A High Performance Fault Tolerant Cache Memory for Multiprocessors

by

Xiao Luo

B.Sc., Huazhong University of Science and Technology, 1982
M.Sc., Memorial University of Newfoundland, 1990

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

We accept this dissertation as conforming
to the required standard

Dr. J. C. Muzio, Supervisor (Department of Computer Science)

Dr. G. C. Shoja, Departmental Member (Department of Computer Science)

Dr. D. M. Miller, Departmental Member (Department of Computer Science)

Dr. K. F. Li, Outside Member (Department of Elec. & Comp. Eng.)

Dr. P. Gillard, External Examiner (Memorial University of Newfoundland, Canada)

Supervisor: Dr. Jon C. Muzio

Abstract

In multiprocessor systems, cache memories serve two purposes, namely the reduction of the average access time to the shared memory and the minimization of interconnection network requirements for each processor. However, in a cache, interference between operations from the processor and operations for data coherence from other caches degrades the cache performance. We propose a cache with only one single dual-port directory which can be operated for both processor accesses and coherence operations simultaneously. The cache can reach high performance at low cost. This cache also has a data-coherence-protocol-independent structure.

To evaluate the cache performance in a multiprocessor environment, two simulation models are created. The system performance is extensively simulated. The results show that the single dual-port directory cache system has higher performance than that obtained by a system with single one-port directory caches. Other design parameters such as cache size, line size, and associativity on system performance are also discussed. Furthermore, simulations indicate that use of multiple buses significantly increases system performance.

In order to improve the reliability of the proposed cache, we design a tag self-purge mechanism and a comparator checker at low cost in the cache management unit. We also propose a new design that provides combinational totally self-checking checkers for 1/n codes in CMOS technology, which can be used to build such a checker for the 1/3 code. Moreover, the total hardware overhead is less than 42%, as compared to the traditional single directory cache management unit.

The dissertation includes a new optimal test algorithm with a linear test time complexity, which can be used to test the cache management unit by either the associated processor or external test equipment. An efficient built-in self-testing variant of the proposed algorithm is also discussed. The hardware overhead of such a scheme is much less than the traditional approach.

Examiners:

Dr. J. C. Muzio, Supervisor (Department of Computer Science)

Dr. G. C. Shoja, Departmental Member (Department of Computer Science)

Dr. D. M. Miller, Departmental Member (Department of Computer Science)

Dr. K. F. Li, Outside Member (Department of Elec. & Comp. Eng.)

Dr. P. Gillard, External Examiner (Memorial University of Newfoundland, Canada)

Contents

List of Figures

List of Tables

Acknowledgements

First of all, I would like to express my sincere gratitude to my Ph.D. supervisor Dr. Jon C. Muzio for his supervision, guidance, suggestions, encouragement, and patience, which have greatly helped me to complete this dissertation.

I would like to appreciate Dr. Kin Li, who has contributed his support and advice toward the development of this work. I would like to thank all members in the VLSI design and test group for their assistance and friendship and the staff members in Department of Computer Science for their technical support during my 3-year Ph.D. study.

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

I am greatly indebted to my mother and my wife for their support, understanding, and patience. Finally, I would like to thank my brothers and sister for their understanding, support, and sacrifices in all these years.

Dedication

To the Memory of My Father.

Chapter 1

Introduction

Until the last two decades, almost all electronic digital computer systems were strictly based on the so-called *Von Neumann architecture*. Although both processors and main memory systems were steadily improved by the development of advanced technologies and novel architectures, there was a persistent mismatch between the speed of processors and that of main memory. That is, the main memory was slow relative to the speed of the processors. The memory system limits how quickly input data can be delivered to a processor and the corresponding results received from the processor. This has come to be called the von Neumann bottleneck of computers. In an attempt to alleviate this problem, many computers have added cache memories between their processors and main memories. This cache memory is a small, comparatively fast memory introduced in the hope that almost all the required instructions and data are in the cache, with the consequent reduction in the number of accesses to main memory by the processor.

In spite of many technological advances in electronics, uniprocessor systems are still inadequate for the most highly computationally intensive problems. Further, we have now reached the point where communication delays between switching elements or integrated circuits play a dominant role in the overall speed of the computation.

New ways have to be found to meet these requirements. An obvious general approach is based on parallelism, implying that computer architectures should depart from the strict Von Neumann concept.

Parallelism in various forms has appeared in computers ever since the early days of their design, and has proved to be an effective approach. 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. Resource replication is the replication or addition of hardware units which can operate simultaneously on a problem. Resource sharing, for example, can be multiple processes using the same hardware in some time-slice order. The specific interest of the work reported in this dissertation is in multiprocessor systems, consisting of a number of processors, I/O devices and main memory connected by an interconnection network.

System reliability has been a major concern since the beginning of electronic computers. The earliest computers used discrete components, such as relays, vacuum tubes, etc., that would fail to operate correctly as often as once every hundred, thousand, or million cycles. This error rate was far too high to guarantee correct completion of even modest calculations. Computer designers tried to use fault tolerant techniques such as error detecting/correcting codes (EDC), match-and-compare methods, and parity checking to improve system reliability. With the evolution of technology, components can be integrated onto single chips so that their reliability increases considerably. However, as the reliability of the components of the system has increased, the complexity of the systems has also increased by several orders of magnitude. Consequently, faults still occur in systems, especially in large and complex systems such as multiprocessors. Since computer systems have been playing a larger role in everyday life, our dependence on such systems has also increased. Furthermore, computer systems are now being used in many more safety-control ap-

plications where system failure could lead to catastrophic results. Automated flight control systems, control systems for nuclear reactors, and the space shuttle are examples of these applications. These applications require computer systems that have both high performance and high reliability. To meet these requirements, one of the approaches is to develop multiprocessor systems with fault-tolerant abilities.

A multiprocessor system increases the computational ability of the system; however, when the number of processors increases, interconnection network traffic may become a serious bottleneck. In order to reduce the traffic, one of the approaches is the incorporation of private cache memories, each of which is associated with a single processor to reduce the direct references to main memory through the interconnection network. However, use of private caches may cause a data coherence problem; multiple copies of data in the shared main memory may reside in several different caches at the same time. Many solutions to keep data in multiple caches consistent have been proposed by implementing data coherence protocols in caches. There exist two kinds of operations in a cache regardless of the coherence protocols that are implemented in the cache: read/write operations from the associated processors (processor operations), and operations to maintain data coherence with other caches or memory in the system (coherence operations). Interference between these two operations is unavoidable in a multiprocessor cache because both operations may be required in the cache at the same time. As a result, cache performance is affected, and in turn performance of the cache-based multiprocessor system is degraded. It is desirable to design an efficient multiprocessor cache which allows the two operations to be carried out simultaneously, but with reduced hardware overhead.

Since cache memory is being increasingly used in modern systems, the reliability of cache is of increasing importance. With rapid developments in technologies, the capacity of cache memory has increased dramatically allowing a significantly enhanced performance. The cache memory management unit has become more

complicated. The reliability of cache memories with large capacity cannot be ignored. Our goal is to find a design for a reliable high-performance multiprocessor cache at a reasonable hardware cost. It is our strategy to make use of some hardware in the cache memory management unit for high cache performance as well as for both off-line testing and on-line concurrent checking. It is also expected that the use of such caches in a multiprocessor system enables the system to detect faults in the caches, including faults in both the cache management units and data memories, as soon as possible. Further, undetected faults in the caches are confined within these caches, protecting information in main memory from pollution.

This dissertation is divided into three main parts. The first part, Chapters 1 and 2, introduces the problems for multiprocessor cache performance and cache reliability as well as giving the foundation for the rest of the dissertation. The second part, Chapters 3 and 4, describes the VLSI design for the multiprocessor cache and evaluates the proposed cache performance and hardware costs. The final part, Chapters 5, 6 and 7, discusses the design for fault tolerance and the design for testability of the cache.

Chapter 3 gives the CMOS design for the proposed cache which can carry out both processor accesses and coherence operations simultaneously. This cache is protocol-independent so that any of the standard data coherence protocols can fit in. We also present an analysis of the hardware overhead for performance enhancement.

Chapter 4 discusses cache-based multiprocessor simulation models with a shared memory and multi-bus. The structure of the simulator and simulation workload are described. The system performance is simulated. Based on extensive simulation results, we show the performance improvements made by the use of our dual-port directory caches. We also investigate the effects of cache parameters such as cache size, line size, and way size, the effects of write reference rates, the effects of data sharing, and the effects of multiple buses on the multiprocessor system performance.

Chapters 5 and 6 describe the design for fault-tolerance in the cache management unit. The design consists of a tag self-purge mechanism, a comparator checker, an error flag, and a totally self-checking checker. In Chapter 5, a comprehensive fault model at the functional level is created for both fault tolerance and off-line testing. The tag self-purge mechanism, the comparator checker, and the error flag are described. The hardware overhead for fault-tolerance and on-line concurrent checking in the cache management unit is discussed. Since the design for fault tolerance is too long to be included in one chapter, we give the detailed design of the totally self-checking (TSC) checkers in the following chapter. This chapter includes a fault model designed to include most physical defects which are likely to occur in MOS implementations. A new design is presented which provides combinational TSC checkers for 1 out-of-n codes in CMOS technology. The checkers retain the TSC properties for any the faults or fault sequences.

Chapter 7 shows a new optimal off-line test algorithm with a linear test time complexity, which can be used to test the cache management unit by either the associated processor in a multiprocessor system or external test equipment. An efficient variant of the proposed algorithm which is suitable for the built-in self testing (BIST) cache management unit is also discussed.

Finally, Chapter 8 gives conclusions. In this chapter, the cache designs and performance evaluation are concluded. The total hardware overhead for both performance enhancement and fault-tolerance/concurrent-checking is discussed. The cache applications and the reliability improvements of a multiprocessor system with the proposed caches are also discussed, and some topics for future work are considered.

Chapter 2

Background

This chapter provides the fundamental foundation on which the following chapters are based. The first section gives a glossary of the relevant terminology required for the dissertation. Sections 2—4 give a brief discussion of multiprocessor systems. This is followed by a review of protocols for bus-based multiprocessor systems. Third, the general cache memory architecture is introduced and the most important cache design considerations for high performance are briefly discussed. In the final sections, we give some general concepts of fault tolerance and testing, and, in particular, review the fault tolerance in multiprocessor systems.

2.1 Glossary of Terms

2.1.1 Terminology for Cache and Multiprocessors

Associativity: Associativity is related to mapping policies which are used to translate the main memory address space to the cache address space. There are three mapping policies according to degree of associativity: full-associative, direct-mapped, and set-associative. The full associative mapping is that any of the lines in main memory can be mapped into any line in cache memory. The

direct-mapped method is that any given line in main memory can reside logically only in a specified line in cache memory. This is a many-to-one mapping. The third mapping method is n-way set-associative mapping which is a hybrid of the direct-mapped and full-associative methods. An n-way set-associative cache has multiple sets which can be selected by the direct-mapping, and n lines in each set which can be simultaneously searched by the full-associative mapping.

Bus Width: Bus data width is the number of information bits a bus can transfer in parallel in one time unit. Usually, bus width is equal to cache data width. Bus data path width must be considered during the design process since it directly determines the time taken when a line is transferred from main memory to cache memory. From the performance point of view, a bus is constructed as wide as possible. However, the wider a bus is, the more expensive. Hence, a trade-off of the path width has to be made during design to achieve a reasonable cost/performance ratio.

Cache Memory: A cache is a small, fast, memory that at any time can hold the most active portions of contents in the overall memory of the machine. Its organization is specified by its size (cache size), line size, way size, set size, fetch strategy, write strategy, and replacement strategy. The cache size is given as the product of the three primary parameters: way size, set size, and line size. Any cache can be a unified cache for both instructions and data or two separate caches for instructions and data respectively. Usually the cache speed is compatible with that of the associated processor.

Cache Size: The cache capacity is usually dictated by many factors connected with the system cost and performance. In general, a large cache capacity can introduce a higher hit ratio, and in turn a better performance. However,

there are limitations on cache size beyond which cache memory has either a high cost, or performance decreases. Therefore, during cache design, a cache should not be made so large that the cache access time is increased beyond the specified limits. A cache also should not be so large that its costs are out of proportion to the added performance, nor should it occupy an unreasonable space.

Coherence Protocols: Coherence protocols are designed for keeping information among main memory and caches in multi-processors consistent. Typically, there are two basic kinds of protocols: write-through and copy-back (or non-write-through). The write-through policy is that whenever there is a write request to a cache memory, the request is also immediately broadcast to the main memory and all other caches to either update or invalidate copies of the requested data, if any. Thus, information in the system is always consistent. The copy-back scheme is that whenever there is a write request to a cache, only the copy of the requested data in that cache is updated with an invalidation signal sent to the main memory and other caches. When there is a line miss in the cache and the cache is full, if the line containing the latest updated data is selected to be purged from the cache for making room for the requested line, the line to be purged is flushed (written-back) to the main memory. Or when there is a demand of a line from other caches, the cache which holds the latest updated copy of the line sends the line to main memory and/or the requested caches. This can reduce interconnection network traffic. However, it is more complicated in logic; and there is a temporary data inconsistency among caches and the main memory.

Data Coherence: 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 processor.

Data Shared Rate: Data shared rate is the fraction of the amount of information shared by all processors to the total amount of information in main memory.

Directory: The directory is a mechanism used in cache memory to translate main memory addresses into corresponding cache memory addresses. It is also called the tag file because it consists of tags each of which is used to record an address of a main memory line that is currently residing in a corresponding cache line.

Fetch Strategy: Fetch algorithms are used to determine when the system fetchs information into a level of memory from the next level in memory hierarchy. In general, types of major fetch algorithms include demand-fetch and prefetch. The demand fetch algorithm is that the requested information is fetched only if needed. The prefetch algorithm, on other hand, gets information before it is needed. Therefore, the prefetch algorithm is based on some kind of prediction as to which line will be used next and obtains it in advance. It must be designed carefully if the machine performance is to be improved rather than degraded, [1]. In addition, implementation of the prefetch is usually more complicated. The fetch size of cache memory is the amount of information that is fetched from main memory as a transfer unit. It can be larger or smaller than the line size, but is frequently equal to it.

Line (Block): A cache line is the unit of data for which there is an address tag. The tag indicates which portion of main memory, called a main memory block, is currently occupying this line in the cache. Usually, it is also the data unit that a system transfers from its main memory into a cache during a line miss. Line size of cache memory is one of the most important parameters affecting cache performance. There are a number of trade-offs for a reasonable line size in terms of architecture and technology.

Line Miss: A line miss occurs when the line containing the data requested by a processor does not reside in the associated cache. Whenever there is a line miss, the cache asks main memory for the transfer of the requested line. There are two types of line misses: read misses and write misses. a read miss is caused by a read request from the processor while a write miss is by a write reference.

Main Memory: The main memory or primary memory is the memory which is directly addressable by the processors. Usually in a multiprocessor system, it is divided into several separate memory modules shared by processors (shared memory module, or SMM).

Memory Hierarchy: Usually a memory hierarchy of a computer system consists of cache memory, main memory, and back-up memory such as disks and tapes. In the memory hierarchy, the top level of memory, e.g. cache memory, has the smallest capacity with highest speed, and the bottom level has the largest capacity with the slowest speed. In this way, the memory hierarchy seems to have nearly the speed of the top-level memory and the capacity of the bottom-level memory. Moreover, the level of cache memory can further be divided into sub-levels.

MIMD: Multiple Instruction Multiple Data. In MIMD architectures, a system has several processing elements which operate in parallel in an asynchronous manner either individually or cooperatively. There are two typical types of MIMD: tightly-coupled multiprocessors and loosely-coupled multiprocessors. However, a MIMD architecture may lie between these two types.

Miss Ratio: The miss ratio is the number of misses, including both read misses and write misses, in a cache divided by total number of references to the cache. If we define the probability of all the references to memory as 1, the hit ratio of a cache memory is (1 - miss ratio) and is the probability that requested

data is found in cache memory. The miss ratio is one of the most important factors for cache performance evaluation.

Reference: A reference is a memory request from a processor which is presented to its cache memory. There are two kinds of requests: read and write.

Reference Locality: The locality of memory references has two aspects: *spatial locality* and *temporal locality* [1, 2]. Spatial locality refers to the property that memory accesses over a short period of time tend to be clustered in space. Both types of behavior can be expected based on the common knowledge of typical program behavior. 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.

Replacement Strategy: The replacement strategy is employed for prediction of a line which probably is least likely to be used in cache memory (or a given set) in the future and can be discarded from cache memory when it is full and a cache miss occurs. The aim is to keep data in the cache optimized for the highest hit ratio or the maximum system throughput.

Set: A set is the collection of lines, the tags for which are checked in parallel. It is also the collection of lines any of which can hold a particular line of main memory. If the number of sets is one, the cache is called full associative, because all the tags must be checked to determine whether a reference causes a line miss.

SIMD: Single Instruction Multiple Data. This system has a single control unit fetches and decodes instructions. The instruction is executed either in the control unit or it is broadcast to some processing elements. These processing elements operate synchronously but their local memories have different

contents.

Snoopy Cache: Caches are used in a bus-based multiprocessor system in which all the caches are watching the bus constantly.

Write Rate: Write rate is the ratio of the number of write references to the total number of references. Since the data coherence problem in multiprocessors is caused by write operations on different copies of data in the system, the write rate is one of many factors to affect system performance.

2.1.2 Terminology for Testing and Fault-Tolerance

Availability: Availability is the probability that a system is operating correctly and is available to perform its functions at a given instant of time.

Dependability: Dependability is the quality of service that a system provides. It encompasses the concepts of reliability, availability, safety, maintainability, and testability.

Error: An error is the manifestation of a fault. Specifically, an error is a deviation from accuracy and correctness. A fault may cause errors in a system, but not necessarily. An error is the result of a fault.

Error Detecting Code: An error detecting code is a code by which errors in codewords are easily detected during normal operations.

Failure: A failure is the nonperformance of some expected actions. A failure is also the performance of some function in a subnormal quantity or quality. In other words, a failure occurs when the behavior of a system first deviates from that specified. It is often used interchangeably with the term malfunction.

Fault: A fault is a physical defect, imperfection, or flaw that occurs within some hardware and software component. There are four possible fault causes. The first cause is specification mistakes, including incorrect algorithms, architectures, or design specifications. The second one is implementation mistakes. Implementation is the process of transforming hardware or software specification into the physical hardware or actual software. The third one is component defects, including manufacturing problems, random device defects, and wear-out. The final one is external disturbance, caused by radiation, electromagnetic interference, environment extremes, and similar phenomena.

Fault Confinement: Fault confinement is the process to isolate faults and to prevent their effects from propagating throughout a system. In other words, it tries to limit faults to one area so that they cannot pollute information in other areas. It is also called fault containment.

Fault Tolerance: Fault tolerance is the ability of a system to continue to operate correctly after the occurrence of faults. The ultimate goal of fault tolerance is to prevent system failures from ever occurring.

Reliability: Reliability is the probability that the system will operate correctly throughout a complete time interval. The reliability is a conditional probability in that it depends on the system being operational at the beginning of the chose time interval. Fault tolerance can improve a system's reliability by keeping the system operational when hardware failures and software errors occur.

Testability: Testability is the ability to test for certain attributes within a system. Testability contains two concepts: observability and controllability. Observability is the ability to observe either directly or indirectly the state of any

node in the system whereas controllability is the ability to set and reset every node internal to the system.

Testing: Testing is the process of exposing defects in the system. In general, there are two types of testing: on-line testing and off-line testing. On-line testing is the process of detecting faults when the system is carrying out normal operations. It is also referred to as the concurrent testing (or checking). Off-line testing is conducted by putting the system into a specific test mode, normal operations being suspended during off-line testing.

2.2 Multiprocessor Systems

It is becoming more attractive to use multiprocessors to increase computational power. In general, a multiprocessor system is defined as a computer system composed of N processors each of which can operate independently [3, 4]. These processors are connected together through an interconnection network to provide a means of cooperating during computation. Therefore, a multiprocessor system has a MIMD architecture (multiple instruction streams and multiple data streams). Multiprocessor systems are suitable for much larger and more varied computation than the SIMD systems (single instruction stream and multiple data streams), because multiprocessors are inherently more flexible. There are two typical kinds of the multiprocessor systems which have become popular: *tightly-coupled* and *loosely-coupled*. However, a multiprocessor system may lie anywhere in between these two extreme cases.

In the tightly-coupled multiprocessors, data can be communicated from one processor to any other processors at rates on the order of the bandwidth of memory. In other words, the tightly-coupled multiprocessors provide a convenient means for information interchange and synchronization through the shared memory since

any pair of processors can communicate with each other directly through a shared location in main memory. Therefore, a tightly-coupled multiprocessor system is generally characterized by the following: (1) multiple processors are used; (2) all the processors share the main memory equally; and (3) each of processors can carry out computation either individually or cooperatively with others via shared main memory, usually partitioned into modules. Therefore, such a multiprocessor system can execute simultaneously a number of tasks required for large computation (or processing) on different processors. Since the regularity of such computer systems, in general, allows duplication of modules of the same type, both time and cost of design are reduced significantly.

In the loosely-coupled systems, communication delays between two processors depend on whether the processors are locally connected to each other or are connected through one or more layers of a routing network. In the loosely coupled systems, each processor has its local private memory and local I/O devices. It supports communication through point-to-point exchange of information. Thus, a loosely-coupled system has the following properties: (1) multiple processors are used; (2) all the processors have their own local main memory and I/O devices; (3) each of processors can do computation (or processing) either individually or cooperatively with others through an interconnection network in a point-to-point style.

There are many tightly-coupled multiprocessors proposed, [5, 6, 7, 8, 9, 10, 11]. However, the bus-based systems are more popular and commercially available because of their effectiveness, simplicity, and relatively low cost. Fig. 2.1 shows a typical architecture of the bus-based shared-memory multiprocessor system where there are four basic components: the processing element (general), main memory, I/O device, and system bus as the interconnection network. In this structure, the processors are replicated, and main memory and I/O devices are equally shared by all the processors. In this way, programs can cooperate using minimum overhead.

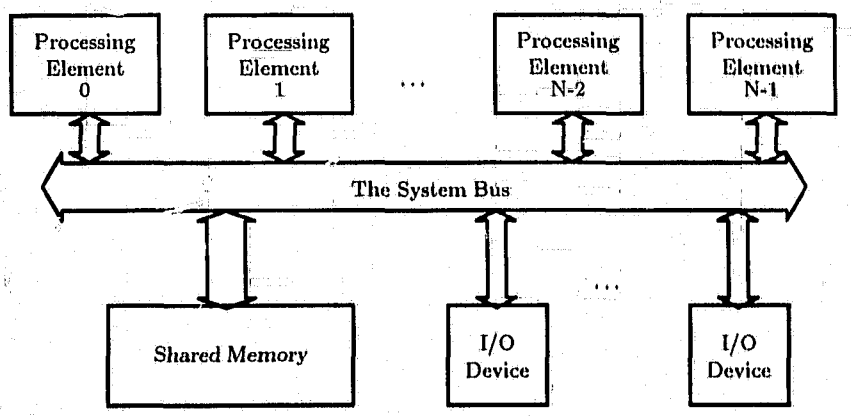


Figure 2.1: A Bus-Based Multiprocessor System

That is, processors can communicate with each other through shared memory without involving the operating system. However, 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 shared main memory at the same time through the system bus. 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 main memory. That is, in order to maximize overall system performance, the bus requirements of each individual processor must be minimized. The long memory reference latency caused by the system bus can greatly be reduced by associating a cache memory with each processing element, since the majority of references from a processing element to main memory can be captured by a cache memory associated with the processing element [2, 12]. Although use of multiple private caches in a multiprocessor system can greatly reduce bus traffic and speed up the system, such a system may have a coherence problem because multiple copies of data in the shared

main memory reside in several different caches at the same time. There have been many data coherence protocols to solve this problem.

2.3 Coherence Protocols

Basically, there are two strategies to solve the problem caused by multiple copies of data in different caches: *write-through* and *non-write-through* (write-back) though there are many variants for high performance. These coherence protocols are called *snoopy* protocols because all the caches in the system always watch the transitions on the system bus.

The *write-through* policy is that whenever there is a write request to a cache memory, the request is also immediately sent to either update or invalidate copies of the requested data in other caches and to update the copy in the main memory. Thus, information in the caches and main memory is always consistent.

On the other hand, the basic idea of the *non-write-through* scheme is that whenever there is a write request from a processor, the copy of the requested data in the associated cache is updated. However, the copy in the main memory is not immediately updated to reduce bus traffic. Instead, the updated data in the cache are either flushed (written-back) to the memory when the cache overflows or invalidated when the data are updated by other caches. A cache overflow occurs when newly referenced data must be brought into the cache from main memory, but, because all the cache blocks that can hold data are already occupied by other data, there is no available space. Therefore, the updated data in the cache must be written back to main memory to make room for the new data. Thus, the *non-write-through* protocols are more efficient in reducing bus traffic, but more complex than the *write-through* protocols. There are many variants of the *write-back* protocols, [2, 13, 14, 15, 16, 17, 18, 19, 20, 21]. However, we describe briefly one of the more

efficient protocols: *berkeley protocols.*

The protocols are implemented in a RISC multiprocessor system designed at the University of California at Berkeley, [18]. The scheme requires four states for each line (block) to indicate status of the data in the line: *Invalid*, *Valid*, *Shared–Dirty*, and *Dirty* (no other copies in caches and modified). It uses the idea of ownership — the cache that has the line in state *Dirty* or *Shared – Dirty* is the owner of that line. If a line is not owned by any cache, the main memory is the owner. And in any case, a line is sent, upon request of the other cache, by the line owner to the requesting cache. Therefore, a line in state *Dirty* can reside in only one cache at any time. Also a line in state *Shared – Dirty* can be in only one cache. However, it might also be present in another cache in state *Valid.* Moreover, a line in either *Dirty* or *Shared – Dirty* has to be written back to main memory if it is selected for replacement by a new line. The consistency solution is the following:

1. Read miss in one cache. If the line is *Dirty* or *Shared – Dirty* in the other cache, the cache with that copy must supply the line directly to the cache of the read miss and set its state to *Shared – Dirty*. Otherwise, main memory has to send the requirted line to the read miss cache. In any case, the line in the requesting cache is set to *Valid.*

2. Write hit in one cache. If the line is already *Dirty* in that cache, the write proceeds with no delay (no bus request is required). If the line is *Valid* or *Shared – Dirty*, an invalidation signal is broadcast to all caches before the write is allowed to proceed. All other caches invalidate their copies of the requested data. The state of the write-hit line in the originating cache is changed to *Dirty*.

3. Write miss in one cache. Like a read miss, the line comes directly from the owner. All other non-owner caches with copies of the requested line change

the local state to *Invalid* and the line in the requesting cache is loaded in state *Dirty*.

It is obvious that the Berkeley protocols are more efficient than the write-through protocols since a requested line comes from a cache, if the cache is the owner, and transfer of a line from a cache is faster that that from main memory. Furthermore, the bus requests are only required when the requested lines are shared or when the lines are the modified ones and need to be written back to make room for new lines. However, this scheme requires more hardware for the line state bits and the protocol controller than that needed by the write-through scheme.

2.4 Cache Concepts and Design Considerations

It is well-known that caches are one of the simplest and most effective ways to improve performance of systems ranging from personal computers to supercomputers. Cache memories are an active area of current research. Cache design for different computers has been extensively studied since the concept was introduced by the IBM. The second bibliography ,[22], includes 487 papers, notes, and books have been published since 1968; and the literature in this area has more than doubled in the last eight years. Many papers focus on the performance effects of the major cache design decisions, such as cache size, line size, way size (associativity), fetch strategy, *etc*, for uniprocessor systems, [23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34]. A survey paper by A. J. Smith, [1], gives the most complete summary of these issues and guideline of choices of these cache parameters and strategies for high performance cache design. There are also a number of books in which cache memories are presented in varying degrees of detail, [4, 35, 36, 37, 38]. More recently, the development of commercial multiprocessors has sparked a great deal of research into cache data coherence protocols and cache design for shared memory multiprocessors. A bibliography for

multiprocessor cache memories lists 251 articles for design of cache coherence, cache-based multiprocessors, performance evaluation techniques, concurrency models, [39]. A number of papers focus on the issues on cache coherence protocols and cache design, [2, 8, 12, 14, 15, 17, 18, 19, 20, 21, 40, 41, 42, 43, 44, 45, 46, 47, 48]. There are also many articles on performance evaluation and modeling of multiprocessors using either analytical modeling techniques or simulations, [5, 49, 50, 51, 52, 53, 54, 55].

The capacity of cache memory is far smaller than that of main memory for high speed; that is, the address space of the cache memory is far smaller than the address space of the main memory. Thus, a cache memory needs an address mapping mechanism to translate the main memory addresses, at a high speed, into the cache memory addresses where corresponding data in the main memory has a copy. Also because the most active portions in the main memory are copied in the cache memory, if the cache memory is full and data requested by the associated processor do not reside in the cache, some line of data in the cache is to be replaced with the new requested line from the main memory. This requires an algorithm which can, hopefully, predict the line, which is unlikely to be used in near future, to be replaced. This decision is determined by a line replacement unit. Clearly, the cache-replacement decision directly affects the performance of the cache. Hence, the basic structure of a cache memory must include at least the three basic hardware components: an address mapping mechanism, a line replacement unit and storage cells.

Each reference from the processor to a memory location is presented to the cache memory. The cache first searches its directory (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 accessed by the processor immediately without disturbing the main memory. Otherwise, a miss signal arises which causes the transfer of the whole line where the requested data reside from the main memory to the requesting

cache. Then the requested data in the new line are referenced by the processor. Before transferring the new line to the cache, some line has to be removed from the cache memory to make room for the new one if the cache is full.

Since a cache memory has to be compatible with the associated processor, the speed of cache memory is a key factor in cache memory design. Thus, all the algorithms of a cache memory have to be implemented in hardware. Therefore, the design of cache memory has to consider how to implement cache functions into practical high-speed hardware. Furthermore, choices of proper parameters of cache memory and tradeoffs between these parameters affect the cache memory performance. Typically, a cache memory system can capture over 90 percent of all references to main memory, provided that the cache is properly designed. Optimization of cache design is very significant for high-performance cache memories. It has four aspects, [1]:

1. minimizing the miss 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 multiprocessor systems, during cache design, considerations have to be given to maximize bus and shared-memory bandwidth and to minimize the bus bandwidth required by each processor in order to maximize the system performance. It is well-known that all of cache size, line size, way size, as well as fetch strategy affect the cache performance, so have to be considered at the design stage. Inevitably, trade-offs have to be made among the cache parameters and algorithms.

2.5 Fault-Tolerance and Testing

Though many of the basic ideas behind fault-tolerant design are conceptually simple, in practice, the design of a computer system to meet desired dependability specifications proves to be very complex. First, it is difficult to statistically characterize beforehand the type and frequency of hardware and software failures likely to afflict a system. Second, after a decision on the types of failures to be covered (protected against), it is difficult to select the fault-tolerant techniques which are best suited to the application with high performance/cost rate under real-life constraints such as weight, volume, power consumption, flexibility, maintainability, and similar considerations.

Today's computer systems are, by their nature, very complex structures. They contain large numbers of sub-systems and components with complex interrelationships, implemented in both hardware and software. For such complex systems, there are many elements that can fail in a wide variety of ways, because of numerous different causes, during the system lifetime. In fact, the increasing complexity of computer systems makes it more difficulty to ensure high dependability. This leaves open the possibility that a fault already present in a circuit might interact with a new failure in the field to defeat a fault-tolerance mechanism designed to only cover one fault at a time. Finally, a more fundamental problem is that it may fail to protect against unforeseen types of failures. Therefore, design decisions for the fault-tolerance have to be included from earliest stages of the systems design.

In order to provide reliable systems, there are, in general, two approaches: *fault prevention* and *fault tolerance*. Fault prevention aims to prevent faults from being present in operational systems, using both fault avoidance as well as fault exposure and elimination. Fault avoidance is concerned with design methodologies and the selection of techniques and technologies, to avoid the introduction of faults during the design and construction of systems. However, faults are usually present in systems

because the enormous complexity inherent in such systems results in oversights and faults in both the design and manufacture of semiconductor devices. Further, systems are subject to a wide variety of physical failures in normal operations. Fault removal checks computer systems and removes any exposed faults before normal operations. One of the most effective techniques for this purpose is off-line testing.

Directly addressing all the possible physical failures is generally intractable, except for very small circuits, so the testing discipline has been built upon *fault models* which assume the potential physical failures result in a definable logical behavior. Usually the fault models are classified into four levels: switch level, gate level, function level, and system level; and testing strategies are based on these defined fault models. In order to increase the testability of a system, proper incorporation of testability as a system (or circuit) design constraint is necessary, which enhances the system reliability.

The application of fault prevention techniques to computer systems has not, in general, proved sufficient for the attainment of high level dependability because physical components or devices age and deteriorate and can consequently become faulty. Failures eventually occur and result in system failure because of faults. Thus, fault tolerance is required, at least to protect the operational system against the effect of such faults. Design of fault-tolerant systems involves the selection of a coordinated response that, depending on the application and system architecture, may combine some or all of the following stages [56, 57]:

1. *Fault confinement* is achieved by limiting the spread of fault effects to one area of the system, thereby preventing contamination of other areas.

2. *Fault detection* is the process of recognizing that a fault has occurred.

3. *Fault masking* is the hiding of the effects of failures.

4. *Retry* is the second attempt to achieve a successful operation.

5. *Diagnosis* is the process of identifying the faulty modules responsible for detected faults.

6. *Reconfiguration* is the process of eliminating a faulty component by reconfiguring the components to replace the failed component or to isolate it from the rest of the system. Logical removal or isolation of a faulty component can be accomplished by switching off the component's power, forcing its output into an inactive state (hardware removal), or instructing all units to ignore or bypass it by updating the available resources tables (software removal).

7. *Recovery* is the process of backing up system operation to a recovery point (rollback checkpoint) for a task prior to fault detection. Establishing a checkpoint for a task involves saving a copy of all necessary information about the current correct state of the task such as values of data objects, registers, status words, etc..

8. *Restart* is the process of resuming system operation.

Our emphasis in this dissertation is the importance of fault tolerance for improving system reliability and availability.

2.6 Fault-Tolerance in Multiprocessors

Many computer applications require more powful computation capability and higher system reliability, availability, and modular expandability. One approach to meet these requirements is to develop multiprocessors with fault-tolerant ability. In general, any multiprocessor system, including both tightly-coupled and loosely-coupled multiprocessors, offers a certain degree of fault-tolerance capability due to the modularity and redundancy. Many authors [9, 57, 58, 59, 60] discuss the fault tolerant systems in general. Also a number of papers present real bus-based fault-tolerant

multiprocessor systems, [61, 62, 10, 63, 64]. These systems can be used for transaction processing, reservation, communication and information applications.

For a loosely-coupled multiprocessor, faults in a processing element, including a processor, its associated local main memory, as well as I/O devices, can be isolated within the element once they are identified, and the system can continue to operate correctly, although system performance is degraded because of removal of the faulty element. That is, a processing element fails independently without corrupting resources owned by other processing elements. A faulty element can be logically purged from the system, and its unfinished process can be transferred to a different processing element. However, the moving of processes between processing elements is expensive; and workload is hard to balance in a loosely-coupled system. In addition, in the case that there are several processing elements operating cooperatively, interprocessor communication significantly reduces the overall system performance; multiple copies of data may reside in the distributed main memory (non-shared) so that memory utilization is relatively low. The loosely-coupled multiprocessors provide easier upgrades and can be fault-tolerant; but expensive tuning, load balancing, and high overhead from interprocess communication reduce their effectiveness.

Unlike a loosely-coupled system, a tightly-coupled multiprocessor system has its main memory shared by all the processors in the system. In the main memory, there is a single queue of ready processes created by users. All the processors share the queue of ready processes so that any processor can assign processes to themselves by looking at the queue in shared memory. That is, when a processor is idle, it examines the queue and selects the next process to perform. Therefore, all the processors can do useful work as long as work is available, and there are no load balancing problems. Only one copy of each software module and data used by the system needs to be kept in shared memory. Interprocessor communication is easy, using the memory locations shared by the processors, without involving the

operating system. Faulty components, processor or main memory modules, can logically be purged from the system as soon as they are detected. Processes can easily be transferred to other processors by inserting the processes into the process queue, and the system continues as long as the errors do not propagate.

The major drawback from the fault-tolerance point of view is that faults in a processor can potentially propagate to other processors via shared main memory, causing the system to fail. The other reliability weakness of the tightly-coupled architectures is that, since all processors share the operating system's memory state, a software error that corrupts that state may cause all processors to fail. Thus, the reliability of a tightly-coupled computer also depends on the reliability of its operating system.

Chapter 3

CMOS Cache Design

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 processors may try to reference shared main memory at the same time. The performance of such multiprocessor systems is limited by the speed and bandwidth of the interconnection network and main memory. The long memory reference latency caused by network traffic can be greatly reduced by associating a private cache memory with each processor, capturing the majority of references from a processor to the main memory [1, 65]. Although use of private caches in a multiprocessor system can greatly reduce network traffic and shared memory contention and in turn speed up the system, such a system can cause a data coherence problem: multiple copies of data in the shared main memory may reside in several different caches at the same time. 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.

In order to find a reliable strategy to keep data in the separate caches coherent, many solutions have been proposed in [6, 17, 18, 19, 65] by implementing various dif-

ferent coherence protocols in caches. There exist two kinds of operations in a cache regardless of what coherence protocol is implemented in the cache: read/write operations from the associated processors (processor operations), and operations to maintain data coherence with other caches or memory in the system (coherence operations). Interference between the processor operations and coherence operations is unavoidable in a multiprocessor cache because both these operations may be required in the cache at the same time. As a result, cache performance is affected, and in turn performance of the cache-based multiprocessor system is degraded. In order for caches to handle such interference, one of two possible schemes is normally used. One is to maintain a single one-port directory in a cache which can be used by these two operations sequentially — one of the operation requests must wait to search the cache directory until the other releases the cache. This single directory scheme causes a performance degradation by serializing two potentially concurrent operations. Since it is simple to implement at low cost, this scheme is commonly used. The second scheme employs two directories in a cache; one for the processor operations and the other for coherence operations. Thus, these two kinds of operations can be carried out simultaneously. Although performance of such a two-directory cache is improved when there are two separate directories, the cache structure becomes more complex and more silicon area is required by such a cache since both an extra directory and a mechanism to keep the information between the two directories consistent are required. The overall cache cycle time may also be increased since the processor must write both tags in the directories and arbitration is required [53]. A single dual-port directory cache is presented in [66] by T. Watanabe. However, in the two-page paper, no details about the structure of the cache are given. The hardware overhead of the cache is not discussed; neither is there any justification of the performance improvements of the cache in a multiprocessor environment.

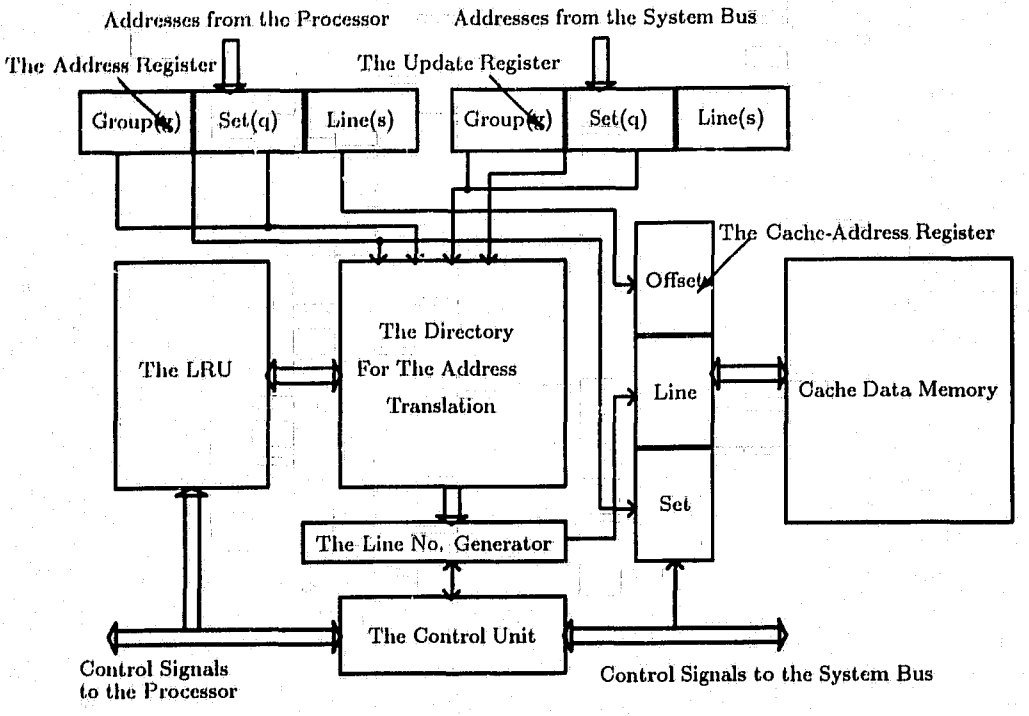


Figure 3.1: Structure of the Proposed Cache Memory

In this chapter, we propose a CMOS design for a multiprocessor cache memory in which only a single dual-port directory is used for both processor accesses and coherence operations simultaneously. Therefore, this cache can obtain the higher performance of a two-directory cache. Also, we show that the complexity of such a single dual-port directory cache is far less than that of a two-directory cache as well as the hardware overhead being lower. Furthermore, the cache cycle is not necessarily slowed down; and the extra silicon area can be used for additional cache memory, or other purposes, such as memory management logic. In the rest of this dissertation, we use the term *single directory cache* for a single one-port directory cache and *dual-port directory cache* for a single dual-port directory cache.

