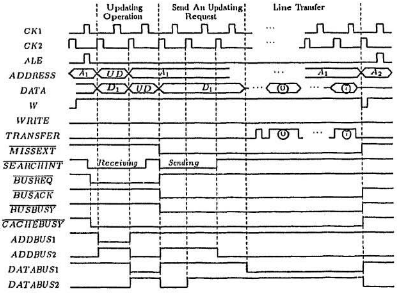


Figure 59: A Timing Diagram for Write Operations with a Line Miss

how the normal read and write operations, including update operations required by other caches, are processed. The operations on the left hand side of the figure are the read operations with an update operation, while on the right hand side are the write operations with an update operation. We can see that the operations are pipelined from the figure in which A1, A2, etc. are an address sequence on the address bus while D1, D2 ... are the corresponding data sequence on the data bus. The valid period of the $\overline{SEARCHINT}$ signal labeled *Receiving* is is for the cache to update the data after receiving an update request from another cache on $\overline{SEARCHINT}$, and the period labeled *Sending* is used to send an update request onto the $\overline{SEARCHINT}$ line. The address labeled *UD* is an address for updating and similarly the data labeled *UD* are the data being updated. Fig. 58 illustrates the signal operations of the cache during a line miss caused by a read operation. The data labeled 0 to 7 on the data bus are eight words of the missing line obtained from the main memory by interleaving. Fig. 59 shows operations during a line miss caused by a write operation. In the figure, the period labeled *Updating Operation* is updating the cache during the request for the use of the system bus caused by a line miss. The period labeled *Send An Updating Request* is used to send the data to be written and the corresponding address on to the system bus for updating other caches and rewriting the main memory. The period labeled *Line Transfer* is transferring the miss line from the main memory into the cache.

5.3 Consideration of the System Bus and Main Memory

A typical multiprocessor system usually consists of a set of processors and of a set of memory and I/O modules linked together by means of an interconnection network. Information exchange between either the processors themselves or the processors and shared main memory is accomplished by the interconnection network. Therefore, the interconnection network is a very important part of the system. No generally accepted standard for an interconnection network exists, and since the interconnection network costs are a significant part of the system cost, the interconnection network is normally designed according to the requirements of the specific application. Here the *Bus-Oriented* network (the system bus) is discussed.

There are several typical implementation policies for the single-bus arbiter. One implementation of the N-user 1-server bus arbiter is based on a first-request first-service policy. In this way, the bus arbiter always serves the request which was made the longest time ago among the bus requests. In the case that there is more than one request being made at the same time, the arbiter satisfies the one made by the processor whose logical number is smallest. Fig. 60 depicts a block diagram of the bus arbiter. It mainly consists of N circuit blocks, each of which corresponds to a cache, and a state storage block. Signals R_0—R_{N-1} are the bus requests from N different caches for use of the system bus. For any block i, there is a signal $\overline{BusAck_i}$ informing the corresponding cache that it may use the bus. Validation of $\overline{BusAck_i}$ depends on the request R_i, the request grant G_i from the storage, and C_{i-1}. All the circuit blocks are connected in a daisy chain by C_0 to C_{N-1} so that

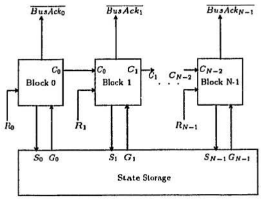


Figure 60: The N-User 1-Server Bus Arbiter

block i can be invoked if and only if block 0 to block $i-1$ are not invoked by bus requests, and the C_i will lock the following blocks (block $i+1$ to block $N-1$) not to be invoked. Therefore, none of the succeeding requests can be responded by the bus arbiter at this time. After the system bus is released by that served request, the next request will be granted use of the system bus under the same strategy. The state storage holds the information about the time the requests are made. Whenever the system bus serves a request, the storage selects the one which is made the longest time ago by asserting G_i. After a request is served, the block i will be reset and the state storage updated for next service.

