Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

HIGH-PERFORMANCE FAULT-TOLERANT VLSI SYSTEMS USING MICRO ROLLBACK

M. Tremblay

September 1991
CSD-910062

UNIVERSITY OF CALIFORNIA

Los Angeles

High-Performance Fault-Tolerant VLSI Systems Using Micro Rollback

A dissertation submitted in partial satisfaction of the requirements for the degree Doctor of Philosophy in Computer Science

by

Marc Tremblay

1991

The dissertation of Marc Tremblay is approved.

Kirby A. Baker

Kirby A. Baker

[illegible]

Asad A. Abidi

[illegible]

Milos D. Ercegovac

[illegible]

Algirdas A. Avizienis

[illegible]

David A. Rennels

Yuval Tamir

Yuval Tamir, Committee Chair

University of California, Los Angeles

1991

To Alexandra who shared this experience
with love, understanding and belief.
A mes parents, pour m'avoir fait
découvrir les nombreuses facettes de la vie.

Table of Contents

List of Figures

List of Tables

ACKNOWLEDGMENTS

Many persons have contributed to creating a positive environment which helped me produce the research described in this dissertation. I wish to express my gratitude to all of them.

My advisor, Yuval Tamir, through his dedication to research, through his desire to produce perfect papers, and through his broad knowledge of computer science, has taught me a great deal. Yuval's method of conducting research and how it relates to other fields will have a lasting impression on my future career. Yuval's comments and inputs have brought this dissertation to a level that I would not have reached on my own. Thanks, Yuval, for all the advice (generally given during one of our many ''late night'' discussions).

I would like to thank members of the committee for their comments on my research. More specifically Prof. David Rennels, who has always been supportive, contributed to the theory of micro rollback. Prof. Milos Ercegovac and Prof. Tomas Lang have helped me with a variety of issues during my stay at UCLA, through their teaching, their insight on research, and even with desk allocation!

This journey would not have been as significant, had I not encountered my fiancée, Alexandra, during this time. She has provided unconditional support, lots of encouragement, and also some of the much needed extra-curricular activities that kept me sane throughout these past several years of research.

My gratitude also goes out to my colleagues, Jaime Moreno, Miquel Huguet, Leon Alkalaj and Paul Tu. Our conversations, our lunches, and our more ''formal'' meetings during the CIGAR seminars, were all moments that I cherish.

The computer science department at UCLA possesses several good people who care about the students. Verra Morgan, Roberta Nelson, Doris Sublette, and Rosemary Murphy are all dedicated to creating a pleasant environment. The help that Verra provided will never be forgotten.

I am very fortunate to have friends like Gary Park, Jeff Bixler, Robert Krasuski and Thuvan Nguyen. They kept my intellect active on topics other than computer science and kept my level of competition in a multitude of sports, quite challenging. Thanks to all of you.

I am also very grateful to my parents, my sister Luci, and my brother Réjean, who have always been there in times of need. Their support, even if sometimes taken for granted, is greatly appreciated and will never be forgotten.

My appreciation also goes to the National Research Council of Canada who supported me for four years, IBM for the Graduate Fellowship, Hughes Aircraft Company and the State of California MICRO Program who provided financial assistance for my research. Special thanks to Dr. T. Basil Smith who made the whole IBM experience an enjoyable and valuable one.

VITA

October 30, 1961	Born in Quebec, Canada
1984	B.S Physics Engineering Université Laval Quebec, Canada
1984-1988	National Research Council Fellowship (Canada)
1985	M.S. Computer Science University of California, Los Angeles Los Angeles, California
1985-1991	Teaching Assistant, Teaching Assosiate University of California, Los Angeles Los Angeles, California
1987-1990	Research Assistant, Research Assosiate University of California, Los Angeles Los Angeles, California
1989-1990	IBM Graduate Fellowship

PUBLICATIONS AND PRESENTATIONS

1. Y. Tamir and M. Tremblay, "High-Performance Fault-Tolerant VLSI Systems Using Micro Rollback," *IEEE Transactions on Computers* 39(4), pp. 548-554 (April 1990).
2. Y. Tamir, M. Tremblay, and D. A. Rennels, "The Implementation and Application of Micro Rollback in Fault-Tolerant VLSI Systems," *18th Fault-Tolerant Computing Symposium*, Tokyo, Japan, pp. 234-239 (June 1988).

3. Y. Tamir, M. Tremblay, and D. A. Rennels, "The Implementation and Application of Micro Rollbacks in Fault-Tolerant VLSI Systems," Computer Science Department Technical Report CSD-880004, University of California, Los Angeles, CA (January 1988).

4. Y. Tamir, M. Liang, T. Lai, and M. Tremblay, "The UCLA Mirror Processor: A Building Block for Self-Checking Self-Repairing Computing Nodes," *21st Fault-Tolerant Computing Symposium*, Montreal, Canada (June 1991).

5. Y. Tamir, M. Liang, T. Lai, and M. Tremblay, "The UCLA Mirror Processor: A Building Block for Self-Checking Self-Repairing Computing Nodes," Computer Science Department Technical Report CSD-900040, University of California, Los Angeles, CA (November 1990).

6. M. Tremblay and Y. Tamir, "Fault-Tolerance for High-Performance Multi-Module VLSI Systems Using Micro Rollback," *Decennial Caltech Conference on VLSI*, Pasadena, CA (March 1989).

7. M. Tremblay and Y. Tamir, "Support for Fault Tolerance in VLSI Processors," *International Symposium on Circuits and Systems*, Portland, OR (May 1989).

8. M. Tremblay and T. Lang, "VLSI Implementation of a Shift-Register File," *Proceedings of the 20th Hawaii International Conference on System Sciences*, pp. 112-121 (January 1987).

9. M. Tremblay and Y. Tamir, "Fault-Tolerance for High-Performance Multi-Module VLSI Systems Using Micro Rollback," Computer Science Department Technical Report CSD-880099, University of California, Los Angeles, CA (December 1988).

10. M. Tremblay and Y. Tamir, "Support for Fault Tolerance in VLSI Processors," Computer Science Department Technical Report CSD-890009, University of California, Los Angeles, CA (January 1989).

ABSTRACT OF THE DISSERTATION

High-Performance Fault-Tolerant VLSI Systems

Using Micro Rollback

by

Marc Tremblay

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1991

Professor Yuval Tamir, Chair

This dissertation addresses the problem of achieving both a high level of fault tolerance and high performance in VLSI computer systems. New implementation techniques are proposed for systems operating in hostile environments where a high rate of errors is expected. Due to the real-time constraints commonly imposed on such systems, long interruptions of service are not acceptable. Hence, both error detection and error recovery must be rapid and robust.

Checkers and encoders on the critical path for intermodule communication are a major cause of performance degradation in many fault-tolerant systems. This performance degradation can be eliminated if error detection is allowed to proceed in parallel rather than in series with intermodule communication.