3.1 Structure of the Cache

Fig. 3.1 depicts structure of the cache which is composed of three basic functional components: the directory, the line replacement unit (LRU), and the cache data memory. The directory functions as an address mapping mechanism to translate main memory addresses, at high speed, into the cache addresses where copies of data in main memory reside. Since the most active portions of software in main memory are copied in a cache, if the cache is full and the associated processor needs data not residing in the cache, some of the data in the cache must be replaced with the newly requested data from main memory. Which data is to be removed from the cache before the new data can be brought in is determined by the LRU. The cache memory is used for storage of data and instructions.

In addition, there are three registers: an address register, an update register, and a cache-address register. The address register is used to latch addresses from the processor for the normal read/write operations, whereas the update register is employed to latch addresses from other caches for the coherence operations. These two registers can operate simultaneously. The cache-address register stores physical cache memory addresses from the directory so that the cache memory can be read/written. Furthermore, there is a *line number generator* which transforms cache memory addresses in one-out-of-n code from the directory into addresses in binary code, and the addresses in binary are latched in the cache-address register.

3.1.1 The processor operations

During the processor operations, an address from the processor is latched in the address register, and is sent to the directory, consisting of tags (line slots), to see if the line (block) containing the requested data is in the cache. If so, the corresponding cache line number generated by the line number generator is placed in the cache-

address register, and the requested data in the cache data memory are accessed by the processor. Otherwise, a line miss occurs. The LRU is asked to select a line to be removed to make room for the new line. Which line is to be selected depends on the line replacement algorithm implemented in the LRU. The LRU sends the tag number to the directory. The address part of this tag in the directory is replaced with the address in the address register, and the status part of the slot may also be modified. The cache address corresponding to this tag is sent to the cache-address register through the line number generator for transfer of the requested line from the main memory to the cache.(Some protocols require that the line to be replaced be written back before transfer of a new one.)

3.1.2 The coherence operations

During the coherence operations, the requested address from the system bus is latched in the update register, and the cache searches its directory. If the requested data is in the cache, the status part in the corresponding tag may be modified, depending on the data coherence protocols. Otherwise, nothing is done.

From the preceding description, it is clear that a tag in the directory has two parts: address and status. The address part stores a main memory line address, and the status part indicates status of the corresponding main memory line in the cache, *e.g.* valid, dirty, *etc.* The address part of a tag can only be updated during the processor operations with line miss occurrence. The status part may be modified by either the processor operations or coherence operations, depending on the data coherence protocols. The cache memory may be accessed by both operations, also depending on protocols. Furthermore, the LRU is only affected by the processor operations.

3.2 The Address Mapping Algorithms

Since the cache, as the fastest part of a memory hierarchy, must be much smaller than the main memory, there has to be a mapping function between the cache address space and the main memory address space.

In general, there are three mapping methods used in cache design: direct-mapping, full-associative, and n-way set-associative. The direct-mapping method is a method that any given line in main memory can reside only in one specified cache line while the full-associative mapping method is a scheme that each main memory line can be mapped into any of cache lines. The n-way set-associative mapping method is a hybrid of the direct-mapped and full-associative methods.

The direct-mapped method is the simplest to implement, but it has the highest miss ratios of the three mapping methods. Usually the fully-associative method has the lowest miss ratios. However, an address from the processor has to be sent to all tags in the directory to see if there is any one matching the address. It introduces longer tag-search delay. Thus, it is difficult to implement on a large scale cache memory.

The set-associative method organizes cache memory into Q sets with n lines per set. When Q becomes one, the cache is a fully-associative cache in which there are exactly n lines. If n becomes one, the organization of the cache is the direct-mapped cache memory. An n-way set-associative cache is required to search n tags whenever there is a request. Therefore, the tag-search delay may be less than that for the full-associative method. Also, since it allows any one of n lines in a referenced set to be replaced when a line miss occurs, this flexibility usually introduces lower miss ratios, without the complexity of a fully-associative cache. Thus, it is a compromise between complexity and performance. Since the direct-mapped and full-associative are two extreme cases of the set-associative scheme, without losing generality, we employ the set-associative mapping method in our model.

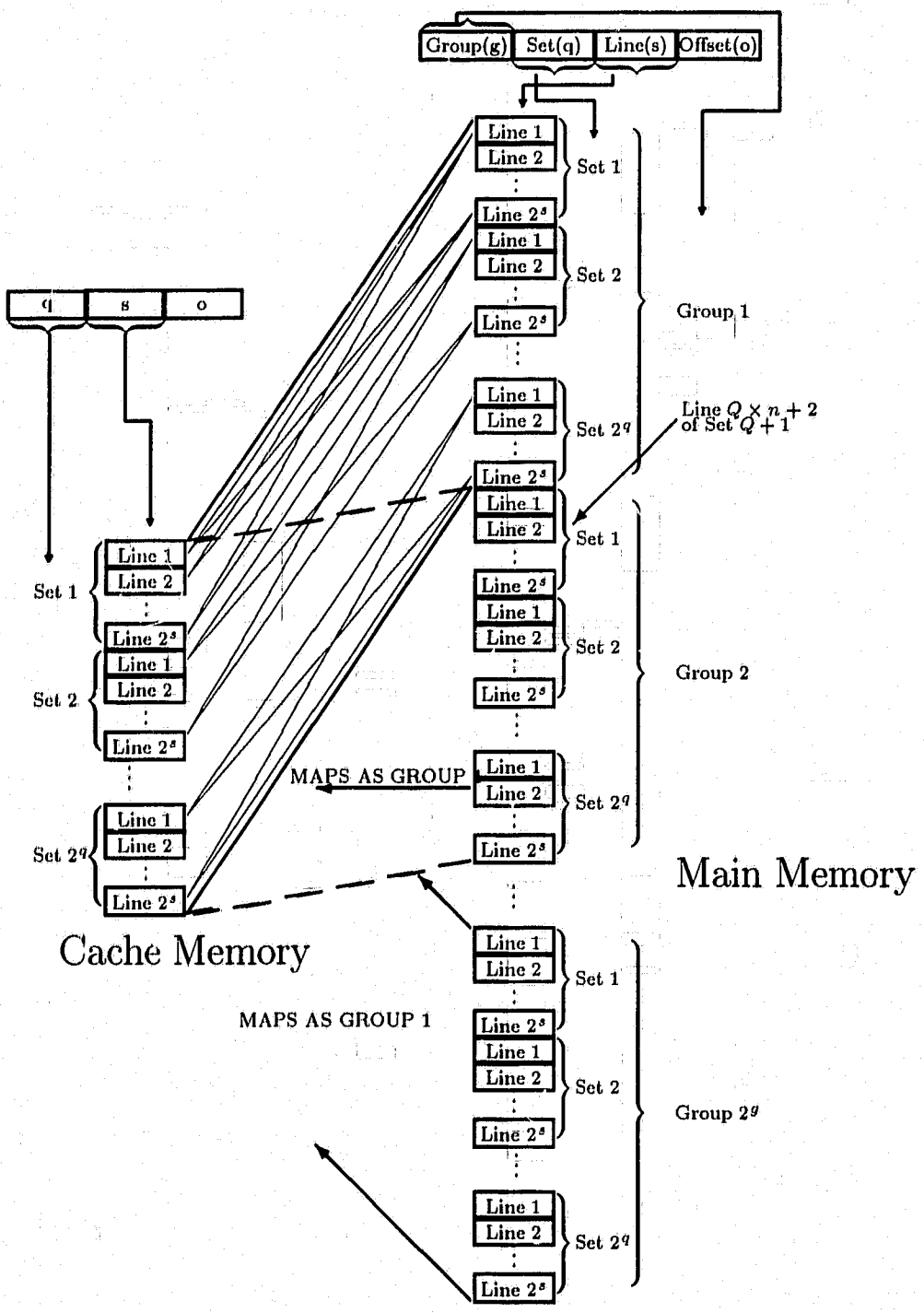


Figure 3.2: The General Set-associative Mapping Function

The principle of *set-associative* mapping is demonstrated in Fig. 3.2. The cache memory is divided into Q ($Q = 2^q$) sets with n ($n = 2^s$) lines in each, and sizes of 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 Q sets. Each set in the cache memory must be shared by several sets of the main memory. For example, as shown in Fig. 3.2, the first set in the cache memory is assigned to hold the sets 1, $1+Q$, $1+2Q$, $\cdots$ of the main memory and the second set is for holding the sets 2, $2+Q$, $2+2Q$, $\cdots$ and so forth. Lines within a set of the main memory are associatively mapped into any of the n lines in the corresponding set of the cache memory (Therefore, the memory is n-way associative). 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 n lines in the corresponding cache set. Note that not all the sets of a given group in the main memory need to be simultaneously resident in the cache memory; sets from different groups can be intermixed within the cache memory. Similarly, lines in a set of the cache memory can come from any sets of different groups of the main memory. For instance, line 1 of set 1 of group 1 is assigned to line 2 of set 1 in the cache memory while line $Q \times n + 2$ of set $Q + 1$ in group 2, as shown in Fig. 3.2, is residing in line 1 of set 1 at the same time.

3.3 Implementation of the Directory

According to the set-associative mapping method, a logical memory address in the address or update register, as shown in Fig. 3.1 and Fig. 3.3, is divided into four parts: g bits represent the group number, q bits are the set number, s bits are the line number, and o bits are the word offset within a line. Meanwhile, a cache address

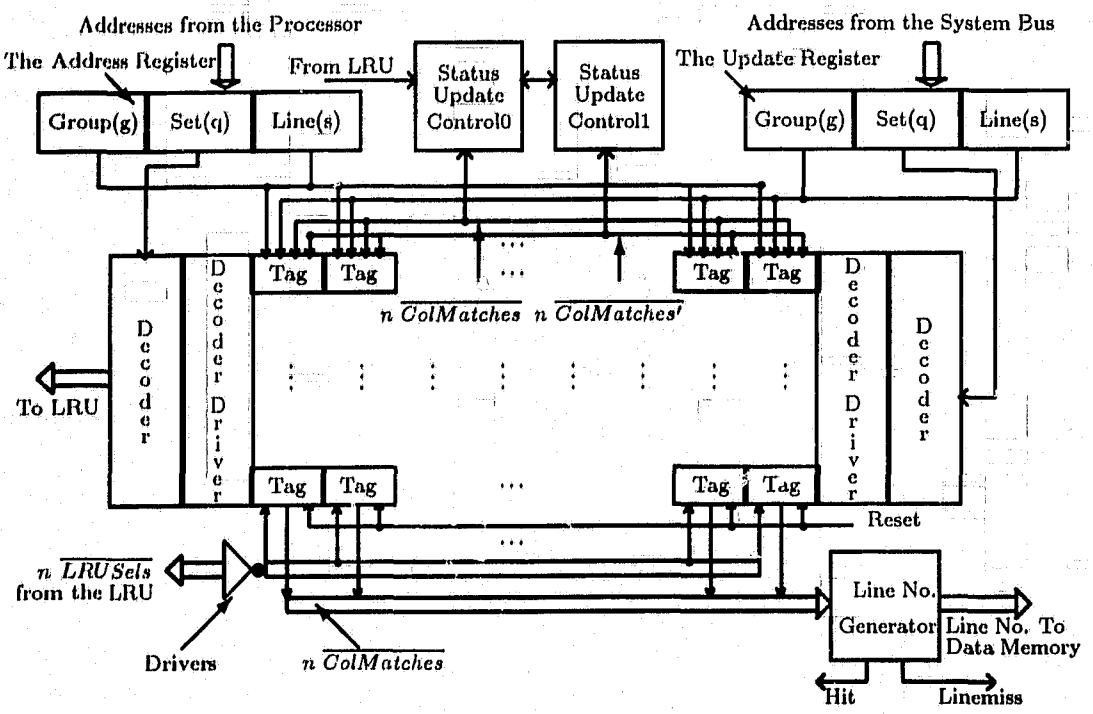


Figure 3.3: The Dual-Port Directory

is partitioned into the set number of q bits, the line number of s bits, and the word offset of o bits. Since, in the set-associative mapping, a set in main memory can only be directly mapped into a given set of cache memory and one main memory line in a given set can be associatively mapped into any of the n lines in a corresponding set in cache memory, the directory is organized as a tag array of Q (2^q) cache sets with n (2^s) tags in each set. Each tag is used to store a main memory line address, indicating the main memory line resides in the corresponding cache line. Set number q in a logical address is directly used to select the cache set, whereas $g+s$ in the logical address is stored in the address part of tags in the directory for associative address search. Note that $g+s$ is the group number concatenated with the line number in the address register and update register. In the sequel, we use $g+s$ to refer to these sections of the registers. In addition, the word offset is sent directly to the cache-address register to access the corresponding words in a given

cache line.

3.3.1 Structure of the directory

Fig. 3.3 shows the structure of the proposed dual-port directory. Each set is represented by a row of the tag array and the n ways are indicated by the columns. Therefore, this directory has a total of $Q \times n$ tags. As shown in Fig. 3.3, there are n $\overline{ColMatch}$s, a $\overline{ColMatch}$ for each column of tags, in the array. Whenever an address from the processor is latched into the address register, the corresponding set is selected through the *directory decoder*. All the n tags in the selected set are simultaneously compared with g+s of the address register. If the address contents in any one of the n tags match the g+s, the corresponding $\overline{ColMatch}$ signal becomes valid (active low), indicating that the requested main memory line resides in the cache line implied by the valid $\overline{ColMatch}$. All other $\overline{ColMatch}$s must be high. Thus, signals on the n $\overline{ColMatch}$s are formed in an (n-1)-out-of-n code ((n-1)/n code), the complement of the one-out-of-n code (1/n code). These signals are sent to the line number generator. The line number generator translates the valid codeword i in the (n-1)/n code into the corresponding cache line number i in binary code, which is latched in the cache-address register. Meanwhile, the Hit flag is generated to inform the processor to access the requested data in the cache data memory.

If all the $\overline{ColMatch}$ signals are invalid, the $Miss$ signal is set high so that the LRU is invoked to find a line in the cache probably not to be used in the near future, in terms of the line replacement algorithm that the LRU employs. The selected tag column number in the (n-1)/n code is sent by the LRU to the directory through wires $\overline{LRUSel}$s, as shown in Fig. 3.3. Each $\overline{LRUSel}$ is connected to all tags in a tag column. The tag to be updated is determined by the valid $\overline{LRUSel}$ (active low) and the valid row selection from the row decoder of the array. The address part of the selected tag is updated with the g+s of the address register. After the update,

the address contents in the tag always matches the *g+s* of the address register. The $\overline{ColMatch}$ corresponding to the tag column that the updated tag locates becomes valid and is sent to the line number generator. The corresponding cache line number is generated and latched into the memory register so that the requested line can be transferred from the main memory to the cache line.

The valid $\overline{ColMatch}$ is also sent to the *Staius Update Control 0* (SUC0). The SUC0 may send a command back to update the status part of the tag in the directory, depending on the coherence protocols employed. Note that, among the n $\overline{ColMatch}$ signals, at most one can be valid at any time since the requested main memory address only can possibly reside in one of the tags.

On the other hand, whenever there is a data coherence request from other caches via the system bus, an address is latched in the update register, and the cache also immediately searches the directory by simultaneously comparing the n tags in the selected set with *g+s* of the update register. If any one of the n tags matches the *g+s*, the comparison results are sent to the *Status Update Control 1* (SUC1) through the n $\overline{ColMatch}$'s, each for a tag column. The SUC1 may modify the status part of the corresponding tag, also depending on the coherence protocols. Otherwise, no operation is carried out.

Note that the SUC0 is to update the status of tags in the directory during the processor operations while the SUC1 is to deal with status modifications during the coherence operations. Both units depend on the specific coherence protocols that are adopted so we leave these units to be designed independently from this discussion. Thus, the structure of this cache is protocol-independent in that any coherence protocols can be implemented.

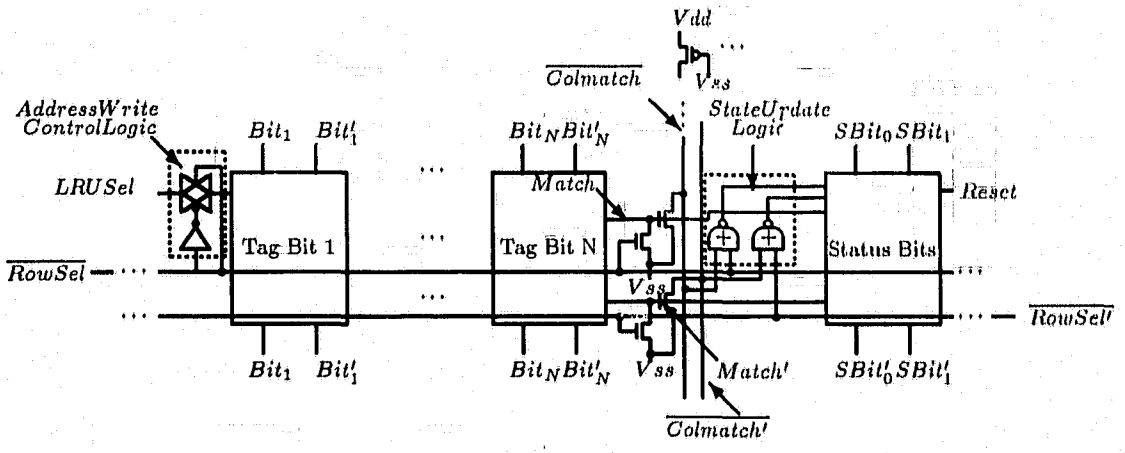


Figure 3.4: A Tag of the Dual-Port Directory

3.3.2 Structure of a tag

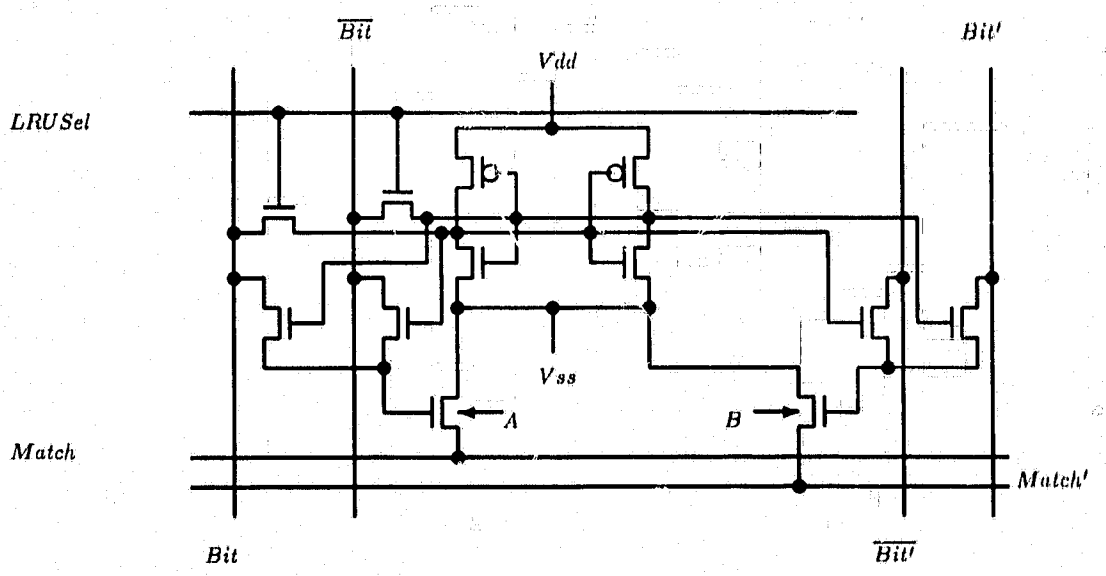
A tag, as shown in Fig. 3.4, consists of an N-bit address part and a 2-bit status part (two status bits used in this example show the connection of more than one status bit, though any number of bits can be chosen depending on protocols). It can be initialized by *Reset*. Other signals can be divided into two groups: one with prime ($'$) and the other without prime. The signals without prime are used for the processor operations, whereas ones with prime are for the coherence operations.

During the processor operations, if set selection $\overline{RowSel}$ from the directory decoder is at logical 0, the path from the *Match* to *Ground* (Vss) is cut off by the N-device. The address bits in the tag are compared simultaneously with the *g+s* (from the address register) on Bit_1 to Bit_N. If all the address bits match the *g+s*, the *Match* signal is high to turn on the N-type device so that the $\overline{ColMatch}$ is pulled low (valid). The result on the $\overline{ColMatch}$ is sent to both the *line number generator* for generation of a valid cache line number. It, along with the current state of the tag selected by the $\overline{RowSel}$, is also sent to the SUC0 for determination of a status-update operation. The decision from the SUC0 may be sent back to the selected tag by setting new state values on the $SBit_0$ and $SBit_1$ so that the status

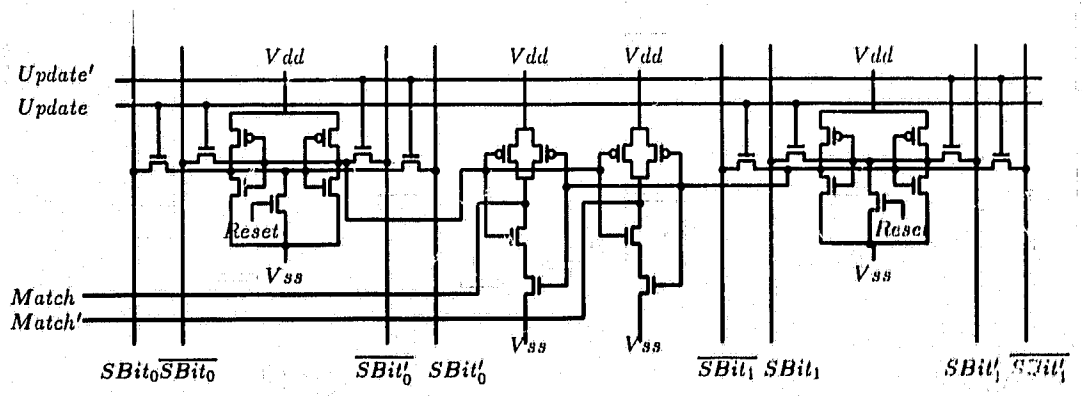
bits are updated. Otherwise, the *Match* remains low so that the $\overline{ColMatch}$ is high to indicate that the requested data is not in the cache line referenced by this tag. If none of the n tags selected by the $\overline{RowSel}$ match the g+s, a line miss occurs. If the LRU chooses this tag as to be replaced, it sends a valid $LRUSel$, the output of the inverted driver from the corresponding $\overline{LRUSel}$ in Fig. 3.3, to the tag column where the tag locates. As shown in Fig. 3.4, address-write control logic, controlled by the valid $\overline{RowSel}$, allows the high $LRUSel$ to update the address part with the g+s on the *Bit*s.

Similarly, during the coherence operations, if the $\overline{RowSel'}$ from the *update register* is at logical 0, the address in a tag is compared with the g+s on the Bit'_1 — Bit'_N. The result is on the $Match'$ and sent to the SUC1 via the $\overline{ColMatch'}$. New state values may be fed back to the status part of that slot through the $SBit'_0$ and $SBit'_1$ so that the status bits are updated. Note that the status bits of the tag to be updated are located by either the valid $\overline{RowSel}$ and $\overline{ColMatch}$ or the valid $\overline{RowSel'}$ and $\overline{ColMatch'}$. Furthermore, both the $\overline{ColMatch}$ and $\overline{ColMatch'}$ are precharged through the P-type devices before evaluation.

Fig. 3.5 (a) shows one bit of the address part. If the $\overline{RowSel}$ is at logical 1 (this tag is not selected), the *Match* line remains low so that no comparison can be done in any case. When the $\overline{RowSel}$ becomes low, this cell becomes a normal content addressable memory cell. If the datum on the *Bit* matches the value in the cell, the *match transistor A* remains off so that the *Match* is high for this bit (*match*). Otherwise, transistor A is turned on so that the *Match* is pulled down to *Ground* (*mismatch*). In a tag, all the *Match* lines of the address bits are cascaded together. If any address bit of the slot produces a *mismatch*, the *Match* line of this slot is low, indicating that this slot does not match the g+s from the *address register*. Otherwise, the *Match* line is charged by the status bits. During a line miss, the $LRUSel$ is asserted. The datum on *Bit* is written into this bit. Also, for



(a) One Address Bit



(b) Two Status Bits

Figure 3.5: Address and Status Bits of the Line Slo

the coherence operations, the $\overline{RowSel'}$ becomes low. When the datum on the Bit' matches the value in the cell, the *match transistor* B is turned off so that $Match'$ of this bit is valid. If all the chained $Match'$ lines of the address bits in the selected tag are valid, the $Match'$ line of this slot is high so that the precharged $\overline{ColMatch'}$ is pulled down. Otherwise, the mismatching bit(s) pulls down the $Match'$ of this slot. Note that, unlike the processor operations, the comparison results for the coherence operations are not sent to the LRU if the requested data is not in the cache.

Fig. 3.5 (b) depicts the structure of two status bits. There are two NAND gates between the two status bits. Outputs $Match$ and $Match'$ of the NAND gates are connected to the corresponding lines of the address part in the tag in Fig. 3.5 (a), respectively. Initially, the status bits (00) are reset by *Reset* so that both the $Match$ and $Match'$ are low. There is no comparison in this slot since the NAND gates pull the $Match$ and $Match'$ down to *Ground*. For other states of the status bits (01, 10, and 11), both lines $Match$ and $Match'$ are charged through the P-type devices of the NAND gates since the pathes of N-type devices are cut off. Thus, the slot can be evaluated as long as the set containing this slot is selected by the corresponding $\overline{RowSel}$. If the $\overline{ColMatch}$ is valid during a processor operation, the $Update$ of the status bits is high so that the current state of the slot is sent on the $SBit_0$ and $SBit_1$ to the SUC0. Similarly, a valid signal on the $Update'$ during a coherence operation allows the state of the slot to be sent to the SUC1 through the $SBit'_0$ and $SBit'_1$. States of these bits can be changed by setting specified values on either the $SBit_0$ and $SBit_1$ or $SBit'_0$ and $SBit'_1$ from either the SUC0 or SUC1.

3.4 The Line Number Generator

The circuit for the *line number generator* is illustrated in Fig. 3.6. The function of the line number generator is to translate the line number of the cache memory from

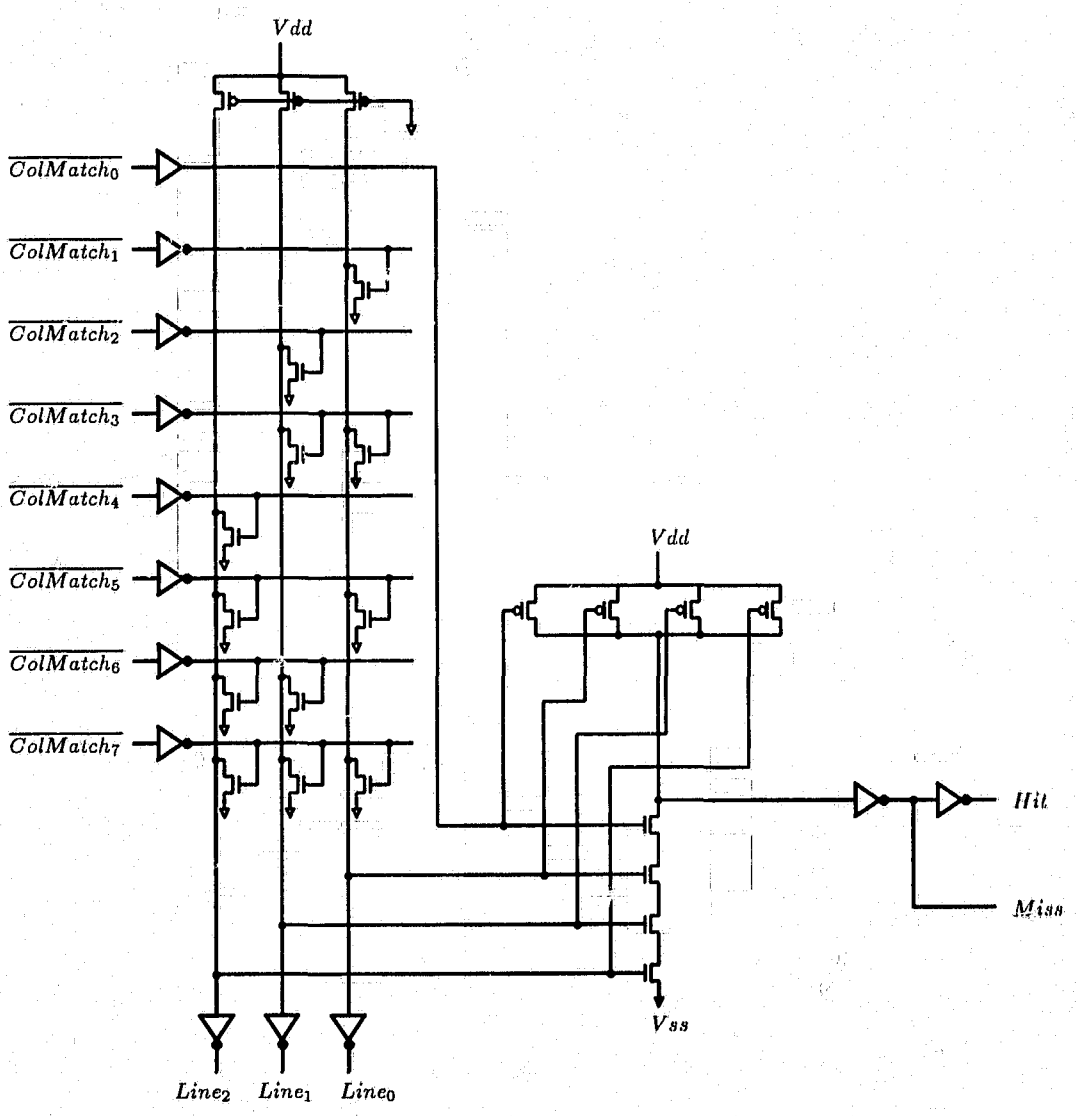


Figure 3.6: The Cache Line Number Generator

7-out-of-8 Code								Binary Code		
1	1	1	1	1	1	1	1	*LineMiss*		
1	1	1	1	1	1	1	0	0	0	0
1	1	1	1	1	1	0	1	0	0	1
1	1	1	1	1	0	1	1	0	1	0
1	1	1	1	0	1	1	1	0	1	1
1	1	1	0	1	1	1	1	1	0	0
1	1	0	1	1	1	1	1	1	0	1
1	0	1	1	1	1	1	1	1	1	0
0	1	1	1	1	1	1	1	1	1	1

Table 3.1: The Truth Table of the 7/8 Code and the Corresponding Binary Code

a *modified* (n-1)/n code, composed of n $\overline{ColMatch}$ lines from the directory, into the corresponding binary code. It is a modified (n-1)/n code because, besides the normal codewords in a (n-1)/n code, there is one codeword "all-one" which is used to indicate an occurrence of a line miss. Therefore, in a modified (n-1)/n code, at most one bit in the n-bit codewords may be a logical 0. Table 3.1 shows the truth table of the 7/8 code (where n is 8) and its corresponding binary code. Each bit of the 7/8 code corresponds to one of 8 match lines $\overline{ColMatch_0} - \overline{ColMatch_7}$ from the directory. If 8 bits of the code are ones, then it means that none of match lines is logical 0. Therefore the line number generator produces a $Miss$. If any one of the $\overline{ColMatch}$s is logical 0, the generator outputs the corresponding line number of the cache memory in 3-bit binary code on the output lines $Line_0$ to $Line_2$; meanwhile, the Hit is asserted. This binary line number is latched into the memory register for accessing data in the cache data memory.

3.5 Conflicts between Two Operations

As discussed, this directory can be searched simultaneously by both operations. Conflicts may only occur when two operations try to modify the status bits of a tag at the same time. That is, the SUC0 tries to update the status bits of a tag during a processor operation while the SUC1, at the same time, is modifying the same status bits if the requested data is found in the corresponding line for a coherence operation. This problem can usually be handled by co-operation between the SUC0 and SUC1 in terms of the coherence protocols that are implemented. For example, assuming that the *write-through* protocols are used in this cache, if a coherence operation tries to modify the status of a tag while a processor operation causes a line miss and the LRU selects the same line to be replaced, only the operations for a line miss are carried out. The worst case is that conflicts cannot be avoided in which situation the cache must delay one of these two operations until the other finishes. The possibility of this occurring is usually quite small, depending on the number of the tags in the directory, and the protocols implemented, as well as frequency of simultaneous occurrence of the two operations.

A simple alternative way to handle this problem is to permit at any time only one operation access a particular row of tags in the directory. That is, at any time, only one of the $\overline{RowSel}$ and $\overline{RowSel'}$ in a row of the directory can be enabled. It is obvious that this scheme is suitable for a directory which has a smaller number of ways of searching in a set since there are fewer slots in a set. The best case for this scheme is that the direct-mapped directory is employed where there is only one slot per set while the worst case is that this scheme is used in a full-associative cache where only one of two types of operations can be executed at any time.

In the case that the coherence operations may update *stale* data in the cache memory instead of invalidation, the dual-port memory can be used for data storage so that both the processor operation and update operation for data coherence can

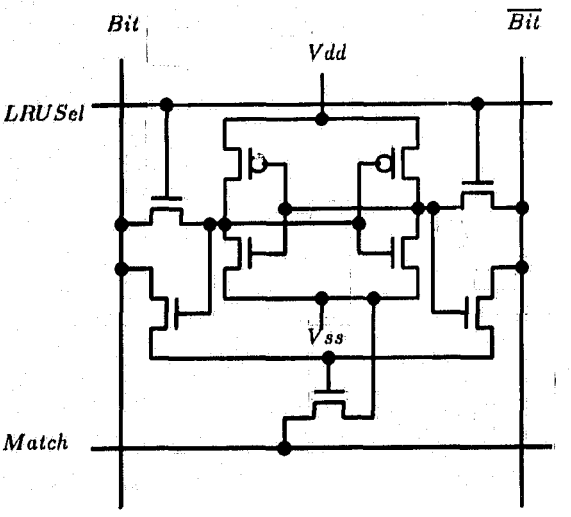


Figure 3.7: A CAM Tag Cell of the Single-Directory Cache

be done simultaneously as long as they do not operate on the same data. Otherwise, the two operations have to access the memory sequentially. As we have seen, the operations of the dual-port directory cache are almost the same as that of a two directory cache.

3.6 Complexity Analysis

In this section, a comparative study of the hardware complexity is carried out on three schemes: single directory, two separate directories, and dual-port directory. There are many different ways to calculate costs of circuits; for example, the number of transistors in circuits or the area used by circuits are commonly used. Since we emphasize circuit designs at the transistor level rather than chips, our evaluation is based on the number of transistors, including both P-type and N-type transistors, used in tags of the cache management units in the different schemes. This is because about 70% to 80% of the area in a cache management unit is used for tags of the cache directory. A tag bit in the single directory cache management unit is shown in

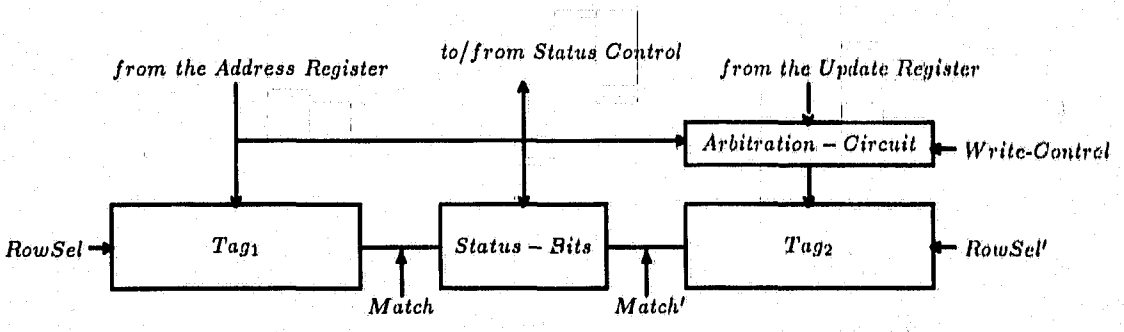


Figure 3.8: A Tag of the Two-Directory Cache

DESIGN BASED ON CAM				
Schemes	N_{tag}	N_{dir}	Overhead(%)	Other Extra Costs
One-Directory	$9N + 28$	$N_{tag} * (9N + 28)$	– – –	– – –
Dual-Port-Dir.	$12N + 42$	$N_{tag} * (12N + 42)$	$\frac{(12N+42)-(9N+28)}{9N+28}\% \approx 33\%$	a decoder
Two-Directory	$18N + 46$	$N_{tag} * (18N + 46)$	$\frac{(18N+46)-(9N+28)}{9N+28}\% \approx 100\%$	a decoder and arbitration
DESIGN BASED ON SRAM				
Schemes	N_{tag}	N_{dir}	Overhead(%)	Other Extra Costs
One-Directory	$6N$	$N_{set} * N_{way} * (6N + 14)$	– – –	– – –
Dual-Port-Dir.	$8N$	$N_{set} * N_{way} * (8N + 18)$	$\frac{(8N+18)-(6N+14)}{6N+14}\% \approx 33\%$	a decoder, N_{way} read/write logic and N_{way} comparators
Two-Directory	$12N$	$N_{set} * N_{way} * (12N + 18)$	$\frac{(12N+18)-(6N+14)}{6N+14}\% \approx 100\%$	the above extra costs plus arbitration logic

N is the number of address bits in a tag

N_{tag} ($= N_{set} * N_{way}$) is the number of transistors in a tag

N_{dir} is the number of transistors in the tag array

N_{set} is the number of sets (rows) in the tag array

N_{way} is the number of ways (columns) in the tag array

Table 3.2: Cost Comparisons for the Three Schemes

Fig. 3.7, which costs nine transistors ($C_{bit} = 9$). The two-directory cache employs two single directories. Fig.3.8 illustrates one tag of the two-directory cache in which tag_1 is used for processor operations whereas tag_2 is for data coherence operations. Each bit in tag_1 or tag_2 is the same as that in the single directory cache. Thus, there are eighteen transistors (C_{bit}=18) for each address bit in a tag (including both tag_1 and tag_2). The status bits are the same as that in Fig.3.5 (b) except that $Match'$ comes from tag_2. The dual-port directory cache requires twelve transistors (C_{bit}=12) for each bit in a tag. Some extra hardware such as the directory row decoder depends on implementation. Therefore, it is only indicated in the table instead of calculated with the number of transistors. Calculation of the cost for each tag in the three schemes can be obtained using the equation:

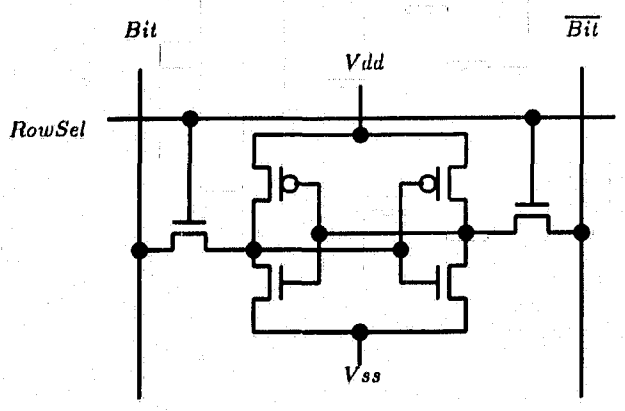
$$C_{tag} = NC_{bit} + 6M + C_{update} + C_{status} + C_{replace}$$

where N is the number of bits in a tag for an address, M is the number of *Match* and *Match'* lines, C_{update} is the cost of state-update logic, C_{status} is the cost of status bits associated with each tag, and $C_{replace}$ is the cost of address-write control logic in a tag, as shown in Fig.3.4. In the case of the single-directory cache, C_{update} is four transistors; and in the cases of the two-directory cache and dual-port directory cache, C_{update} is eight. In the comparison, although the number of the status bits varies depending on data coherence protocols, we assume that there are two status bits in a tag for the three schemes. There are nine transistors in each status bit for either the two-directory cache or the dual-port directory cache, and seven transistors for the single directory cache. $C_{replace}$ is eight transistors for the two-directory cache because there are two tags for an address, and four for other schemes. Table 3.2 shows the evaluation for the three schemes. Each row in the table corresponding to a named scheme. Column 1 indicates the number of transistors in each tag. Column 2 shows the number of transistors in a tag array for each scheme. Column 3 is the hardware overhead in percentage that the scheme uses. That is, the hardware

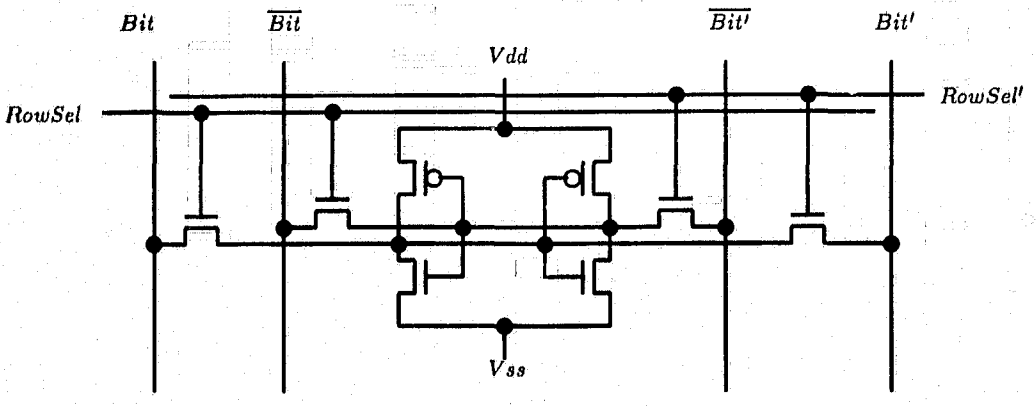
overhead in the tag array of dual-port directory cache is about 33%, and that of two-directory cache is approximately 100%, compared to the single-directory cache. Column 4 gives some other extra hardware that is needed in the dual-port directory cache and the two-directory cache, compared to the single directory cache scheme. Note that the single directory cache, however, requires an extra mechanism, which is not needed in other schemes, to serialize the two concurrent operations. From the comparison, the proposed cache is a compromise between the single-directory cache and the two-directory cache in terms of hardware complexity.