In general, devices in a multiprocessor system have different priorities for use of the system bus. They can be grouped in terms of their priorities. The scheme as shown in Fig. 61 can be used for the arbitration unit in which there is a two-level

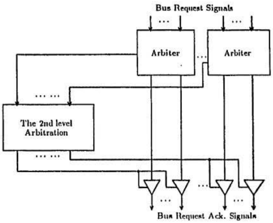


Figure 61: The Arbitration Unit

parallel bus arbitration. The first level is organized with arbiters shown in Fig. 60. For each group of requests with the same priority, an arbiter can be employed to select one request. The $CHAIN$ signal lines from all arbiters are connected and the second-level arbitration selects the highest priority arbiter using a daisy chain.

As indicated previously, the main memory can be divided into modules which are connected to the system bus. It is assumed that the shared main memory for the system under consideration is partitioned into eight modules as shown in Fig. 62. The data bandwidth of each module is one word (32 bits). The memory is organized in such a way that 8 words of a line are stored in 8 modules, respectively. That is, the first word of line i resides in the module 0 while the second word of line i resides in the module 1 and the third is in the module 2, and so on. Hence, a line can be transferred easily by interleaving. When there is a request for transfer of a missing line, each module sends a word in that line and the system bus delivers all of them by interleaving so that the delay is reduced.

Also for each module there is a buffer queue for write requests. Thus, the specified cache only needs a short time to send the write request (including the data to be written and the corresponding address) to the given module without waiting for the main memory to complete the request. Whenever there is a write request entering one module, the module controller first checks to see if there is a write request in the buffer queue which is accessing to the same location as the entering request. If so, the data of the request already waiting in the queue will be replaced by that of the entering request, and the entering request removed. Otherwise, the entering request is inserted at the end of the queue. Thus, the *write* — *write* competition is eliminated in the memory module. When there is a cache miss, a line transfer is required, and all the modules transfer the missing line immediately without inserting the request in the queues. In the case that the line miss is caused by a write operation, first the module being overwritten checks the queue to see if there is a request in the queue for the same location. If so, the request in the queue is removed. Otherwise, the queue is unchanged. Then the module serves that write request causing a line miss by updating the requested memory location with the data on the system bus; the updated word is sent to the requesting cache with the other 7 words from respective modules by interleaving. Meanwhile, the other 7 modules serve the transfer request as they do for a transfer request caused by a read operation. When the line miss is caused by

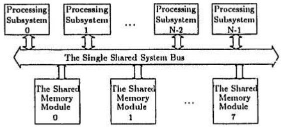

Processing Subsystem 0
Processing Subsystem 1
...
Processing Subsystem N-2
Processing Subsystem N-1
The Single Shared System Bus
The Shared Memory Module 0
The Shared Memory Module 1
...
The Shared Memory Module 7

Figure 62: The Shared Main Memory Partitioned into 8 Modules

a read operation, each module checks its buffer queue to see if there is a request in the queue, for a write into the location to which the transfer request will access. If there is such a request, it is removed from the queue; then it is served immediately. Thus, the so-called *read — write* memory competition is handled. The module then sends the requested word onto the system bus. All 8 words from different modules are sent on the bus by interleaving. This can be done by the bus controller. Note that there is no *read — read* memory competition since the main memory only serves read operations during a missing line transfer and the single system bus only serves one request at a time.

5.4 Simulations of the Cache-based Multiprocessor

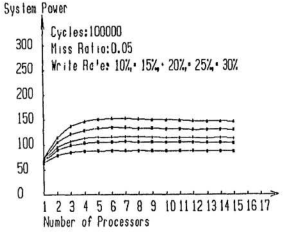
In previous sections, functions and structures of the cache memory and a multiprocessor environment were described, based on a *write-through with updating* cache coherence protocol, in which the multiple caches are used. In this section, a typical simulation model, as shown in Fig. 62, will be used to study the efficiency of such a shared-memory multiprocessor system. That is, we would like to determine how many processors can be used in the system without reaching saturation of this system. For simplification, a single bus is employed as the interconnection network between multiple caches and the shared memory, although using a more complex interconnection network such as a multi-bus system makes the system more efficient. Furthermore, in the model all the processors are identical and each processor has a private cache.