The fundamental new technique proposed in this thesis, *micro rollback*, allows the use of extensive error detection mechanisms without compromising system performance. Error detection is performed in parallel with normal execution. If errors are detected, micro rollback restores modules to a previous error-free state. Micro rollback is based on the *optimistic assumption* that system components are fault free *most of the time*. This allows modules to be optimized for the normal case when no errors occurs.

Efficient techniques for adding micro rollback to a processor and other modules are described. VLSI layouts and extensive simulations of key building blocks demonstrate that the proposed techniques involve small area overheads and minimal performance degradation.

In a multi-module system, rollback to a consistent global state requires coordination among the modules. We introduce a novel hardware mechanism for logging the occurrence of recent intermodule transactions. During a rollback, these logs are used to compute the appropriate *rollback distance* for each module. Through the design of specific *rollback domain interface units*, we also show that standard modules not capable of rollback can interact with modules belonging to the *rollback domain*.

The integration of micro rollback with speedup techniques for uniprocessors, such as out-of-order execution and register renaming, shows that a high level of fault tolerance can be achieved for high-performance processors.

Chapter One
Introduction

The widespread use of VLSI digital systems running critical applications in ''hostile'' environments, has resulted in the development of systems able to operate reliably in the presence of isolated faults. The reliability requirements for systems operating in harsh environments, such as air-borne or space-borne computers, demand that only momentary cessation of processing be tolerated. Since most errors are caused by transient failures [Cast82], the ability to detect and recover quickly from transient faults is directly related to the reliability of a computer system. Systems capable of operating *reliably* despite the occurrence of errors, are said to be *fault-tolerant*.

One of the keys to achieving a high degree of fault tolerance is the ability to detect errors immediately after they occur and prevent erroneous information from spreading throughout the system. Traditionally this had been achieved through concurrent error detection, which is designed to detect the first error resulting from a fault in the system. To achieve concurrent error detection and confine the damage caused by the error to the failed module, it is often necessary to check the outputs of the module continuously (during every clock cycle for synchronous systems). These requirements are usually satisfied by connecting checkers and isolation circuits in the communication path between each module and the rest of the system (Figure 1.1(a)). Consequently, concurrent error detection results in longer clock cycle times in order to allow the checks to complete. Alternatively, checking delays can be pipelined, resulting in additional pipeline stages thus diminishing throughput whenever the pipeline needs to be flushed or is not full due to data dependencies. Hence, systems with high-coverage concurrent error detection often experience significant performance penalties due to checking delays. Moreover, these delays can compound as, for example, when a processor reads a memory word, and (1) Hamming Code checks are made, (2) the word is encoded for bus transmission, and (3) the word is checked when it arrives at the processor before use.

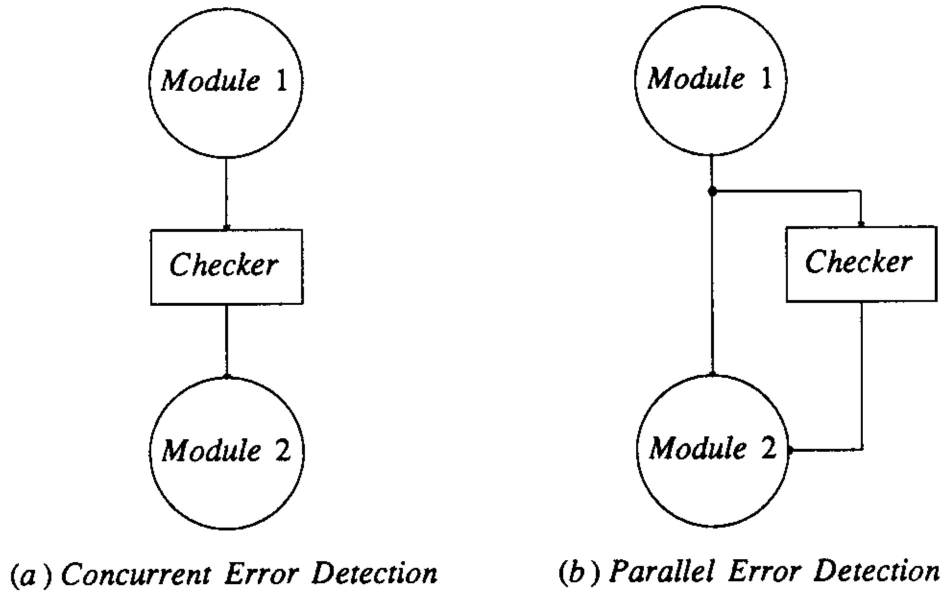


Figure 1.1: Concurrent error detection vs. parallel error detection.

One way to solve the problem described above is to perform checking *in parallel* with the transmission of information between modules (Figure 1.1(b)). The receiving module does not wait for checks to complete. It proceeds with execution as the check is being carried out and the checking result is sent one (or a few) cycles later.

Performing error checking in parallel with inter-module communication largely solves the problem of checking delays, but it introduces a new problem in recovery. The state of the system (starting with the receiving module) may be polluted with damaged information before an error signal arrives. Therefore it is necessary to back up processing to the state that existed just before the error first occurred. This returns the system to an error-free state where the offending operation can be retried (or correction may be attempted by other means such as restoring information from a redundant module or initiating higher level rollback). We call the process of backing up a system several cycles in response to an error signal, *micro rollback* [Tami88b, Tami90].

This dissertation addresses the problem of achieving a high level of fault tolerance in computer systems while maintaining high performance. With *micro rollback*, parallel error detection can be used, thus removing error detection circuits from critical paths. General techniques for efficient implementation of micro rollback in processors and other modules are described throughout the thesis. It is shown that micro rollback can be integrated with many speedup techniques, thus demonstrating that micro rollback can serve as a basis for high-performance, highly reliable computer systems.

1.1. Applications of Micro Rollback

Systems using concurrent error detection can benefit from micro rollback by replacing the error detection mechanism with a parallel scheme. For example, if data in a cache memory is stored along with check bits so that its validity can be checked upon retrieval, the logic necessary for the coding and decoding may be inserted serially between the processor and the memory (Figure 1.2(a)). The added delay due to the error detection logic would contribute to the access time to the cache memory which is often a critical path for high performance processors [Kane87]. With micro rollback, the checkers (or correction circuitry) can be removed from this critical path. The error detection can occur in parallel (Figure 1.2(b)) and the error signal can be sent after the data has arrived.