Note that the above design of the tag array is based on the *content addressable memory* (CAM). However, a similar result is obtained, as shown in Table 3.2, for a design of the tag array based on conventional *static random access memory* (SRAM) and comparators for each way in the tag array. We assume that a single-port SRAM cell requires six transistors, and a dual-port SRAM cell needs eight transistors, as shown in Fig 3.9. Again, in the comparison, we do not count the costs for the directory decoder, read/write logic, and the way comparators since they depend on VLSI design implementation as well as the technology used. Instead we list these in the column of the table as *other extra costs*. Although the cost of a cache employing dual-port RAM cells seems less than that of a cache using dual-port CAM cells, there are some concerns about cache speed. On one hand, since speed of a cache is required to be compatible to that of its associated processor, the cache speed is the major concern in cache design. On the other hand, since the area of the tag array mainly depends on the size of tag cells, usually the size of transistors used in the tag cells is as small as possible. Use of RAM cells in a cache tag array may have two problems concerning to the cache speed.

1. As shown in Fig. 3.9 (b), each RAM cell has two transistor gates connecting to the row selection from the directory decoder. Each tag consists of N cells (usually N is greater than 20 if the address length is 32 bits). Thus, the ca-



(a) A RAM Tag Cell for the Single-Directory Cache



(b) A RAM Tag Cell for the Dual-Port Directory Cache

Figure 3.9: The Cache Tag Cells in the SRAM Implementation

pacitance on the row selection is very large, when the ways (columns) of the tag array are large, so that the row selection signal is considerably delayed. Although use of intermediate drivers on the row selection and some design techniques are useful, this problem can not be eliminated. This design can solve this problem, since there is only one transistor, for each tag, connecting to the row selection shown in Fig. 3.4. Consequently, there is only the capacitance of n transistors for n-way associative tag array on the row selection. Hence, this design is consistent with our assumption that n can be any value between 1 to the number of tags, corresponding to the directed mapping and full-associative mapping respectively.

2. Since the size of transistors in a tag cell is small, the driving power of such a transistor is weak. If RAM is employed and there are many rows in the tag array, all the bit lines of cells are connected together so that the capacitance on each bit line is large, especially for a large cache. When there is a searching action, delays on these bit lines may be considerable as related to the expected cache speed even though read amplifiers are used (there is no problem for the similar RAM to be used in main memory since speed of main memory is slower than that of cache). If a cache to be designed has a large capacity, this problem is more serious so that it may not be tolerated. This situation is improved in the proposed design. Because of the built-in comparators, there is no need to read an address out of the tag array for comparison. The comparison result on *March* in Fig. 3.4 drives a relatively large size of transistor; and, in turn, the delay for the result to be on the corresponding $\overline{ColMatch}$ is small.

Furthermore, although use of RAM in the tag array makes an error-detection code easily employed in the directory, it would cause further delays in the searching actions. Based on the above reasons, in this design, we still use the dual-port CAM cells in the tag array instead of the dual-port RAM cells.

Both a dual-port directory cache and a two-directory cache can handle processor operations and coherence operations simultaneously. However, if there is a line miss in the two-directory cache during a processor operation, an address must be written into tag_1 and tag_2 in the directories, respectively, since contents in tag_1 must be identical to that in tag_2. An arbitration circuit, as shown in Fig.3.8, is used to determine whether the tag is written with a new address or it is checked for data coherence. This impacts the two-directory cache write time. Furthermore, the addition of the extra directory may increase capacitance on the address lines, and in turn increase the overall cycle time. Thus, the result of both of these effects may slow down the overall cache cycle time. For the dual-port directory cache, if a coherence operation checks a tag while a processor operation causes a line miss and the LRU selects the same tag to be replaced through the corresponding *LRUSel*, the signal *LRUSel* invalidates the output *Match'* of the tag produced by the coherence operation while writing an address into the tag. Therefore, no arbitration circuit is required in the dual-port directory cache. Hence, the overall cycle time of a dual-port directory cache can be shorter than that of a two-directory cache if the caches have the same VLSI implementation.

3.7 Summary

Using VLSI technology, the proposed cache only has one single dual-port directory which can be operated for both processor accesses and coherence operations simultaneously. This cache has a protocol-independent structure so that any of the standard data coherence protocols can be implemented. About 33% extra hardware, compared to a single directory cache, is needed for high performance, which is far less than that required by a two-directory cache which has over 100% extra hardware. The overall cycle time of a dual-port directory cache may be shorter

than that of a two-directory cache. In the next chapter, we discuss the simulation models created for a multiprocessor environment and performance evaluation based on simulations.

Chapter 4

Simulation and Evaluation

A typical multiprocessor system (a tightly coupled 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 and shared main memory or the processors themselves (depending on coherence protocols used in the system) is accomplished by the interconnection network. 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 main memory. As mentioned before, use of caches in multiprocessor systems can greatly increase the overall system performance since the majority of references from processors to main memory can be captured by the private caches. Performance evaluations of various multiprocessor systems have been discussed from different points of view in [2, 5, 6, 12, 17, 40, 43, 49, 53, 54, 55].

In the last chapter, we designed a multiprocessor cache memory which can carry out both the processor operations and data coherence operations simultaneously.

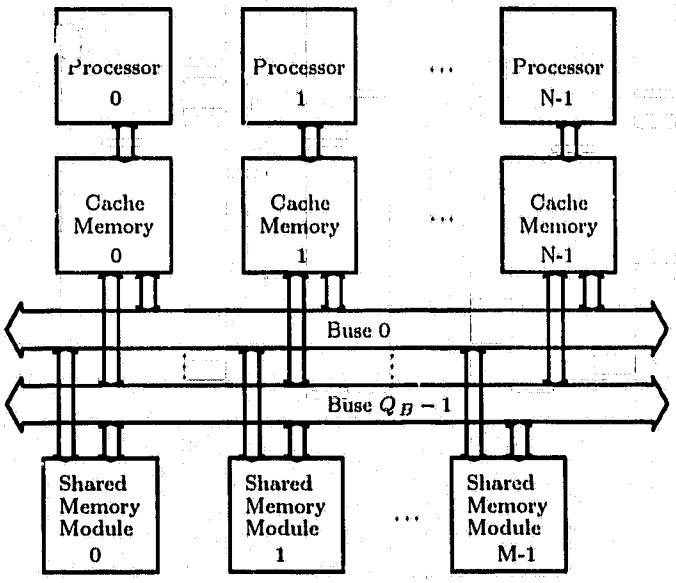


Figure 4.1: A Cache-Based Multiprocessor System with Multiple Buses

In this chapter, we evaluate the performance of the proposed cache memory in a multiprocessor environment. In order to do this, we create two simulation models in which different caches are employed and simulate them for multiprocessor system performance. Finally, based on extensive simulation results, the system performance improvements made by the use of the proposed caches are discussed. Also the effects of the performance of the caches used in the system as a function of the write reference rates, shared rate of data, number of buses and other design parameters for the multiprocessor system are evaluated.

4.1 A Cache-Based Multiprocessor System

Fig. 4.1 illustrates a typical cache-based multiprocessor system with a shared memory, in which each processor has an attached cache memory and a shared main memory is connected to the caches via a multi-bus system. The shared main mem-

ory is partitioned into modules which can be accessed individually by processors and transfer a requested line by interleaving. Multiple buses are used as the system interconnection network to bridge the communication between the processors and the main memory. We choose a bus system as the interconnection network in our simulations since it is simple to implement, popular in multiprocessors, and commercially available, though it is only suitable for a system which pools a small number of processors. Use of the dual-port directory caches in other systems with an interconnection network rather than a bus system may not have the same effects as to be discussed. For example, use of such caches in a hypercube system may not have much performance improvement over the use of single directory caches since the coherence traffic in such a system is usually less than that in a bus system.

As previously described, each reference from the processor to a memory location is presented to the cache memory. The cache first searches the directory to see if the requested data reside in the cache memory. If so, the data are accessed by the processor immediately without disturbing the main memory. Otherwise, a cache miss occurs, causing transfer of the new line, in which the requested data reside, from the main memory to the cache. Then the data can be referenced by the processor. Before transferring a new line to the cache, if the set where the requested line resides is full (for the n-way associative mapping), some line in that set will be purged to make room for the new one. The line that seems least likely to be used in the near future is predicted by the LRU and is removed from the cache. Since there exist many replacement algorithms, in our simulation models, the *least recently used line replacement* algorithm is employed in the LRU. That is, under this strategy, *the line into 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.

Cache performance can be described with reference to two aspects [1]: access time

Cache Type: Unified Cache	Miss Ratio					
	Line Size (Bytes)					
Cache 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	0.671	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	0.023	0.016
32768	0.124	0.07	0.04	0.024	0.014	0.009

Table 4.1: The Design Target Miss Ratios of Unified Cache

and cache miss rate. The cache access time is 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. Therefore, the effect of design changes on access time is difficult to predict without specifying the circuit technology used. The miss ratio of the cache memory is 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 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.) required by the processor at a cache reference.

Although there are many parameters that affect cache performance, it is well-known that the cache miss ratio is strongly affected by cache size, line size, and way

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 4.2: The Relevant Cache-mapping-type Ratio

size. After extensive simulation, Smith [25] presented practical values for the miss ratio as a function of cache size and line size for uniprocessor systems, called the *Design Target Miss Ratios* (DTMR). They are proposed for unified caches, instruction caches, and data caches, respectively. A unified cache is one in which both data and instructions coexist. Also a cache system can be split into two separate caches: one for data and the other for instructions, called a data cache and an instruction cache respectively. Table 4.1 shows only the *Design Target Miss Ratios* for unified caches since the unified caches are used in our simulations. The DTMR in Table 4.1 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. The cache miss ratio is also related to the mapping methods used. Values in Table 4.2 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 [48]. 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 4.1 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 4.1 by the corresponding relevant cache-mapping-type ratio from column three labeled *Ratio of Miss Rate to Full Associative* of Table 4.2. In our simulations, construction of the reference streams for each processor is based on these parameters given in these tables to obtain the expected miss ratios, which will be discussed later.

Information in memory can be classified as *shared* and *private* as well as *readable* and *writable* [65, 12]. The data are defined as *shared*, including readable or writable variables, if they can reside in more than one cache at the same time, while the term *private 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. Private read/write data, meaning the data can only be read or written by one processor at any time.

4. Private only-read data are defined as those which can only reside in one cache at any time.(In our simulation, this type of data is not considered).

Although use of multiple private 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. There are two kinds of data incoherence [2]:

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

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

Both cases exist for the shared data, and *Case 1* can happen to the private data while *Case 2* does not. For private data, since they cannot exist in more than one cache at a time, the modified write-back policy is used in this system: whenever a *modified* private line is selected by the LRU to be purged from a cache due to a line miss, it is written-back to the main memory for data consistency before transfer of a new line to its positions in the cache. Otherwise, no write-back actions are taken.

For shared data, to eliminate the first case, a *buffered write-through* policy is chosen in this system to keep the shared 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 [36]: 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 shared 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 system with a shared main memory. It is called the *buffered* write-through policy because all the write requests from caches are immediately stored in the buffers in main memory modules.

In the second case, an *updating* algorithm is employed rather than *invalidation*. That is, whenever there is a write request to the associated cache from a processor, this request will be broadcast to all the caches in the system through the interconnection network. All the caches search immediately 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 of this scheme 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 if copies of the data to be updated do not reside in most of the caches.

For shared data 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 values of the data. The *write-through with updating* are 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. This policy is more appropriate for the case where many shared data are to be processed concurrently among processors (a number of caches are sharing the same data). When a processor rewrites shared data in its cache, copies of the data in all other caches are updated immediately so that other associated processors need not transfer the updated data from the main memory when they have to use the updated copies.

For the *write-through with updating* protocols, it is assumed 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 degree of 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. For private data, the *write back* protocols are employed.

Since all information exchanges in a multiprocessor system go through the interconnection network, 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 applications, [67]. In our models, the *Bus-Oriented* network (the multiple system buses) is employed.

In the system, the main memory can be divided into modules which are connected to the system buses. It is assumed that the shared main memory of the system under consideration is partitioned into M modules as shown in Fig. 4.1. The data bandwidth of each module is 32 bits, consistent with that of the buses. The memory is organized in such a way that M words of a line are stored in M modules, respectively. That is, the first word of a given line resides in the module 0 while the second word of the line 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 given 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, a cache only needs a short time to send a write request (including both the data and the corresponding address) to a 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 targeted by the write operation checks its 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 $M-1$ words from respective modules by interleaving. Meanwhile, the other $M-1$ modules serve the transfer request as they do for a transfer request caused by a read operation. When the line miss is caused by 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 M words from different modules are sent on the corresponding bus by interleaving. This can be done by the bus controller.

Note that there are no read operations on the buses because of the use of private

caches. Therefore, no *read* — *read* memory competition exists in this system with a single system bus. For a multi-bus system, for each memory module, there is a module interface connected to the multiple buses. In the interface, a buffer is used for all the miss requests from the multiple buses to the memory module at the same time. If there is more than one request to the same module at any time, the requests are again served by the first-come first-serve policy. A write miss request is served as described above. Furthermore, if more than one read miss request accesses to the same location at a time, the memory module controller responds only one read request and the information in the location is sent to satisfy all the requests. Therefore, the *read* — *read* memory competition caused by multiple buses is handled.

4.2 The Simulation Model

In order to study the efficiency of the dual-port directory caches in a multiprocessor environment, we create two simulation models, based on the architecture shown in Fig. 4.1. One model embeds single directory caches and the other employs dual-port directory caches. As discussed in the previous chapter, operations of the two-directory caches are almost the same as that of the dual-port directory caches. Performance of the two-directory caches is very close to that obtained by the dual-port directory caches. Therefore, performance of the two-directory caches is not discussed. We also study the efficiency of such a dual-port directory cache-based multiprocessor system with shared memory That is, we would like to determine what effects on system performance the various cache parameters and write references have, how many processors may be pooled in the system without reaching saturation of this system, how efficient are multiple buses, as the interconnection network, to a multiprocessor system, and what effects of shared data on the system

performance.

The simulation models are developed using a simulation queueing network model, as shown in Fig. 4.2. They are driven by synthetic reference streams rather than by actual traces of a multiprocessor system because no such multiprocessor traces exist, [49]. Such traces would be created artificially, which have no difference with the one we use. In simulation models, written in C, all the processors are identical and acting independently. Each processor has an associated private cache. The shared main memory is partitioned into modules so that a line can be transferred from the main memory to a cache by interleaving. The number of the memory modules is equal to that of words in a line, assuming that each module provides a word each time to the system bus. Multiple buses are employed as the interconnection network between multiple caches and the shared memory in order to evaluate the efficiency of multiple buses. Note that there are no buffers either between processors and their private caches or between the caches and the system bus. Therefore, those caches waiting for use of the system bus must halt until they are released after service, and in turn the processors have to wait until their private caches are released. This lack of buffering makes it easier to observe the performance difference of the different caches. Since the intention of the simulations is to study the efficiency of the proposed caches, all other system parameters for each simulation run such as the main memory cycle time, the number of processors, *etc.*, are identical. Thus, the workload is the same: for each simulation run, the reference streams generated by processors are identical for both models. The basic simulation time unit is a cache cycle, assuming that the speed of the cache is compatible to that of the processor, for each cache cycle a processor can send a reference to its cache and the cache can respond to the reference.

The models consist of a process for each processor, a process for each cache (different caches in the models), and a process for each bus. Each processor generates

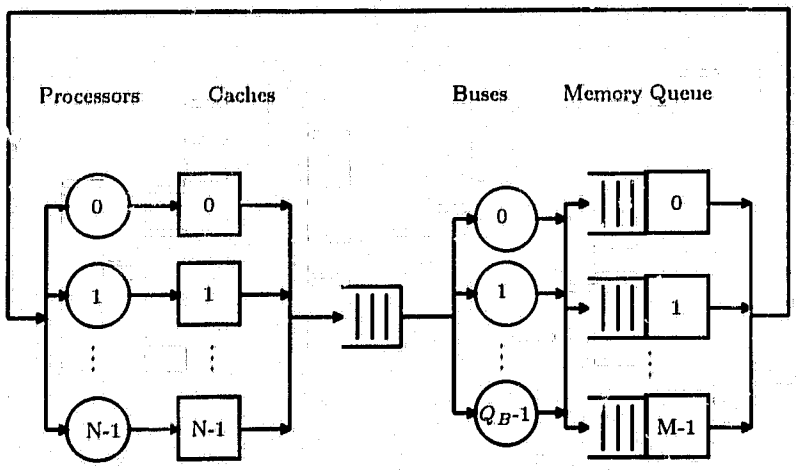


Figure 4.2: Multiprocessor System Queueing Model Diagram

a memory reference sequence to the associated cache and waits for the associated cache's response within a cache cycle. Each cache services each request from its processor within a cycle, either carrying out the request if the requested data reside in it or generating a bus request to the bus queue and waiting for bus service. Each bus picks a request from the bus service queue if it is free. The bus queue controller receives service requests from all caches and allocates them to available bus processes in first-in first-out order in the simulation model. Note that we do not consider the conflicts between multiple buses in the simulations, that is, two bus requests to the same memory address may be serviced by different buses at the same time, since it is assumed that such conflicts can be handled by operating systems and compilers. Each cache, either a dual-port directory cache or a single directory cache, is implemented as has been described. The cache parameters are changed in simulation to evaluate their effects on system performance.

4.3 Simulation Control

For each processing subsystem, including a processor and its associated cache, 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 buses, and spends two cycles to update all the caches and the main memory via a specified bus as soon as the update request is acknowledged by the bus controller. It is assumed that there is a large enough 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. Consequently, each buffer queue has at least $Q_B \times 4$ buffers, where Q_B is the number of buses in the system, since it is assumed that each main memory access requires 4 cache cycles. Otherwise, a miss occurs. In this case, the cache needs to insert the transfer request into the service queue of the system buses and waits, and once a given bus serves the transfer request, the bus takes $R + (M - 1)$ cache cycles to transfer a requested line into the cache, where R is the number of cache cycles that a main memory module needs to respond for an access request from a cache and M is equal to $line\_size/bus\_width$, the times (or cache cycles) the bus has to be accessed for transfer of a line by interleaving. Memory interleaving means that whenever there is a read request to the main memory, the request line address is sent to all memory modules so that each module will send one different word in the request line onto the bus in a such way that the first module sends the first word in the line to the bus, say, after $R-1$ cache cycles (the memory module read response time); and each following module will send a word to the bus one cache cycle later than the previous module. The order of modules is determined when the system is constructed.

When the cache receives an update request from the bus, if the requested data

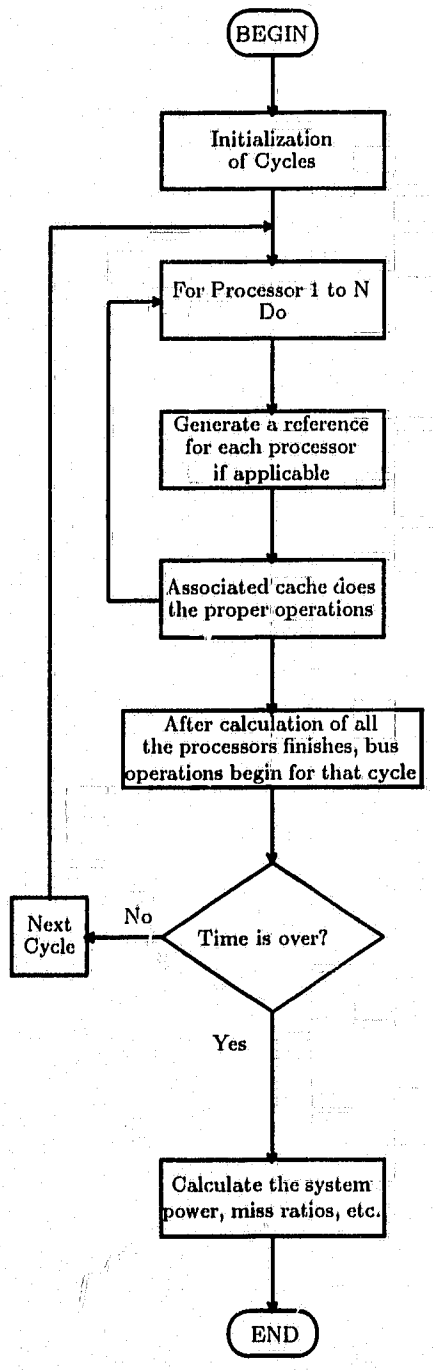


Figure 4.3: The Simulation Control for N Processors

are found, two cache cycles are required for a cache to search the directory and update the requested data in the cache data memory. 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. For a request in the queue, there are four items: *Cache Number*, *Write/Miss*, *Address*, and *Cache Cycles*. Cache Number indicates the request is sent by the corresponding cache while Address is the reference's address. Cache Cycles means the time of the system bus that is required to service a request, as discussed above. Write/Miss indicates the operation types of the bus requests. Requests are one of four types: read miss, write miss, write-back of a "dirty" private line, and a request for a broadcast of a new value of a word, dealing with write hits on shared lines. In the models, it is assumed that there is no delay when the bus services a request. A bus is held during the entire time of each bus transaction, including two cache cycles for update requests, $R+(M-1)$ cycles for transfer of a entire line, and $R+2M-1$ cycles during a read miss with a write back.

The simulation control is shown in Fig. 4.3. For each cache cycle, each of N processors produces a memory reference address. Each associated cache operates in terms of read operation, write operation, update operation, line miss, or wait for bus. After all operations of N processors and their associated caches are completed for that cycle, the buses operate for that cycle. If the simulation time for N processors is over, the miss ratio, system power, etc. are calculated. Furthermore, after all the simulation results are completed, a plot is created showing the simulation results.

4.4 Workload Model and System Parameters

Table 4.3 shows the parameters and ranges used in the simulations. The initial memory references are generated at random as the starting points of programs executed

Parameters	Range
shd	10—100 %
wrt	10—30 %
miss	1.0—15 %
Main Memory time	4 Cache Cycles
Line Size	4—128 Bytes
Cache Size	2—128 KBytes
Way Size	1—16
Number of Processors	1—15
Bus Number	1—3
Bus Width	4 Bytes
Simulation Time	450,000 Cycles

Table 4.3: Summary of System Parameters and Ranges

in multiple processors. In order to simulate locality of references, a line number is repeatly generated by a processor several times (hit) before a new line number is generated (miss). The maximum number of repeats of a line depends on the cache size, line size as well as way size, based on the Target Miss Ratios in Table 4.1. However, the number of repeats for each new line is generated at random, ranging from 1 to the maximum number of repeats. Therefore, for an individual processor, the miss ratio generated is close to those indicated in Table 4.1 which are from realistic traces of single processors.

The reference stream of each processor is viewed as the merging of two reference streams: one is the program reference stream and the other is the data reference stream. Here we assume that program codes and data are stored in separate areas in the main memory. All the program codes are shared by the processors, that is, all the memory lines for the program codes are *shared lines*; and they can not be updated. The data are either ones shared among the processors (shared data) or ones which are used by at most one processor at any time (private data). Each time

a memory reference is called for, the processor generates a reference to a shared line with probability *shd* and to a private line with probability *1-shd*. In our models, all lines for data in main memory are allocated to either shared lines or private lines using a uniform distribution random generation of data line numbers. Furthermore, memory reference streams for data in the system are produced with a specified read/write operation ratio. Write operations are produced at random with a given write rate *wrt*. The write rates showed in the simulation results are the overall write rates for all references including program codes and data. The percentage of references to shared lines indicated in the simulation results means the amount of actual sharing divided by the total references, including references for both program codes and data. If a cache miss occurs, the cache selects a line in a given set in terms of the least recently used line replacement algorithm described above. If the line chosen is a private line which is modified, determined by the local state of the line, it needs to be written back to the main memory before transfer of a new line into the cache. Otherwise, no written-back actions are taken before a new line is transferred. If it is a shared line, according to broadcast with update protocols, it need not be written back since all the copies in the main memory are valid. After transfer of a new line for a cache line miss is complete, the tag and the corresponding local state is changed properly for that new line.

As shown in Table 4.3, the main memory access cycle time (or read response time R) is four cache cycles. The time required to transfer a missing line is based on this assumption; 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 following word ($M-1$ cycles). It is also assumed that a requested line is sent in the same order, regardless of which word is referenced; and the requesting cache does not proceed until the entire line is loaded. The line size varies from 4 to 128 bytes.

Each bus has a width of 32 bits, or a word, which is the unit of data that can be transferred on the bus in a single cache cycle. The cache size varies from 2K to 128K bytes. The way size varies from 1 to 16. The total number of cache cycles for each simulation is 450,000. The number of processing elements in the system, including processors and their associated caches, varies from 1 to 15.

4.5 Simulation Results

Figures 4.4 to 4.14, selected as representative of our extensive simulations, summarize the simulation results for performance evaluation of the multiprocessor system with the proposed caches as private caches. Simulation outputs in figures include bus utilization figures and other parameters under which the simulations are run. The performance of the cache-based multiprocessor system is measured by the *system power*. The system power is defined as total sum of the processor utilization in the multiprocessor system, and processor utilization is measured by the ratio of time spent doing useful work in a processor to the total runing time, [49]. In each figure, the parameter *Cycles* gives the total cache cycles during simulation, assuming that during each cycle a cache executes a reference, while the *Miss Ratio* indicates the overall cache miss ratio of the system. Also the capacity, line size, way size, as well as set size of caches are shown in the figures. Both the number of lines in a cache shared by other caches and the shared rate to the total cache lines are indicated by *Share Line* and % in the figures, respectively. *Bus Number* indicates the system buses in the system. Note that all the parameters in the two models are the same, and also the reference streams from processors in the two models are identical. Thus, performance comparison of these models reflects the performance difference between the two different caches in the models.

The figures can be divided into three areas: *area I*, *II*, and *III* in terms of degree

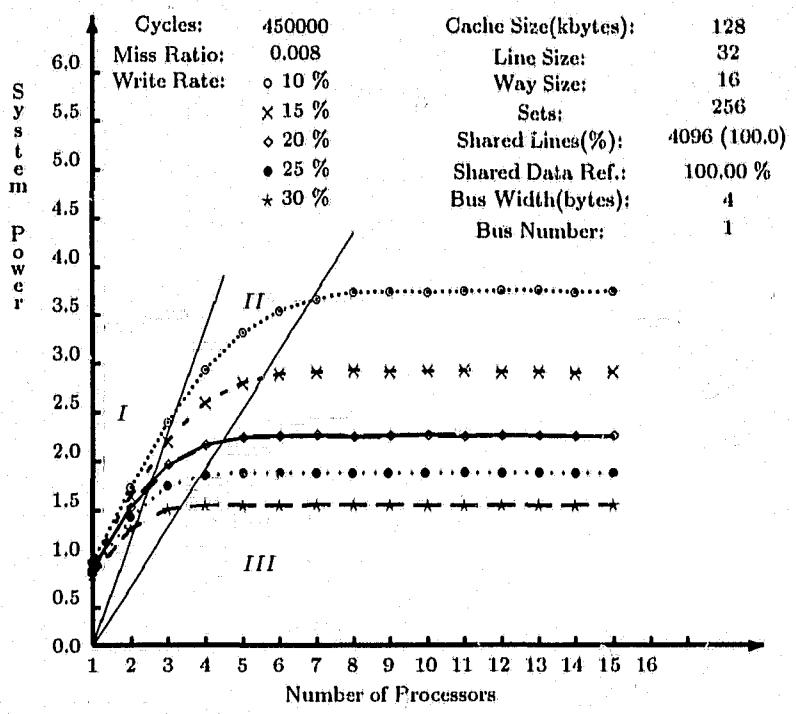


Figure 4.4: Typical Simulations of the Dual-Port-Directory Cache System

of the bus saturation. Fig. 4.4 gives a typical figure as an example. *area I* is the linear area in which the bus is not saturated. In this area, the system power increases linearly with the number of processors used in the system. *area II* is the pre-saturation area where the system power increases nonlinearly with the number of processors used. *area III* is the saturation area in which the system power shows virtually no increase with increasing number of processors (The system power levels out). In this area, the bus utilization is approximately 100 %.

4.6 Performance Evaluation

Fig. 4.5 shows the simulations for both the systems with the single directory caches and the dual-port directory caches, in which all lines are shared (a shared

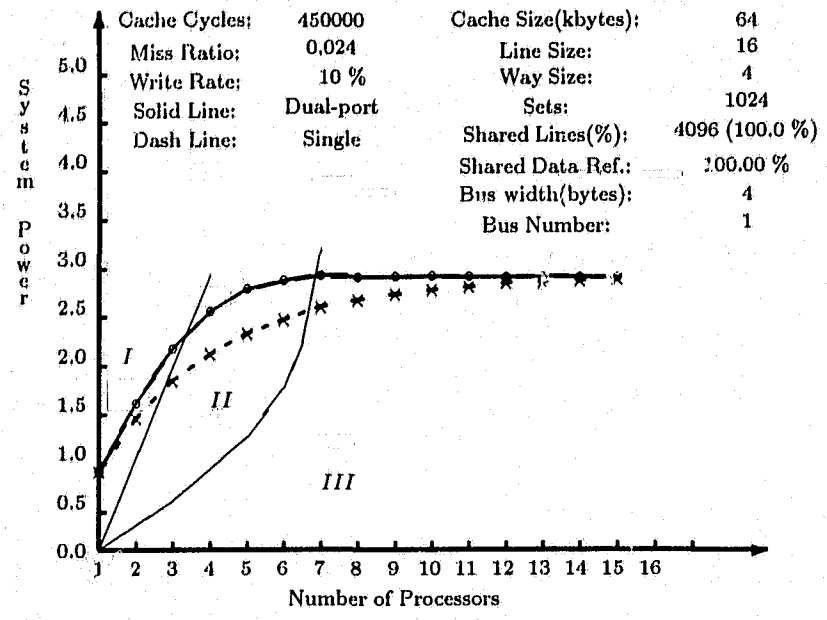


Figure 4.5: Comparison of Dual-Port-Directory Cache with Single-Directory Cache

rate of 100%). The dashed line is the simulation result for the single directory caches while the solid line is the result for the dual-port directory caches. When there is one processor in the each system, the system powers are identical because there is no data coherence interference. Comparison of the two curves indicates that use of the dual-port directory caches increases the system power rapidly in *area I* and *area II*. The maximum number of processors pooled in the system with the dual-port directory caches is larger than that with single directory caches in *area I*. The dual-port directory cache system reaches the bus saturation quickly in *area II* (The dashed curve in Fig. 4.5 is sharply changed). In *area III*, the solid curve levels out horizontally. The system performance hardly increases no matter how many more processors are added to the system since, during saturation, all the processors wait for use of the bus, if they are idle, not because of coherence operations but because of line missing operations of their associated caches. Because the dual-port directory caches increase utilization of each processor, the system performance as a function

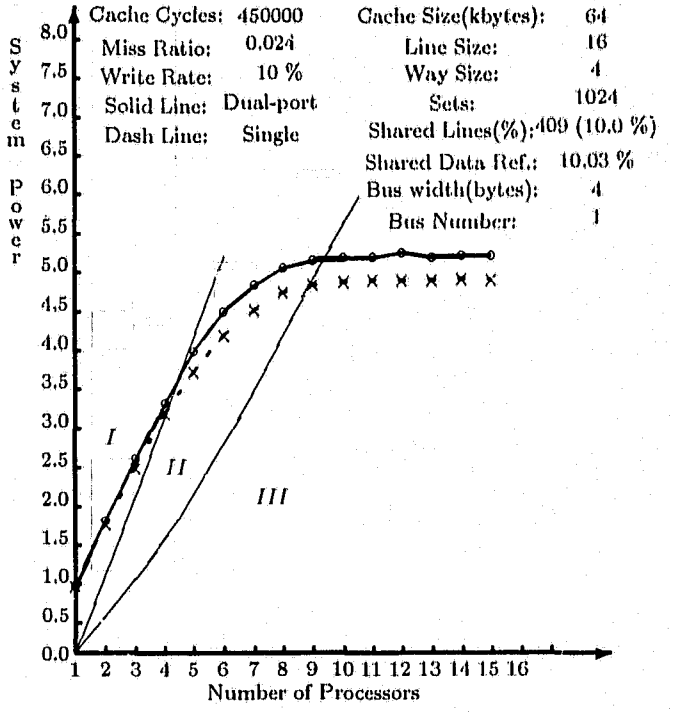


Figure 4.6: Simulation of a System with 10 Percent of Shared Lines

of the number of processors increases rapidly before bus saturation. Even after bus saturation, utilization of each processor is still higher than that in a system where single directory caches are employed because of elimination of processor waiting time for data coherence. As a result, the system power, for a given number of processors used in a dual-port directory cache system, is higher than that for a single directory cache system. For the single directory cache system, during bus saturation (in *area III*), the system performance still increases since not all the processors wait for use of the bus if they are idle. Some processors are idle because of waiting for data coherence operations. Thus the system power obtained by the single directory cache system increases smoothly, especially in *area II*. It implies that use of the dual-port directory caches makes the interconnection network bottleneck more obvious and serious since utilization of each processor increases because of higher cache performance.

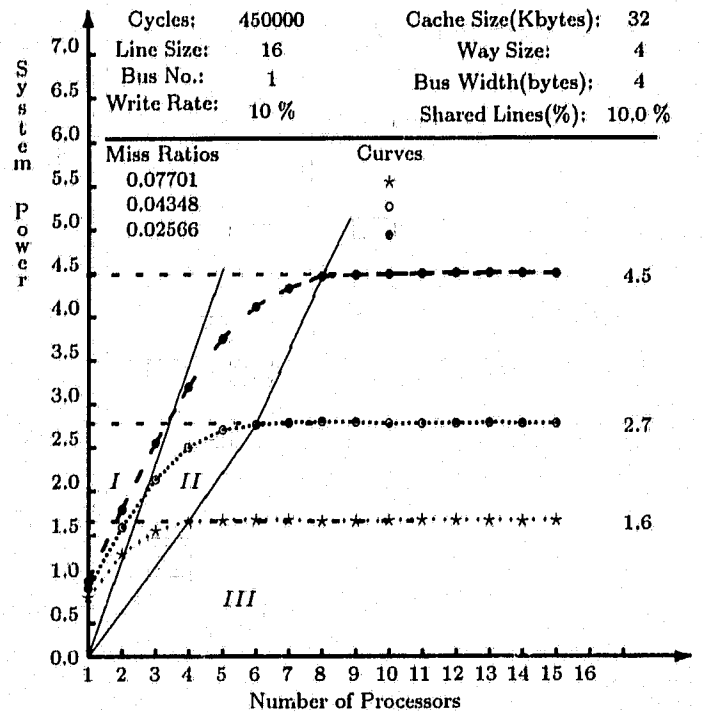


Figure 4.7: Simulation for Miss Ratios

Fig. 4.6 shows the simulations with a shared data rate of 10 percent. Compared to that in Fig. 4.5, since write operations on private data do not produce bus requests for data coherence, with decrease of shared lines, the write operation requests on the system bus decrease; and in turn, the system power increases for both the systems with the dual-port directory caches and single directory caches respectively. In *Area I*, the system powers obtained by both systems almost have no difference. However, in *Area II* and *Area III*, the system power of dual-port directory cache system is higher than that of the single directory cache system.

Fig. 4.7 shows the simulations of different miss ratios obtained by different reference streams. all the parameters used in the simulations, such as the cache size, line size, way size, and number of buses, are the same as 32 Kbytes, 16 bytes, 4 ways, and a single system bus, respectively. There are three curves with miss ratios of 0.077, 0.043, and 0.026. Comparison of these simulation results (curves) indicates that a decrease in the overall miss ratio of the caches increases the system power.

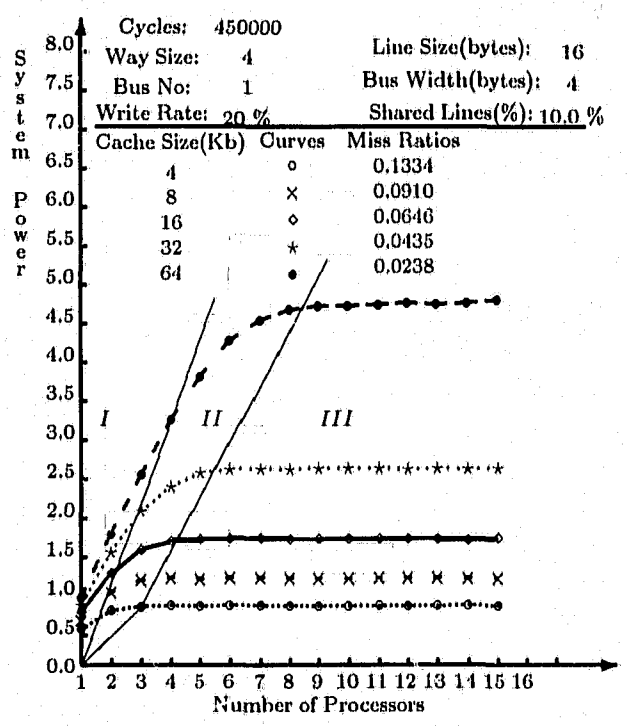


Figure 4.8: Simulation for Systems with Caches of Different Cache Sizes

Figure 4.8 gives simulation results for different cache sizes. System parameters, except the cache sizes, and the reference streams in the simulations are the same. In the figure, there are five curves representing simulation results with five different cache sizes of the dual-port directory caches in the multiprocessors. The figure indicates that the overall miss ratios are 0.133, 0.091, 0.064, 0.043, and 0.024 while accordingly the cache sizes are 4, 8, 16, 32 and 64 Kbytes. The simulations show that cache size affects the overall cache miss ratios (that is, increasing the cache size causes a decline of the overall cache miss ratio of the system), and the system power in turn increases because the miss ratio decreases.

Figure 4.9 shows simulation results for different line sizes. Similarly, all the system parameters, except the line size in the dual-port directory caches, and the reference streams are the same in the simulations. The line size of the caches varies from 4 to 64 bytes as power of 2. The corresponding miss ratios are 0.133, 0.078,

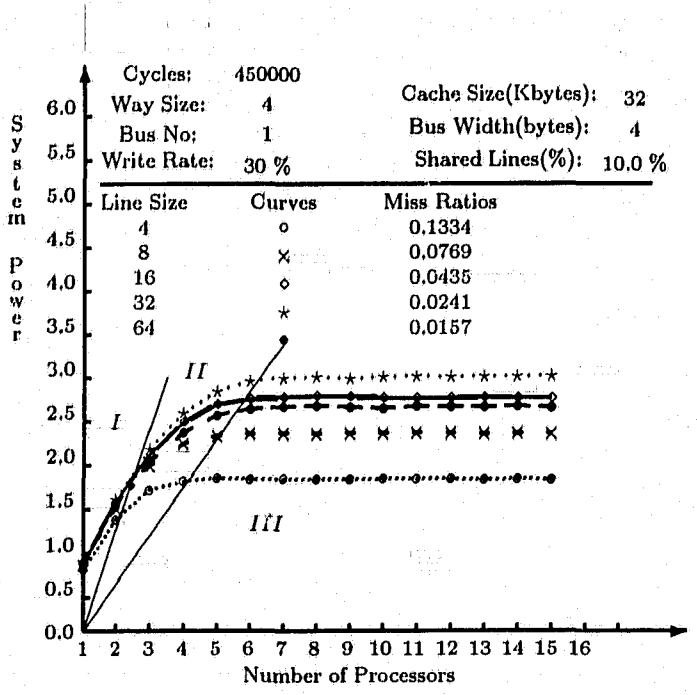


Figure 4.9: Simulation for Systems with Caches of Different Line Sizes

0.043, 0.024, and 0.016. As indicated in the figure, the miss ratios decrease with an increase of line size of the caches. Therefore, the system power should increase while the overall miss ratio decreases because of the larger cache line size. However, as shown in the simulations, the system does not behave in precisely that way. Comparison of the curve indicated with dark dots, with others in the figures indicates that even though the cache line size of 64 bytes exhibits the lowest overall miss ratio, the corresponding system power is smaller than that of a system with caches where the line size is 16 bytes. The reason is that although a cache with a larger line size has a lower miss ratio, the larger line size causes the bus to spend more cycles transferring the lines from the main memory to the caches during line misses, which makes the corresponding cache wait. Therefore, in this case, the bus limits the system performance. Hence, cache line size should be chosen optimally and a trade-off has to be made between cache line size and bus width during cache design. In this example that the bus width is 4 bytes, the optimal line size of caches is 32

bytes.