The model consists of a process for each processor, a process for each cache, and a process for the single bus. Each processor generates a memory reference sequence to the associated cache. Memory reference streams in the system are produced with a specified write operation ratio and a given cache miss ratio in the steady state. Write operations are produced at random with a given read/write ratio. Each cache is implemented as has been described. For each processing subsystem, if the reference is a read operation and the requested line is present in the cache, the cache spends one cache cycle and the processor continues. If the reference is a write operation and the requested line is found in the cache, the cache puts an update request into the service queue of the system bus, and spends two cycles to update all the caches and the main memory via the single bus as soon as the update request is acknowledged by the bus. It is assumed that there is a buffer queue in each memory module for write requests so that a write request can be sent to an appropriate memory module in two cache cycles, otherwise, a miss occurs. In this case, the cache needs to insert the transfer request into the service queue of the system bus, and the bus takes 11 cycles to transfer a requested line into the cache when the request is served by the bus. The time required to transfer a missing line is based on an assumption that a main memory cycle time is four cache cycles and the main memory transfers the missing line by interleaving, one cache cycle per word. So the transfer of a missing line requires one main memory cycle time (4 cache cycles) for the first word plus one cache cycle for each additional word (7 cache cycles). When the cache receives an update request from the bus, if the requested data are found, the cache spends two cache cycles. Otherwise, it only takes one cycle to search the directory. Note that those caches, waiting in the bus service queue for use of the system bus, must halt until they are released from the bus queue after service. The bus process receives service requests from all caches and serves them in first-in first-out order, implemented by a first-in first out service queue in the simulation model. For one request in the queue, there are four items: *Cache Number* from which the request is made, *Write/Miss*, *Address*, and *Cache Cycles* to be used by the system bus. In this model, it is assumed that there is no delay when the bus services a request.

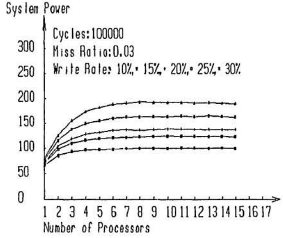
Fig. 63 summarizes the results of simulation for evaluation of the multiprocessor system with the proposed caches as private caches. Simulation outputs include bus utilization figures and other parameters under which the simulations are run. The system power is defined as total sum of the processor utilization in the multiprocessor system, multiplied by 100, and processor utilization is measured by the ratio of time spent doing useful work in a processor to the total runing time. In each figure, the simulation results obtained with the indicated parameter values are shown with from one to fifteen processors. The parameter *Cycles* gives the cache cycles executed during simulation. Fig. 63 (a) shows the simulations with an overall cache miss ratio of 0.05 while Fig. 63 (b) gives the simulation results with an overall miss ratio of 0.03. In each figure, there are five curves, each of which indicates a simulation with write operation ratios varying from 10 percent to 30 percent for memory references. The system power rises until the system bus begins to reach saturation. When the bus utilization approaches 100 percent, the system power levels out. For each figure, it is seen that the system power becomes higher and the minimum number of processors at which the single system bus reaches saturation increases as the write operation ratio decreases. Comparison of the two figures indicates that a decrease in the overall miss ratio of the caches increases system power.

Although use of the specified caches in this given multiprocessor structure can greatly reduce references to the slow main memory, the single bus system seems to be a bottleneck in the multiprocessor since write operations and transfer of miss lines under the *write through* protocol still make the bus busy. In order to further improve the system performance, an increase in system power and bus ability can be achieved as follows:

First, data can be labeled as either private or shared. If there is a write operation



(a)



(b)

Figure 63: Simulations for the Multiprocessor System

on private data, the cache does not issue an update request to all other caches while it sends an overwrite request to the main memory via the system bus. In this case, other caches can do useful work without being intercepted for an update operation. Thus, the system power would be increased although the system bus has the same traffic as that without this enforcement. Note that in this scheme, there are two signals required, instead of $\overline{SEARCHINT}$, one is used to inform all caches to update copies of the shared data and the other is used to request the main memory for a write operation.