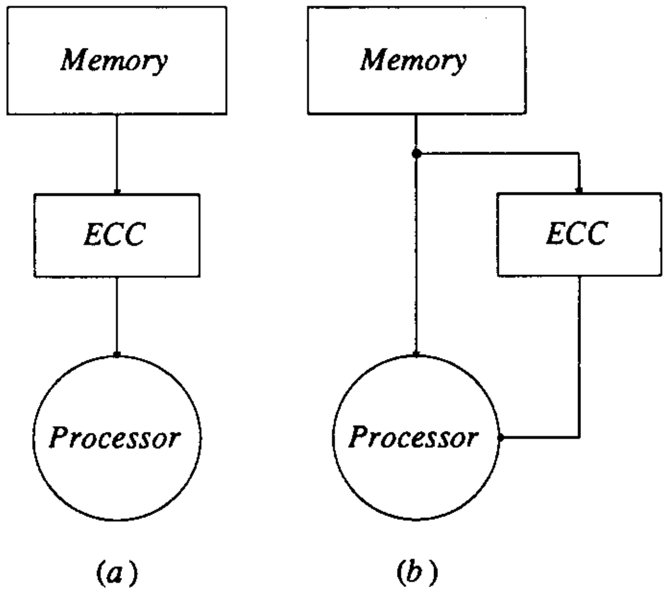


Figure 1.2: Application of parallel error detection for a processor-memory system

Another important use of parallel checks is in duplex systems [Down64] where error detection is accomplished by running two identical subsystems in parallel and comparing their outputs (Figure 1.3). With this technique, which is supported by some current commercial chips [AMD87], the two subsystems may be on different chips and there is thus a significant delay in getting both outputs to the comparator (off chip communication) and obtaining the results of the comparison. With micro rollback, the processors do not have to wait for the output of the comparator to resume operation — upon a mismatch both processors are forced to roll back to a point in time where they were both in agreement. A system based on triplication and voting (TMR) can benefit from micro rollback in a similar way (Figure 1.4). During normal execution, the three modules execute instructions without waiting for the outcome of the voting. When voting indicates a disagreement, the modules are brought back to an error-free state using micro rollback.

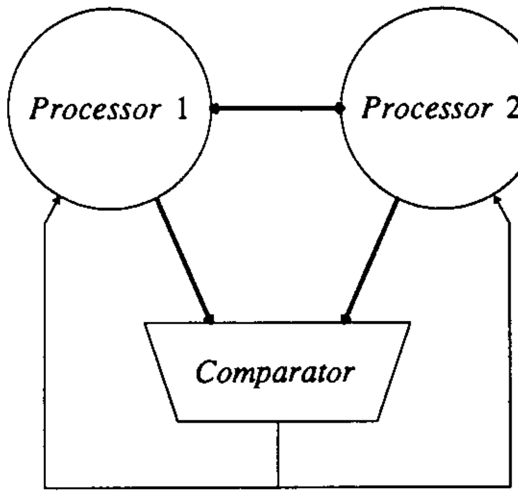


Figure 1.3: Processors operating in duplex mode

Duplex mode operation and TMR both use redundancy in space. Micro rollback can also be used for systems in which checking is done using time redundancy. With time redundancy, the same hardware is used to compute logic or arithmetic operations that can confirm the validity of the results. The operation performed by the hardware to validate the results, can be the same — in which case a simple comparison between the two results is sufficient, or different — in which case a "match" is established through a relation between the two results. For

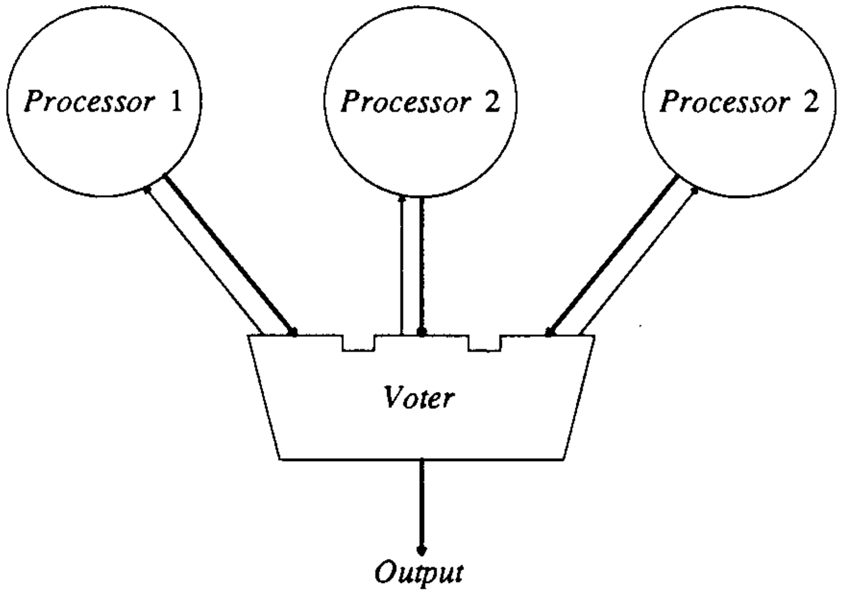


Figure 1.4: Processors running in Triple Modular Redundancy (TMR).

instance a combined multiplier/divider functional unit can be used to validate a multiplication through a division.

Time redundancy can be achieved at small cost (small hardware overhead) by using alternating logic [Reyn78]. Circuits such as adders and multipliers can be slightly modified so that when fed with complemented operands they produce a result which is the complement of the one obtained with non-complemented operands. Recomputing with Shifted Operands (RESO [Pate82]) is another method for using redundancy in time — the second computation is done using shifted operands (shifted left), producing a result which is then shifted right in order to compare it with the original result. In a fault-free circuit, the second computation matches the original result (Figure 1.5). Although savings in area are achieved through time redundancy, the main drawback of this method is the added delay required to recompute the result and compare the two values. Using micro

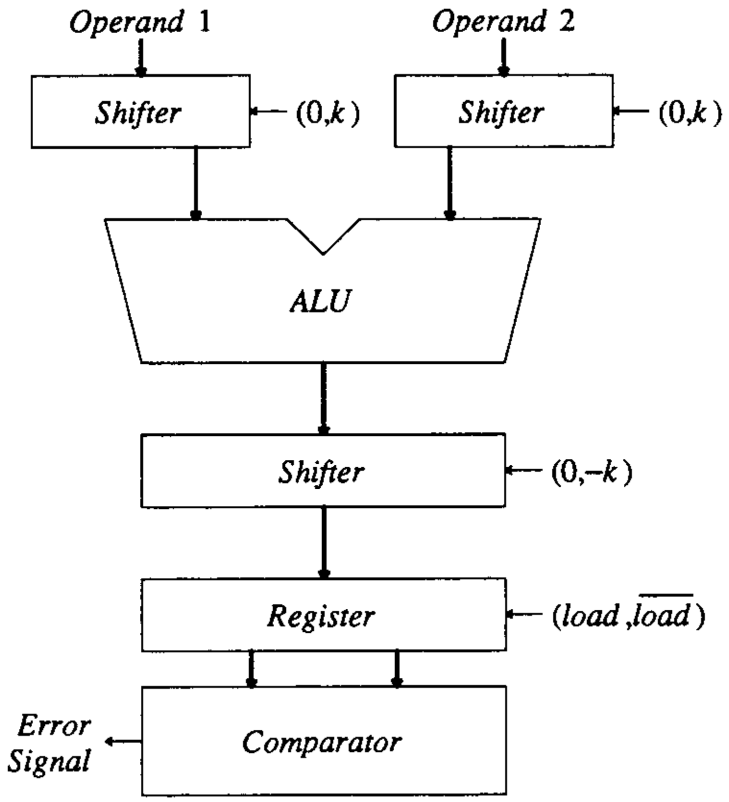


Figure 1.5: Error detection by Recomputing with Shifted Operands (RESO). The first result is computed with the original operands and stored into the register (using first set of control signals). The second result is calculated using shifted operands (shifted left k positions). The second result is shifted right by k positions and compared with the original result.

rollback, the second result can be computed in parallel with normal execution. If an error is detected, the processor is rolled back to a point before the computation error occurred.