Figures 4.10 and 4.11 give simulation results for different way sizes in caches. The only difference in the simulations is the way number of the caches used in the model. In each figure, there are also five curves obtained from the simulations with the cache way size varying from 1 to 16 (Note that the way size = 1 means that the directly mapped method is applied in the caches). In Fig. 4.10 where the cache size is 32 Kbytes, the overall miss ratios are 0.065, 0.051, 0.046, 0.044, and 0.043, corresponding to way size of 1, 2, 4, 8, and 16. Although the miss ratios decrease when the way sizes of the cache increase, the decrease in the overall miss ratios seems not to have as large an effect as do changes in the cache size and line size. In particular, when the way size is larger than 4, the miss ratio increases a little. In Fig. 4.11 where the cache size is 128 Kbytes, The system with caches of way size of 1, 2, 4, 8, and 16 has the overall miss ratios of 0.00805, 0.00780, 0.00779, 0.00779, and 0.00781. The change of way size hardly decreases the overall miss ratio. Thus, the system power gains a little from the change of cache way size. The curves in the figure are close to each other. Therefore, the system power is not affected much by the way size. Consequently, the way size of caches does not appear to be important for large caches, and for very large caches, direct mapping may be adequate, which is consistent with studies for uniprocessor systems in [26].

Although use of the private caches in this multiprocessor structure can greatly reduce references to the slow main memory, the bus system seems to be a bottleneck in the multiprocessor since write operations and transfer of miss lines under the *write through* protocols still make the bus busy. The simulations, as shown in Fig. 4.12, indicate the effects of the bus system on performance of the multiprocessor system with the dual-port directory caches. The number of buses used in the simulations is printed out in the figure. The results show that the system power increases with the number of buses. Especially, when the bus utilization reaches saturation (*area III*),

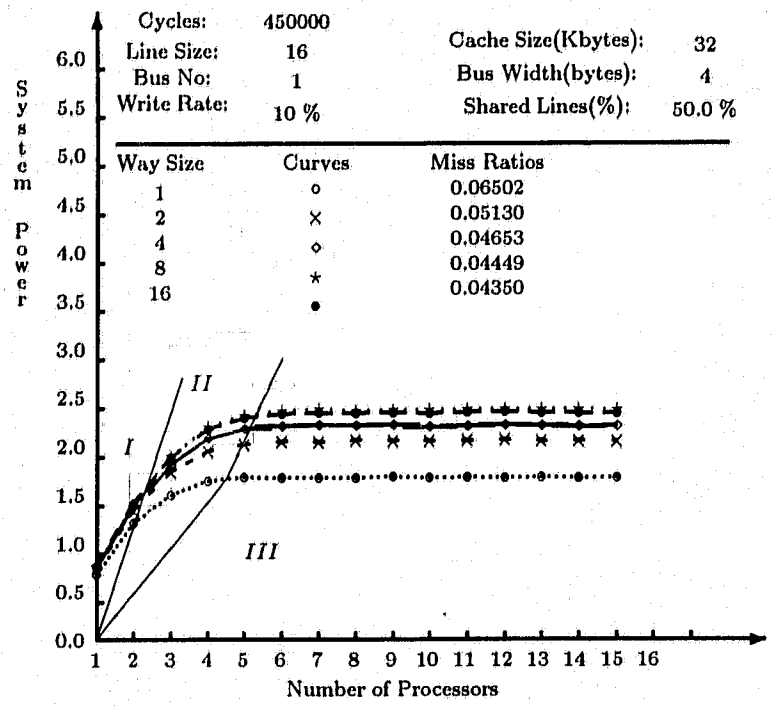


Figure 4.10: Simulation for Systems with Caches of Different Way Sizes

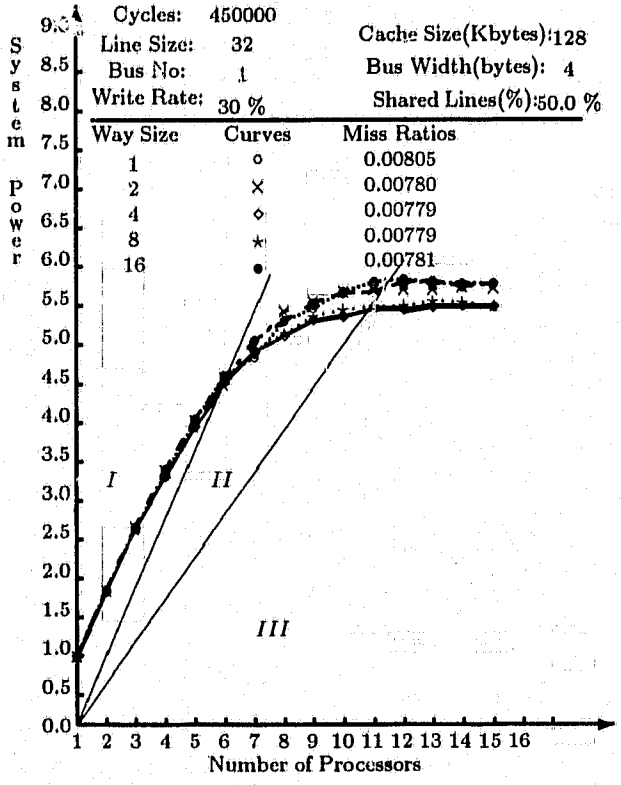


Figure 4.11: Simulation for Systems with Caches of Different Way Sizes

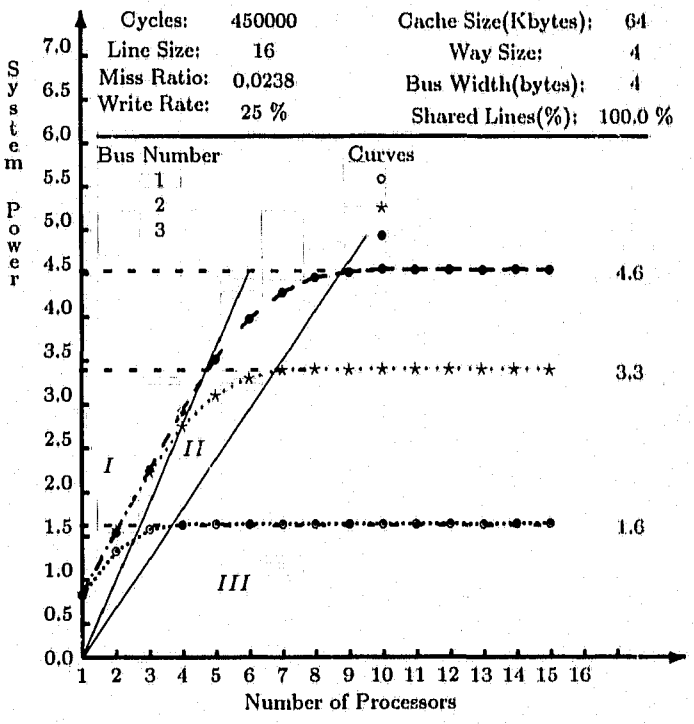


Figure 4.12: Simulation for Systems with Multiple Buses

the system power increases almost linearly with an increase in number of busses. For instance, the system power is almost doubled (from 1.6 to 3.3) with a change of the number of buses from 1 to 2 when the system has 7 processors or more. Similarly, when the bus number increases from 1 to 3, the system power of the tri-bus system is as nearly three times as that of the uni-bus system. Thus, use of multiple busses would significantly increase performance of a multiprocessor system with the dual-port directory caches because the waiting time of each cache for use of the system bus would certainly be decreased.

Fig. 4.13 gives the results, for dual-port directory caches, in which there are five curves, each of which indicates a simulation with write ratios (*Write Rate* in the figures) varying from 10 to 30 percent for memory references. In the simulations, all the lines are shared by processors. In the figure, it is obvious that the system power

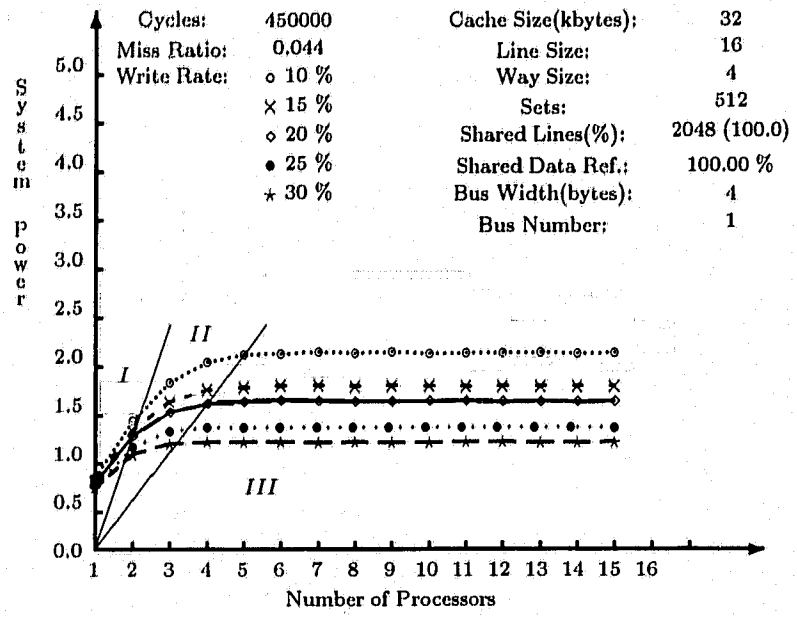


Figure 4.13: Simulation for Systems with Different Write Rates

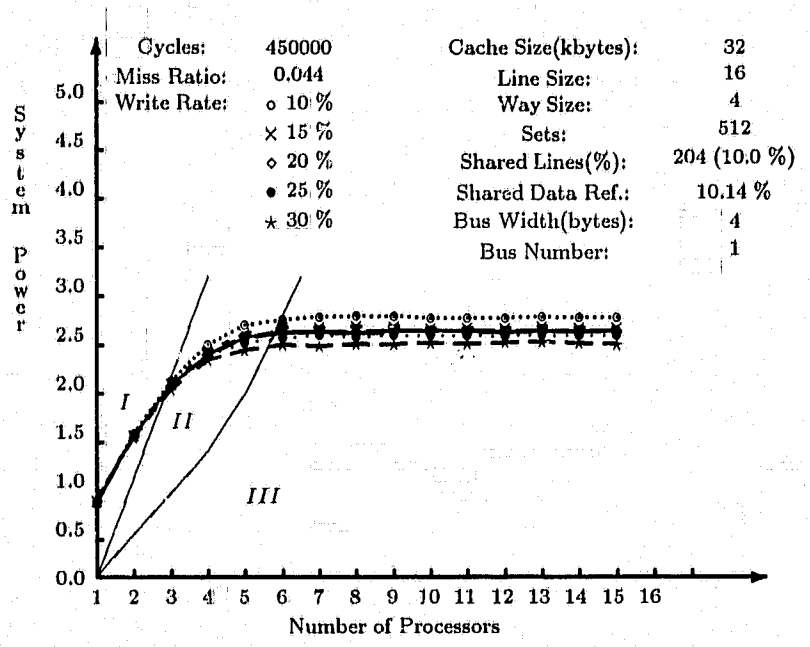


Figure 4.14: Simulation for Systems with Different Write Rates

becomes higher with a decrease of the write ratios since each write operation requires that the other caches are updated. The minimum number of processors at which the system bus reaches saturation increases as the write ratio decreases. For example, in Fig. 4.13, the curve with a write ratio of 10% indicates the minimum number of processors pooled in the system during bus saturation (*Area III*) is 5 and the system power is about 2.2 whereas the curve with a write ratio of 30% shows the minimum number of processors is 3 with system power of about 1.2. This is expected, since each write operation requires use of the bus. It implies that decreasing write ratios reduces use of the system bus for each processor, when the *write-through* protocols are employed, and in turn increases system performance. The reasons that the system power, as shown in the figure, seems slightly low are the lack of buffers between the processor and the caches as well as between the caches and the bus in the models and sharing of all lines by the processors in the system. Moreover, simulations in Fig. 4.14 indicate that, when sharing of data decreases, the impact of write operations in the system decreases considerably so that the curves are close to each other. Especially, the curves in *area I* are almost overlapping. The reason is that the dual-port directory caches can deal with most update operations caused by write references without delaying their normal operations when the bus does not become a bottleneck. Therefore, the write rate has almost no effect on the system power whenever the bus does not saturate.

Use of more efficient protocols such as the *berkeley protocols* decreases coherence operations, and in turn reduces network traffic. Performance difference between the two systems with single caches and dual-port directory caches may be smaller than that shown in Fig. 4.5. Usually the "snoopy" protocols can benefit more from this design since these protocols require caches in a system to watch transactions on the bus and take actions, if necessary.

The simulation results show that, in general, the system power increases mono-

tonically with the number of processors until eventually a saturation point is reached, caused by bus saturation. Moreover, as the system power increases, the number of processors, pooled in the multiprocessor system, required to produce bus saturation increases. It is also seen that the system power increases almost linearly before the bus utilization becomes saturated (before *Area II* is reached). That means that before the bus system reaches saturation, improvement of system performance is roughly proportional to the number of processors in the system.

4.7 Summary

A cache-based multiprocessor system with a shared memory and multi-bus has been described. Strategies and structures for the cache memory, shared main memory, and multiple buses used in simulation have been discussed. In the proposed cache, an n-way set-associative mapping is employed for address translation while the least recently used line replacement algorithm is used for line replacement after a cache miss. The write-through with updating algorithm is employed to keep the shared information in the multiprocessor system coherent while the write-back policy is used for private data. The simulation model and its data structure were presented; and the structure of the simulator and workload were described. The system performance is simulated. During simulation, the processors generate the memory reference streams at random with randomly-generation for write rates and program locality. The simulation time unit was based on the cache cycle, and it is assumed that speed of the cache memory matches that of processors. Based on extensive simulation results, the performance improvements made by the use of the proposed caches are discussed. Furthermore, we investigate the effects of the write reference rate, data sharing rate, multiple buses, as well as cache parameters such as the cache size, line size and way size on the multiprocessor system performance.

The simulation results show that the multiprocessor system with dual-port directory caches has higher performance than that obtained by a system with single directory caches. Sharing of data affects greatly the system power for both the dual-port directory cache system and the single directory cache system, though the system power obtained by the former is still higher than that by the later. Since each write request for shared data updates the main memory and other caches via the bus system in terms of the buffered write-through policy, the workload of the bus system increases while the overall write rates becomes larger. The impacts of write rates on system performance decrease with a decrease of shared data rates. Simulation results indicate that larger cache size, line size, and way size will decrease system overall cache miss ratios. Although, in general, decreasing the cache miss ratios can increase system performance since it reduces the bus traffic, a lower miss ratio caused by increasing cache line size may not be effective. The reason is that with a larger line size the system bus will spend more time transferring lines from the shared memory to caches, which in turn decreases the system performance. Therefore, cache line size must be selected carefully during cache design to match the bus system so that the bus traffic does not reach the point at which the system performance will decrease. Furthermore, it implies that use of the dual-port directory caches makes the interconnection network bottleneck more serious. Use of multiple buses in the multiprocessor system significantly increases system performance. After bus utilization reaches saturation, the system power is approximately proportional to the number of buses. The results also show that before bus utilization starts to saturate (in *area I*) the system power increases almost linearly with the number of processors in the system. And, as the system power increases, the number of processors in the multiprocessor system, required to produce bus saturation increases. Thus, use of multiple buses and more efficient coherence protocols can greatly reduce bus traffic and improve the system performance.

In order to further improve the system performance, an increase in system power and bus ability can be achieved as follows: First, 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. Second, multiple port memory system would be used to increase performance and the memory competition caused by multiple buses is easier to handle. Each bus can be connected to a port of the memory, which decreases the memory delay time. The major drawback of multiport memory is the cost of such a memory system.

Chapter 5

Fault-Tolerant Design

Since cache memory is being increasingly used in modern systems, the reliability of the cache is of increasing importance. In the previous two chapters, we designed a dual-port directory cache, and evaluated the performance of bus-based multiprocessor systems with the proposed caches. In our evaluation, we can see the cache memory size is one of the most important factors affecting the system performance. With rapid developments of technology, the capacity of cache memory has increased dramatically; and more efficient data coherence protocols have been developed and incorporated into cache memory. Cache memory has become more complicated. The reliability of cache memory with large capacity cannot be ignored.

Although an error detection code (EDC) can easily be employed to protect data against errors in cache data memory, errors occurring in a cache management unit, especially in the address translation mechanism, *e. g.* the directory, are difficult to detect during operation because a cache memory management unit differs from a conventional static random access memory management unit in that:

- a write/read operation is on a tag (a word). That is, all bits in a tag are read or updated at the same time.

- Once addresses are written into the tags in a cache directory they are never read out until they are purged for new requested lines due to cache line misses. Instead they can only be "seen" via the *hit/miss* signals.
- Several tags are checked for a match with a requested memory address at the same time (n-way associativity).
- If the tags in a set are full, there is a replacement algorithm to select a cache line (block) to be replaced with the requested line. Which line is replaced can be predicted but cannot be controlled by the outside world.

Because of the above properties, traditional fault tolerant approaches and on-line concurrent checking techniques used in memory such as EDC's cannot be simply transferred into a cache management unit. Thus, a cache memory management unit is more difficult for testing and error detection.

In the following two chapters, we discuss the fault-tolerance and on-line concurrent checking design in the proposed cache management unit for high reliability during normal operation. The hardware design primarily includes a tag self-purge mechanism for errors in directory tags, a comparator checker for errors on the cache address lines, and a totally self-checking checker for errors on the signals from the LRU ($\overline{LRUSel}$'s). In this chapter, we create a fault model for the cache management unit for both testing and fault-tolerance, including the faults that most frequently occur in MOS technologies. We also discuss the implementations of the tag self-purge mechanism, the comparator checker, and an error flag. Finally, we study the hardware costs for fault-tolerance and on-line concurrent checking. In the next chapter, we concentrate on the design of the required totally self-checking checker for (n-1)-out-of-n codes.

5.1 Faults and Fault Model for the Directory

5.1.1 Physical Faults in CMOS

A system fails to operate correctly because of an error, a part of a system state which is liable to lead to failure. Errors can be caused by faults, or physical defects in circuits.(Errors can also be caused by incorrect design or by environment conditions). These defects occur because of problems during manufacture, or because of physical changes in the manufactured circuit which may be caused by wearout, the environment, etc.. These faults can be classified into permanent faults and temporary faults.

Permanent faults are relatively easy to detect since they permanently exist once they occur. These permanent faults can be:

1. physical faults caused by the manufacturing process
2. problems due to wearout.

In general, these faults can be categorized as follows:

- stuck-on and stuck-open faults in transistors
- shorts to V_{dd}
- shorts to V_{ss} ($Ground$)
- opens or breaks of interconnection wires
- bridges between interconnection wires which should not be connected. In general, there are two types of bridge faults: AND-bridge faults and OR-bridge faults.

Temporary faults are far more difficult to detect since they occur randomly. Typical temporary faults include:

1. intermittent faults due to environmental conditions such as electrical noise
2. transient faults due to electromagnetic interference
3. soft errors caused by the interaction of charged particles with the silicon substrate.

In order to detect the faults in a specified circuit, it is necessary to have a representation for all the faults to be tested in the circuit, *i.e.* a fault model.

5.1.2 Fault Model

As we know, a cache directory mainly contains tags (line slots) to store the logical addresses most likely to be used by the associated processor in the near future and some comparison logic built in each tag. Which faults (physical defects) in each of those parts are the most important depend on the specific layout of the entire directory. For the proposed cache, several types of faults at the function level are considered as follows.

- For the tags, we assume that most of the common faults occurring in *static random access memory* (SRAM) are likely to occur. Thus, the faults in the storage part of the directory are:
  1. single or multiple *stuck-at* faults in individual storage bits in tags.
  2. *stuck-open* faults in individual cells of a tag or a whole tag — the cells or tag are not accessed by any action on the cells. Note that the stuck-at faults in the address decoders that no set is selected by one address can be mapped into this type of faults in the tag array. Similarly, stuck-at-0

faults on word lines *LRUSel* of tags, as shown in Fig. 3.4, can also be mapped into these faults. In the presence of these faults, the faulty tags cannot be written with any addresses.

3. *transition* faults in memory cells — the cells fail to have new values when the complement of their present values are written into the cells. That is, a cell fails to undergo at least one of the transitions 0-to-1 or 1-to-0. Failures in the write control circuitry can be mapped into this class as well as certain cell faults.

4. *coupling* faults between two bits in a tag— bit i is coupled with another bit j in the same tag so that a change of the state (from 0 to 1 or from 1 to 0) in bit j incorrectly changes the state of bit i as well. State coupling is a non-symmetric relation. That is, the state in bit i may not change the state of bit j. Note that the coupling fault that a state transition in bit j incorrectly changes the state of bit i may not occur since all the bits in a tag are updated simultaneously at any time.

5. *pattern-sensitive* faults — the behavior of a bit in a tag is affected by some patterns of neighboring bits in the tag. *E.g.* for bit i in an m-bit tag, $1 < i < m$, its state can only be affected by the patterns of bit *i-1* and bit *i+1*. Meantime, bit 1 is affected by bit 2 and bit m can be affected by bit *m-1*. Since cells in the directory are partitioned into tags and each tag is physically distanced from other tags by some logic circuitry such as comparison logic, etc., we assume that the pattern-sensitive faults only occur within tags.

6. *multiple access* faults — the faults that an access to one memory cell results in access to other unaddressed cells in combination with the addressed cell. In the directory, this kind of fault may only occur between tags since all the cells in a tag have the same operations, such as search-

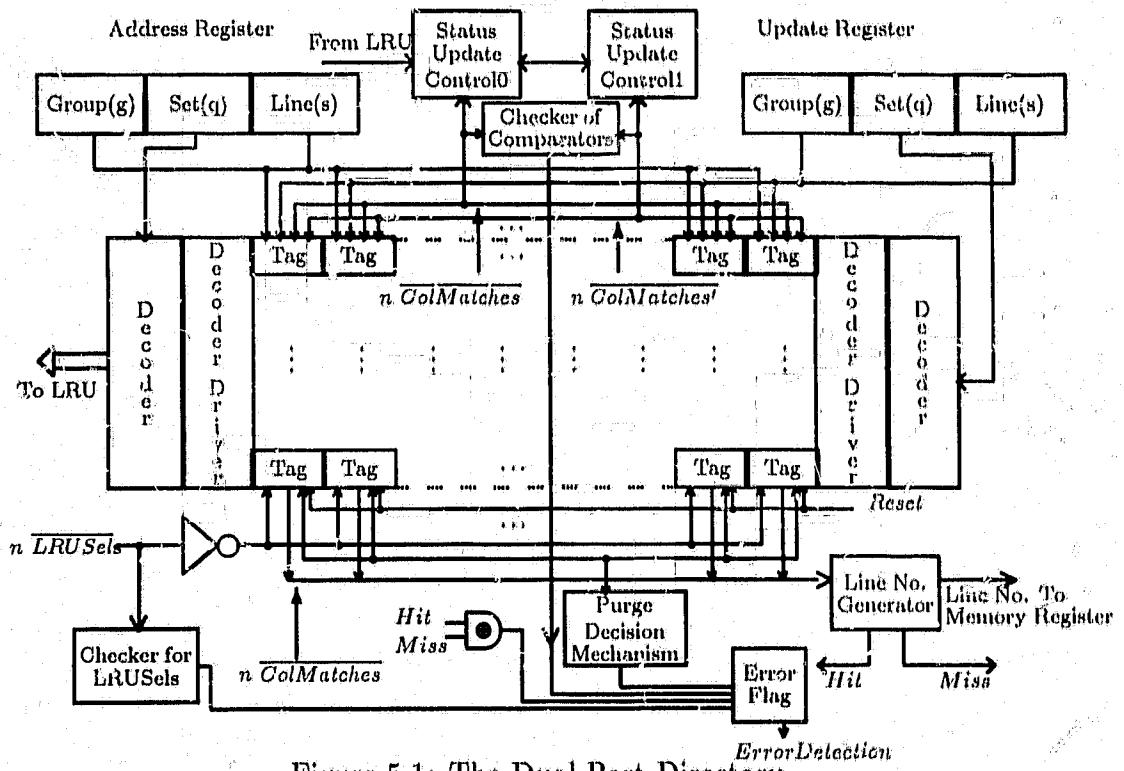


Figure 5.1: The Dual-Port Directory

ing or updating. The multiple access is a non-symmetric relation. Note that faults in the address decoders that more than one set is accessed by one address can be mapped into this type of fault in the tag array.

7. data retention faults — the faults which cause a cell to fail to retain its valid logical value after some units of time.

- For the comparators associated with tags:

1. comparators are not performing correctly because of permanent faults.
2. the signal lines ($\overline{COLMATCH}$es in Fig. 5.1) from the comparators to the line number generator contain permanent faults, including stuck-at and bridge faults.

- For the address register and update register, multiple stuck-at faults are in the bit cells of the address and update registers.
- For address lines connecting the registers with the tag array and with the decoders, permanent faults occur.
- For the lines connecting the Line Replacement Unit with the tag array ($\overline{LRUSel}$s in Fig. 5.1), stuck-at faults and bridge faults may occur. These stuck-at faults cause incorrect write actions to the tags selected by the decoder. That is, a stuck-at-0 fault at a $\overline{LRUSel}$ incorrectly writes the address into the selected tag, and a stuck-at-1 fault makes the selected tag not be updated with an address during a line miss. Since $\overline{LRUSel}$s are encoded in an (n-1)-out-of-n code, at any time, only one of $\overline{LRUSel}$ lines must be low to update a selected tag in a given set during a line miss. On one hand, if the occurring fault is an AND-bridge fault between two $\overline{LRUSel}$ lines and one of these lines is selected by the LRU to update a corresponding tag in the directory during a cache line miss, the other line with a logical 1 is forced to be low. On the other hand, if the fault is an OR-bridge fault, both of the involved lines are high due to the bridge fault.

For fault-tolerance and on-line concurrent checking in the cache management unit, multiple access faults and the data retention faults cannot be detected during normal operation. However, all the faults in the model can be detected by the off-line cache memory testing algorithm which are described in Chapter 7.

5.2 Fault-Tolerance in the Directory

In the directory, as shown in Fig. 5.1, there are a tag purge mechanism and two error checkers designed for fault-tolerance to protect the directory against faults.

The *tag purge mechanism* is used to purge faulted tags with limited interruption of the normal operations. The *comparator checker* is used to detect errors in the two comparators in a tag and errors on both the $\overline{ColMatch}$s and the $\overline{ColMatch}$'s. Note that the faults occurring in the address register or the update register, and on the address paths from the registers to the directory, including the directory decoder, are also detected by the comparator checker. The *LRUSel checker* is designed to detect errors on the $\overline{LRUSel}$s from the LRU to the directory.

5.2.1 Self-Purge of Faulty Tags

As we know, with the advent of VLSI technology, a comparator can be easily built into each tag; only the results of the comparison are required to be output. Thus, once an address from the processor is written into a tag, it does not have to be read out for normal operation. Thus, the well-known fault-tolerant solutions for RAM, such as error correcting codes, can not be simply transferred to a cache directory. Otherwise, the overhead for fault-tolerance is considerable, since extra special hardware is required for fault detection/correction. The cost for the special hardware is high, especially for the full-associative cache which requires that addresses in all the tags have to be read out for fault checking. Furthermore, the time delay for the cache to search its directory increases because of employing error detection codes in the cache management unit, and in turn, the cache cycle time increases as well, which is critical to the cache speed and the cache-based system speed.

In order to achieve the fault-tolerant goal in the directory, a tag purge mechanism is designed to purge faulted tags with limited interruption of normal operations. That is, faulted tags in the directory can logically be "self-purged" from the directory and never be reused during late normal operation. The idea is that whenever a cache line miss occurs during processor operation, the LRU finds a line in the cache unlikely to be used in the near future; and the corresponding tag number is sent to

the directory through a specified $\overline{LRUSel}$ signal. The address in the selected tag is updated with that of the address register. After the address is over-written into the selected tag, the address in the address register is immediately compared with the updated contents in the tag using the corresponding built-in comparator. If any inconsistency occurs, a purge-decision mechanism sets the purge bit of the selected tag as " fault " (one purge bit for each tag) so that the faulted tag is disabled and cannot be used later. Meantime, the LRU reconfigures itself so that the faulty tag number in the given set cannot be chosen again; and it selects the other line number for the new line. Thus, the faulted tag is transparently purged from the directory while the cache continues operations correctly though performance of the cache may be slightly degraded because the useful cache lines for normal operation decrease due to the tag purges. The corresponding memory cells in the purged cache lines can be used for other purposes, depending on how the fault-tolerant cache data memory is designed. For example, these cells can be used as spare cells so that if any faults are detected in cells of the data memory, the faulty cells can be replaced using the spare ones.

In general, detection of a faulty tag and re-selection of a new tag by the LRU for the missing line can be carried out within one cache cycle. However, considering the possibility of the fault occurrences is usually low and this strategy increases the time of cache cycles for normal operation, the cache is designed that the re-selection of a new tag is carried out in the following cache cycle using the *retry* technique. The implementations of the retry technique will be described in section 5.2.4 of this chapter.

5.2.1.1 Fault detection in tags

We recall the operations of a directory tag cell, as shown in Fig. 3.5 in Chapter 3. When the $\overline{RowSel}$ is low, this cell becomes a normal content addressable memory

cell. This cell takes one of the following actions:

1. If the datum on the *BIT* matches the value in the cell, the *match transistor A* remains off so that the *Match* is high for this bit (*match*).

2. Otherwise, transistor A is turned on so that the *Match* is pulled down to *Ground* (*mismatch*).

In a tag, the *Match* lines of all the cells are cascaded together. If any address bits of the tag produce *mismatch*s, the *Match* line of this tag is pulled down to *Ground* via those *mismatch* bits to generate a high $\overline{ColMatch}$ for the tag column where this tag resides (see Fig. 5.1), indicating the this tag does not match the *g+s* from the address register. If all tags in the selected set generate high $\overline{ColMatch}$s, a cache line miss occurs. In this case, if the tag including the cell, as shown in Fig. 3.5 (a), is selected by the LRU through a high *LRUSel*, the datum on the *BIT* is written into this cell. After the write operation, the updated content in this cell is immediately compared with the *BIT*, and the *Match* line is re-evaluated. If the *Match* line remains *mismatch* through the cache cycle, the tag or/and the corresponding comparator must be faulted and the faulty tag is detected.

5.2.1.2 Purge bit and column purge mechanism

In the directory, there is one column purge decision mechanism for each column of tags and one purge bit for each tag. Fig. 5.2 illustrates structures of a purge bit of a tag and a corresponding purge decision mechanism for tag column *i*, whereas Fig. 5.3 gives the timing diagram of the column purge decision mechanism in which there are three operation periods. In the timing diagram, there are arrows, indicating that arrowed signals are affected by the transitions of arrowing signals. Furthermore, there are three cache cycles showing three periods of fault-free, purge-bit setting, and tag-purged operations respectively. Initially, the purge bit is reset by *Reset*

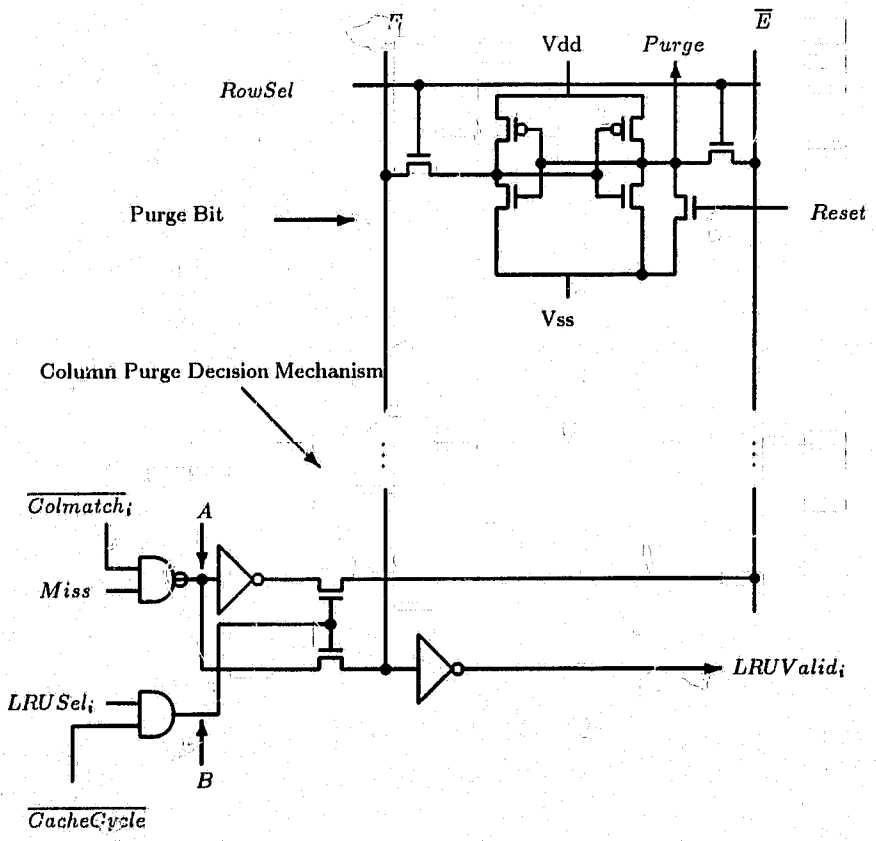


Figure 5.2: The Self-Purge Mechanism for One Column Tags

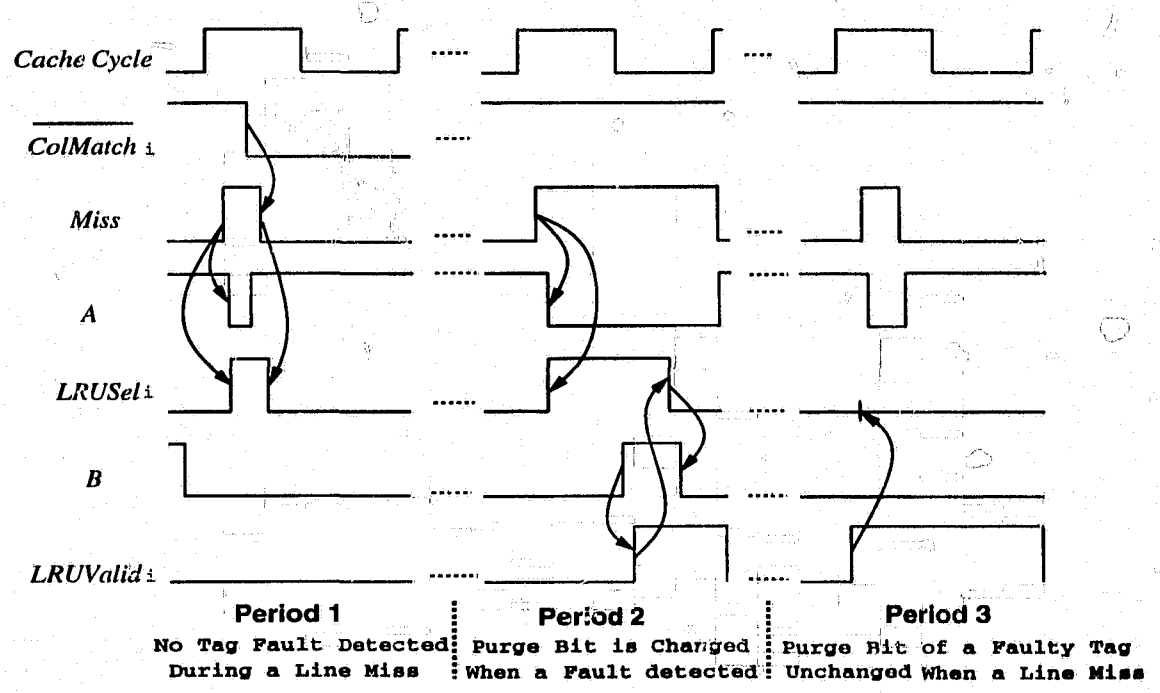


Figure 5.3: The Timing Diagram of the Tag Column Purge Mechanism

so that the signal *Purge* is low. The status bits of the corresponding tag cannot be affected by the purge bit. During *period 1*, when the *Miss* is high (a line miss occurs), the $LRUSel_i$, the inverted signal of $\overline{LRUSel_i}$ from the LRU, is valid to update the address in the tag with that in the address register. Normally, after a certain period of time delays, the corresponding $\overline{ColMatch_i}$ must become low since contents in the tag are identical to that in the address register. A valid $\overline{ColMatch_i}$ will cancel the signal *Miss* (at this moment the miss flag still remains to tell the memory to transfer the requested line to the cache); and in turn the signal $LRUSel_i$ becomes inactive (low). Thus, the logical value of the point B is 0 so that the two N-devices are open. The E and $\overline{E}$ are unchanged from the previous values. That is, E is at logical 1 and $\overline{E}$ is at 0; Signal $LRUValid_i$ remains 0 to tell the LRU that there are not faults in the tag, and no purge actions take place in this cycle.

In *period 2*, The $\overline{ColMatch_i}$ remains high even though both the *Miss* and

$LRUSel_i$ are at logical 1; there must be faults in the selected tag or the $\overline{ColMatch_i}$ line. At the falling edge of the cache cycle, the tag update signal $LRUSel_i$ remains high so that the two N-devices are close. Due to inconsistency between the tag and the address register, both the *Miss* and the $\overline{ColMatch_i}$ are high. Thus, a logical 0 is on the E while a logical 1 is on the $\overline{E}$ through the N-devices. The state of the purge bit in the tag selected by the *RowSel* changes; and a high *Purge* signal is produced, which makes the corresponding tag permanently unable to carry out associative searches for requested addresses. Meanwhile, a high $LRUValid_i$ is generated which disables the tag update signal $LRUSel_i$, and in turn, the N-devices are set open so that the paths from point A to the purge bit are cut off. A valid $LRUValid_i$ (high) is sent to the LRU to prevent from selecting this tag any more. Once the purge bit is set the purge bit cannot be changed any more whenever the corresponding faulty tag is selected by the *RowSel* later, as shown in *period 3*. That is, whenever a set of tags containing a faulty tag is selected, the $LRUValid_i$ of logical 1 from the purge bit of the faulty tag always prevents the *LRU* from producing a valid tag update signal $LRUSel_i$. The open N-devices cut off the paths so that the state of the purge bit is never changed, remaining "faulty". Note that the AND gate in Fig. 5.2 can be replaced with a NOR gate, which needs four transistors, since the signal $\overline{LRUSel_i}$ is already available in the cache.

Although the tag self-purge mechanism allows the cache to continue normal operations correctly, continuous fault occurrences in tags are considered as the faults that are caused by other problems that the self-purge mechanism cannot handle. Therefore, in the case when a faulty tag is detected during a cache cycle, the cache informs the processor to allow the cache to try other tags for the missing line once more in the immediately following cache cycle by using a *Retry* from the error flag. The LRU must then select another tag instead of the faulty one in the following cache cycle. If a fault is still detected by the purge mechanism during the following

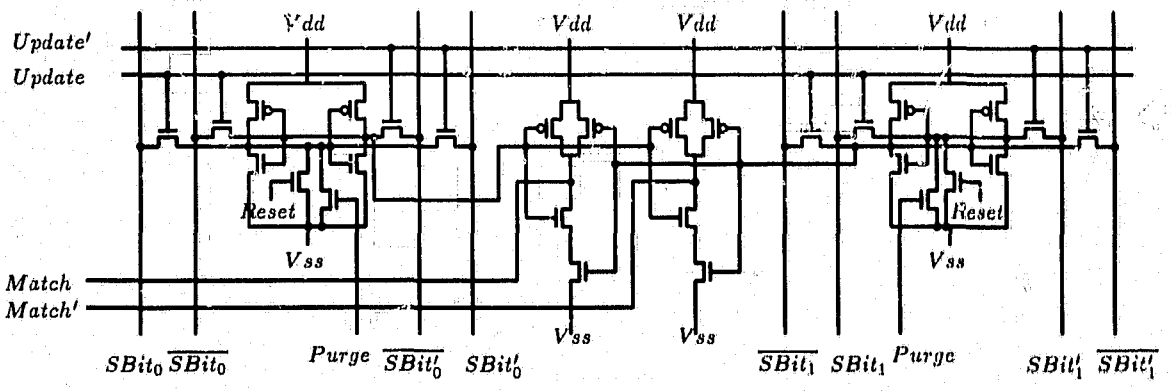


Figure 5.4: Modified Status Bits of a Tag

cache cycle, the error flag rises a signal $FAULT$ to the system and the cache will stop normal operations (see discussion of the error flag).