Second, use of caches with larger sizes can increase the system performance, since a lower overall miss ratio for the caches results in both a higher system power and a higher bus utilization from the simulations. Also, the larger cache size makes the cache have a lower miss ratio. The multiprocessor system can take advantage of the efficient caches since a large size cache memory for each processor can easily be formed by using several cache chips selected by *chip selects* without decreasing the cache speed.

Third, the use of multiple busses would significantly increase system performance, because the waiting time of each cache for use of the system bus would certainly be decreased.

5.5 Testing the Cache Memory

Testing the circuit is the final step of a VLSI design, to determine if the circuit being tested has both correct logic and circuit operations. Although testing will be done only after fabrication, how to complete the testing task has to be considered during circuit and architecture design.

Testing of the cache memory can be done using a microcomputer. All the I/O lines of the cache memory chip to be tested are connected to the microcomputer via an interface. All the signals for controlling operations of the cache memory and addresses and corresponding data are sent to the tested cache and all the corresponding results are received by the microcomputer. The microcomputer will check to see if the cache operations are correct. The microcomputer, step by step, will test all the functions of the cache memory.

In order to test this cache memory chip, there is a simple additional logic block in the chip to shift out the cache memory address from the memory register. Therefore, for each main memory address referenced, the corresponding cache memory address and its content can be observed. Thus, we can create a table containing the main memory addresses for testing, the corresponding cache memory addresses, and the corresponding data during testing.

Fig. 64 illustrates the test circuit which can shift out the cache memory addresses. The 8-bit cache addresses are imposed on bits $\overline{BIT_0}$ — $\overline{BIT_7}$ and the multiplex is controled by a 3-bit counter. Whenever there is a pulse at $TEST-IN$, the counter is increased by 1. One of the 8 bits of the cache address passes the multiplex to the output $TEST-OUT$. Thus, for shift-out of a cache address, 8 pulses are imposed on the $TEST-IN$ and 8 bits of the address can be received at $TEST-OUT$ one by one.

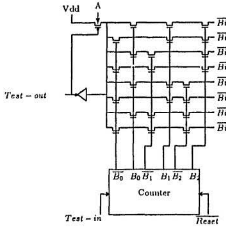


Figure 64: The Testing Circuit for Shifting-out Cache Line Addresses

In this chapter, use of the cache in a multiprocessor environment is described in which a system bus and a shared main memory are assumed. A *write-through with updating* strategy is proposed and employed to keep data in the system coherent. The system bus and shared memory structures are discussed. A queueing model is created and the system simulations have been done to evaluate the system performance.

6 CONCLUSIONS

This cache memory system has been laid out within a chip, using the 3 micron NTCMOS3 technology, and simulated. It has an 8K-byte cache memory (4 bytes for each word, 8 words for each line), and it is organized as an *8-way set-associative* cache. The cache memory is directly accessable to one, two, three, or four bytes (one word) once by the associated processor. A two-phase clock is used to synchronize and pipeline the system. The clock period is 40 nanoseconds.

In the directory, there are 32 sets, therefore 8 line slots for each set can be simultaneously compared. The address translation can be finished in 18 nanoseconds. Thus, the cache can safely trun out a result in 20 nanoseconds during read operations without line misses.

The *least recently used line replacement* strategy is employed in the replacement unit. There are 32 components, each one implemented by a bit matrix corresponding to a set.

This cache memory can be used in a multiple processor system to improve the system performance; a *write-through with updating* policy, combination of a *write-through* and a *updating* algorithm, is employed to keep the information in the main memory consistent with that of the caches and to make the multicaches in the system coherent. The hit ratios of this cache memory, in terms of the Cache Design Target Miss Ratios Table, are predicted to be over 95 percent.

Compared with on-chip cache memory, this cache memory chip has a larger cache size and a low miss ratio. Unlike a cache system consisting of a cache controller and RAM chips, it is more flexible to build a cache system which has a larger cache capacity (more than 8K bytes) for one processing element with several of the proposed cache memory chips by using the chip select signal; this does not decreases the system speed. It is also easy to implement a cache system with separate cache memories for data and instructions. This multi-chip cache system also eliminates delay time caused by wire connections between the cache controller and RAM chips (off-chip delay).