In a multi-module system, a high level of fault tolerance can be achieved by making each module self-checking (Figure 1.6). When an error is detected by one of the modules, it is reported to the rest of the system C cycles later where C

represents the error detection latency. A recovery procedure such as software rollback can then be initiated so that process can resume within a few milliseconds. Although quite useful in many situations (e.g. general purpose computing), the method described above cannot be used in a context of real time control or in an environment subject to a high rate of faults.

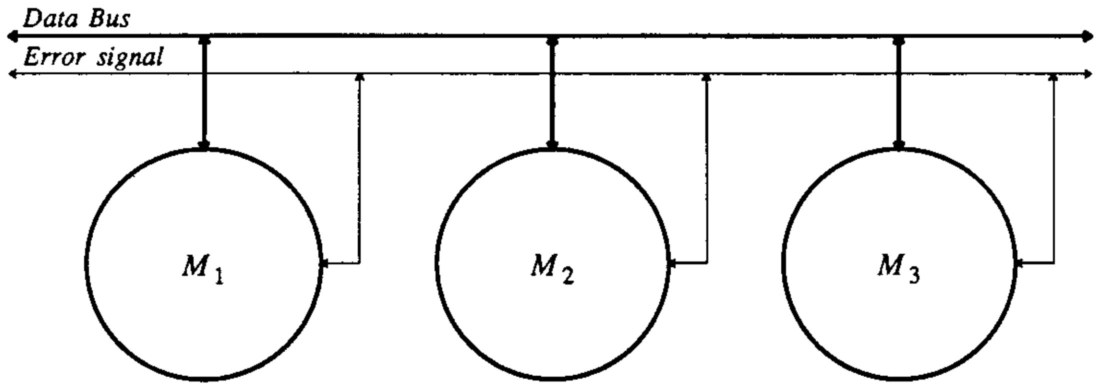


Figure 1.6: System with Self-Checking Modules

In tightly-coupled multi-module systems, micro rollback can be used to provide faster error recovery than software-based techniques. Upon detecting an error, a module sends a rollback signal to the other modules in the system, thereby requesting a rollback to a global fault-free state. Using records of previous inter-module interactions, all the modules must then coordinate their individual rollbacks, to achieve a consistent, error-free, system state. After all the modules have rolled back (in parallel), normal system operation can resume.

1.2. Implementation of Micro Rollback

The previous examples have shown how micro rollback, once combined with some error detection mechanisms, can reduce or even eliminate overheads encountered in traditional fault tolerant systems. This, as we will see in the next chapters, is possible only if micro rollback can be implemented efficiently without adding significant delay. A straightforward implementation of micro rollback could lead to a significant increase in area and could introduce delays by attaching circuitry to critical paths.

Throughout this work we present novel architectures that exploit micro rollback capabilities for achieving a high degree of reliability but also try to minimize the overhead on the system. While presenting these new ideas and exposing their benefits, we also *implement* (detailed layouts, logic simulations, extensive SPICE simulations, etc) the designs in order to know in details the impact of our ideas on the system. Bearing in mind that fabrication of the modules would lead to better absolute measurements, we compare our delays and area metrics to other modules implemented using the same technology, and when possible laid out by the same designer. This work has led to the development of a complete processor through class projects [Trem88] and other efforts [Tami91].

1.3. Organization of this Work

In Chapter 2 we provide a brief overview of common error detection and correction techniques for computer systems. We also examine hardware mechanisms for reversing recent state changes in processors. These mechanisms are directly related to micro rollback even though they were originally developed for supporting precise interrupts and efficient handling of mis-predicted conditional branches.

An additional benefit of using micro rollback is the possibility of using ''cheaper'' checkers (in terms of area) for error detection. Since long error detection latencies have little impact on performance, it is possible to use slow checkers, which are small and simple. If error detection is on the critical path for performance, it is necessary to use fast checkers, which may be large and complex. In Chapter 3 we describe and evaluate the VLSI implementations of checkers that are commonly used for error detection and correction.

We present in Chapter 4, techniques for an *efficient* implementation of micro rollback in VLSI systems. We focus on building blocks for a VLSI RISC processor that is capable of micro rollback. We show how the updated state of the entire processor can be checkpointed after every cycle without replicating all the storage. Based on VLSI layout and circuit simulation of key modules, it is shown that the micro rollback functionality can be added with only a small performance penalty and with a low area penalty relative to the size of the entire chip. We will also discuss the implementation of a cache capable of micro rollback as well as the problems (with their solution) encountered in a multi-processor environment with shared memory.

The implementation of micro rollback in a multi-module system is discussed in Chapter 5. Asynchronous and synchronous systems are both covered. Transducer modules performing translation between a number of cycles to rollback and a number of transactions to rollback (and vice-versa), are introduced in this chapter. Those relatively small modules are implemented through full pass gate logic and perform the mapping without interfering with the basic number of cycles required to rollback.

Chapter 6 describes the problem of interfacing off-the-shelves modules with modules capable of rollback. It is shown that modules that can roll back and modules that are not capable of rollback cannot communicate directly and must exchange information through a rollback domain interface unit (RDIU). The purpose of the RDIU is to delay (buffer) all transfers to modules which are incapable of rollback and whose state might be corrupted if these transfers are erroneous. Transfers *from* modules which are incapable of rollback are recorded in a *replay memory* so that they can be retransmitted to modules inside the rollback domain following a rollback. We describe several off-the-shelf memory systems that can be part of a rollback system by connecting them to the system through a RDIU. We address the issue of integrating a processor capable of micro rollback with standard loosely-coupled and tightly-coupled coprocessors.

The combination of micro rollback and techniques used to increase the performance of uniprocessors is discussed in Chapter 7. Techniques such as out-of-order execution of instructions, branch repair, and register renaming are combined in a single structure (called a standby reorder buffer) which also provides micro rollback. Sharing of the circuitry for the techniques mentioned above, leads to a low area overhead.