5.2.1.3 Modified tag status bits

Fig. 5.4 depicts the structure of modified two status bits as an example of the connection of more than one status bit. There are two NAND gates between the two status bits. Outputs $Match$ and $Match'$ of the NAND gates are connected to the corresponding lines of the address part in the tag in Fig. 3.5 (a), respectively. Initially, the $Purge$ signal from the corresponding purge bit is low and the status bits (00) are reset by $Reset$ so that both the $Match$ and $Match'$ are low. There is no comparison in this tag since the NAND gates pull both the $Match$ and $Match'$ down to $Ground$. Other states of the status bits (01, 10, and 11) charge both lines $Match$ and $Match'$ through the P-type devices of the NAND gates since the pathes of the series of N-type devices in the NAND gates are cut off. Thus, the tag can be evaluated as long as the set containing this tag is selected. States of these bits can be changed by setting specified values on the $S0$ and $S1$ during a high $Update$ from

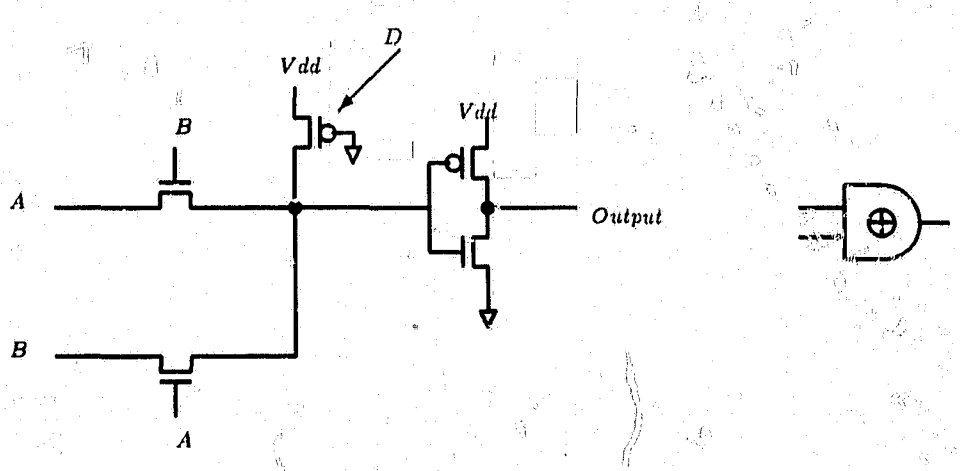
either the SUC0 or SUC1.

If there are errors detected in the address part, the *Purge* signal from the purge bit is high so that the status bits are permanently reset to 00. The tag cannot carry out comparison any more. Thus, this tag is logically self-purged from the normal operations. Hardware overhead for self-purge is N_{tag} bits for purge bits, where N_{tag} is the number of tags in the directory, plus a purge decision mechanism to detect faults and to set the corresponding bits. A column purge decision mechanism requires 14 transistors.

Furthermore, if the cache has to write a line to be purged back into the main memory before transferring the new line, depending on the data coherence protocols, the address in the selected tag for purge has to be read out and stored in a register for flushing the line back to the main memory. Then the address of the missing line is written into the tag. Since the faulty tags are never selected by the LRU, erroneous data/addresses are confined in the cache and never written back to the main memory from the cache. Thus, data pollution in the main memory caused by the faulty cache is prevented.

5.2.2 Comparator Checker

In order to detect faults in comparators in tags for both the operations, the address/update registers, address lines from the registers to the tag array, and the output lines $\overline{ColMatch}$s and $\overline{ColMatch}$'s to the line number generator, two comparators associated to each tag for both processor operation and coherence operation are used during operations. That is, whenever there are no coherence operations, an address from the processor is latched in not only the address register but also the update register as well during the processor operations. The comparator for the processor operations in a selected tag is used to compare the address in the address register with the contents in the tag, whereas the comparator for the coherence



(a) Structure of an Exclusive-OR Gate (b) Logic Diagram of an Exclusive-OR Gate

(a) Structure of an Exclusive-OR Gate

Figure 5.5: The Exclusive-OR Circuit

operations compares the address in the update register with the contents in the same tag. The results from two separate comparators are sent on the $\overline{ColMatch}$s and $\overline{ColMatch}$'s, respectively, and evaluated by the on-line comparator checker, as shown in Fig. 5.1, to see if there is any inconsistency.

Fig. 5.5 depicts an XOR gate, which is used in the error detection circuits. In the gate, P-device D is weak and used to eliminate a possible high-resistance state on the input of the inverter. If inputs A and B are at logical 1, the N-devices connected to the two inputs are closed so that *Output* is low (logical 0) to indicate that the two inputs have the same values. If both inputs are at logical 0, the N-devices are open so that connections between the gate inputs, A and B, and the input of the inverter are cut off. However, the input of the inverter is charged through the P-device so that the *Output* is also low. Otherwise, one gate input is at logical 1 while the other input is at logical 0. The path connecting the input of logical 0 to the input of the inverter is conducted, under control of the input of logical 1, so that the input of the inverter is pulled down to *Ground*. Thus, the *Output* is high to indicate that values on the two inputs are different with each other. One of the advantages of this

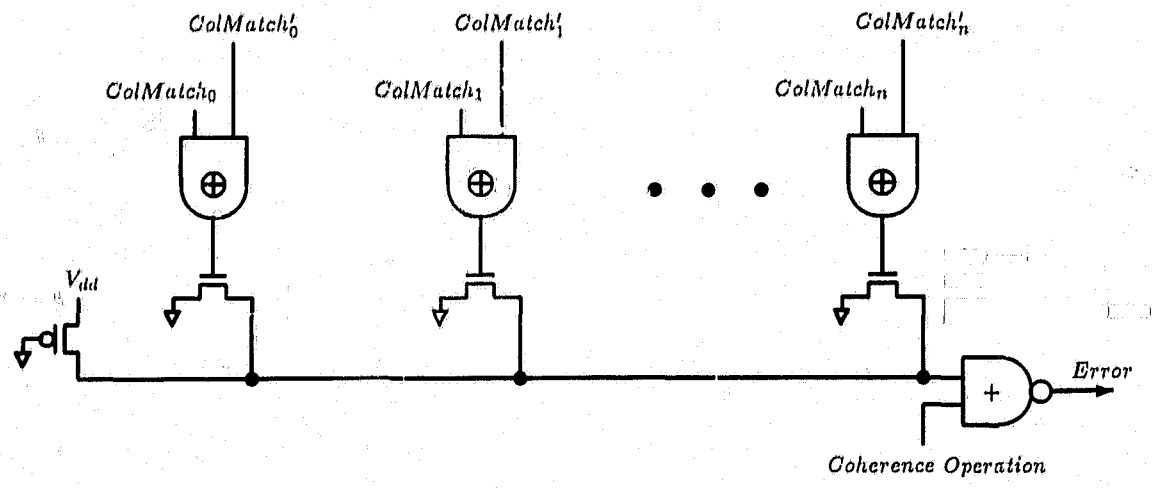


Figure 5.6: Circuit of the Comparator Checker

circuit is low overhead without the requirement of complementary input variables. It can also provide reasonable driving power to meet various circuit requirements by changing the size of the inverter. It only requires five transistors.

Fig. 5.6 shows the comparator checker. The output of each comparator for the processor operations in a selected tag is sent out through the corresponding $\overline{ColMatch}$ while that for the coherence operations is through the corresponding $\overline{ColMatch'}$. Each pair of these two signals from a tag column is connected to two inputs of an XOR gate, as shown in Fig 5.6. When the signal *Coherence Operation* is high, indicating that a coherence operation is carrying out, the NOR gate output, *Error*, is low. This means that fault detection functions are disabled. When the *Coherence Operation* is low, if all the outputs of the XOR gates are at logical 0, the input of the NOR gate is charged to logical 1 by the P-device since all the paths from the NOR gate input to *Ground* are cut off by the N-devices. The checker output, *Error*, is at logical 0 to indicate that comparison results are consistent. If any of the outputs from the XOR gates are at logical 1, indicating that the corresponding pair of $\overline{ColMatch}$ and $\overline{ColMatch'}$ from a selected tag has different values, the NOR

gate input is pulled down to *Ground* so that the *Error* is high to tell the system that errors occur and are detected. Then, the cache error flag rises to inform the system.

Note that the normal operations continue while error detection is proceeding. Thus, system performance is not degraded, and there is not much overhead for fault-tolerance (the checker is relatively simple). Since the comparators for the coherence operations have to perform normal coherence operations if there are any requests, comparison of the results from two comparators for error detection cannot be carried out at all times. Therefore, transient errors may not be guaranteed to be detected when they occur during the coherence operations.

5.2.3 Totally Self-Checking Checker

As mentioned earlier, the LRU is used, during a line miss, to select a cache line probably not to be used in the cache in near future; and the selected cache line is replaced with a new one requested by the associated processor. Meanwhile, the tag array is updated with the requested line address of the main memory. To do so, the tag number corresponding to a selected cache line in a given set is sent to the directory through the n $\overline{LRUSels}$ during a line miss. Tag numbers are encoded in an (n-1)-out-of-n code which is used for detection of any single errors in codewords to prevent writing an address into multiple or zero tags. The *LRUSels checker* in Fig. 5.1 is designed not only to detect any single errors, including transient ones, on the $\overline{LRUSel}$'s but also to detect any defined single faults in the checker itself as well, *i.e.* the *totally self-checking* (TSC) checker. The details of the design are discussed in the next chapter.

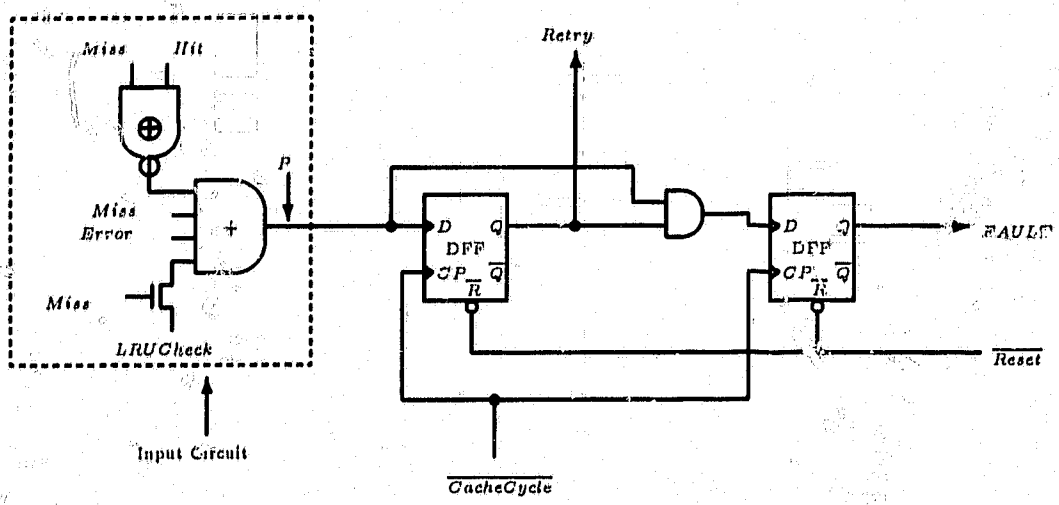


Figure 5.7: The Circuit of the Cache Error Flag

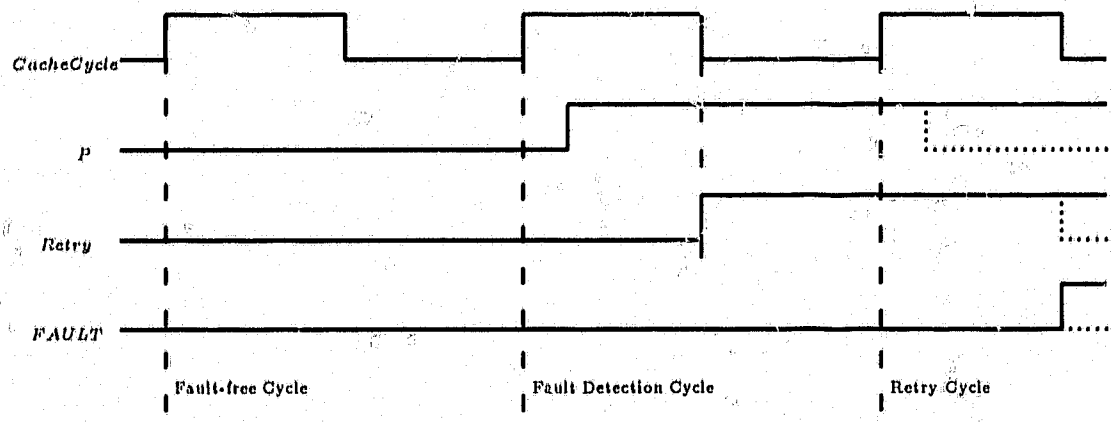


Figure 5.8: The Timing Diagram of the Error Flag

5.2.4 Error Flag

The error flag, as shown in Fig. 5.7, is used to inform the system that errors occur in the cache. Fig. 5.8 illustrates the timing diagram of the error flag in which there are three cache cycles: fault-free, fault detection, and retry cycles. In Fig. 5.7, there are two D-type positive-edge-triggered flip-flops, triggered by the falling-edge of a cache cycle. Initially, the D flip-flops are reset so that signals *Retry* to the processor and $FAULT$ to the multiprocessor system are at logical 0, indicating that there are no faults in the cache.

As discussed earlier, the signal *Error* from the comparator checker provides a valid result of fault detection when there is no request for the coherence operations. The error detection signal from the LRUSels checker, $LRUCheck$, is under control of the signal *Miss* via an N-device. This is because the detection signal can only be valid during a line miss; and it may normally disappear with a low *Miss* signal at the end of a cache cycle (see *Period 1* in Fig. 5.3). The N-device locks the detection signal on the input of the OR gate at the end of a valid *Miss*. Thus, at the end of a cache cycle, the detection signal is still available on the input D of the first flip-flop and stored in the flip-flop. Also, at any time, only one of the *Hit* and *Miss* signals from the line number generator can be at logical 1 under the normal circumstances. Otherwise, both of them are the same values, 0 or 1, indicating that there are faults in the line number generator, or on the $\overline{ColMatch}$s. Furthermore, strictly speaking, the conditions that a faulty tag is detected and purged at end of a cache cycle are that the signals *Miss*, $LRUSel$ to the faulty tag, and $\overline{ColMatch}$ from the faulty tag remain high. Here, we loose the conditions as only to detect *Miss* after each cache cycle. This is because, after a cache cycle for the processor operations, *Miss* must be low if the cache is fault-free. Otherwise, there are faults in the cache and fault indication has to be sent out anyway. Thus, only the signal *Miss* is connected to one of the inputs of the OR gate. Hence, these error signals, if any, are together

to produce the detection signal at the output P of the four-input OR gate.

As shown in Fig. 5.7, if there are no faults, the value at P is logical 0. At the end of a cache cycle (the falling edge of a cache cycle), both flip-flops remain unchanged so that *Retry* and $FAULT$ are kept at logical 0. The operational timing is shown as the *fault-free cycle* in Fig. 5.8. Otherwise, any of these four fault signals brings P to logical 1. And at the end of a cycle, the faulty signal at P is latched in the first flip-flop so that *Retry* rises to inform the processor to allow the cache to try once more in the immediately following cache cycle for any transient errors. Meantime, the second flip-flop remains in the previous state so that $FAULT$ is still at logical 0. The *fault detection cycle* in Fig. 5.8 shows the timing of fault detection operations.

After the processor receives the *ReTry* signal, it tries a second time for an attempt to recover from transient errors. During the second cache cycle, if there are no faults in the cache, the value at P is set to logical 0. Thus, at the end of the second cycle, the two flip-flops are reset so that *Retry* is canceled and $FAULT$ remains zero to indicate that the errors have been recovered, which is shown as the dotted lines in the *retry cycle* of Fig. 5.8. Otherwise, the second flip-flop is triggered and the error flag raises $FAULT$. Meanwhile, the first flip-flop is unchanged so that both *Retry* and *Fault* are at logical 1, indicated by the solid lines in the *retry cycle* of Fig. 5.8, to inform the system that the cache detects permanent faults. The system will take proper actions to deal with this problem.

Note that the input circuit in Fig. 5.7 can be simplified as:

$$\overline{Hit \oplus Miss} + Miss = \overline{Hit} \bullet \overline{Miss} + Hit \bullet Miss + Miss = \overline{Hit} + Miss$$

Thus, the modified circuit is shown in Fig. 5.9. Also fault detection signals from the data memory of the cache and/or the LRU can also be connected to inputs of the OR gate. As we will see, the hardware designed for fault-tolerance and on-line concurrent checking can also be used for off-line cache testing. Thus, utilization of the hardware is increased.

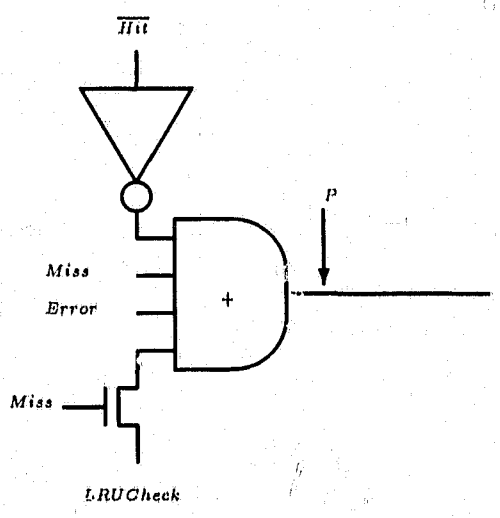


Figure 5.9: The Input Circuit of the Error Flag

5.2.5 Costs for Fault Tolerance and Concurrent Checking

There are three main components in the cache management unit for fault-tolerance and concurrent checking during normal operation: the tag purge mechanism, the comparator checker, and the (n-1)/n code checker. The hardware costs for fault tolerance and concurrent checking are based on the number of transistors in these components. For the purge mechanism, as shown in Fig.5.2, there is one bit (it requires seven transistors) for each tag to purge the corresponding tag by permanently reseting the the status bits (it needs two extra transistors) if a fault occurs in the tag. In addition, there is an error detecting circuit for each way (each column in the tag array of the directory) to detect any permanent faults occurring in a tag during normal operation. It costs fourteen transistors for each way. The total cost for the purge mechanism can be calculated using the equation:

$$N_{purge} = 9N_{tag} + 14N_{way}$$

Components	Direct Mapped	2 Ways	4 Ways	8 Ways	16 Ways
Purge Mechanism	10	$9N_{tag} + 28$	$9N_{tag} + 56$	$9N_{tag} + 112$	$9N_{tag} + 224$
Comparator Checker	5	16	28	52	100
(n-1)/n Code Checker	—	—	24	72	168
Total Cost	15	$9N_{tag} + 44$	$9N_{tag} + 108$	$9N_{tag} + 236$	$9N_{tag} + 492$
N_{tag} is the number of tags in the directory					

Table 5.1: The Hardware Cost for Fault-Tolerance and Concurrent Checking

where N_{tag} is the number of tags in the directory; and N_{way} is the number of ways in the tag array. The cost of the comparator checker, as shown in Fig.5.6, can be obtained by:

$$N_{comp} = 6N_{way} + 4$$

Calculation of the cost of the (n-1)/n code checker is shown in Chapter 6.

Table 5.1 shows the costs for fault-tolerance and concurrent checking in the cache management unit. Columns indicate the costs for the different way sizes the cache uses. When the direct mapped method is employed in the cache, there is no purge mechanism required since there are no other tags available for replacement of the faulty tag. In this case, only an error detection circuit is necessary which can be implemented with a four input AND gate, instead of the circuit in Fig. 5.2. Furthermore, an exclusive-or gate is required for the comparator checker since there are only two output lines from the two comparators in the selected tag in terms of the direct mapped method. For the case that the direct mapped method or 2 way associative method is employed, the TSC (n-1)/n code checker is not necessary. The table shows the costs for a cache in which the number of ways is equal or less than sixteen since most caches have ways of equal or less than sixteen. From the table, we can see that the hardware cost for fault-tolerance and concurrent checking increases when the number of ways in caches increases. Note that the cost of the

error flag is not counted in Table 5.1 since the cost is constant and depends on the implementations of the D-type flip-flops.

The percentage of the hardware overhead used for fault-tolerance and error detection over the cost for the tag array of a single directory cache, which requires $N_{tag}(9N + 28)$ as discussed in Chapter 3, can be obtained using the equation:

$$C_f = \frac{9N_{tag} + 14N_{way} + 6N_{way} + 4 + N_c}{N_{tag}(9N + 28)} \times 100\%$$

Where N_c is the cost of the TSC (n-1)/n code checker. When $N_{way} \leq 16$, $N_c + 4 \leq 11N_{way}$; and $N_{tag} = N_{set} \times N_{way}$. Therefore, the equation is:

$$C_f \leq \frac{9N_{set} \times N_{way} + 31N_{way}}{N_{set} \times N_{way}(9N + 28)} \times 100\%$$

Since usually the number of the sets in a tag array is much larger than 31, the equation is further simplified as:

$$C_f \leq \frac{10N_{set} \times N_{way}}{N_{set} \times N_{way}(9N + 28)} \times 100\% \leq \frac{10}{9N + 28} \times 100\%$$

If we assume that the number of address bits in a tag is 20, $C_f \approx 5\%$, which means that the overhead for fault-tolerance and error detection is quite small! Since a cache tag has rarely less than 16 bits, $C_f \leq 6\%$. Thus, the overhead for fault-tolerance and error detection in a cache management unit is smaller than 6% of the cache management unit with a single directory.

5.3 Summary

In this chapter, in order to improve the reliability of the proposed cache, the designs of the tag self-purge mechanism and the comparator checker in the cache management unit are described. The hardware increase for fault-tolerant and on-line concurrent checking is less than 6 percent of the cache management unit with the single directory.

Chapter 6

TSC Checker Design

In the previous chapter, we designed the tag self-purge mechanism, the comparator checker and the error flag. In this chapter we propose a new design for combinational checkers for 1-out-of-n codes which are used to detect any faults on the $\overline{LRUSels}$ in the directory. It is common to use m-out-of-n (m/n) codes to detect unidirectional errors in VLSI chips, that is, faults within a circuit that either change a number of ones into zeros or vice versa, but not both at the same time. They are also becoming more important with the increasing use of concurrent checking in digital circuits, [68]. m/n codes contain n information bits of which exactly m bits are ones and $(n-m)$ bits are zeros. Checkers are employed to detect errors in the codes during normal operations. A checker should have the capability of detecting errors, not only in codewords, but also in the checker itself during normal operations, namely, it should be a *totally self-checking* (TSC) checker. Thus, the problem of 'who checks the checker' is solved.

One of the most frequently used m/n codes in computer systems is the 1-out-of-n code (m=1), or its complementary (n-1)-out-of-n code. There have been many approaches to the design of totally self-checking checkers for a 1-out-of-n (1/n) code.

In [69, 70, 71, 72], a 1/n code is converted into a k/2k code, and a TSC checker for the k/2k code is used. The schemes, [69, 70, 71], are implemented with combinational circuits while the scheme, [72], is implemented using an nMOS PLA. In [73], a 1/n code is converted into a two-rail code, and a TSC checker for the two-rail code is employed. It is implemented using an nMOS PLA. All of these schemes require the encoding of the 1/n code into some other intermediate code by a TSC translator and use a TSC checker to check the intermediate code. None of these schemes implements a TSC checker for the 1/3 code with combinational circuits. Since TSC checkers for the 1/3 code are impossible to build, from the logical point of view, because of the lack of information provided by the 1/3 code, [70], many researchers have been using other techniques to design TSC 1/3 code checkers.

David proposes a method, [74], to implement a TSC checker for the 1/3 code by using a sequential circuit to map the 1/3 code into a 1/4 code, then developing a TSC checker for the 1/4 code. Another method proposed by Golan, [75], suggests first translating the 1/3 code into an (m+1)-out-of-(n+3) code by combining it with an m-out-of-n code, and translating this new code into a 1-out-of-p code, and then using a TSC checker for the 1-out-of-p code. Both designs are based on gate level implementations and only single stuck-at faults in the gate model are considered. Paschalis *et al* also propose a TSC checker for the 1/3 code, [76], by combining the 1/3 code with other codes. However, a translator initially translates the 1/3 code into an incomplete two-variable two-rail code; this incomplete two-rail code is combined with the 1/2 code outputs of another TSC checker of any codes. This checker is also implemented in nMOS and basically covers the single stuck-at faults. Again, these schemes do not directly implement a TSC checker for the 1/3 code using combinational circuits. Tao *et al* presents a design of a TSC 1/3 code checker using nMOS technology with 17 transistors in [77], and Jha indicates that the inverters at the checker outputs can be removed, [78]. The faults considered in the design are

physical defects occurring in MOS technology. However, stuck-on faults in pull-down transistors and interconnection defects, such as shorts between drain and source of pull-down transistors, *etc.*, are not included in the fault model. Lo *et al* designs a TSC 1/3 code checker, [79]. Again it is implemented using nMOS technology with only 11 transistors. However, some single faults cannot be detected by any input codewords; and the self-checking properties can be defeated by some fault sequences (two faults in the checker followed by a non-codeword at the checker inputs).

In this chapter, we propose a new combinational design of totally self-checking checkers for 1/n codes using CMOS technology. In particular, this design can directly be used for a TSC 1/3 code checker. we initially discuss TSC definitions and the possible faults occurring in 1/n code checkers. In section 6.1, we recall the work of Jha, showing that the *strongly code disjoint* properties in 6.6 are inadequate in some cases and the property of fault secure should hold as well as the *strongly code-disjoint* properties, whenever an undetectable fault exists in the checker. We also create one of the most comprehensive fault models at the transistor level by far in which all the single physical defects frequently occurring in MOS implementations are covered. Then we explain the design of the checkers and prove that they are totally self-checking. Finally, we discuss this design and other TSC checker implementations.

6.1 TSC Definitions and Fault Model

The design of totally self-checking checkers has been widely discussed since the TSC definitions were introduced by Anderson *et al* in 1973. The conventional definitions, [69], are as follows:

Definition 6.1 *A circuit is self-testing for a set of faults F, if for every fault in F, the circuit produces at least one non-code output for at least one code input.*

Definition 6.2 *A circuit is fault-secure for a set of faults F, if for every fault in F, the circuit never produces an incorrect code output from a code input.*

Definition 6.3 *A circuit is code-disjoint if the circuit maps the input code space into the output code space, and the non-input code space into the non-output code space.*

Definition 6.4 *A circuit is totally self-checking if it is self-testing, fault secure, and code-disjoint.*

In any self-checking design, it is assumed that the inputs and outputs are encoded. Thus, a non-codeword means that a word is not in the code space. For example, 100 is not in the 2-out-of-3 code space; however, it is in the 1-out-of-3 code space. Therefore, an n-bit code space is normally a subset of the space of all possible n-bit vectors. An incorrect codeword means that a codeword other than the intended one is produced. That is, the word produced is in the code space, but it is not the one that is expected. For a given fault set for a circuit, the *self-testing* property guarantees the detection of any single fault in the circuit by at least one input codeword. The *fault-secure* property does not allow the circuit to generate an incorrect output codeword in the presence of any single fault in the circuit. The *code-disjoint* property guarantees that a codeword in the input code space maps into the output code space.

TSC circuits work under the assumptions that faults occur one at a time and that the time interval between occurrences of any two faults is long enough for all input codewords to be applied to the circuit. The first assumption is required because a TSC circuit is designed with respect to any *single* fault in a given fault set. The TSC properties may not hold if more than one fault occurs simultaneously. Since the self-testing property requires one or more input codewords to detect a fault, the second assumption is required to make sure that a fault can be detected before the next one occurs in a circuit, or a non-codeword is present on the inputs of a circuit.

Given these fault assumptions, a TSC circuit always produces a non-codeword as the first erroneous output due to a fault. This behavior is referred to as the TSC goal, [80]. Another alternative to achieve the TSC goal is to design a circuit which is *strongly fault-secure* (SFS), [80].

Definition 6.5 *A circuit is strongly fault-secure for a set of faults F, if for every fault in F, either*

1. *the circuit is self-testing and fault-secure, or*
2. *the circuit is fault-secure, and if another fault from F occurs in the circuit then either property 1 or 2 is true for the fault sequence.*

This definition implies that a SFS circuit never produces an incorrect codeword as the first erroneous output of the circuit because, according to the definition, the circuit continues to be fault-secure even though one or more faults is present in the circuit. Thus, a SFS circuit achieves the TSC goal. In fact, SFS circuits are the largest class of functional circuits which meet the TSC goal [81].

For checkers, there are two output lines from a TSC checker. If a checker has only one output line, the output line may be stuck at its "good" logic value and the fault cannot be detected by applying any codewords to the checker input. For this reason, the outputs are always encoded into the 1/2 code. The main function of the checker is to indicate the first occurrence of a non-codeword at its inputs (the outputs from the circuit under test). The faults in the checker should be detected as much as possible with the input codewords; and the undetected faults should not interfere with the checker's mission. Based on the TSC goal proposed by Smith and Metze in [80], the TSC goal for a checker can be stated as, [79], "given the fault assumptions, a code checker always produces a non-codeword as the first erroneous output due to a fault in the circuit under check and the fault(s) in the checker must either be detected or not interfere with its capability to produce a non-code output

for a non-code input from the circuit under check". For checkers, in general, the fault-secure property is not necessary, [82], although the conventional TSC checker definition (Definition 4) requires this. The reason is that it does not matter if the output of a checker is changed from 10 to 01 or *vice versa* in the presence of a fault as long as the fault is detected later, or the checker keeps mapping non-codewords at its inputs into non-codewords at its outputs. Thus, a new concept of *strongly code-disjoint* (SCD) is defined, [83]:

Definition 6.6 *A circuit is strongly code-disjoint with respect to a set of faults F if before the occurrence of any fault, the circuit is code-disjoint, and for every fault in F, either*

1. *the circuit is self-testing, or*
2. *the circuit always maps non-codewords at its inputs to non-codewords at its outputs, and if another fault from F occurs then either property 1 or 2 is true for the fault sequence.*

An SCD checker can continue to be code-disjoint even though undetected faults or fault sequences remain in the checker. Although SCD checkers can usually achieve the TSC goal, their use in a multi-level checker may defeat the self-checking properties of the checker (since the outputs of the SCD checkers can not directly be observed). That is, the checker may allow an erroneous output vector from the circuit under check to go through the checker without detection.

An example is given in [81]. Consider the self-checking system, as shown in Fig. 6.1, in which there are two functional circuits under check. The outputs of these functional circuits are checked by SCD checker 1 and SCD checker 2 respectively. The checker at the final level is a TSC two-rail checker as usual. Consider a fault ϕ_1 in checker 1 with respect to which the checker is neither self-testing nor fault-secure (such a fault is allowed in an SCD checker). Suppose that in presence of ϕ_1,

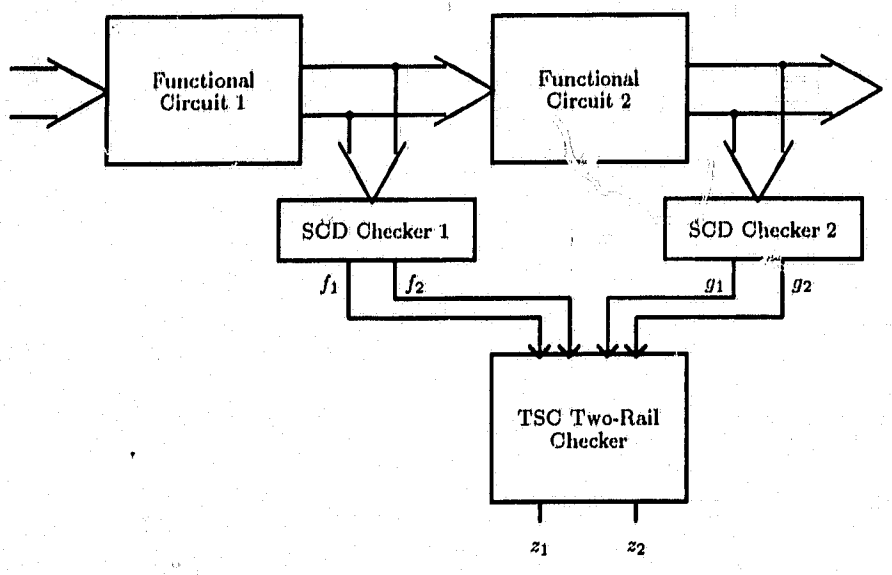


Figure 6.1: A Self-Checking System with SCD Checkers

checker 1 produces an output 01 at f_1 and f_2 for only one input codeword and 10 for all others. Thus, either the codeword, consisting of f_1, f_2, g_1, and g_2, at the inputs of the two-rail checker is equal to 0101 or to 0110, but not both because the functional circuit 2 can only produce, with the input codeword which has checker 1 to generate 01, one output which makes checker 2 to generate either 01 or 10, but not both. Therefore, in all, only three codewords can be fed to the two-rail checker, the other two words are 1001 and 1010. However, the two-rail checker requires all the four words for self-checking purposes. Thus, a subsequent fault, say, ϕ_2 in the two-rail checker, which was otherwise detectable, may no longer be detected. If a non-codeword occurs at the outputs of functional circuit 1, it will map to 00 or 11 at the outputs of checker 1. However, this may result in a codeword either 01 or 10 at z_1 and z_2, the outputs of the two-rail checker, due to the presence of undetected fault ϕ_2. This means the checker TSC goal is defeated by the fault sequence.

Thus the SCD properties in a checker are not necessarily sufficient for all circuits. A checker should also have the fault-secure property as well as the SCD properties. Therefore, combining the SFS and SCD properties, the concept of the strongly self-checking (SSC) for checkers is proposed by Jha in [84, 85] as follows:

Definition 6.7 *A circuit is strongly self-checking with respect to a set of faults in F if before the occurrence of any fault, the circuit is code-disjoint, and for every fault in F, either*

1. *the circuit is self-testing, or*
2. *the circuit is fault-secure and always maps non-codewords at its inputs to non-codewords at its outputs, and if another fault from F occurs then either property 1 or 2 is true for the fault sequence.*

This definition implies that a SSC checker continues to be fault-secure and code-disjoint even if undetectable faults or fault sequences are present in the checker. In fact, SSC checkers are the largest class of checkers that achieve the TSC goal of a checker under any circumstances. Our design of the checkers for 1/n codes is based on the SSC properties to achieve the TSC goal.

Since this design is implemented in CMOS technology, the fault model considered is at the transistor level and includes physical defects frequently occurring in MOS technology [86, 87]. Faults in the fault set are important to a TSC checker because the TSC properties of the checker are based on these faults. We use the similar set of single faults from [86, 87], and expend it to also include the faults of 1, 7, and 8. Thus, the following single faults are considered in the checker:

1. stuck-at faults on the gate of transistors (s-a-1 and s-a-0).
2. stuck-on and stuck-open in transistors (s-on and s-open). The difference between these faults is that stuck-on and stuck-open faults occur in transistors themselves whereas stuck-at faults occur on transistor gates.

3. shorts between the gate and drain of transistors (g-d), shorts between the gate and source of transistors (g-s), and shorts between the source and the drain of transistors (s-d).

4. opens at the drain contacts (d-c), the source contacts (s-c), and the gate of transistors, *e.g.* floating gate, (f-g).

5. shorts and opens in interconnect lines (short and cut).

6. stuck-at faults on input and output lines (s-a-0 and s-a-1).

7. bridging faults, including both AND bridge and OR bridge faults, between input lines or output lines (bridge).

8. transistor missing faults; i.e. transistors missing from their positions (the channel of transistors is not created) due to manufacturing problems (miss).

Note that the P-type transistors whose gates are connected to ground have no stuck-on faults since the transistors are always on. The fault effect of the floating gate of a transistor depends on the charges stored on the gate before the open fault occurs. Since these eventually leak away, this fault has the behavior of a s-open in the transistor if the transistor is an N-type device; and it behaves as a s-on if the transistor is a P-type device. Circuit level studies relate to the exact values of voltages at the observation points. These voltages can be translated into 5 logic values as: hard 0, soft 0, indeterminate, soft 1, and hard 1, [86]. Usually, a digital circuit produces results of hard 1 or/and hard 0. However, due to faults in the circuit, the output values of the circuit may be any of these 5 logic values.

6.2 TSC Checker Design for 1-out-of-n Codes

This proposed design is for a TSC (n-1)/n code checker, but it is easy to implement 1/n code checkers with (n-1)/n code checkers. Our strategy is to partition all possible input binary codewords of n-tuples $a_1a_2\ldots a_n$, into two sets S_1 and S_2 such that all the codewords are in S_1 and the non-codewords in S_2. Since there are only n codewords for an $(n-1)/n$ code, S_1 has n elements. The codewords in S_1 are further partitioned arbitrarily into 2 disjoint subsets $X = \{x_i | x_i \in S_1, 1 \leq i \leq k\}$ and $Y = \{y_j | y_j \in S_1, 1 \leq j \leq l\}$, where $k + l = n$, $X \cup Y = S_1$, and $X \cap Y = \phi$. Considering time delay and hardware costs, it is preferable for k and l to be equal. Consequently, whenever n is even, $k = l = n/2$; and whenever n is odd, one of k and l is $\lfloor n/2 \rfloor$ and the other is $\lfloor n/2 \rfloor + 1$. We construct two functions f_1 and f_2 to map, for all $x_i \in X$ and all $y_j \in Y$, respectively, such that $f_1(x_i) = 1$, $f_1(y_j) = 0$, $f_2(x_i) = 0$, and $f_2(y_j) = 1$. That is,

$$f_1 = x_1 + x_2 \cdots + x_k + w$$

$$f_2 = y_1 + y_2 \cdots + y_l + w$$

where w is a special non-codeword 'all-one' (no zeros in the word) which produces an output of 11 on f_1f_2. It is included because it is very useful for the construction of a larger TSC checker using several smaller TSC checkers, as discussed below. Let $F = f_1f_2$ form the 1/2 code output space of a checker, output codewords being 10 and 01, whereas non-codewords are 00 and 11. Thus, $F(x) = 10$ $(\forall x \in X)$, and $F(y) = 01$ $(\forall y \in Y)$. For w, $F(w) = 11$ and $F(z) = 00$ $(\forall z \in S_2$ with more than one 0). Hence, the checker is code-disjoint.

The design is implemented in CMOS using modified transmission gate logic. To illustrate the method, we initially design a TSC checker for the 2/3 code (see Fig. 6.2). This is expanded to other TSC checkers for (n-1)/n codes. The high-resistance state at the output line of the transmission gate logic is eliminated by adding a P-

type transistor (transistor l_3 in Fig. 6.2) between the output line of the logic and V_{dd}. It can also be implemented easily in other technologies such as nMOS using load transistors instead of P-type transistors.

The input codewords of the 2/3 code can arbitrarily be partitioned into X and Y, and in this example, they are divided as $X = \{110, 101\}$ and $Y = \{011\}$. Fig. 6.2 shows the structure of the checker. There are three inputs A, B, and C (B and C are not shown in the figure to avoid confusion), as well as two outputs f_1 and f_2 from the circuit. Input line A is connected to transistors a_1, a_2, and a_3 whereas B is connected to b_1, b_2, and b_3. C is only linked to c_1. Initially, we show that the circuit implements the correct functions and is code-disjoint:

(a) if the checker input(CBA) is 110, both C and B are 1's and A is 0, transistors c_1, b_1 and b_2 are closed while a_1 and a_2 are open. $\overline{a}$ is charged to V_{dd} and transistor t_1 is closed. Since b_2 and t_1 are both closed, $\overline{b}$ is pulled down to ground. Meanwhile, transistor t_2 is cut off. Thus, the path from $\overline{f}$ to ground via c_1 and b_1 is conducting so that $\overline{f}$ is at logic 0 and f_1 is at logic 1.

(b) Similarly, if the input is 101, b_1 and b_2 are open while a_1, a_2 and c_1 are closed. $\overline{b}$ is charged to V_{dd} so that t_2 is closed. The path controlled by c_1 and a_1 is conducting to pull $\overline{f}$ down to ground so that f_1 is high.

(c) If the input is 111 (the special codeword w), both the paths are conducting so that $\overline{f}$ is low, and, in turn, f_1 is high.

(d) If the input is $011 \in Y$, open transistor c_1 cuts off the paths so that $\overline{f}$ is always 1 and f_1 is 0.

(e) It is also easy to see that if inputs have codewords with more than one 0 (001 010 100 000), either c_1 or both a_1 and b_1 are open. all the paths are cut off so that f_1 remains 0. Hence, function f_1 is correctly implemented by the circuit.