Although this cache has many advantages, there are several drawbacks, due to limitations of the VLSI technology used. It does not have a "snoop" directory which can be used to snoop the system bus for update operations, and in turn to eliminate the directory search time for updates. It does not further reduce the references to the main memory caused by write operations, which can make heavy interconnection network traffic under the *write-through* policy, especially in a single-bus shared-memory multiprocessor system.

An implementation using a more modern process technology, say a 1.5 micron technology, would permit a larger cache memory chip, or rather a large on-chip cache memory, with dual-directories and faster address translation. Also it could allow the implementation of both the *write-through* and *write-back* policies in a cache memory, which could make the cache have a great performance improvement.

For higher performance, assuming the data are classified into *shared* and *unshared* as mentioned before, if there is a request for writing a shared read/write variable, the *write-through* policy is used to keep the multicaches and main memory consistent since the shared read/write variable may be modified by several processors. If the accessed data are an unshared read/write variable, the *write-back* policy is employed to decrease the network traffic since only one *valid* copy of this variable can exist in one cache. In the case that the line containing the unshared read/write variables is to be replaced by a new requested one at a cache miss, if this line has been updated since it existed in the cache, it is written-back to the main memory before transfer of the new line to make information in the main memory correct. If it has been unchanged since it resided in the cache, it, like the line which contains the read-only data (including the shared and unshared read-only variables as well as instructions), is simply overwritten by the new requested line, since it is consistent with that in the main memory.

References

[1] Derek De Solla Price, "A History of Calculating Machines", *IEEE Micro*, Vol. 4, No. 1, 1984 pp. 24-36

[2] Harold S. Stone, *High-performance Computer Architecture*, Addison-Wesley, Reading, Mass., 1987

[3] R. E. Matick, *Computer Storage Systems and Technology*, John Wiley & Sons, New York, 1977

[4] T. Kohonen, *Content-Addressable Memories*, Springer-Verlag, Berlin, 1980

[5] A. V. Pohm and O. P. Agrawal, *High-speed Memory Systems*, Prentice-Hall, Reston, Virginia, 1983

[6] J. Edler, et al., "Issues Related to MIMD Shared-memory Computer: the NYU Ultracomputer Approach", *The 13th Ann. Int'l Symp. on Computer Architecture*, 1986, pp. 127-135

[7] W. C. Yen, D. W. L. Yen, and K. S. Fu, "Data Coherence Problem in a Multicache System", *IEEE Trans. on Computers*, Vol. C-34, 1985, pp. 56-65

[8] S. Frank and A. Inselberg, "Synapse Tightly Coupled Multiprocessors: a New Approach to Solve Old Problems", *AFIPS Conf. Proc.*, Vol. 53, 1984, pp. 41-50

[9] A. J. Smith, "Cache Memories", *ACM Computing Surveys*, Vol. 14, No. 3, 1982, pp. 473-528

[10] A. Gottlieb, et al., "The Ultracomputer — Designing a MIMD, Shared Memory Parallel Machine", *The 9th Ann. Int'l Conf. on Computer Architecture*, 1982, pp. 27-42

[11] A. J. Smith, "Line (Block) Size Choice for CPU Cache Memory", *IEEE Trans. on Computers*, Vol. C-36, No. 9, 1987, pp. 1063-1074

[12] R. H. Katz, et al., "Implementing a Cache Consistency Protocol", *The 12th Ann. Int'l Symp. on Computer Architecture*, 1985, pp. 276-283

[13] P. Sweazey and A. J. Smith, "A Class Of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus", *The 12th Ann. Int'l Symp. on Computer Architecture*, 1985, pp. 415-423

[14] J. R. Goodman, "Using Cache Memory to Reduce Processor-memory Traffic", *10th Ann. Int'l Symp. on Computer Architecture*, 1983, pp. 124-131

[15] P. Bitar and A. M. Despain, "Multiprocessor Cache Synchronization — Issues, Innovations, Evolution", *The 12th Ann. Int'l Symp. on Computer Architecture*, 1985, pp. 425-433