Chapter Two Previous Work

The development of fault tolerance techniques has improved the ability of computer systems to cope with hardware component failures. This has led to the development of highly reliable computer systems, such as the STAR [Aviz71a, Aviz72], FTMP [Hopk78], SIFT [Wens78], Tandem [Mack78], and Stratus [Wils85]. Several error detection methods often found in those computer systems are summarized in this chapter. A micro program control unit with extensive concurrent error detection is described in details in Section 2.2. Forward and backward recovery methods are described in Section 2.3. Specifically, we discuss *instruction retry* as an example of a backward recovery method. Rollback techniques used for supporting precise interrupts and handling of mispredicted conditional branches are covered in Section 2.3.

2.1. Error Detection

The first step in fault-tolerance is to detect errors occurring in the system. The purpose of error detection is to prevent system failures by recognizing that they may be about to happen and initiate corrective actions.

Possible errors in VLSI chips include: corruption of the contents of storage elements, incorrect results produced by computation modules (e.g., an ALU), and corruption of data and control signals (e.g., buses). These errors are the result of transient or permanent faults due to design and fabrication flaws (e.g., marginal timing, incorrect dosage of ion implants), environmental factors (e.g., noise, radiation), and wear-out mechanisms (e.g., electromigration) [Doyl81]. Since micro rollback is aimed at removing delays introduced by checkers it is important to understand how error detection is performed and what are the different methods used. The VLSI implementation of some of these methods is described in Chapter 3. Among several error detection methods, we mention:

Parity: a bit representing the XOR of every data bits in a word is sufficient to detect all single bit errors as well as all odd-weight errors [Hamm50]. This technique is used in many systems, either to detect errors in storage elements, or to detect errors after transmission through a bus [Kane87].

Codes for memories: several types of codes have been applied successfully to memory systems. Most of the codes fall can be categorized into two families [Rao89]; linear codes, which are usually derivatives of Hamming Code [Hamm50], and cyclic codes such as linear feedback shift register (LFSR) [Froh77]. The principle behind these codes is simple. Words are encoded prior to transmission or storage, and they are checked upon reception or retrieval. Besides detecting errors, most of the codes offer the possibility (with larger overhead) to correct a number of errors.

Arithmetic error codes: this family of codes is useful for the design of computational unit since they serve to detect errors in the results produced by arithmetic processors as well as the errors which have been caused by faulty transmission or storage [Aviz71b].

Duplex mode operation: errors observable at the boundary of a module can be detected by running two identical modules in lockstep and having their outputs feed a comparator [Down64, John84, AMD87].

Alternating logic: this technique achieves its fault detection capability by utilizing redundancy in time instead of the conventional space redundancy and is based on the successive execution of a required function and its dual [Reyn78]. Logic designs of binary adders and multipliers using alternating logic have been described in [Take80].

Self-exercising self-checking modules: to prevent accumulation of dormant faults, modules can be made self-exercising [Renn86]. Additional logic is added to the basic circuitry of a module in order to periodically activate and test all parts of the module. This logic can be combined with a simple recovery mechanism (e.g. Hamming code) to allow rapid recovery before multiple errors build up. Memory designs making use of this technique have been proposed in [Renn86] and the methodology has been extended to the design of a complete processor in [Chau88].

2.2. Micro Program Control Unit with Concurrent Error Detection

The techniques mentioned above can be used individually or they can be combined in order to enhance the level of fault tolerance that a system can reach. As an example, we look at the VLSI implementation of a microprogram control unit with concurrent error detection [Yen87]. The paper describes several techniques used in order to make the unit self-checking. Specifically, the techniques used are:

— coding techniques for detecting unidirectional errors

— dual-rail codes for a few control signals

— duplication for some of the submodules

— layout-based code-word checkers

— the PLA is strongly fault secure and strongly code disjoint

— the PLA is implemented using a modified Berger Code [Mak82]

The addition of these techniques to a basic control unit (in this case, an Am2910 look-alike) is costly in terms of area, and impairs performance because of additional delays required to compute codes and because of added capacitance on various parts of the circuitry. The Concurrent Error Detection (CED) method is then compared with a simpler method consisting of a duplication of the complete functional modules (duplex mode) [Down64, John84, AMD87]. The conclusions extracted from this article (in quotes) and our own remarks are:

(1) ''The duplex method is easier to design''.

(2) ''The duplex method has extensive fault coverage for faults in one of the duplicated modules''.

(3) The authors estimate that ''the duplex mode method would take 138% more area than a simple Micro Control Unit (MCU) which represents 20% more area than the Concurrent Error Detection (CED) method''. This calculation based on an estimate of the area taken by check bits generators. In [Trem89a] we discuss the implementation of data compression circuits and we believe that the area overhead could be reduced significantly compared to the estimates in [Yen87]. Specifically,

we believe that efficient compression and comparison circuits combined with the duplication of the modules should lead to a more reasonable overhead of 110% (compared to 138%).

(4) ''The duplex method has slightly less performance''. The extra logic needed for the CED method is part of the control circuitry and is thus bound to affect the timing of critical paths on the chips, resulting in decreased performance. With the duplex method, the basic chip is kept intact so that the performance of the basic processor is not degraded. However, if the processors must wait for the outcome of the comparison at the end of every cycle, performance can be significantly reduced. If there is no need to wait for the output of the comparison, the performance degradation can be eliminated. This is possible if either there is no need to support fast local recovery or micro rollback is used. In the first case, after an error occurs, the processors will continue to operate, with erroneous data, until the mismatch signal triggers system-level recovery. This recovery can then restore the processors to a state that was saved (checkpointed) many cycles prior to the occurrence of the error [Rand78]. In the latter case, as described in Section 1.1, micro rollback is used to ''hide'' the error detection latency. Once a mismatch is detected, the processors are quickly rolled back a few cycles, to their state just prior to the occurrence of the error.

(5) ''The CED method is able to perform fault diagnosis with greater resolution''. This is important for error recovery. If a low latency recovery is desired, a combination of the two methods can be performed as we shall see in Chapter 4.

The conclusions drawn from the article and our own observations indicate that the duplex method is more viable than the CED method, based on overhead in terms of area, performance, and ease of design.

2.3. Error Recovery

Once an error is detected by hardware checkers, actions must be taken to ''repair'' the damage caused by the fault and resume program execution. Two techniques described in [Rand78], called *forward error recovery* and *backward error recovery*, attempt to place the system in a *valid* state from which processing can resume. After describing these two techniques, a backward error recovery method related to micro rollback, namely *instruction retry*, is discussed.