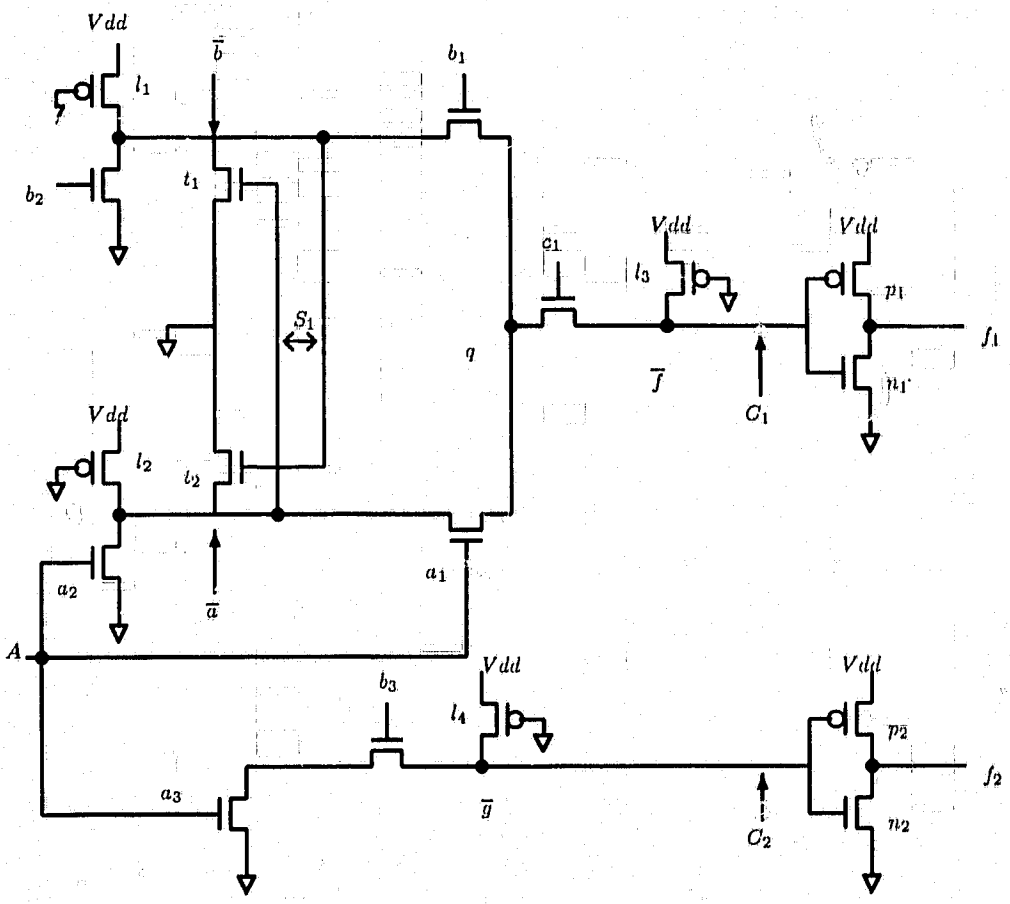


Figure 6.2: The 2-out-of-3 TSC Checker Using Pass-Transistor Logic

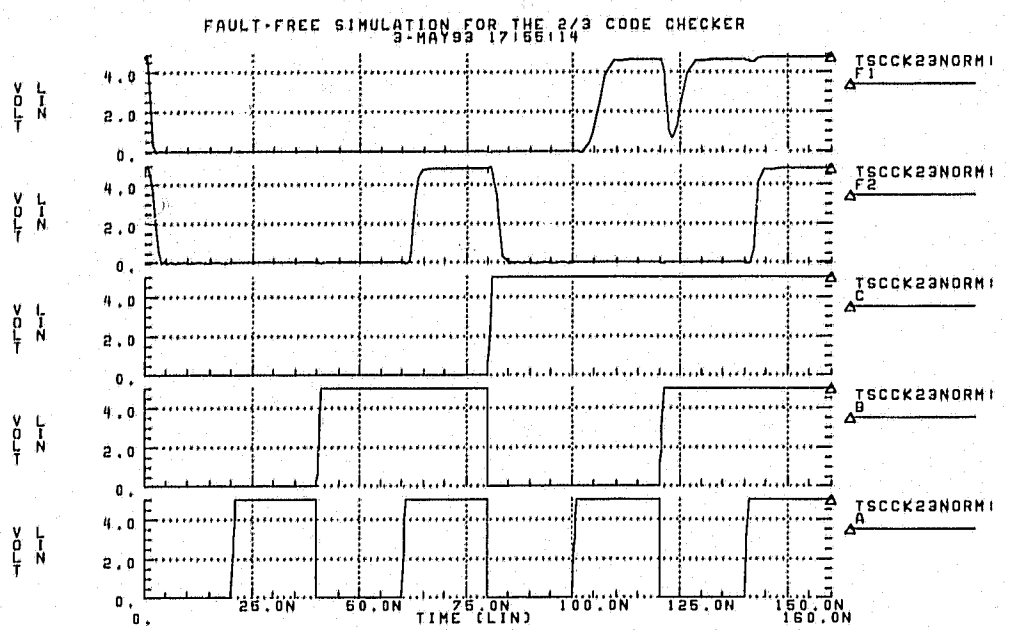


Figure 6.3: The Fault-Free Simulation for the 2-out-of-3 TSC Checker

(f) f_2 is high if and only if signals on both a_3 and b_3 are high, regardless of signal C. That is, f_2 is high only on inputs $011 \in Y$ and 111 (w). Otherwise, f_2 is at logic 0 because the path is cut off by an open transistor either a_3 or b_3.

Therefore, the circuit performs the correct functions. Table 6.1 shows the truth table of the functions. In order to verify the checker's functional correctness and to prove the TSC goal be achieved for all single faults in the checker, we use the HSPICE circuit simulator for the function verification and the fault simulations. The circuit simulation results shown in Fig. 6.3, produced by the HSPICE simulator, also illustrate the functional correctness of the fault-free checker. Thus, the circuit is *code-disjoint*. In summary, output 01 or 10 shows the codeword on CBA is a 2/3 codeword while output 11 ('all-one') is indicating the non-codeword 111 ('all-one') on the input and output 00 is for other non-codewords with more than one 0 on the input.

Input Nodes (CBA)	Output Nodes $f_1 f_2$
000	00
001	00
010	00
011	01
100	00
101	10
110	10
111	11

Table 6.1: Truth Table of the 2-out-of-3 Code Checker

All the single faults except bridge and stuck-at faults at the input lines affect only one of the outputs. Thus, any single one of these faults causes the checker to produce either a correct output codeword or a noncodeword, but never an incorrect codeword. Obviously, any single stuck-at faults on input lines A, B and C can be detected by at least one codeword in which the bit to the faulty line is complementary to the fault; and the remaining codewords make the faulty checker generate correct outputs. For example, if input line A is stuck at 1, a codeword 110 is applied. This is equivalent to applying a non-codeword input 111 to the fault-free checker. The checker produces 11. The checker produces correct outputs 01 and 10, respectively, on the remaining inputs 011 and 101. If A is stuck at 0, 011 or 101 is applied, which is equivalent to applying a non-codeword with more than one 0 to the fault-free checker. The checker generates a nonvalid output 00. The checker generates a correct output 10 on 110. Moreover, any single AND or OR bridging fault on the input lines is also detected by input codewords. For instance, an AND bridge

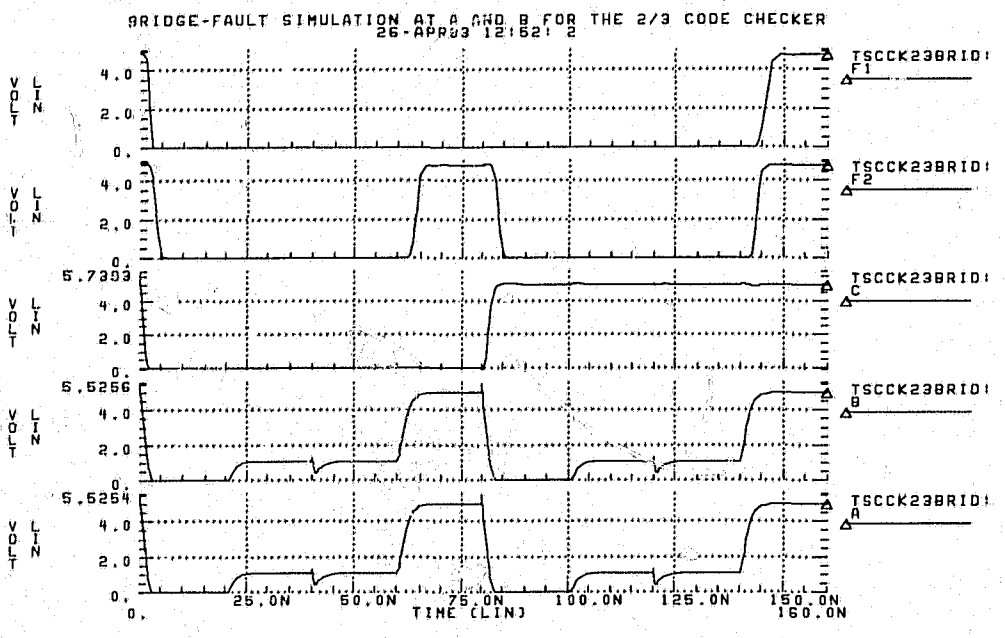


Figure 6.4: AND-Bridge Fault Simulations between Checker Inputs A and B

between A and B is detected by applying 101 or 110 to the input. It is equivalent to inputting a non-codeword 100, and the resulting circuit output is 00. The checker produces a valid output 01 on inputting 011. Also, an OR bridge between A and B causes the circuit to produce 11 by applying 101 or 110, which is equivalent to the input of a non-codeword 111 on the fault-free checker; and application of 011 has the faulty checker output a correct codeword 01. Therefore, any single stuck-at faults and bridging faults on the input lines cause the circuit to produce either nonvalid outputs with some input codewords, or correct outputs with other input codewords, but never to produce incorrect outputs. Hence, the checker is fault-secure for all the defined faults. Furthermore, from the above discussion, the checker is also self-testing in the presence of the bridge or stuck-at faults at the checker inputs.

In order to simulate the AND-bridge faults, we attached an inverter to each input line of the 2/3 checker. The inverters are the same as that at the checker outputs:

the inverters are the N-dominant CMOS implementations. A CMOS gate is said to be N-dominant (P-dominant) if its output is 0 (1) when both its pMOS and nMOS networks conduct due to a fault, [81]. Fig. 6.4 shows the results for the bridge fault between A and B, which is consistent to our above discussion. After bridging the voltages of inputs A and B are the same. Similarly, the simulations for OR-bridge faults are obtained if the inverters are the P-dominant implementations.

Given the large number of single-fault simulations for the self-testing property of the checker, it is not possible to include them all here. Therefore, for those faults which obviously can be detected by some codewords, such as s-a-1 faults at output lines, a s-d short at l_3, a g-d short at p_1, *etc.*, are not illustrated. Also, prior to our discussions of the self-testing property of the checker due to the faults other than the bridge and stuck-at faults at the input lines, we consider the equivalence of some faults. For the N-type transistors, a s-open fault in the transistor causes the circuit to have the same behavior as a s-a-0 fault on the gate of the transistor, cut faults at the drain or source of transistors, a floating gate fault, a short fault between the gate and the source of transistors (except n_1 and n_2), and transistor missing faults do. A s-on fault forces the circuit to behave as a s-a-1 on the gate, or a drain-source short at transistors (except n_1 and n_2). For the P-type transistors, a s-open fault in transistors is equivalent to a s-a-1 on the gate, cuts of the drain or the source of transistors, or gate-drain short fault of transistors while a s-on fault behaves as a s-a-0 on the gate, or floating gate (note that these faults are only for p_1 and p_2 shown in Fig. 6.2). Furthermore, because of the symmetry of the circuit; faults at b_1, b_2, l_2, and t_2 have similar behavior as those at a_1, a_2, l_1, and t_1.

If there is a stuck-open in c_1, f_1 is permanently at logic 0, which is equivalent to a stuck-at-0 fault on output line f_1 (as well as a s-d fault at n_1 and a g-d fault at p_1). An input pattern 110 or 101(CBA) can be used to detect this fault by generating a non-codeword 00 since f_2 is not affected by the fault. If c_1 is stuck on, c_1 is always

closed. When the input is 011, the faulty checker produces the non-codeword of 11 by changing f_1 from 0 to 1. If transistor a_1 is stuck open and 101 is applied, since a_1 and b_1 are both open, the two paths are cut off. Thus, f_1 is 0, while f_2 remains 0, so that circuit output is 00. In the case that a_1 is stuck on and 110 is applied, the faulty checker performs as follows: Here we use the bidirection characteristics of pass-transistor gates for detecting this fault. Since current is injected, via the faulty a_1 (stuck-on), into the path controlled by c_1 and b_1, $\overline{a}$ and $\overline{b}$ are both at neither logic 1 nor logic 0, but somewhere between 1 and 0. t_1 and t_2 are both half-conducting so that the voltage at $\overline{f}$ is between V_{dd} and ground. Note that transistors b_2, a_2, t_1 and t_2 are carefully ratioed so that, in this case, the voltage value at $\overline{f}$ is above the threshold of the inverter (soft 1 at $\overline{f}$) so that f_1 is forced to be 0. Thus, the circuit output is a non-codeword 00. If a_2 is stuck open (or a s-d short at l_2) and the input is 101, $\overline{b}$ is high and t_2 is closed. Since a_2 is open, the channel of t_2 is not wide enough to pull $\overline{f}$ under the inverter threshold so that f_1 is 0. If a stuck-on fault occurs on a_2 (or cut-equivalent faults at l_2) and 110 is applied, $\overline{a}$ is pulled down to ground, though a_1 is open, so that t_1 is open. The channel of b_2 is not wide enough to pull $\overline{f}$ under the invertor threshold so that f_1 is at logic 0. The checker output is 00.

If a stuck-open fault occurs on t_1, the bypass from t_1 is permanently cut off. When 110 is on the checker input, the voltage at $\overline{b}$ rises to some value between V_{dd} and ground. Therefore, $\overline{f}$ is above the inverter threshold so that f_1 is 0. In the case that t_1 is stuck on, when 101 is applied, although b_1 and b_2 are both open, $\overline{b}$ is still pulled down to ground through the faulty t_1. Thus, t_2 is open so that $\overline{a}$ is between logic 1 and logic 0; and f_1 is 0. In the both cases, the circuit outputs are 00.

A cut-equivalent fault at transistor l_3 (l_4) causes the output f_1 (f_2) to be permanently 1, which can be easily detected. A s-open-equivalent fault in the inverters, such as a s-open, s-a-1, s-c, or d-c faults at the P-type transistor or a s-open, s-a-0,

s-c, and d-c fault at the N-type transistor, causes the checker to have a "sequential" behavior, since some charges may be locked at f_1 or f_2 before the fault occurs. A sequence of two input vectors is required to detect these kinds of faults. For example, if there is a s-open (or equivalence) fault at p_1 and f_1 is high, a sequence of 011 and 101 (or 110) at the checker inputs results in 01 and 00 at the outputs.

If b_3 or a_3 is stuck open, the path from $\overline{g}$ to ground is blocked, which is equivalent to a stuck-at-0 fault on the output line f_2. The circuit produces 00 by applying 011 to its input. If b_3 or a_3 is stuck on and 101 or 110 is applied, respectively, $\overline{g}$ is pulled down to ground so that f_2 is 1. Thus, the circuit output is 11 from 10. Simulation results from Fig. 6.5 to 6.13 illustrate the above analysis on these s-on and s-open faults. Furthermore, Fig. 6.14 to 6.18 show the drain-gate shorts at a_1, a_2, a_3, c_1, and t_1, are also detected by the 1/3 codewords. Note that short S_1 in Fig. 6.2 is equivalent to the drain-gate short at t_1 while cuts, C_1 and C_2 are equivalent to the s-a-1 faults at f_1 and f_2, respectively. Therefore, it can also be detected. Fig. 6.19 shows a simulation for the short between the input $\overline{f}$ and the output f_1 of the inverter (or g-s fault at p_1 or g-d fault at n_1). It shows that when the input codeword is 011, the value of f_1 is about 3.9 volts, which is considered to be a soft 1. Therefore, the fault is detected. However, the faulty checker has about 2.0 volts on f_1 with the other two codewords at its inputs. We consider that the corresponding logic values are indeterminate. The stuck-on faults in the inverters are handled by the N-dominant CMOS design so that the outputs of the inverters are logic 0 when both P-type and N-type transistors conduct due to the s-on fault in either transistor. A s-on fault in the N-type transistor (n_1 or n_2) is detectable with a logic 0 at the inverter input and a logic 0 at the output. However, a s-on equivalence in the P-type transistor (p_1 or p_2) remains undetectable. Such a s-on fault makes an inverter behave as an nMOS inverter so that the logic behavior of the inverter is not changed. However, it does not prevent the detection of any other

single faults that may occur later in the circuit. Thus, the self-checking properties of the checker are not destroyed even though there is a stuck-on fault in a transistor of the inverters. Hence, the circuit is either (1) self-testing and fault-secure or (2) fault-secure and continues to map codewords (non-codewords) at the inputs to codewords (non-codewords) at the outputs. When the next fault occurs, either (1) or (2) is true for the fault sequence. The checker is strongly self-checking.

Table 6.2 shows that all the defined single faults, except s-open-equivalent faults in the inverters which can be detected by two vectors, cause the checker to produce non-codewords at its outputs for some input codewords. The faults in the table are represented by concatenating the names of either transistors or lines with the faults. For example, the fault a_1-s-d stands for a short between the source and drain of transistor a_1. If a single fault can be detected by more than one input codeword, only one codeword is shown in the table.

A TSC checker for a 3/4 code is shown in Fig. 6.20. It is easy to show that the checker is strongly self-checking using similar arguments.

6.3 Building TSC Checkers for (n-1)/n Codes

The drawback of using the pass-transistor implementation is that a TSC checker with a large number of inputs is not practical because of the time delay in a series of pass-transistor gates. Therefore, if n is large, a TSC (n-1)/n checker is designed as a tree of the TSC checkers discussed above, with each as a node in the tree. n input bits of the tree-style checker are partitioned into q groups; and group i has n_i bits ($1 \leq i \leq q$), the number of inputs of leaf node i. Therefore, q is the total number of leaf nodes. Such a checker built as a tree of TSC checkers is also a TSC checker. For example, Fig. 6.21 shows a checker for a 7/8 code as a tree of three TSC 3/4 code checkers. At the bottom level (leaf nodes), if the input of the 7/8

Inputs	Faults Detected
110	The follows are the single faults with output 00: a_1-s-a-1, a_1-s-on, a_1-s-d, a_2-s-a-1, a_2-s-on, a_2-s-d, b_1-s-a-0, b_1-s-open, b_1-miss, b_1-f-g, b_1-d-c, b_1-s-c, b_1-g-d, b_1-g-s, b_2-s-a-0, b_2-s-open, b_2-miss, b_2-f-g, b_2-d-c, b_2-s-c, b_2-g-d, b_2-g-s, t_1-s-a-0, t_1-s-open, t_1-miss, t_1-f-g, t_1-d-c, t_1-s-c, t_1-g-d, t_1-g-s, t_2-s-a-1, t_2-s-on, t_2-s-d, n_1-s-a-1, n_1-s-on, n_1-s-d, p_1-g-d, p_1-miss, l_2-miss, l_1-s-d, s_1, l_2-g-s, l_3-s-d, B-s-a-0, C-s-a-0, bridges between A-C or f_1-f_2. The follows are the faults with output 11: a_3-s-a-1, a_3-s-on, a_3-s-d, l_4-miss, p_2-s-d, A-s-a-1, f_2-s-a-1
101	The follows are the single faults with output 00: a_1-s-a-0, a_1-s-open, a_1-miss, a_1-f-g, a_1-d-c, a_1-s-c, a_1-g-d, a_1-g-s, a_2-s-a-0, a_2-s-open, a_2-miss, a_2-f-g, a_2-d-c, a_2-s-c, a_2-g-d, a_2-g-s, b_1-s-a-1, b_1-s-on, b_1-s-d, b_2-s-a-1, b_2-s-on, b_2-s-d, t_1-s-a-1, t_1-s-on, t_1-s-d, t_2-s-a-0, t_2-s-open, t_2-miss, t_2-f-g, t_2-d-c, t_2-s-c, t_2-g-d, t_2-g-s, c_1-s-a-0, c_1-s-open, c_1-miss, c_1-f-g, c_1-d-c, c_1-s-c, c_1-g-s, c_1-g-d, l_1-miss, l_1-g-s, l_2-s-d, A-s-a-0, f_1-s-a-0, bridges between A-B and B-C. The follows are the faults with output 11: b_3-s-a-1, b_3-s-on, b_3-s-d, n_2-g-s, l_4-g-s, n_2-miss, B-s-a-1, C_2.
011	The follows are the single faults with output 00: a_3-s-a-0, a_3-s-open, a_3-miss, a_3-d-c, a_3-s-c, a_3-f-g, a_3-g-s, a_3-g-d, b_3-s-a-0, b_3-open, b_3-miss, b_3-d-c, b_3-s-c, b_3-f-g, b_3-g-s, b_3-g-d, n_2-s-a-1, n_2-s-on, n_2-s-d, n_2-g-s, n_2-d-s, p_2-g-d, p_2-miss, l_4-s-d, f_2-s-a-0. The follows are the faults with output 11: c_1-s-a-1, c_1-s-on, c_1-s-d, C-s-a-1, n_1-g-d (11*), n_1-g-s, n_1-miss, p_1-g-s (11*), p_1-s-d, l_3-miss, f_1-s-a-1 l_3-g-s, C_1.
Note that the faults in this table do not include the sequential behavior faults. These faults are the open faults at P-type transistor and N-type transistors in inverters The faults followed by (11*) means that the faulty outputs has a soft 1 marked as 1*.	