[16] D. R. Cheriton, G. A. Slavenburg, and P. D. Boyle, "Soft-controlled Caches in the VMP Multiprocessor", *The 12th Ann. Int'l Symp. on Computer Architecture*, 1985, pp. 366-374

[17] M. Dubois, C. Scheurich, and F. Briggs, "Memory Access Buffering in Multiprocessors", *The 12th Ann. Int'l Symp. on Computer Architecture*, 1985, pp. 434-443

[18] M. D. Hill, "A Case for Direct-Mapped Caches", *Computer*, Dec., 1988, pp. 25-40

[19] M. Dubois and F. A. Briggs, "Effects of Cache Coherency in Multiporocessors", *IEEE Trans. on Computers*, Vol. C-31, No. 11, 1982, pp. 1083-1099

[20] N. H. E. Weste, K. *Principles of CMOS VLSI Design*, Addison-wesley, Reading, Mass., 1985

[21] B. Randell and P. C. Treleaven, *VLSI Architecture*, Prentice-Hall International, London, 1983

[22] F. J. Hill and G. R. Peterson, *Digital Systems*, John Wiley & Son, Toronto, 1987

[23] M. A. Marsan, G. Balbo, and G. Conte, *Performance Models of Multiprocessor Systems*, The MIT Press, Cambridge, Mass., 1986

[24] X. Luo, and P. Gillard, "An Efficient Cache Memory Management Unit", *Proc. of APICS Computer Science Conference*, 1988, pp. 1-12

[25] H. J. Mitchell, *32-Bit Microprocessors*, William Collins Sons & Co. Ltd, London, UK, 1986

[26] M. Stansberry, "Cache Memory Design in 32-bit Microprocessor Systems", *VLSI Systems Design*, 1988, pp. 32-42

[27] B. Merril, "370/168 Cache Memory Performance", *Share Computer Measurement and Evaluation Newsletter*, No. 26, 1974, pp. 98-101

[28] D. Phillips, "The Z80000 Microprocessor", *IEEE Micro*, Vol. 5, No. 6, 1985, pp. 23-36

[29] D. Alpert, et al., "32-bit Processor Chip Integrates Major System Functions", *Electronics*, July, 1983, pp. 113-119

[30] R. Gregory, "Caching Designs Eliminate Wait States to Relieve Bottlenecks", *Computer Design*, Oct., 1988, pp. 65-73

[31] D. MacGregor, D. Mothersole, and B. Moyer, "The Motorola MC68020", *IEEE Micro*, Vol. 4, No. 4, 1984, pp. 101-118

[32] H. Scales, P. Harrod, "The Design and Implementation of the MC68030 Cache Memories", *IEEE Conf. On Computer Design*, 1987, pp. 578-581

[33] Goldman, "First Look at Motorola's Latest 32-bit Processor", *Electronics*, Vol. 59, No. 31, 1986, pp. 71-75

[34] D. W. Clark, "Cache Performance in VAX-11/780", *ACM Trans. on Computer Systems*, Vol. 1, No. 1, 1983, pp. 24-37

[35] R. E. Matick, "Functional Cache Chip for Improved System Performance", *IBM J. Res. Develop.*, Vol. 33, No. 1, 1989, pp 15-32

[36] P. Stenstrom, "Reducing Contention in Shared-Memory Multiprocessors", *Computer*, Vol. 21, No. 11, 1988, pp. 26-37

[37] S. Thakkar, P. Gifford, and G. Fielland, "The Balance Multiprocessor System", *IEEE Micro*, Vol. 8, No. 1, 1988, pp. 57-69

[38] J. Archibald and J. L. Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model", *ACM Transactions on Computer Systems*, Vol. 4, No. 4, 1986, pp. 273-298

[39] A. Ayyad and B. Wilkinson, "Multiprocessor Scheme with Application to Macro-dataflow", *Microprocessors and Microsystems*, Vol. 11, No. 5, 1987, pp. 255-263

[40] J. K. Muppala and L. N. Bhuyan, "Arbiter Designs for Multiprocessor Interconnection Network", *Microprocessing and Microprogramming*, No. 26, 1989, pp. 31-43