2.3.1. Forward Error Recovery

Forward error recovery schemes attempt to make use of the erroneous system state, to make further progress. For example, in several operating systems, recovery procedures following a crash are based on the state of the system at crash time. Some jobs not affected by the crash will be able to proceed while other jobs, which were executing in main memory, may not be able to resume. Because correct identification of the error is often necessary to allow continuation of the process, forward error recovery schemes are designed as integral parts of the system they serve [Rand78]. An example of a detection of an error in a TMR system (Figure 2.1), and a subsequent recovery through forward error recovery, is shown in Figure 2.2.

At a lower level we can consider the use of Error Correcting Codes (ECC) for various systems such as the Cray X-MP, as a forward error recovery strategy. The detection of a bit error in a memory word is accomplished through separated

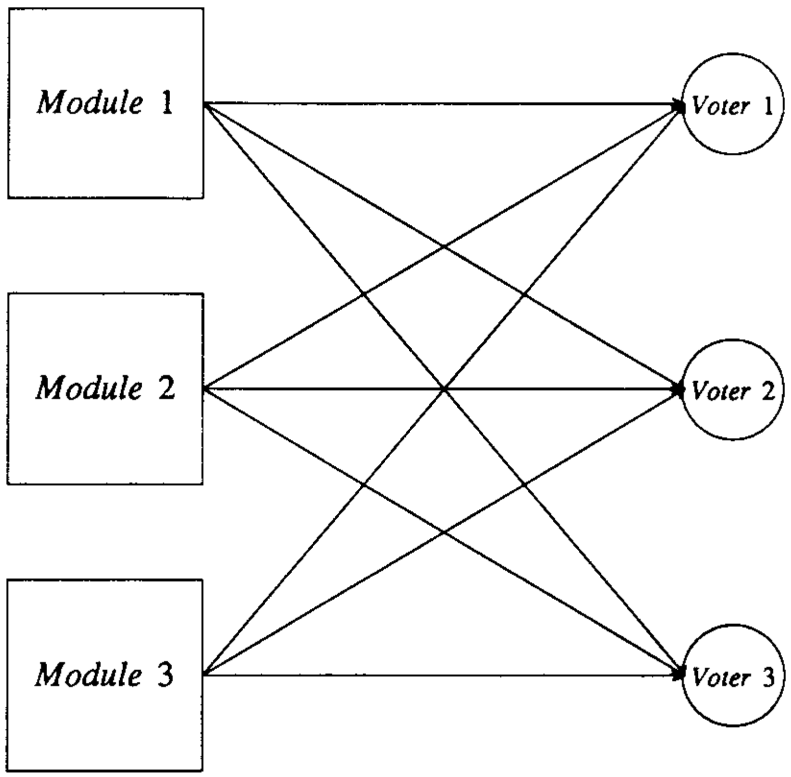


Figure 2.1: Triple modular redundancy with triplicate voters.

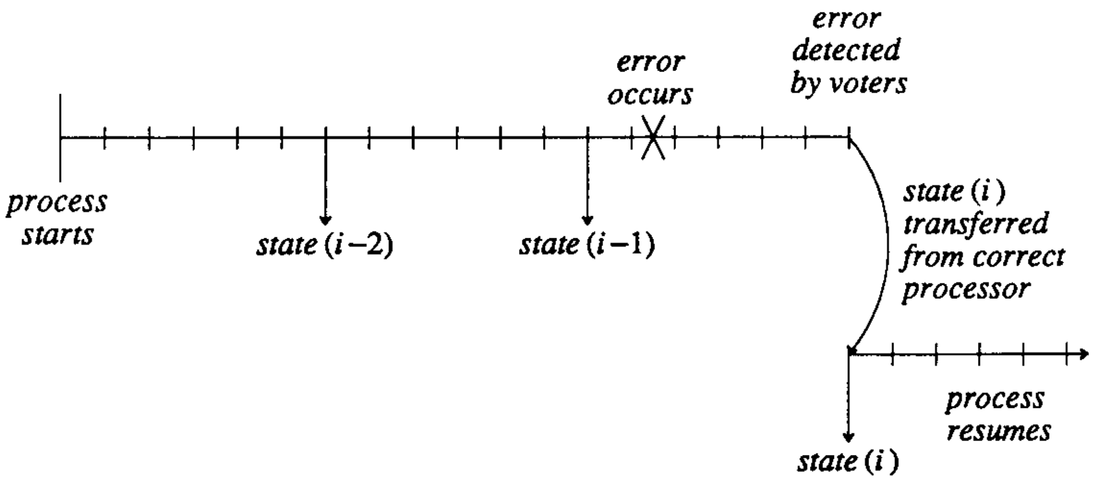


Figure 2.2: Forward error recovery through a ''roll forward''.

dedicated hardware which, upon the detection of an error, gives a *state* from which the correction circuitry can flip the specific bit so that process can go on [Cray84]. The circuitry for both error detection and error correction is connected serially with accesses to memory. Thus, even in case of an error correction, the process is not interrupted. However, delay is added to each memory access. In the case of the Cray X-MP, serial ECC takes 0.5 processor cycle (the latency of a memory access is 14 processor cycles) [Cray84].

Davis in [Davi85] describes the use of ECC for a high density CMOS memory chip, using the same principles discussed above. In his paper Davis mentions that the forward recovery scheme using ECC does not add any delay to the memory access time. This is only true because during a word access (16 bits), one of the two data bytes is delayed for 15ns by an RC chain to avoid excessive power-supply noise. During that extra delay of 15ns the ECC circuitry can correct any one-bit error, an operation taking 10ns, without adding extra delay. On the other hand, for a chip operating at its full potential, i.e. when both bytes are presented at the same time, the ECC delay would be added to the critical path, making the chip significantly slower.

2.3.2. Backward Error Recovery

Backward error recovery relies on the periodic saving of the *state* of the system, an action called checkpointing [Rand78]. The *recovery points* thus created, represent error-free states from which the system can resume execution. The action of bringing a process back to an error-free state is called *rollback*.

If errors do not occur often and if long recovery procedures and loss of work during recovery are acceptable, software methods can be used. Thus, checkpointing may be performed only once every few millions (or billions) of cycles and recovery may be software controlled so that the recovery process itself may take many thousands of cycles [Koo87].

In our research we are more concerned with systems designed for environments where error rates are high and/or real-time constraints prohibit significant delays for recovery. As we shall explain later, our work provides hardware-supported checkpointing and rollback.