Table 6.2: Input Test Patterns for the Single Faults

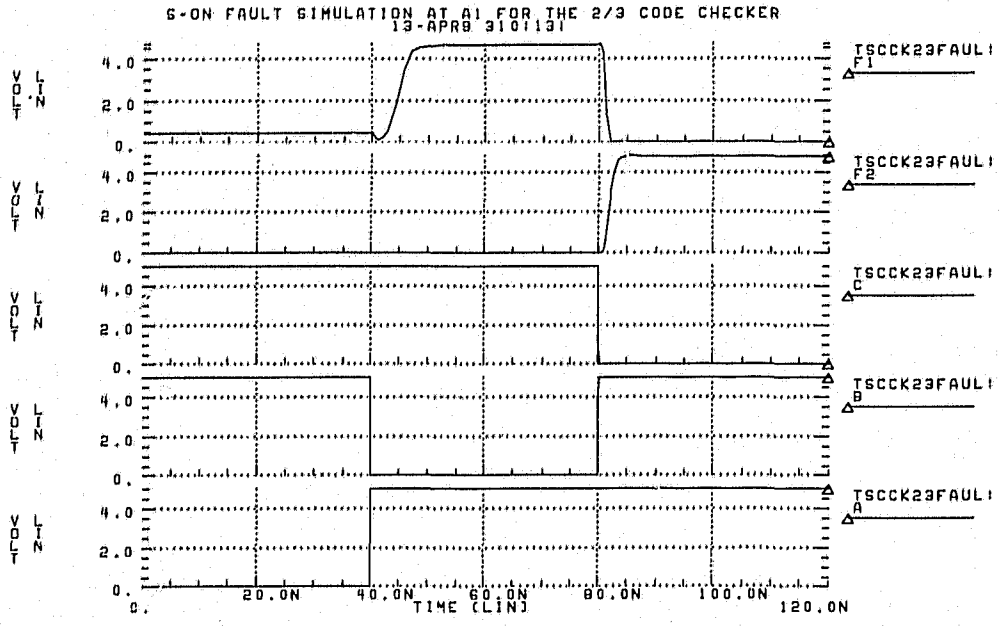


Figure 6.5: Simulation for the 2/3 TSC Checker with Stuck-On Fault at a1

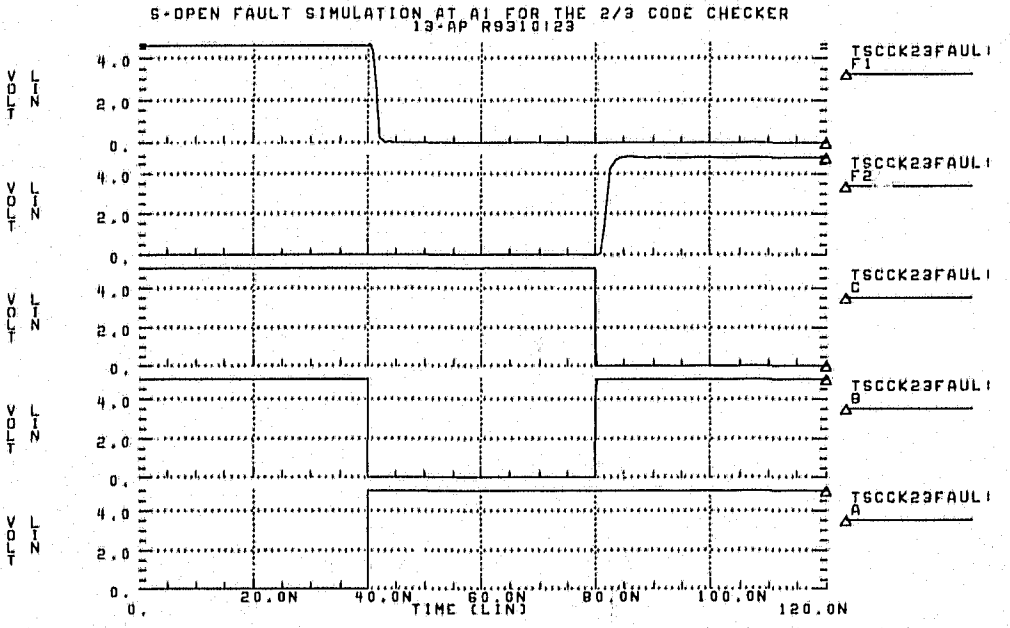


Figure 6.6: Simulation for the 2/3 TSC Checker with Stuck-Open Fault at a1

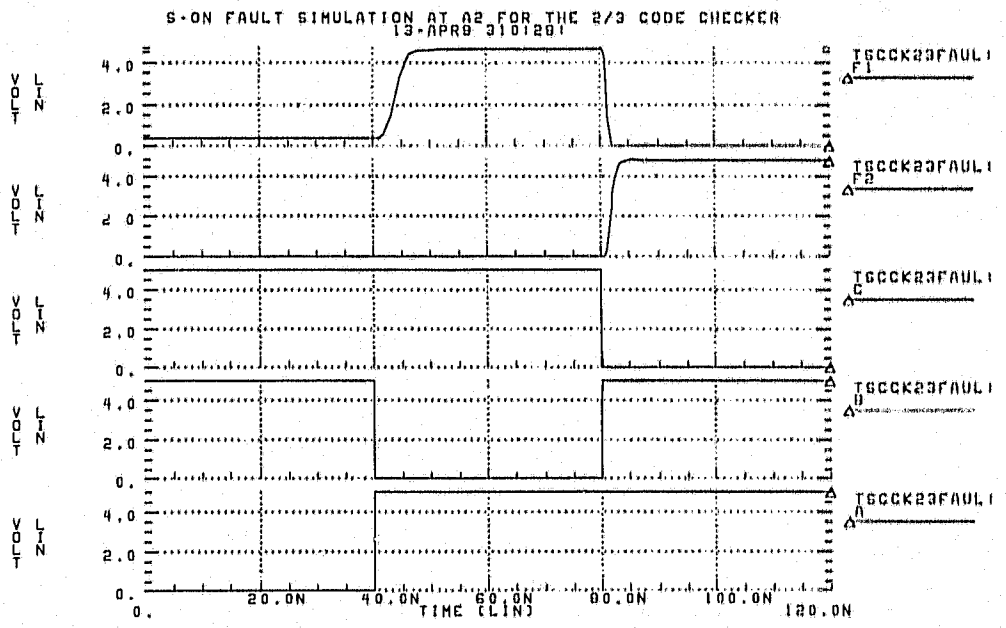


Figure 6.7: Simulation for the 2/3 TSC Checker with Stuck-On Fault at a2

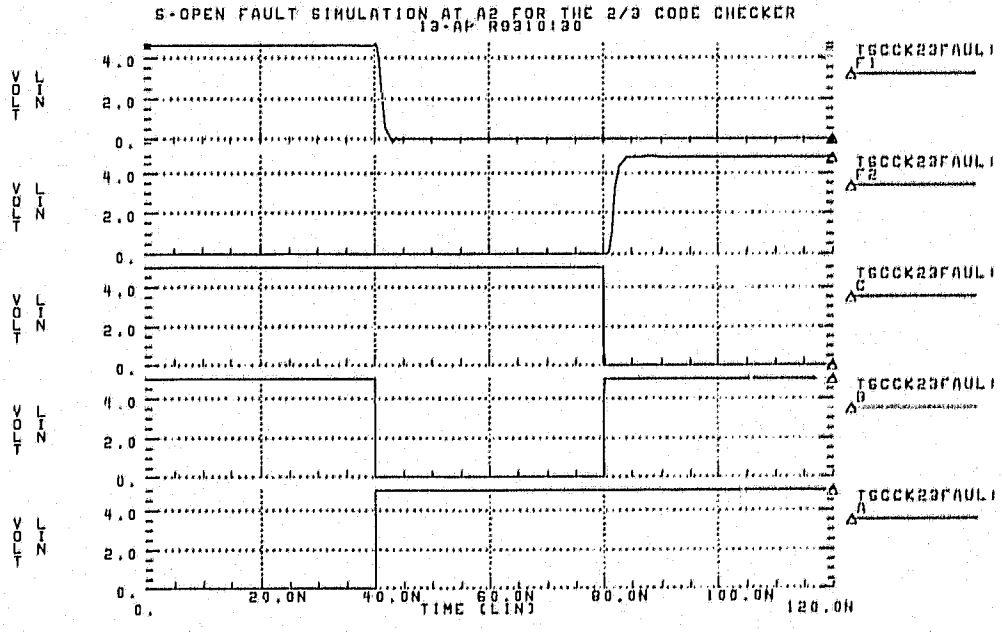


Figure 6.8: Simulation for the 2/3 TSC Checker with Stuck-Open Fault at a2

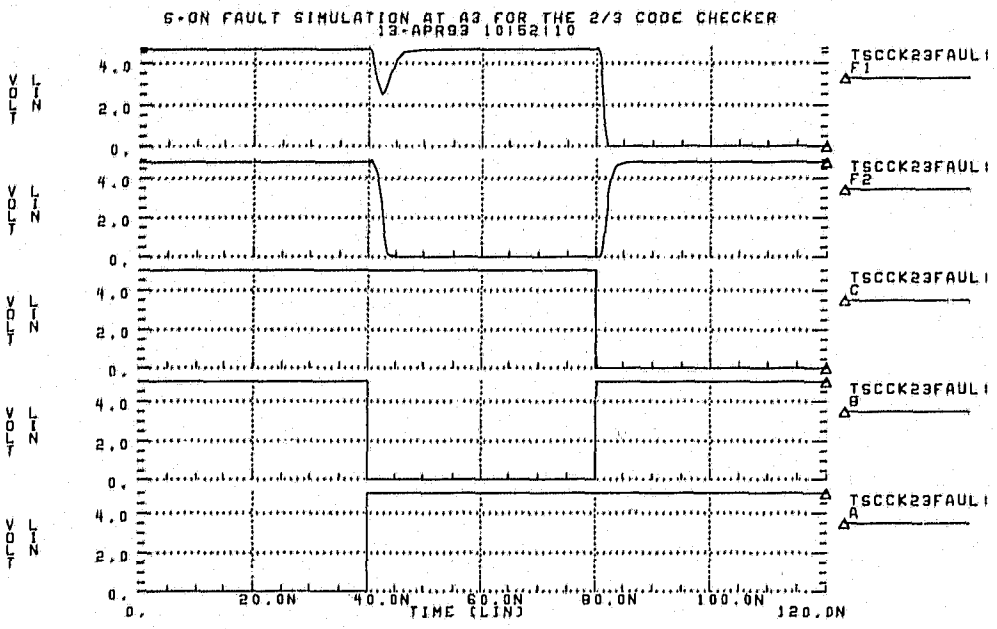


Figure 6.9: Simulation for the 2/3 TSC Checker with Stuck-On Fault at a3

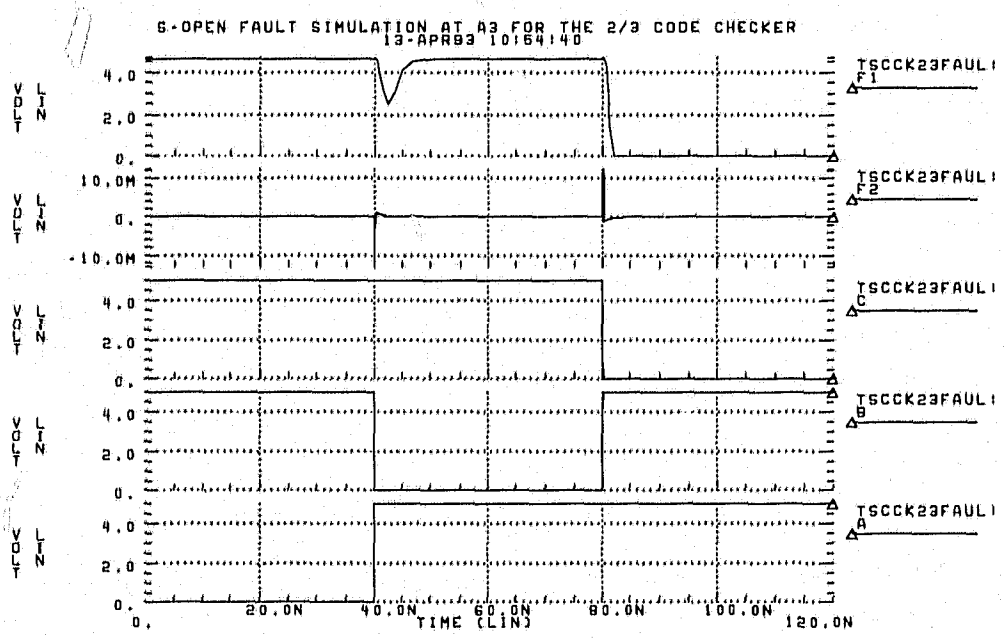


Figure 6.10: Simulation for the 2/3 TSC Checker with Stuck-Open Fault at a3

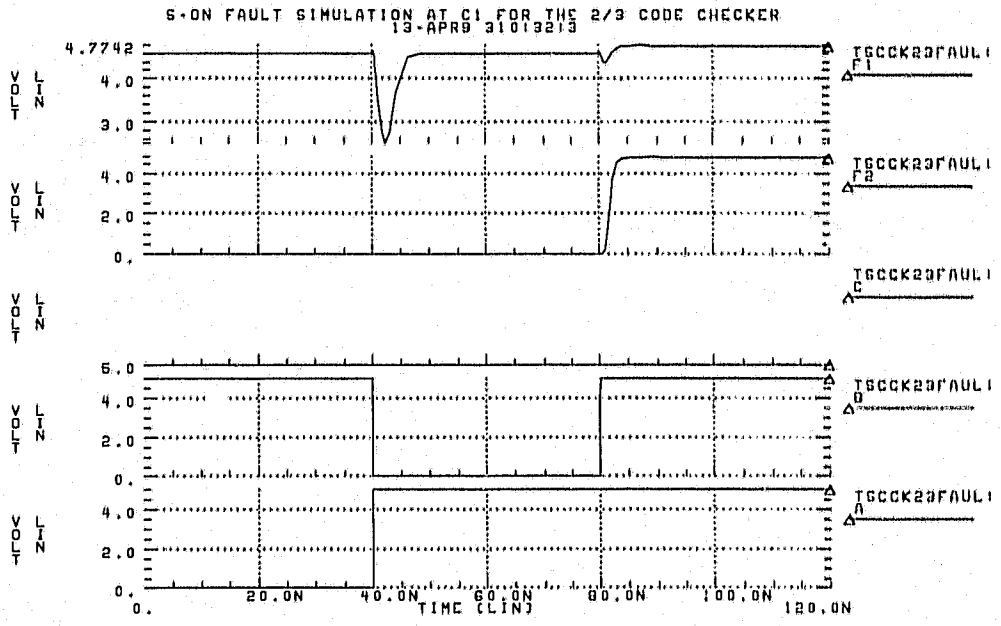


Figure 6.11: Simulation for the 2/3 TSC Checker with Stuck-On Fault at c1

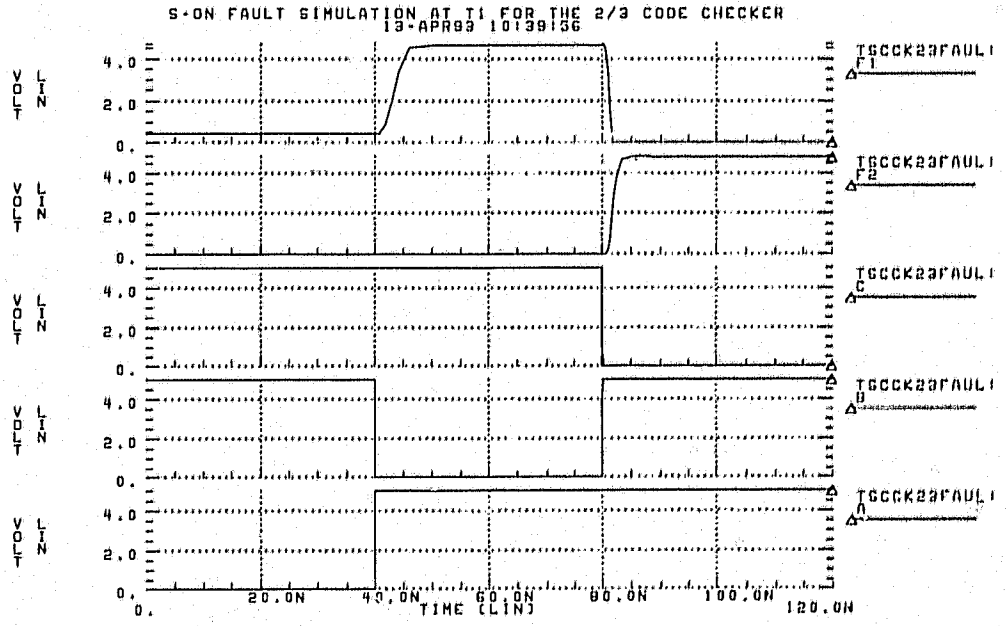


Figure 6.12: Simulation for the 2/3 TSC Checker with Stuck-On Fault at t1

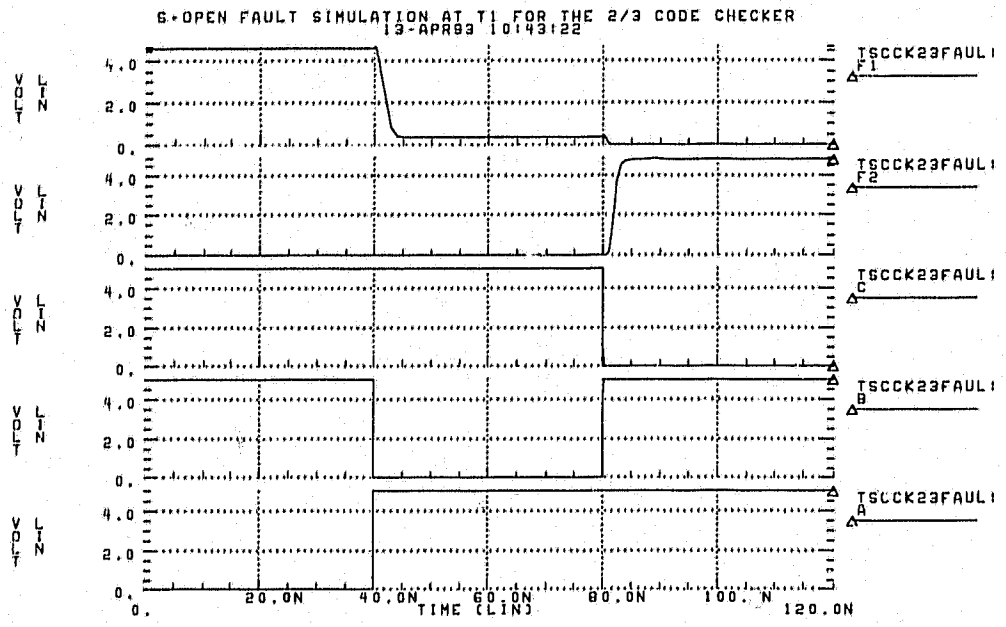


Figure 6.13: Simulation for the 2/3 TSC Checker with Stuck-Open Fault at t1

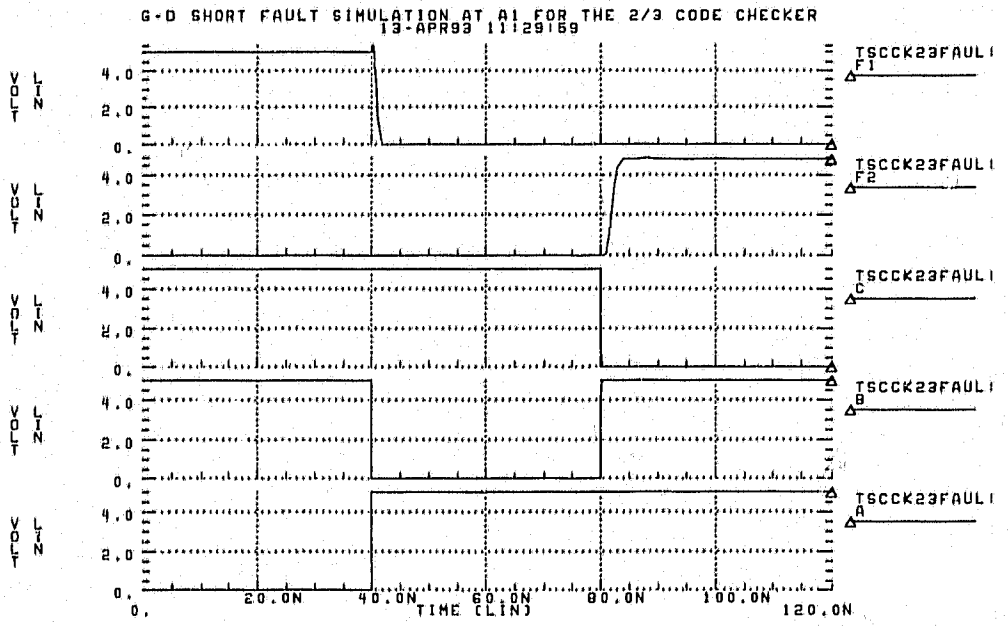


Figure 6.14: Simulation for the 2/3 TSC Checker with Gate-Drain Short at a1

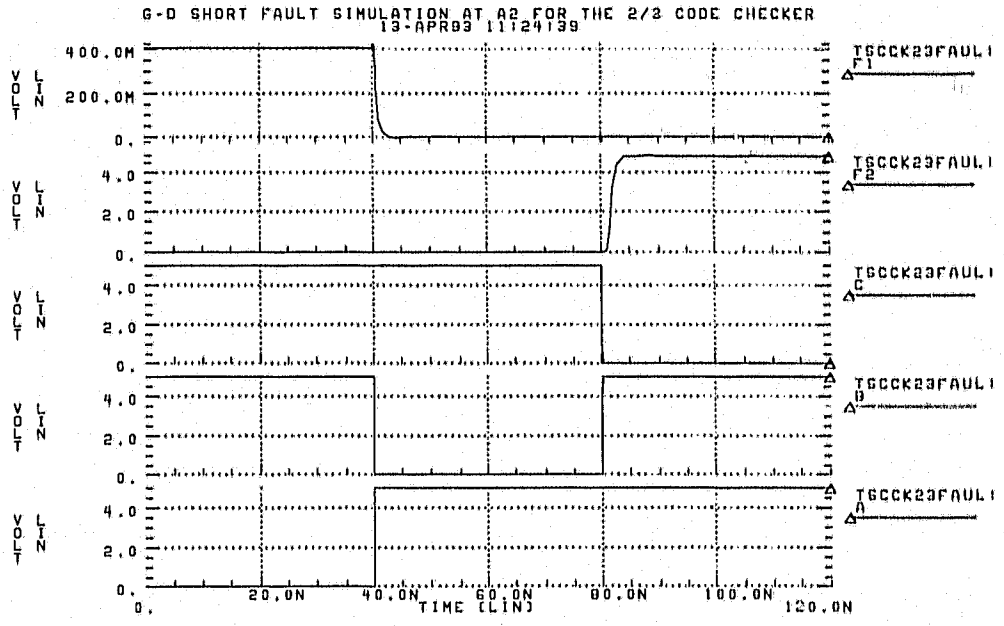


Figure 6.15: Simulation for the 2/3 TSC Checker with Gate-Drain Short at a2

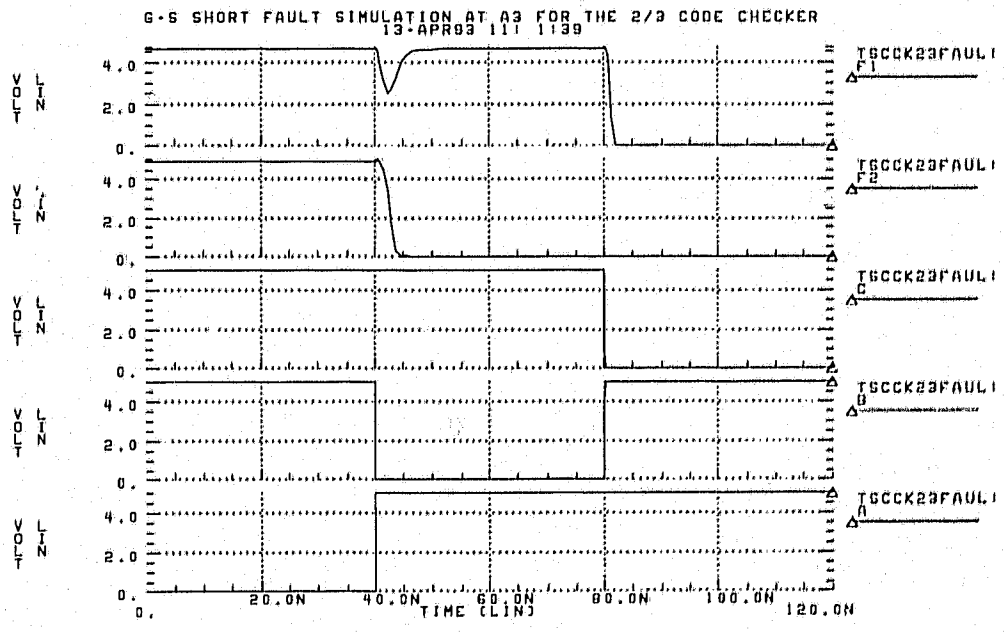


Figure 6.16: Simulation for the 2/3 TSC Checker with Gate-Drain Short at a3

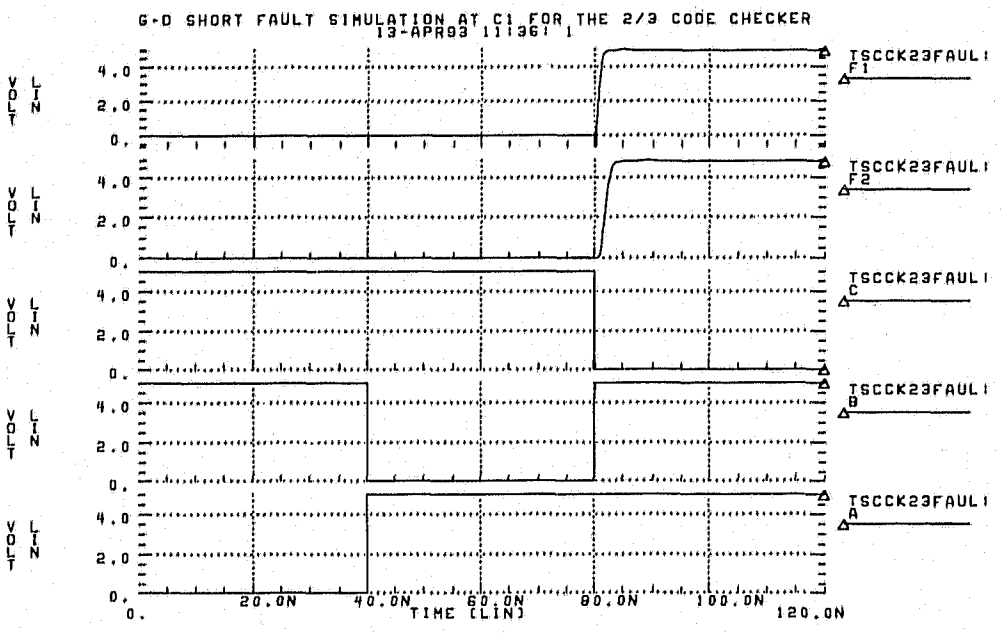


Figure 6.17: Simulation for the 2/3 TSC Checker with Gate-Drain Short at c1

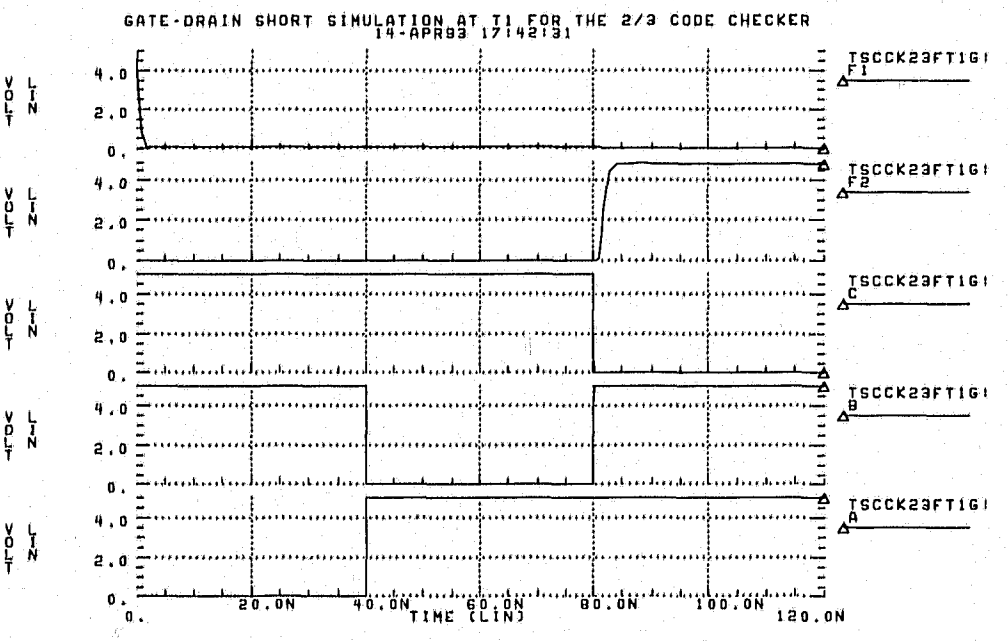


Figure 6.18: Simulation for the 2/3 TSC Checker with Gate-Drain Short at t1

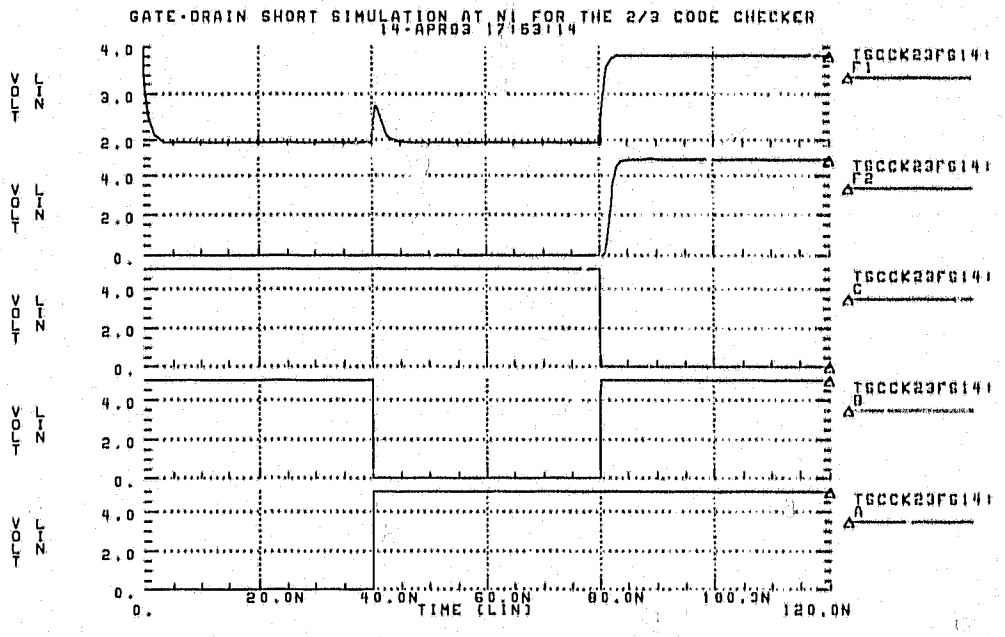


Figure 6.19: Simulation for the 2/3 TSC Checker with Gate-Source Short at p1

checker is the non-codeword 'all-one', all the signals on f_1, f_2, g_1, and g_2 are 1's so that the 7/8 checker output is 11. Suppose that the tree-style checker has an input codeword which has only one 0. After partitioning, one of the leaf checkers has inputs with one 0, and its output is 10 or 01. The other leaf node must have 'all-one' on its input and generate an output of 11. Thus, there is only one 0 on f_1, f_2, g_1, and g_2. The codeword formed by f_1, f_2, g_1, and g_2 is a 3/4 codeword as an input of the top level TSC checker. the checker output (E_1E_2) is a 1/2 codeword 01 or 10. If there is more than one 0 in a codeword on the tree-style checker input, there are two cases to consider:

CASE 1: after the input codeword is partitioned into two parts for two leaf nodes, the input codeword on one of the two leaf checkers has more than one 0. That TSC checker must produce 00 on its output so that the codeword on f_1, f_2, g_1, and g_2 is a non-codeword of the 3/4 code with at least two 0's. When this non-codeword is applied to the top-level checker, the tree-style checker

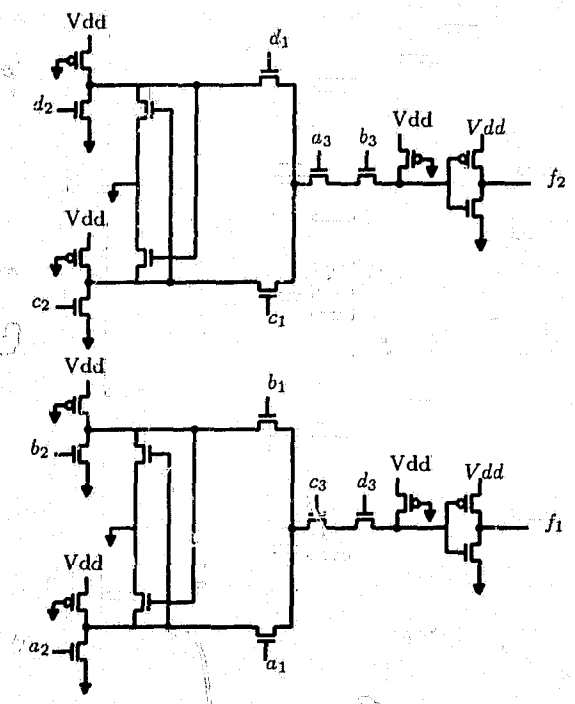


Figure 6.20: The 3-out-of-4 TSC Checker

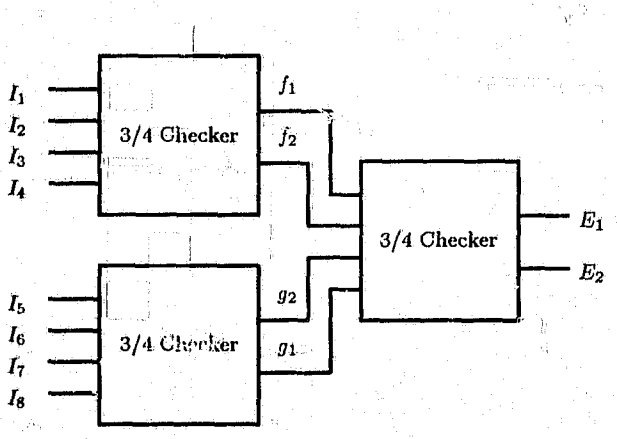


Figure 6.21: A Tree Configuration of a TSC Checker for a 7/8 Code

generates 00.

CASE 2: if each of the leaf TSC checkers has exactly one 0 on its input, f_1f_2 and g_1g_2 are both either 10 or 01. The checker produces 00.

In general, suppose there is a tree-style checker for an (n-1)/n code with a codeword at its inputs. We form the outputs of TSC checkers at each level in the tree as (p-1)/p codewords for some p, $2 \leq p < n$ (in the above example, there are two levels and the output of the bottom level is a 3/4 code). At any time, only one TSC checker at each level can have a valid input after partition of a valid input for this level and this checker produces 10 or 01. Other TSC checkers at the same level must have 'all-one' on their inputs and produce 11. Therefore, the output from this level is a (p-1)/p codeword. If there is a non-codeword with more than one 0 at the input of a level, as shown in the above example, the output from this level must be a non-codeword with two 0's or more. If the input codeword at a level is 'all-one', each checker must have 'all-one' on its input. Consequently, the output of this level must be 'all-one'. Hence, a tree-style checker remains code-disjoint.

If there is a single fault occurring in any TSC checker, the faulty TSC checker must produce a non-codeword 00 or 11 on its output for some input codewords, depending on the fault, while the other fault-free TSC checkers at the same level produce 11 on their outputs (since their inputs must be all-one's the input at this level is a codeword). Therefore, the output from this level must be a non-codeword (either 'all-one' or a codeword with two 0's). Because the tree-style checker is code-disjoint, the following levels generate non-codeword outputs, as each of them has a non-codeword input. Thus, the tree-style checker produces a non-codeword at its output. The defined faults occurring on connection lines between levels in a tree-style checker are equivalent to the faults on the inputs of the TSC checkers to which the faulty lines connect. These faults also cause nonvalid outputs on the level of TSC checkers; and, in turn, the tree-style checker produces a non-codeword.

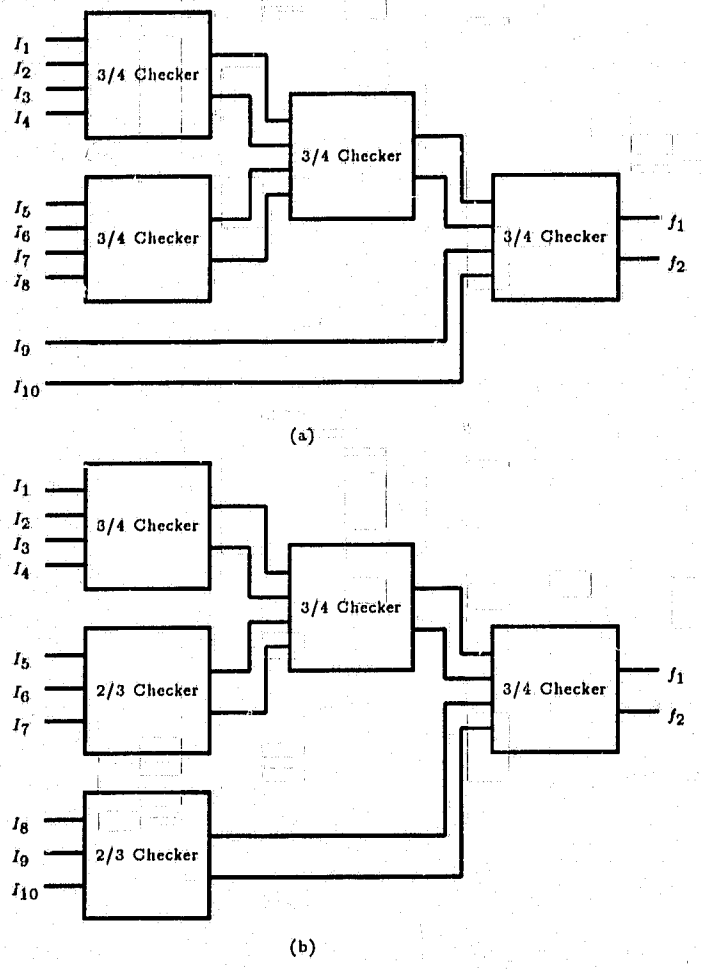


Figure 6.22: Two configurations of a TSC Checker for a 9/10 Code

In order to detect all possible defined faults in a checker for an (n-1)/n code, n input test patterns (input codewords) are required. Thus, a tree-style checker is also fault-secure and self-testing. Hence, such a checker is totally self-checking.

For the basic TSC checkers, the 2/3 and 3/4 code TSC checkers need only 11 and 18 N-type transistors, respectively. These basic TSC checkers have one logic gate time delay. The 1/3 and 1/4 code TSC checkers can be obtained by attaching an inverter to each input of the TSC checker, with resulting costs of only 14 and 22 N-type transistors, respectively. If a TSC checker is implemented as a tree, the number of transistors is $N_t = \sum_{i=1}^{j} n_i$, where j is number of nodes (checkers) in the tree and

Schemes	1/3 Code Checker			2/3 Code Checker		
	T_t	T_n	Gate Delay	T_t	T_n	Gate Delay
David	48+4m	35+4m	7+t	52+4m	37+4m	6+t
Golan	44	32	3	44	32	3
Paschalis	30	22	3	32	22	3
Tao	17	10	2	11	7	1
Lo	11	9	1	17	12	2
Proposed	23	14	2	17	11	1
T_t is the total number of both P-type and N-type transistors T_n is the cost for N-type transistors						

Table 6.3: Comparisons for the Methods for the 1/3 and 2/3 Codes

n_i is number of transistors used in TSC checker i. Since there are many possible tree configurations for an (n-1)/n code checker, construction of a tree-style TSC checker with basic TSC checkers is very important to both time delay and hardware cost. For example, Fig. 6.22 depicts two configurations of a 9/10 code checker with different costs. The time delay for a tree-structured TSC checker depends on the number of levels in the tree. The total time delay is $T_t = \sum_{i=1}^{L} t_i$, where L is the number of levels in the network and t_i is the maximum time delay in level i. Thus, it is preferable for the tree to have as few levels as possible and each level of a network to have as many TSC checkers for the 3/4 code as possible. In order to build a checker for a 9/10 code, for example, the configuration in Fig. 6.22 (a) is preferable to that in Fig. 6.22 (b).

6.4 Comparison of TSC Checkers

In this section, we compare the proposed TSC checkers for both the 1/3 and 2/3 codes with other designs. For the comparison of TSC checkers for the 1/3 and 2/3

codes, the proposed checker is implemented in CMOS whereas other designs are either based on gate level or implemented in nMOS. The faults considered in the proposed TSC checker design are the most comprehensive ones at switch level, and the TSC properties can hold in the checkers with respect to any the faults or fault sequences. Table 6.3 shows the comparison of five schemes for the TSC 1/3 (2/3) code checker for hardware cost in transistors and time delay at gate level. In the table, we consider the TSC checkers for both the 1/3 and 2/3 codes. We also list the costs where only the N-type transistors in circuits are taken into account.

The delay of the proposed checker for the 2/3 code is approximately that of a 3-input AND gate, considered as one gate delay. The corresponding 1/3 code checker requires two-gate delays by inverting the inputs of the 2/3 checker. The numbers of transistors and the corresponding gate delays in the MOS implementations of the 1/3 code checkers for both David's and Golan's designs are those from [79]. In David's design, we assume that m transistors and t time delay are needed for each delay element. In both designs of Golan and Paschalis, we assume that the 1/3 code is merged with the 1/2 code, the simplest of the m-out-of-n codes. As mentioned before, only stuck-at faults was considered in these designs. We can see that the proposed design requires fewer transistors than these three schemes. It also has less time delay for the gates than these schemes. Tao's nMOS implementation needs fewer transistors. However stuck-on faults in pull-down transistors and interconnection faults are not included in the fault model. Lo's design requires fewer transistors, but some single faults are undetectable; more importantly, the self-checking properties could be defeated by some sequences of three faults. None of these TSC combinational checking schemes is implemented in CMOS technology. Furthermore, the proposed design can be used to build TSC checkers for (n-1)/n codes using the TSC checkers for the 2/3 code and 3/4 code. The proposed design takes as many physical defects, which frequently occur in MOS implementations, as

possible into account, whereas other combinational checkers for 1/n codes proposed by Anderson, [69], Reddy, [70], and Smith, [71], only consider the single stuck-at faults which is not sufficient for realistic circuits testing in MOS implementations.

6.5 Summary

Using VLSI pass-transistor logic, we propose a new design that provides combinational TSC checkers for 1-out-of-n codes in CMOS technology, which can be used, especially, to build the TSC checker for the 1-out-of-3 code. The fault model considered in the proposed TSC checker design is very comprehensive, consisting of most physical defects which are likely to occur in MOS implementations, and the checkers retain the TSC properties for any the faults or fault sequences. Both analysis and circuit simulation show that the proposed checker is a TSC checker with respect to the faults in the model. Although our design is more suitable for TSC checkers for (n-1)-out-of-n codes, it is also used for TSC checkers for 1-out-of-n codes — the complement of (n-1)-out-of-n codes — by inverting the inputs of the (n-1)/n code checkers. In the next chapter, we discuss an off-line testing algorithm for the cache management unit, which has a linear test time complexity. We also describe a variant of the proposed algorithm which is suitable for a built-in self testing cache management unit.

Chapter 7

Testing Algorithm

Advances in VLSI technology allow circuits, especially memory, to become larger and denser in single chips. The testing of such circuits has become a major problem. There have been many test algorithms for testing *static random access memory* (SRAM), [88, 89, 90, 91, 92, 93] as well as some algorithms for multi-port SRAM, [92, 93]. There have also been several test methods for testing *content addressable memory* (CAM) published in the past five years, [94, 95]. However, there is little literature published so far to support the testing of a cache management unit. The cache directory in the proposed management unit differs from both CAM and multi-port RAM in the following ways:

- A write/read operation is on a tag (a word). That is, all bits in a tag are read or updated at the same time.
- Several tags are checked for a match with a requested memory address at the same time (n-way associativity).
- If the tags in a set are full, there is a replacement algorithm to select a cache line (block) to be replaced with the requested line. Which line is replaced can

be predicted but cannot be controlled by the outside world.

In addition, the directory of the cache differs from CAM in that a tag in the dual-port directory cache has two access ports, instead of one port that traditional CAM has, while it differs from the multi-port RAM in that once addresses are written into tags in the cache directory they are never read out until they are purged. Thus, the proposed cache memory management unit is more difficult for testing and error detection. The existing test algorithms for CAM and/or for multi-port RAM cannot be simply transferred to test the cache management unit. In this chapter, we discuss the testability of the proposed cache management unit.

The faults in the fault model discussed in Chapter 5 are quite comprehensive and try to cover most of the typical faults that can be encountered in the cache directory. A test strategy to be discussed tries to cover the faults in the model at reasonable cost. It is applicable not only to post production testing, together with other test techniques such as the scan design, but also to service or maintenance testing in the field. Although this strategy is designed for the proposed cache, we hope that it should also be useful in the development of test algorithms for other cache implementations. There are several possible approaches for off-line testing of the cache memory management unit:

- implement a special bus to increase the testability of the directory, as shown in Fig. 7.1, so that the tag in a set to be overwritten with an address can directly be controlled from outside. Thus, a test algorithm can be developed to test the cache directory using the corresponding test pattern set discussed in Section 7.1 One of major advantages is easy implementation of an efficient testing algorithm. The main disadvantages are the requirements of extra hardware, which is dedicated to the testing of the directory, and extra pins, which are often critical, to input the testing control signal.

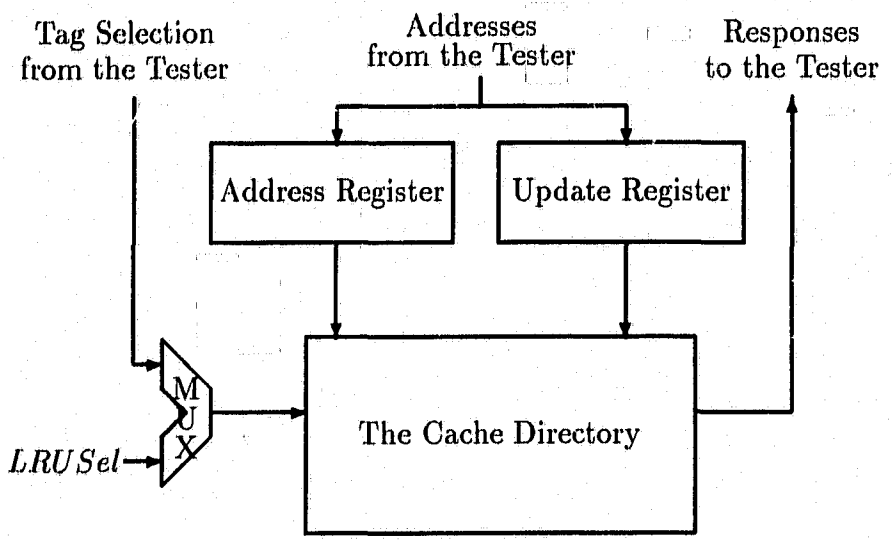


Figure 7.1: The Test Implementation for an External Tester

- use built-in self test (BIST) techniques for the test of the unit. The main advantage is that testing can be completed within the cache memory management unit at speed without outside control; and no extra pins for the testing control signals, except a pin for selection of test mode/operation mode, are required. One of the major disadvantages is the extra hardware, which is only used to support the test of the directory.

- develop a testing algorithm based on the hardware already in the cache memory management unit, including the LRU. During the testing procedures, by predicting which tag is to be overwritten for a new address during a line miss, a selected pattern for testing can be sent into that tag. The functional correctness of the tags can be verified through the *Hit*, *Miss*, and error signals from the on-line concurrent checking mechanisms. The testing algorithm would be more delicate since the testing patterns have to be carefully ordered so that all faults in the fault models can be detected. But it requires no extra hardware or pins which are dedicated to testing. Therefore, the hardware overhead re-

mains low. If the testing algorithm is carefully designed, the time complexity can be optimal.

In the next sections, we present an efficient method to generate the test pattern sets for the cache management unit. Then, based on the test pattern sets, we develop a test algorithm with linear time complexity to test the unit. The algorithm can be used by either the associated processor in a multiprocessor system or a special external tester. Finally, a BIST implementation of the cache management unit is discussed, which requires less hardware overhead for testing purposes, compared to the traditional BIST for other circuits.

7.1 Test Pattern Generation

In order to generate test patterns for the cache management unit, without losing generality, we assume that tags in the unit have a length m varying from 8 bits to 32 bits. To detect coupling faults in an m-bit tag, any pair of two cells c_i and c_j in a tag, where $1 \leq i, j \leq m$ and $i \neq j$, has to exercise 4 states 00, 01, 10, 11. Furthermore, to test pattern-sensitive faults in an m-bit tag, any three adjacent cells c_{i-1}, c_i, and c_{i+1}, where $2 \leq i \leq m-1$, have to exercise 8 states 000, 001, 010, 011, 100, 101, 110, and 111. Table 7.1 shows all the initial test patterns for tags with lengths from 8 to 32 bits. The test patterns for a tag of a given length are generated by first setting a corresponding initial test pattern shown in the table and then left-shifting it cyclically bit by bit. The initial patterns basically consist of four sub-patterns: 0011, 1100, 0101, and 1010. Patterns 0011 and 1100 are used to guarantee any two adjacent bits in a tag experience state 00 and 11 during shifting while patterns 0101 and 1010 make any two adjacent bits have states 10 and 01. In the table, there are three cases of initial test patterns in terms of the range of tag lengths.

Tag Length	Initial Pattern	Number of Patterns
Case One ($8 \leq m \leq 16$) with the base pattern 1100010110100011		
8 bits	11000101	8
9 bits	110001011	9
10 bits	1100010110 *	11
11 bits	11000101101	11
12 bits	110001011010 *	13
13 bits	1100010110100 *	14
14 bits	11000101101000 *	15
15 bits	110001011010001	15
16 bits	1100010110100011	16
Case Two ($17 \leq m \leq 24$) with the base pattern 00111010 1100010110100011		
17 bits	00111010 110001011 **	11
18 bits	00111010 1100010110 *	11
19 bits	00111010 11000101101 **	13
20 bits	00111010 110001011010 *	13
21 bits	00111010 1100010110100 *	14
22 bits	00111010 11000101101000 *	15
23 bits	00111010 110001011010001 *	16
24 bits	00111010 1100010110100011 *	17
Case Three ($25 \leq m \leq 32$) with the base pattern 1100010110100011 0011101000110101		
25 bits	1100010110100011 001110100 *	17
26 bits	1100010110100011 0011101000 *	17
27 bits	1100010110100011 00111010001 *	17
28 bits	1100010110100011 001110100011 *	17
29 bits	1100010110100011 0011101000110 *	17
30 bits	1100010110100011 00111010001101 *	17
31 bits	1100010110100011 001110100011010 *	17
32 bits	1100010110100011 001110100011010 *	17
* means there is an additional pattern all-1 in the test pattern set besides the patterns produced by shifting. ** means there is additional patterns all-1 and all-0 in the test patterns.		

Table 7.1: The Initial Patterns to Generate the Test Patterns for Tags

As shown in Table 7.1, Case One is suitable to generate the initial test patterns for the tags of 8—16 bits($8 \leq m \leq 16$). In this case, the initial patterns are based on a base pattern 1100010110100011 by fitting the base pattern with the tag length from the left. For example, the initial pattern for an 8-bit tag is 11000101 while the initial pattern for a 13-bit tag is 1100010110100. In order to obtain all the patterns for an m-bit tag ($8 \leq m \leq 16$), the initial pattern is cyclically left-shifted bit by bit $m-1$ times to generate m patterns. This sequence of patterns makes any two bits in the tag exercise all the four states to capture any coupling faults. It also has any three adjacent bits in the tag to experience all the eight states, but 111 for some m ($m = 10$, 12, 13, and 14) indicated by * in Table 7.1, for pattern sensitive faults. For instance, the sequence of the 8 test patterns generated for an 8-bit tag can be used to detect any coupling and pattern sensitive faults in the tag. However, the test patterns produced by cyclically shifting for a 13-bit tag cannot make three adjacent bits in the tag exercise the state 111. Therefore, for those tags that the state 111 cannot be provided by the corresponding test patterns, a pattern all-1 is added so that any coupling and pattern sensitive faults can be captured during testing. The total number of test patterns used for the m-bit tags ($8 \leq m \leq 16$) varies from 8 to 16, depending on the length of tags in the directory.

Case Two is the initial test patterns for the m-bit tags ($17 \leq m \leq 24$). In this case, a 24-bit base pattern 001110101100010110100011 is used for the m-bit initial patterns. This base pattern is based on two sub-patterns: an 8-bit left base pattern 00111010 and a 16-bit right base pattern 1100010110100011, That is, an m-bit initial pattern consists of the 8-bit left part 00111010 and the right part of $m-8$ bits by fitting the right base pattern 1100010110100011 with $m-8$ bits from the left. For example, the initial pattern for a 17-bit tag consists of 00111010 and 110001011 while a 23-bit initial pattern is formed by concatenating the left part 00111010 with the right part 110001011010001. In order to generate all test patterns for an m-

bit tag, both the left part and right part are simultaneously left-shifted cyclically bit by bit $m-9$ times, respectively, to generate $m-9$ test patterns following the initial pattern. The reason that both the left part and right part of a pattern are cyclically left-shifted independently $m-9$ times to generate the pattern set is that the length of the right part is greater than that of the left part. Therefore, in order to generate all possible distinct sub-patterns for each part using the cyclic-shift, the total number of shifts is at least the number of bits in the right part minus one. That is, the minimum number of left-shifts is $m-9$ for $17 \leq m \leq 24$. However, the pattern of all-1's has to be added into the test patterns for some m-bit tags to guarantee that all the coupling faults and pattern sensitive faults can be detected. These patterns are indicated with * in Table 7.1. For $m = 17$ and $m = 19$, the pattern of all-0's has to be added to detect all coupling faults; otherwise, not all two bits in a tag have state 00 during testing. These patterns are indicated with ** in Table 7.1. In this case, the total number of the test patterns for an m-bit tag ($17 \leq m \leq 24$) is from 11 to 17.

Case Three is for an m-bit tag where $25 \leq m \leq 32$. Similarly, a 32-bit base pattern is used for producing all the m-bit initial patterns. This pattern has two sub-patterns: a 16-bit left base pattern 1100010110100011 and a 16-bit right base pattern 0011101000110101, An m-bit initial test pattern is composed of the left part 1100010110100011 and the right part of $m-16$ bits by cutting the right base pattern 0011101000110101 with $m-16$ bits from the left. In order to generate the following patterns for an m-bit tag, both the left and right part of the initial pattern are simultaneously left-shifted cyclically 15 times, respectively, to generate 15 test patterns following the initial pattern. Also the pattern of all-1's is employed to test all the coupling and pattern sensitive faults. The total number of test patterns for any m-bit tag ($25 \leq m \leq 32$) is equal to 17.

The test patterns generated for a tag varying from 8 to 32 bits are verified

through simulations to detect all the coupling and pattern sensitive faults by checking that any of two bits in a tag have four states for the coupling faults and any of three adjacent bits experience eight states for the pattern sensitive faults. Table 7.2 shows the examples of the test pattern sets for the tags of 8 bits, 19 bits, and 25 bits. Note that the test pattern sequence for a tag has to be used to stimulate the tag, during testing, in a cyclic order of the patterns shown in Table 7.2; but the leading pattern of the sequence for stimulation can be any one in the corresponding test pattern set. For instance, the test sequence for an 8-bit tag can be in the order: $P_5, P_6, P_7, P_8, P_1, P_2, P_3, P_4$, starting from P_5. Moreover, as shown in Table 7.2, any bit in a tag experiences the state transitions 1-0-1 and 0-1-0 at least once. Therefore, it is obvious that the stuck-at faults, transition faults, as well as stuck-open faults in individual cells are covered by a test pattern sequence generated in this order. However, if any changes from the pattern sequence for a tag shown in Table 7.2 may not make all the bits in the tag exercise the state transitions 1-0-1 or 0-1-0. For example, use of a pattern sequence $P_1, P_2, P_8, P_6, P_7, P_5, P_3, P_4$ to stimulate an 8-bit tag cannot guarantee that the bit on the left side of the tag exercises 1-0-1 and 0-1-0 transitions. Thus, the transition faults can not be detected by this sequence. From the above discussion, we see the maximum number of test patterns for detection of all defined faults in a tag is 17 and the minimum number of the test pattern sets is 8 for an 8-bit tag.

7.2 The Test Algorithm

In this section, we discuss a test algorithm which can be used to test the cache management unit by the associated processor in a multiprocessor system without extra overhead. As we know, when there is a request from a processor to the corresponding n-way associative cache, the cache has to simultaneously search all

Pat. No.	8 8-bit Patterns	9 9-bit Patterns	13 19-bit Patterns	17 25-bit Patterns
P_1	11000101	110001011	00000000 00000000000	1100010110100011 001110100
P_2	10001011	100010111	00111010 11000101101	1000101101000111 011101000
P_3	00010111	000101111	01110100 10001011011	0001011010001111 111010000
P_4	00101110	001011110	11101000 00010110111	0010110100011110 110100001
P_5	01011100	010111100	11010001 00101101110	0101101000111100 101000011
P_6	10111000	101111000	10100011 01011011100	1011010001111000 010000111
P_7	01110001	011110001	01000111 10110111000	0110100011110001 100001110
P_8	11100010	111100010	10001110 01101110001	1101000111100010 000011101
P_9		111000101	00011101 11011100010	1010001111000101 000111010
P_{10}			00111010 10111000101	0100011110001011 001110100
P_{11}			01110100 01110001011	1000111100010110 011101000
P_{12}			11101000 11100010110	0001111000101101 111010000
P_{13}			11111111 11111111111	0011110001011010 110100001
P_{14}				0111100010110100 101000011
P_{15}				1111000101101000 010000111
P_{16}				1110001011010001 100001110
P_{17}				1111111111111111 111111111

Table 7.2: Examples of the Test Patterns

the n tags in the given set required by the request. If the contents of the n tags do not match the address from the processor or the tester, there is a line miss so that the LRU selects a line to be purged to make room for the new one. Therefore, which line is to be purged during a line miss is normally controlled by the LRU. Although special signals from outside can be added to directly control the tag to be purged for testing purposes, this adds to the hardware and time overhead. Here we present a solution to avoid this increase in overhead. Although there are many line replacement algorithms, such as the least recently used line relacement algorithm and the pseudo random line replacement algorithm, the line to be purged can be predicted for these algorithms. Therefore, in terms of the prediction, we can arrange

	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8
$Pass_1$	P_1	P_9	P_8	P_7	P_6	P_5	P_4	P_3
$Pass_2$	P_2	P_1	P_9	P_8	P_7	P_6	P_5	P_4
$Pass_3$	P_3	P_2	P_1	P_9	P_8	P_7	P_6	P_5
$Pass_4$	P_4	P_3	P_2	P_1	P_9	P_8	P_7	P_6
$Pass_5$	P_5	P_4	P_3	P_2	P_1	P_9	P_8	P_7
$Pass_6$	P_6	P_5	P_4	P_3	P_2	P_1	P_9	P_8
$Pass_7$	P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_9
$Pass_8$	P_8	P_7	P_6	P_5	P_4	P_3	P_2	P_1
$Pass_9$	P_9	P_8	P_7	P_6	P_5	P_4	P_3	P_2

Table 7.3: An Example of Testing 8 Tags in a Given Set

the test pattern sequence to stimulate the cache under test so that each tag is stimulated by the corresponding test pattern sequence discussed in the previous section. To illustrate the method with use of the prediction, we assume that the cache to be tested employs the least recently used line replacement algorithm and the number of test patterns for an m-bit tag is greater than the number of ways of the tag array; for instance, the number of test patterns for a 9-bit tag is 9 while the number of ways is 8.

Table 7.3 shows how to test 8 tags in a given set. Each row $Pass_i$ $(1 \leq i \leq 9)$ indicates a pass of testing the 8 tags while each column C_j $(1 \leq j \leq 8)$ represents a corresponding tag in the set. We suppose that initially all tags are reset. During $Pass_1$, pattern P_1 is sent to the tag array, which causes a line miss. Since there are no valid addresses in the tags, the LRU selects tag C_1 so that P_1 is written in C_1. Then another pattern, say P_9, is sent to the set in the cache, the cache searches all tags in the given set against P_9 simultaneously and does not find any tags matching it. A line miss occurs. The LRU selects C_2 for P_9; and so on until P_3 is written into C_8. Now the patterns in the tags from C_1 to C_8 at $Pass_1$ in Table 7.3 are

distinct. P_2 is the only pattern in the test pattern set which is not used at $Pass_1$ in this example. Thus, at the beginning of $Pass_2$, P_2 is sent to the given set of the cache. The cache searches all tags in the set to see if any of the tags match P_2, which certainly causes a line miss. The LRU selects C_1 to be replaced since the tag contains the address of the least recently used line in the set. Thus, P_1 is purged from C_1, and P_2 is written into C_1. Note that test pattern P_2 is necessary to keep the test patterns, used to stimulate C_1, in a proper order in this example since the previous pattern in C_1 is P_1. Then P_1, which is now not in the set, is sent to the cache, and a line miss occurs. C_2 is chosen to be updated with P_1 according to the line replacement algorithm. The following pattern is P_9 to be used to update C_3; and then P_8 for C_4, and so on until C_8 is updated with P_4. Thus, $Pass_2$ is finished. Now P_3 does not reside in the set, and it is used to start $Pass_3$. We continue sending the corresponding test patterns into the set by repeating the above procedures until $Pass_9$ in Table 7.3 is completed. Thus, all the tags in the given set are tested with the required test patterns in the proper order. The reason that the number of test patterns for an m-bit tag is required to be greater than the number of ways in the tag array is that, at any time during the test of a given set, at least one pattern in the required test pattern set is not in tags in that set so that this pattern can be used to continue the test procedures. In order to test all tags in the tag array, we can fill up all the tags for $Pass1$ in an order of set 0, then set 1, and so on until the last set Q. Then we can test the tags for $Pass_2$ in the same order, and so on until the last pass $Pass_9$ is finished.

In order to verify that a tag is functionally correct, after each pattern is written into a tag under test in a given set, the same pattern is sent to the cache management unit once more. The unit searches tags in the set, and the searching result causes either a line hit or a line miss. If there is a line hit, after the pattern is sent out, the tag under test is working properly with this pattern. Otherwise, a line miss

indicates there are faults in the unit. Thus, these testing procedures can detect all the defined faults in tags except the multiple access faults and retention faults. The time complexity is $16N_{tag}$ to $34N_{tag}$(including testing and verification), depending on the length of the tags in the tag array. Note that N_{tag} is the number of the tags in the tag array, rather than the number of memory cells which is commonly used to evaluate memory testing algorithms. A complete pseudo test procedure is given to test the faults as follows:

Pseudo Testing Procedure 1

FOR each *Pass* **FROM** one **TO** the max. number of patterns **DO**
  FOR each *Set*(Row of the tag array) **FROM** 0 **TO** $Q-1$ **DO**
    FOR each *Column* **FROM** one **TO** the max. number of ways **DO**
      send out a proper test pattern to update
        the corresponding tag in the *Column*;
      verify that tag by sending out the same test pattern;
    END of *Column*;
  END of *Set*;
END of *Pass*;

If, after each pass, the next pass is delayed for a certain period, the data retention faults in tags can also be captured by this strategy. The multiple access faults in the tag array can be tested by the following procedure. Suppose that two rows of the tag array can be selected by sets i and j where $i \leq j$, respectively, if there is not a multiple access fault. Because the multiple access faults are non-symmetric, there are two possible cases for accessing to the two rows:

Case A: set address j can access both rows while set address i can access the corresponding row.

Case B: set address i can access both rows while set address j can access the corresponding row.

In order to detect the multiple access faults, the following procedure is carried out:

Pseudo Testing Procedure 2

1. write all the tags in the directory with distinct addresses in an order from set 0 to $Q-1$.

2. using the previous written addresses, search each set, from 0 to $Q-1$, to see if the expected addresses reside in the tags of the set by examining the signal *Hit*. If so, write a new address in the set and continue this step until the last set is searched. Otherwise, Case A or Case B is happening and detected.

The procedure 2 has a time complexity of $3N_{tag}$.

7.3 Testing Other Faults in the Directory

Permanent faults in the address register and the update register, address lines from the registers to the tag array, comparators for both the processor operations and the coherence operations in the tag array, the lines $\overline{ColMatch}$s and $\overline{ColMatch}$'s, and the lines *LRUSel* from the LRU can be detected by the hardware designed for fault tolerance during the above testing procedures. During the testing procedures, whenever each pattern is sent to the address register for verifying a tag in a given set with the searching results on the *ColMatch*s, it is sent simultaneously to the update register as well so that the tag under test is also checked against the update register with the comparison results on the *ColMatch*'s. These two results are checked immediately by the comparator checker, as shown in Fig. 3.3. If there is

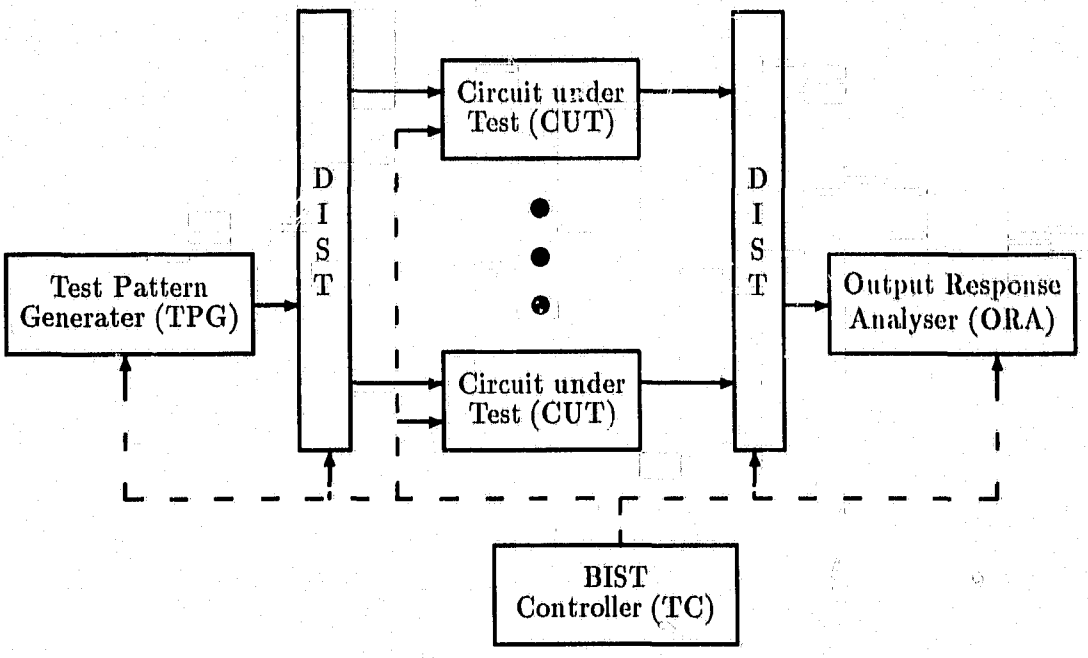


Figure 7.2: Generic Form of Centralized and Separate BIST Architectures

any inconsistency, the checker will send out an error signal via the error flag. As soon as an error signal is received, a fault is detected. Furthermore, the checker for LRUSels also monitors the operations of the LRU. Any errors detected by the checker set the error flag. Therefore, during the testing procedures, the error flag, *Hit*, and *Miss* signals are observed.

In summary, in order to detect all the defined faults in the cache management unit, the two proposed test procedures have to be used. That is, the procedure 2 is applied after procedure 1 finishes. Thus the total time complexity is from $19N_{tag}$ to $37N_{tag}$ where N_{tag} is the number of the tags in the directory, depending on the tag length in the tag array.

7.4 The BIST Implementation

In the previous section we discussed a testing algorithm for external testing for the proposed cache management unit. In this section, we discuss a Built-in Self-Test (BIST) implementation of the cache management unit. BIST is a design method in which parts of a circuit are used to test the circuit itself. That is, testing is incorporated into circuits at the design stage so that testing can be completed without the help of special testing equipment. Usually there are two types of BIST techniques: on-line BIST and off-line BIST(or implicit testing and explicit testing). On-line BIST refers to the concurrent checking or concurrent testing which is used to detect errors that occur during normal system operation. We have discussed the design for the concurrent checking in the cache management unit in Chapter 5. In the discussion of this section, the BIST mean the off-line BIST which tests a circuit when it is not carrying out its normal operation.

A general BIST(off-line) structure at chip and board level is illustrated in Fig. 7.2. It consists of the following key elements in a circuit:

1. test-pattern generators (TPG).
2. output-response analysers (ORA).
3. the circuit under test (CUT).
4. a distribution system (DIST).
5. a testing controller (TC).

The TPGs are used to generate test patterns to stimulate the circuit under test. The test patterns generated by TPGs are usually pseudo-random binary patterns the number of which is greater than that of deterministic test patterns, given a specified fault coverage. The ORA compares the response of the CUT to reference

patterns for fault-free circuits and indicates if the circuit is faulty or fault-free. A DIST is employed for transmitting patterns from TPGs to CUTs and/or from CUTs to ORAs. The test controller is to control the operations of all the components in the circuit during self-test. The basic BIST operations are that the controller tells the TPGs to generate patterns and has the DIST to transmit the patterns to the inputs of CUTs. The outputs of the CUTs are transmitted to ORAs through the DIST; and the final decision whether the CUTs are faulty or fault-free is made by the ORAs at the end of test.

BIST techniques for RAM have only recently become of interest. They can reduce test generation cost because low-cost pseudo-random number generators are available. They provide an alternative, to the costly automatic test equipment, which enables circuits to be tested at speed. They also provide memory chips, in a simple way, to spend a minimal amount of time for tests performed during their operational life. However, the implementation of BIST techniques in chips has disadvantages:

1. it increases hardware overhead for testing in memory chips. The silicon area required for the BIST circuitry reduces the area available for memory cells.

2. It may affect the memory access time since the addition of the BIST circuitry in the chips may increase the chip complexity.

3. It requires chips to have extra pins, which are an extremely-important cost item for chips, for testing because the BIST circuit has to be controlled in both normal operation mode and test mode.

In this section, we explore a BIST method in the proposed cache management unit. In the BIST scheme, the parts (shown as $g + s$) of both the address and update registers can be implemented as left-cyclic-shift registers, as shown in Fig. 7.3, and the other parts operate as counters (counters for set) in test mode. At the beginning

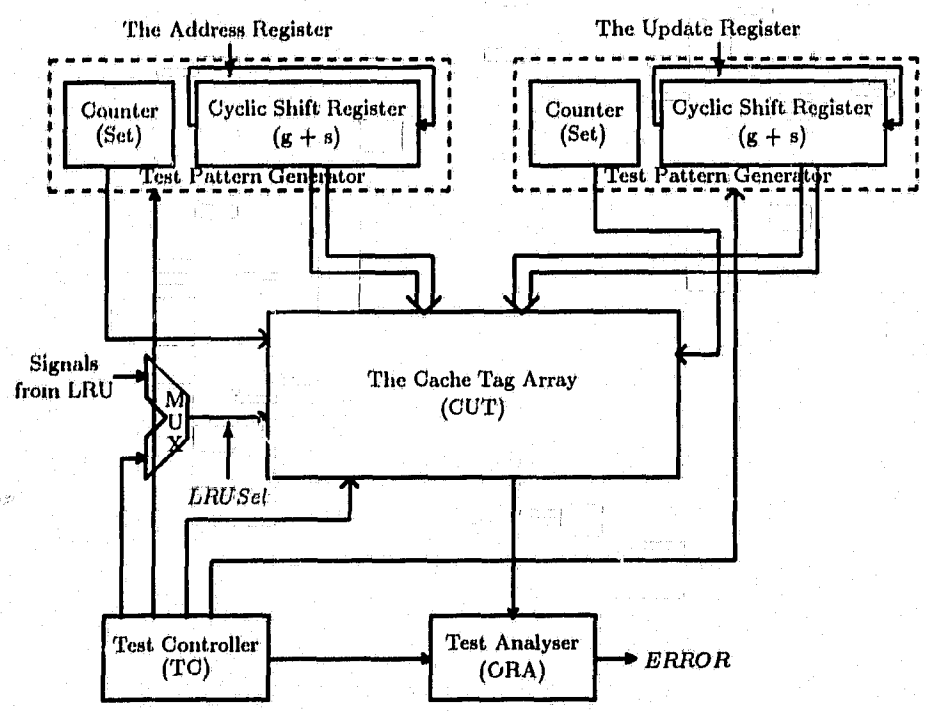


Figure 7.3: The BIST Implementation of the Dual-Port Directory Cache

of a test, both the registers become the shift registers and are set with the same initial test pattern. This pattern is sent into the tag array as the first step of testing a tag. Then they are cyclically left-shifted as discussed in section 7.1 to generate the following test patterns one by one to stimulate the tag. Additional patterns, such as all-zero and all-one, if any, can directly be set into the shift registers. After each pattern is sent to the directory, a line miss occurs, indicated by the *Miss*, so that the pattern is rewritten into a corresponding tag in a given set. That pattern is also sent to the directory once more for verification by observing the *Hit*. Thus, the output analyser is required to monitor the signals *Hit* and *Miss* which needs only 2-bit information.

In the BIST implementation, we test tags in the directory in such a way that one tag is completely tested, except the test for the multiple access faults, before another tag begins to be tested. That is, all the required test patterns for an m-

bit tag have to be generated and sent to a tag; and verification is made for that tag. Then, another untested tag can begin to be tested. In this case, the LRU is disconnected during testing, and the signals $LRUSels$ are directly controlled by the BIST controller. This can be implemented by adding a multiplexer on the wires $LRUSels$, as shown in Fig. 7.3, so that during normal operations the LRU is connected to the directory through $LRUSels$ and during testing, controlled by the test mode, the BIST controller sets the signals on the $LRUSels$ through the multiplexer. The controller, for a given set (a tag array row), selects the tag in the first column of the tag array, and completely tests that tag. Then, the tag in the second column in the given row is selected to be tested, and so on until the tag in the last column is tested. The controller selects another row of untested tags by incrementing the counters for set and testing them using the same procedures as discussed above. The controller repeats these until all the rows in the tag array are tested. Note that the LRU operates as normal operations during testing and can be detected by the checker for $LRUSels$, as shown in Fig. 7.3, though the outputs of the LRU are not used during testing. Therefore, a cache with any line replacement algorithms can be implemented in such a BIST structure.

In order to detect the multiple access faults, both the shift registers become the counters while the counters for set remain as counters. Distinct "addresses" will be generated by the counters and be written into different tags. This is controlled by the BIST controller through both the multiplexer and the counters for sets. After all tags in a given set are written with "addresses", another set can be selected and the tags in this new set start to be written with new distinct "addresses". Until all the tags in the tag array are written, the counters are reset. The above test procedures are repeated after a wait of a certain period of time for testing the retention faults. This time, the operations are the read operations instead of the write operations. After each tag is verified by the read operation, the pattern of all-1's is overwritten

into the tag. We suppose that the all-1's pattern was not used in the first pass of the write operations for the multiple access faults. The time complexity of the BIST scheme is the same as that of the test algorithm in Section 7.2 which is $19N_{tag}$ to $37N_{tag}$, depending on the tag length in the tag array. Note that N_{tag} is the number of the tags in the tag array. The pseudo test algorithm carried out by the BIST controller as follows:

Reset the shift-registers for $g + s$.
FOR each *Set*(Row of the tag array) **FROM** 0 **TO** $Q - 1$ **DO**
    FOR each *Column* **FROM** one **TO** the max. number of ways **DO**
        FOR each P **FROM** one **TO** the max. number of patterns **DO**
            send out a proper test pattern to update
                the corresponding tag in the *Column*;
            Wait for a certain time period to detect retention faults;
            verify that tag by sending out the same test pattern;
        END of P;
    END of *Column*;
END of *Set*;
Reset the counters for $g + s$;
FOR each *Set*(Row of the tag array) **FROM** 0 **TO** $Q - 1$ **DO**
    FOR each *Column* **FROM** one **TO** the max. number of ways **DO**
        send out an "address" from the counter for $g + s$
            to the corresponding tag in the *Column*;
        the counter for $g + s$ increments;
    END of *Column*;
END of *Set*;
Reset the counters for $g + s$ again;
FOR each *Set*(Row of the tag array) **FROM** 0 **TO** $Q - 1$ **DO**

FOR each *Column* **FROM** one **TO** the max number of ways **DO**
verify an "address" in the counter for $g + s$
with that in the corresponding tag in the *Column*;
write an all-1 in the tag under test;
the counters for $g + s$ increments;
END of *Column*;
END of *Set*;

As discussed above, since the test pattern generation is simple, the TPG, based on the address and update registers, requires little extra hardware. Since there is only one CUT the distribution system is not needed. Moreover, no data compression in the ORA is required. The ORA needs only to observe the *Hit* and *Miss* during testing to verify the functional correctness of the directory. Therefore, faults can be detected at any time of the test, instead of checking the signatures at the end of testing. The BIST scheme generates a smaller number of test patterns than that using a pseudo random test pattern generator because of deterministic test pattern generation. Testing time may be relatively less than the normal BIST approaches and fault coverage is relatively higher. Hence, this scheme eliminates most of the disadvantages that a usual BIST has so that it has less hardware overhead and is faster for testing. The potential disadvantage is that the complex registers and the addition of a multiplexer between the LRU and the directory may increase the cache cycle for normal operations. Careful design and implementation of these components can limit this problem to the minimum. Moreover, an extra pin is required to select the test mode or normal operation mode for the cache management unit.

7.5 Summary

The proposed cache management unit differs from the traditional cache management units. In this chapter, a new optimal test algorithm with a linear test time complexity is presented which can be used to test the cache management unit by either the associated processor in a multiprocessor system or external test equipment. An efficient variant of the proposed algorithm which is suitable for the BIST cache management unit is also discussed. The hardware overhead of such a BIST scheme is much less than a traditional BIST implementation for other circuits. Furthermore, this BIST scheme spends less time for self-testing. The test algorithm and the BIST implementation can also be employed for a single directory cache. In this case, instead of the comparator checker used in the dual-port directory cache, another TSC 1/n checker, which is the same as the checker for $LRUSel$, can be attached to the n $\overline{ColMatch}$s which is also formed as a one-out-of-n code. Thus, all the defined faults in a single directory cache management unit can be detected with the proposed algorithm with little change.

Chapter 8

Conclusion

8.1 Design and Evaluation

In this dissertation, using VLSI technology, we propose a new multiprocessor cache. The cache has one single dual-port directory which can be searched for both the processor accesses and coherence operations simultaneously. This cache has a protocol-independent structure so that any of the standard data coherence protocols can be implemented. About 33% extra hardware, compared to a single directory cache, is needed for the dual-port directory cache required for high performance, which is far less than that required by a two-directory cache (normally over 100% extra hardware). Furthermore, the overall cycle time of a dual-port directory cache may be shorter than that of a two-directory cache because no arbitration is required.

In order to evaluate the cache performance in a multiprocessor environment, two simulation models of multiprocessors with both dual-port-directory caches and single directory caches are created, respectively. Strategies and structures of the cache memory, shared main memory, and multiple buses used in simulation have been discussed. In the proposed cache, an n-way set-associative mapping is employed for

address translation while the least recently used line replacement algorithm is used for line replacement after a cache miss. The write-through with updating algorithm is employed to keep the shared information in the multiprocessor system coherent while the write-back policy is used for private data. The structure of the simulator and workload are described. During simulation, the processors generate the memory reference streams at random with random-generation for write rates and program locality. The system performance has been extensively simulated. Based on the simulation results, the performance improvements made by the use of the proposed caches are discussed. Furthermore, we investigate the effects of the write reference rate, data sharing rate, multiple buses, as well as cache parameters such as the cache size, line size and way size on the multiprocessor system performance.

The simulation results show that the multiprocessor system with dual-port directory caches has higher performance than that obtained by a system with single directory caches. Sharing of data affects greatly the system power for both the dual-port-directory cache system and the single-directory cache system, though the system power obtained by the former is still higher than that of the latter. The effects of write rates in reference streams on the system performance show that, because each write request updates the main memory via the bus system under the buffered write-through protocols for shared data, the workload of the bus system increases when the overall write rate becomes larger. The impacts of write operation rates on system performance decrease with a decrease of shared data rates. Simulation results indicate that larger cache size, line size, and way size decrease system overall cache miss ratios. Although, in general, decreasing the cache miss ratios can increase system performance since it reduces bus requests, a lower miss ratio caused by increasing cache line size may not be effective. The reason is that with a larger line size the system bus spends more time transferring lines from the shared memory to caches, which in turn decreases the system performance. Therefore, cache

line size must be selected carefully during cache design to match the bus system so that the bus traffic does not reach the point at which the system performance will decrease. Furthermore, it implies that use of dual-port directory caches makes the interconnection network bottleneck more serious. The use of multiple buses in the multiprocessor system increases system performance. After bus utilization reaches saturation, the system power is approximately proportional to the number of buses. The results also show that, before bus utilization starts to saturate, the system power increases almost linearly with the number of processors in the system. As the system power increases, the number of processors in the multiprocessor system required to produce full bus saturation increases. Thus, use of multiple buses and more efficient coherence protocols can greatly reduce bus traffic and improve the system performance.

In order to further improve the system performance, an increase in system power and bus ability can be achieved as follows: First, 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. Second, a multiple port memory system would be used to increase performance and the memory competition caused by multiple buses is easer to handle. Each bus can be connected with a port of the memory, which decreases the memory delay time. The major drawback of multiport memory is their cost.

In order to improve the reliability of the proposed cache, the tag self-purge mechanism, the comparator checker, are designed as part of the cache management unit. Also a new CMOS design for the combinational TSC checkers for both 1/n codes and (1-n)/n codes is described. This design can also be used to build the TSC checkers for the 1/3 code and 2/3 code. A comprehensive fault model is created, in which most physical defects which are likely to occur in MOS implementations are included. Both analysis and circuit simulation show that the checkers retain the

Total Overhead of the Dual-Port Directory Cache	
Tag Range	16 Bits — 32 Bits
Overhead Range	41.8 % — 38.2 %
Average Overhead	39.7 %
Extra Overhead	A decoder, a register

Table 8.1: Total Overhead of the Dual-Port-Directory Cache Management Unit

TSC properties for any of the faults or fault sequences in the fault model. The total hardware increase for fault-tolerance and on-line concurrent checking is less than 6 percent of the cache management unit with the single directory.

8.2 Total Overhead Estimates

In order to calculate the percentage of the total hardware overhead in the cache management unit over the cost for the tag array of a single directory cache, we combine the hardware overheads for both performance and fault-tolerance in Chapters 3 and 5. The percentage of the total hardware overhead can be obtained using the equation:

$$C_{total} \leq (\frac{3N+14}{9N+28} + \frac{10}{9N+28}) \times 100\%$$

In the above equation, the first term is the overhead for performance while the second one is for fault tolerance. The equation can be simplified to:

$$C_{total} \leq \frac{3N+24}{9N+28} \times 100\%$$

Assuming the length of cache tags, N, ranges from 16 bits to 32 bits, Table 8.1 shows the corresponding total overhead of the dual-port directory cache management unit. The table gives the average overhead of less than 40%, which means that the total

average overhead is relatively small. Also the extra overhead is listed in the table for reference.

A new optimal test algorithm with a linear test time complexity is presented which can be used to test the cache management unit by either the associated processor in a multiprocessor system or external test equipment. An efficient variant of the proposed algorithm which is suitable for the BIST cache management unit is also discussed. The hardware overhead of such a BIST scheme is much less than that of a traditional BIST implementation for other circuits. Furthermore, this arrangement spends less time for self-testing. The test algorithm and the BIST implementation can also be employed for a single directory cache with only minor changes.

8.3 System Applications

Architectures of tightly-coupled multiprocessor systems with shared memory provide *dynamic* hardware redundancy, modularity and self-tuning so that the workload in the systems can be smoothly switched and balanced automatically among processors. Therefore, such systems offer high performance, and, as well, usually have a certain degree of fault-tolerance. A system has dynamic hardware redundancy because, when there are no faults, all the processors in the system can do useful work simultaneously. The system is running with no redundancy. However, if a processor fails, another one picks up the task of the faulty processor, by self-tuning, so that the system can continue correct operations with minimal degradation of its performance. Therefore, tightly-coupled multiprocessors fit not only scientific applications, but also applications requiring high performance with high reliability and availability, such as transaction processing.

Use of the proposed cache in a multiprocessor system can detect faults not only

in cache data memory(by using EDC), but also in the cache management unit. This is important, from the fault-tolerant point of view, because any errors occurring in the directory will eventually pollute data in the main memory system. Pollution is more serious if coherence protocols based on *write-back* are employed since the lines to be purged from the cache to make room for the new requested lines during line misses are flushed into main memory. If a line to be flushed has a wrong address because of a faulty tag, the line will be written into the wrong location of shared main memory. Furthermore, the polluted data may be used by other processors without knowledge, which is one of major concerns in a multiprocessor system with shared memory. The cache can improve the fault confinement in a multiprocessor system. Erroneous data are limited to a faulty cache before the errors are detected; and when data in the cache are written back to the main memory it is checked for any faults. Thereby contamination of other areas in the system is prevented. Usually, whenever the system senses a faulty component, the component is purged from the system and the system is able to continue operations with some degradation of performance. A fault in a tag of the cache causes the corresponding processing element to be purged from the system so that the system performance greatly decreases. The self-purging of faulty tags in the cache can avoid some of degradation of system performance because the cache is still able to work correctly. The cache performance slightly decreases because of the purging of the faulty tags.

Note that there are multiprocessor systems, such as *Sequoia* in [10], in which all processors are paired. In these systems, the paired processors are doing the same computation during operations, and the results from the paired processors are compared for consistency. The use of the proposed caches in these systems may not yield the best solution, because the faults in tags of a cache can be detected by result-comparisons of the two corresponding paired-processors if the faulty data are accessed by the paired-processors before the faulty data are flushed from the faulty

cache back into the shared memory to make room for new line requests. However, the use of the proposed caches can help the system to confine erroneous data within faulty caches to prevent information in the main memory from pollution caused by the faulty caches. Thus, the proposed fault-tolerant cache can improve the system reliability. That is, it increases the system's fault-tolerant ability.

8.4 Further Work

As we know, network traffic is caused by write references and data transfer for cache misses. Since the coherence protocols are one of the major factors affecting network traffic and the system performance, it would be valuable to evaluate the effects of efficient protocols such as the Berkeley protocols on performance obtained by the proposed caches, and to study the impacts of data sharing rate and write reference rate, and cache design parameters on the system performance.

In addition, the trend of processors is moving to 64-bit address and data paths, and their effects on the proposed cache memory and the cache-based multiprocessor systems, specifically the effect on miss ratio, cache sizes, line size, bus width, data sharing, and write reference rate should be studied. Also, we would like to investigate the effects of the increased addressing range on the cache.

Bibliography

[1] A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–528, 1982.

[2] W. C. Yen, D. W. L. Yen, and K. S. Fu. Data coherence problem in a multicache system. *IEEE Transactions on Computers*, 34:56–65, 1985.

[3] H. S. Stone *et al. Introduction to Computer Architecture.* The SRA Computer Science Series, 1980.

[4] H. S. Stone. *High-performance Computer Architecture.* Addison-Wesley, Reading, Mass., 1990.

[5] A. N. Choudhary and S. Krishnamoorthy. Experimental evaluation of multilevel caches for shared memory multiprocessors. *Proc. of Hawaii Int'l Conf. on Systems and Sciences*, pages 409–420, Jan. 1991.

[6] S. Frank and A. Inselberg. Synapse tightly coupled multiprocessors: a new approach to solve old problems. *Proc. of AFIPS National Computer Conf.*, 53:41–50, 1984.

[7] S. Thakkar, P. Gifford, and G. Fielland. The balance multiprocessor system. *IEEE Micro*, 8(1):57–69, 1988.

[8] A. Ayyad and B. Wilkinson. Multiprocessor scheme with application to macro-dataflow. *Microprocessors and Microsystems*, 11(5):255–263, 1987.

[9] J. Kuhl and S. Reddy. Fault-tolerance considerations in the large, multiprocessor systems. *Computer*, 19(3):56–67, March 1986.

[10] P. A. Bernstein. Sequoia: A fault-tolerant tightly coupled multiprocessor for tansaction processing. *Computer*, 17(8):19–30, 1984.

[11] J. Edler *et al.* Issues related to MIMD shared-memory computer: the NYU Ultracomputer approach. *Proc. of the 13th Int'l Symposium on Computer Architecture*, pages 127–135, 1986.

[12] P. Bitar and A. M. Despain. Multiprocessor cache synchronization — issues, innovations, evolution. *Proc. of the 12th Int'l Symposium on Computer Architecture*, pages 425–433, 1985.

[13] J. Archibald and J.-L. Baer. An economical solution to the cache coherence problem. *Proc. of the 11th Int'l Symposium on Computer Architecture*, pages 355–362, 1984.

[14] L. M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, 27(12):1112–1118, 1978.

[15] S. Frank. Tightly coupled multiprocessor systems speed memory access times. *Electronics*, 57(1):164–169, 1984.

[16] P. Sweazey and A. J. Smith. A class of compatible cache consistency protocols and their support by the IEEE futurebus. *Proc. of the 12th Int'l Symposium on Computer Architecture*, pages 415–423, 1985.

[17] J. R. Goodman. Using cache memory to reduce processor-memory traffic. *Proc. of the 10th Int'l Symposium on Computer Architecture*, pages 124–131, 1983.

[18] R. H. Katz *et al.* Implementing a cache consistency protocol. *Proc. of the 12th Int'l Symposium on Computer Architecture*, pages 276–283, 1985.

[19] M. Papamarcos and J. Patel. A low overhead coherence solution for multiprocessors with private cache memories. *Proc. of the 11th Int'l Symposium on Computer Architecture*, pages 348–354, 1984.

[20] L. Rudolph and Z. Segall. Dynamic decentralized cache schemes for MIMD parallel processors. *Proc. of the 11th Int'l Symposium on Computer Architecture*, pages 340–347, 1984.

[21] C. K. Tang. Cache system design in the tightly coupled multiprocessor system. *Proc. of AFIPS National Computer Conf.*, pages 749–753, 1976.

[22] A. J. Smith. Second bibliography on cache memory. *Computer Architecture News*, 19(4):138–153, June 1990.

[23] A. Agarwal. *Analysis of Cache Performance for Operating Systems and Multiprogramming.* PhD thesis, Stanford University, 1987.

[24] A. J. Smith. Cache evaluation and the impact of workload choice. *Proc. of the 12th Int'l Symposium on Computer Architecture*, pages 64–74, June, 1985.

[25] A. J. Smith. Line (block) size choice for CPU cache memory. *IEEE Transactions on Computers*, 36(9):1063–1074, 1987.

[26] M. D. Hill. A case for direct-mapped caches. *Computer*, pages 25–40, Dec., 1988.

[27] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38:1612–1630, Dec. 1989.

[28] D. Alpert and M. Flynn. Performance trade-offs for microprocessor cache memories. *IEEE Micro*, pages 44–54, Aug. 1988.

[29] A. J. Smith and J. R. Goodman. Instruction cache replacement policies. *IEEE Transactions on Computers*, 34:234–241, Mar. 1985.

[30] J. R. Goldman. First look at Motorola's latest 32-bit processor. *Electronics*, 59(31):71–75, 1986.

[31] H. Scales and P. Harrod. The design and implementation of the MC68030 cache memories. *IEEE Int'l Conf. On Computer Design*, pages 578–581, 1987.

[32] R. E. Matick. Functional cache chip for improved system performance. *IBM J. Res. Develop.*, 33(1):15–32, 1989.

[33] D. Alpert *et al.* 32-bit processor chip integrates major system functions. *Electronics*, pages 113–119, July, 1983.

[34] M. Stansberry. Cache memory design in 32-bit microprocessor systems. *VLSI Systems Design*, pages 32–42, 1988.

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

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

[37] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, San Mateo, CA., 1990.

[38] S. A. Przybylski. *Cache and Memory Hierarchy Design: a Performance-Directed Approach.* Morgan Kaufmann, San Mateo, CA., 1990.

[39] G. Ramamoorthy and A. N. Choudhary. A bibliography for multiprocessor cache memories. *Computer Architecture News*, 19(4):154–182, June 1990.

[40] P. Stenstrom. Reducing contention in shared-memory multiprocessors. *IEEE Computer*, 21(11):26–37, 1988.

[41] J. L. Baer and W. H. Wang. Architectural choices for multilevel cache hierarchies. *Proc. of int'l Conf. on Parallel Processing*, pages 258–261, 1987.

[42] M. Dubois and S. S. Thakkar. Cache architecture in tightly coupled multiprocessors. *IEEE Computer*, 23(6):9–11, 1990.

[43] D. R. Cheriton, G. A. Slavenburg, and P. D. Boyle. Soft-controlled caches in the VMP multiprocessor. *Proc. of the 12th Int'l Symposium on Computer Architecture*, pages 366–374, 1985.

[44] C. A. Prete. RST cache memory design for a tightly coupled multiprocessor system. *IEEE Micro*, pages 16–19, April 1991.

[45] X. Luo and P. Gillard. A VLSI design for an efficient multiprocessor cache memory. *Int'l Journal of Computers & Electrical Engineering*, 16(1):3–21, 1990.

[46] X. Luo and J. C. Muzio. A novel multiprocessor cache memory. *Proc. of IEEE Pacific Rim Conf. on Comm., Comp. and Signal Processing*, pages 145–148, May 1993.

[47] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. *Proc. of the 12th Int'l Symposium on Computer Architecture*, pages 434–443, 1985.

[48] R. Gregory. Caching designs eliminate wait states to relieve bottlenecks. *Computer Design*, pages 65–73, Oct., 1988.

[49] J. Archibald and J. L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, 1986.

[50] A. Agarwal. Cache performance for operating systems and multiprogramming workload. *ACM Transactions on Computer System*, 6:393–431, Nov. 1988.

[51] S. J. Eggers and R. Katz. Evaluating the performance of four snooping cache coherency protocols. *Proc. of 16th Int'l Symposium on Computer Architecture*, pages 2–15, 1989.

[52] H. Bugge, E. Keistiansen, and B. Bakka. Trace-driven simulations for a two-level cache design in open bus systems. *Proc. of the 17th Int'l Symposium on Computer Architecture: in Computer Architecture News*, pages 250–259, June, 1990.

[53] A. Agarwal, R. Simoni, and M. Horowitz. An evaluation of directory schemes for cache coherence. *Proc. of the 15th Int'l Symposium on Computer Architecture*, pages 280–289, 1988.

[54] X. Luo. Performance evaluation of a multiprocessor system with dual-port directory caches. *Proc. of IEEE Pacific Rim Conf. on Comm., Comp. and Signal Processing*, pages 215–218, May 1991.

[55] M. Dubois and F. A. Briggs. Effects of cache coherency in multiporocessors. *IEEE Transactions on Computers*, 31(11):1083–1099, 1982.

[56] V. P. Nelson. Fault-tolerant computing: Fundamental concepts. *Computer*, 23(7):19–25, July,1990.

[57] D. P. Siewiorek. Fault-tolerance in commercial computers. *Computer*, 23(7):26–38, July,1990.

[58] O. Serlin. Fault-tolerant systems in commercial applications. *Computer*, 17(8):19–30, Aug. 1984.

[59] D. P. Siewiorek. Architecture of fault-tolerant computers. *Computer*, 17(8):9–18, Aug. 1984.

[60] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, Feb. 1991.

[61] O. Serlin. New microprocessor-based computer architectures. *AFIPS, Proceedings of National Computer Conference*, pages 123–130, July 1984.

[62] A. Inselberg. An approach to successful online transaction processing applications. *AFIPS, Proceedings of National Computer Conference*, pages 419–427, July 1985.

[63] P. A. Bernstein. Synchronizing shared memory in the sequoia fault-tolerant multiprocessor. *Database Engineering*, 5:17–23, 1986.

[64] F. Pollack and *et al.* A VLSI-intensive fault-tolerant computer architecture. *Proceedings of IEEE Spring 1990 Computer Conference*, pages 134–142, 1990.

[65] A. Gottlieb *et al.* The Ultracomputer — designing a MIMD, shared memory parallel machine. *Proc. of the 9th Int'l Symposium on Computer Architecture*, pages 27–42, 1982.

[66] T. Watanabe. An 8kbyte intelligent cache memory. *Proc. of IEEE Int'l Solid-State Circuits Conf.*, pages 266–267, Feb., 1987.

[67] J. K. Muppala and L. N. Bhuyan. Arbiter designs for multiprocessor interconnection network. *Microprocessing and Microprogramming*, 26:31–43, 1989.

[68] M. Abramovici, M. A. Breuer, and A. D. Friedman. *Digital Systems Testing and Testable Design.* Computer Science Press, New York, 1990.

[69] D. A. Anderson and G. Metze. Design of totally self-checking check circuits for m-out-of-n codes. *IEEE Transactions on Computers*, 22(3):263–269, 1973.

[70] S. M. Reddy. A note on self-checking checkers. *IEEE Transactions on Computers*, 23(10):1100–1102, 1974.

[71] J. E. Smith. The design of totally self-checking checker circuits for a class of unordered codes. *Design Automation and Fault Tolerant Computing*, pages 321–343, Oct. 1977.

[72] D. L. Tao and P. K. Lala. Three-level totally self-checking checker for 1-out-of-n code. *IEEE Int'l Symposium on Fault-Tolerant Computing*, pages 108–113, 1987.

[73] J. Khakbaz. Totally self-checking checker for 1-out-of-n code using two-rail codes. *IEEE Transactions on Computers*, 31(7):677–681, 1982.

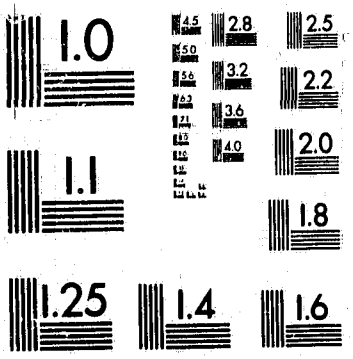
[74] R. David. A totally self-checking 1-out-of-3 checker. *IEEE Transactions on Computers*, 27(6):570–572, 1978.

[75] P. Golan. Design of a totally self-checking checker for 1-out-of-3 code. *IEEE Transactions on Computers*, 33(3):285, 1984.

[76] A. M. Paschalis, C. Efstathiou, and C. Halatsis. An efficient TSC 1-out-of-3 code checker. *IEEE Transactions on Computers*, 39(3):407–411, 1990.

[77] D. L. Tao, P. K. Lala, and C. R. P. Hartmann. A MOS implementation of totally self-checking checker for the 1-out-of-3 code. *IEEE Journal of Solid State Circuits*, 23(3):875–877, June 1988.

[78] N. K. Jha. Comments on "a MOS implementation of totally self-checking checker for the 1-out-of-3 code". *IEEE Journal of Solid State Circuits*, 24(4):875–876, Oct. 1989.



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No. 2)

[79] J. C. Lo and S. Thanawastien. On the design of combinational totally self-checking 1-out-of-3 code checker. *IEEE Transactions on Computers*, 39(3):387–393, 1990.

[80] J. E. Smith and G. Metze. Strongly fault secure logic networks. *IEEE Transactions on Computers*, 27(6):491–499, 1978.

[81] N. K. Jha and N. Kundu. *Testing and Reliable Design of CMOS Circuits.* Kluwer Academic Publishers, New York, 1990.

[82] Y. Tamir and C. H. Sequin. Design and application of self-testing comparators implemented with MOS PLAs. *IEEE Transactions on Computers*, 33(6):493–506, 1984.

[83] M. Nicolaidis and B. Courtois. Strongly code disjoint checkers. *IEEE Transactions on Computers*, 37(6):751–756, 1988.

[84] N. K. Jha. SFS/SSC Domino-CMOS implementations of TSC circuits. *Proc. of Annual Allerton Conf. Communication, Control & Computing*, pages 763–777, Sept. 1988.

[85] N. K. Jha. Fault detection in CVS parity tree: Application to SCC CVS parity and two-rail checkers. *Proc. of Int'l Symp. on Fault Tolerant Computing*, pages 407–414, June 1989.

[86] P. Banerjee and J. A. Abraham. Characterization and testing of physical failures in MOS logic circuits. *IEEE Design & Test Comput.*, pages 76–86, Aug. 1984.

[87] M. S. Cheema and P. K. Lala. A new technique for totally self-checking CMOS circuit design for stuck-on and stuck-off faults. *Proc. of IEEE VLSI Test Symposium*, pages 155–159, 1992.

[88] W. Daehn and J. Gross. A test generator IC for testing large CMOS-RAMs. *Proc. of IEEE Int'l Test Conf.*, pages 16–24, 1986.

[89] K. T. Le and K. K. Saluja. A novel approach for testing memories using a built-in self testing techniques. *Proc. of IEEE Int'l Test Conf.*, pages 830–839, 1986.

[90] R. Dekker, F. Beenker, and L. Thijssen. Fault modeling and test algorithm development for static random access memories. *Proc. of IEEE Int'l Test Conf.*, pages 343–352, 1988.

[91] A. Noore. Memory design to improve testability. *IEEE Circuits and Systems*, pages 825–828, 1990.

[92] M. J. Raposa. Dual port static RAM testing. *Proc. of IEEE Int'l Test Conf.*, pages 362–368, 1988.

[93] V. C. Alves, A. Nicolaidis, P. Lestrat, and B. Courtois. Built-in self-test for multi-port RAMs. *Proc. of IEEE Int'l Test Conf.*, pages 248–251, 1991.

[94] k. E. Grosspietsch, H. Huber, and A. Muler. The concept of a fault-tolerant and easily-testable associative memory. *Proc. of Int'l Symp. Fault Tolerant Computing*, pages 34–39, 1986.

[95] P. Mazumder and J. H. Patel. Methodologies for embedded content addressable memories. *Proc. of Int'l Symp. Fault Tolerant Computing*, pages 270–275, 1987.