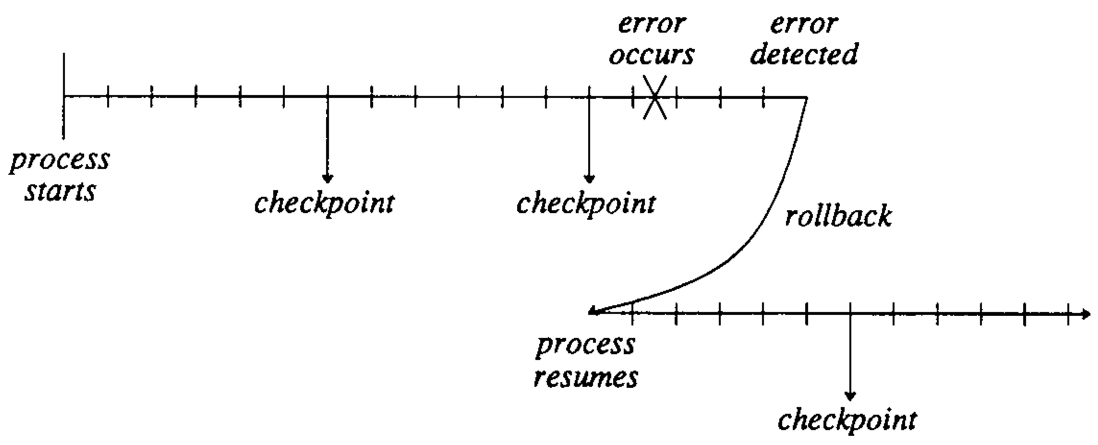


Figure 2.3: Checkpointing and recovery through rollback.

The backward error recovery concept can be used to ''hide'' the time to perform error detection and correction. Based on the assumption that errors do not occur often, it is wise to take any extra delay out of the critical path for a certain module or out of the communication path between two modules. If a module can checkpoint periodically, then error detection for actions occurring after a checkpoint can have a relatively long latency since the *state* of the module can always be brought back to the recovery point (Figure 2.3).

If there is enough time between two successive *states* to perform error detection and/or correction (plus a certain margin), rollback simply consists of aborting the execution of the current operation before an erroneous modification of the state occurs (Figure 2.4). In this case, the checkpoint is the state reached before the operation started, which is the current state since the operation was aborted before any damage occurred. The Memory Management Unit (MMU) of the IBM ROMP processor used the latter feature to take error correction out of the critical path for each access request by the processor to the MMU [Wald85]. The two-cycle memory access time required for the chip set includes: address translation, address and data buffering, and ECC error detection. Error correction time is not included in the access time, if an error is detected, the data sent to the processor is "intercepted" and the corrected data is then resent on a subsequent cycle. According to [Wald85] this practice reduced the impact of ECC on access time from 80ns to about 30ns.

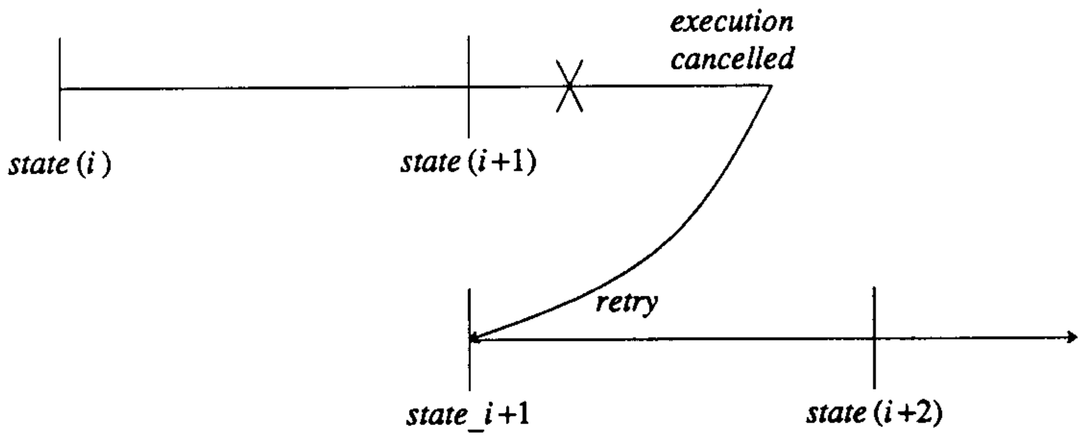


Figure 2.4: A short error detection latency allows cancellation of an instruction.

2.3.3. Instruction Retry

For systems in which the latency of the error detection mechanisms is too long for a simple cycle cancellation, a method called ''instruction retry'' can be used.

With the instruction retry method, the state of the processor is checkpointed at each *instruction* boundary. Then, upon the detection of an error *during the execution* of instruction 'j', the state present after instruction 'j-1' is restored, and execution resumes. In this way the error detection latency is stretched from one machine cycle to the instruction latency of the fastest instruction (smallest latency). But because of the various instruction latencies present in the instruction set of a processor, the storage for checkpointing each instruction must be based on the longest instruction or on the instruction which changes the most states. For example, if a processor executes instructions with latencies varying from 2 cycles to 8 cycles, the error detection latency must be smaller than 2 and the storage must be large enough to accommodate up to 8 changes of state. The controls for such a ''temporary storage'' become quite complex when a wide range of instruction latency is present in the instruction set of a processor.

The IBM 4341 processor uses instruction retry to recover from errors detected by the three following error detection methods: [Ciac81]

— duplication and comparison of the lines

— parity bit

— detection of special error conditions (e.g. invalid control register)

When a malfunction is detected, either the operation is successfully retried or the operation is aborted. Instructions are re-executed by restoring state information that is continuously saved and periodically purged by hardware. If the instruction needs to be aborted, information regarding the cause of the termination is sent to the *machine check interrupt process*. Ciacelli claims that the saving and purging does not affect the performance of the machine. For a VLSI processor we anticipate that instruction retry would impair performance slightly. Based on our experience [Tami91] we believe that the insertion of *shadow registers* for saving the processor state as well as the extra control signals needed for the retry mechanism would lead to increased bus capacitance and more complex controls, which both contribute to lower performance.

An implementation of error detection and *instruction retry* for a VLSI microprocessor is described in [Tsao82]. The error detection techniques implemented are:

— a self-checking PLA for the control part,

— parity checkers for the buses,

ALU errors are not detected unless the chip is matched with an identical chip running in master-slave configuration (duplex mode). The authors acknowledge that the error detection hardware is limited (due to area consideration) but they claim that if the chip is used in a master-slave configuration, single transient errors are all detected.

In [Tsao82], upon the detection of an error occurring *during* the execution of the *current* instruction, the processor has to capability to retry that same instruction. To accomplish this, a shadow register is attached to every single state register on the chip. The shadow registers hold the contents of the state registers which were present at the end of the previous instruction. If an error is detected, the modifications made to the state registers during the execution of the current instruction are discarded and the previous contents are restored from the shadow registers.

The method described is efficient for simple error detection mechanisms and for simple processors, such as the Fairchild F8 used as the target machine, but is not viable for more sophisticated error detection techniques combined with modern VLSI pipelined processors. Using the proposed instruction retry method, the detection of the error must occur before the execution of the following instruction (or when the *normal* state register is copied to the shadow register). In this case the detection must either (a) be executed serially with the execution of each instruction — a severe loss in performance, or (b) be executed in parallel — a severe time restriction limiting the complexity of the of the detection technique that can be used.

For modern processors, there are several other problems with instruction retry. First, the state of modern VLSI processor is large and precludes the use of shadow registers for every single register (the register file for example would have to be duplicated, adding from 32 to 128 extra registers to the layout). Secondly, modern RISC processors are heavily pipelined to increase performance. This has the effect of having several instructions executing in parallel in different stages of the pipeline. To ''retry'' a single instruction becomes more complicated due to the following reasons:

(1) Different instructions take different number of cycles to execute. The instruction retry mechanism would have to be designed to accommodate several

''kinds'' of instructions.

(2) Modern processors achieve high performance by supporting out-of-order execution of instructions. By definition, instruction retry requires the processor to roll back to a *precise* instruction boundary [Smit88]. The ability to roll back to a precise instruction boundary is also needed in order to support the normal semantics of interrupts and exceptions [Smit88]. Since instructions modify processor state out-of-order with respect to the order in which they are issued, complex dedicated hardware is required in order to support precise exceptions [Smit88, Hwu87]. The overheads associated with this hardware are significant, leading to many recent processor designs which avoid full support for precise exceptions [Sun91, Groh90]. Identical hardware is required to support instruction retry. Hence, instruction retry mandates the overheads associated with support for rollback to precise instruction boundaries. Micro rollback, on the other hand, can be implemented for these processors at a significantly lower cost (Section 4.2).

We will describe, in Chapter 3, error detection methods that offer high coverage at the expense of long error detection latencies (beyond a single instruction). We will also describe, in Chapter 5, multi-modules systems where errors signals are propagated throughout the system without having any knowledge of what kind of instructions different modules are executing. With instruction retry, it would be very difficult to synchronize the error signal with the execution of a particular instruction. The results from Chapter 3, Chapter 4 and Chapter 5, will show that it is desirable to have a repair mechanism that offers:

(1) Independence between processor performance and the error detection latency.

This allows the use of error detection mechanisms with long latencies without decreasing the processor performance.

(2) Independence between the retry mechanism and the instruction semantics of the module (in order to simplify the implementation)

(3) Independence between different modules. Modules do not need to know the specific instructions that other modules are executing. The synchronization of the retry mechanism should be simple and based on semantics-free synchronization points (e.g. cycles).

The technique that we propose, micro rollback, offers the advantages described above. It differs from instruction retry since rollback is performed on the basis of *clock cycles* rather than instructions. This allows rollback to be executed at the logic level, without keeping track of instruction semantics and instruction pipeline conditions. As a result, the micro rollback capability can be *independently* implemented in each module of a synchronous system, regardless of its function, by following very simple specifications — it must be possible to roll back all storage elements by any number of cycles up to a specified limit (Chapter 4). Building blocks which are capable of micro rollback can be interconnected in arbitrary ways to construct synchronous *systems* capable of micro rollback (chapter 5). Such flexibility is difficult to achieve if the semantics of rollback are tightly coupled to the specific function of each module.

2.4. Related Work

Comparison with precise interrupts methods

At the micro-architecture level, the technique we present for micro rollback of the register file has some similarities with schemes for precise interrupts in high-performance processors [Smit88, Hwu87]. Because high-performance processors are heavily pipelined, so that several instructions can executed simultaneously, and because of the different latencies of the processor's functional units, instructions are often allowed to modify the state of the processor in a different order from which they were issued.

During the execution of a program, if an interrupt occurs, the state of the running processor may not reflect the state that the processor would have if the instructions were executed sequentially. This occurs when instructions modify the processor state out-of-order. For example, an integer addition (short latency) issued after a floating-point divide (long latency) may complete and send its result to the register file before the divide completes. If the divide instruction later causes an exception (e.g. divide by zero, overflow, etc.), the current processor state does not represent the state present when the divide was issued. In order to resume operation quickly after an exception, it is essential to have access to the state present before the instruction causing the exception was issued. Special techniques have been proposed [Smit88, Hwu87, Mele89] to "recover" the proper *state* so that execution can proceed upon termination of the interrupt.

An analogy can be drawn between *repairing* the state of a processor in order to process an interrupt, and *repairing* the state of a module following the detection of an error. In both cases we have to undo portions of instructions that should not have been executed. In the context of out-of-order execution machines, higher performance is achieved by issuing instructions speculatively hoping that previously issued instructions will not cause an exception. In the context of fault tolerance, modules proceed conditionally hoping that the error detection mechanisms will not signal that an error has occurred a few cycles ago. In both cases special techniques must be provided so that the *state* of the processor/module can be repaired for execution to resume. We will give in Chapter 4 a more extensive comparison between the different techniques. Specifically we will compare the Delayed-Write Buffer (DWB) with a reorder buffer with bypasses [Smit88] and with the forward difference scheme [Hwu87]. We will show that micro rollback is inherently simpler — there is no need to keep track of instruction boundaries since the rollback event is transparent to the software. Also many of the proposed schemes for precise interrupts will be shown to require multi-cycle rollback, increased bandwidth to storage elements, and complex control.

Exception repair mechanism for the Motorola 88000

A high performance VLSI RISC processor, such as the Motorola 88100, in which several *Special Function Units* (SFU) can operate in parallel, requires special hardware to be able to handle exceptions. The processor's SFUs have different pipeline latencies and exceptions can occur at various stages of the pipeline. Because of the complexity of building an *exception repair mechanism,* Motorola chose a combination of hardware components and software routines to handle exceptions.

The processor contains several "shadow" registers which are copies of the register they accompany. For example, each one of the Execute, Next and Fetch instruction (pointers to instructions) have a shadow register. Upon the detection of an external interrupt or an internal exception, the shadow-freeze (Sfrz) bit of the processor-status register (PSR) is set and all the shadow registers freeze. All SFUs also freeze and the instruction unit fetches the instruction pointed to by an exception vector.

Handling of a trap instruction is slightly different. The machine "synchronizes" itself before taking the trap. The SFUs are emptied, memory accesses are completed, and the shadow registers are frozen. Trap processing then starts at the instruction pointed to by the trap vector.

The shadow register method offers the following advantages:

— it restores the processor state *quickly* following an exception,

— it is relatively inexpensive,

— it "frees" the normal logic for exception processing.

But it also presents the following disadvantages:

— if nested exceptions occur, the shadow registers must be saved by a software routine before jumping to the next exception (the shadow registers must also be restored upon returning from the second exception).

— it does not provide precise exception for all cases.

Imprecise exceptions occur for the Motorola 88000 because some exceptions are reported late in the floating-point pipelines. To resume operation following an imprecise exception, several registers containing the 5-bit opcode that identifies the instruction type, the exception handler that became enabled, and the destination register for the result (plus other information), are provided. Using that information, Motorola claims that software routines can complete instructions that caused imprecise exceptions. Unfortunately using this method the processor state cannot always be restored to a correct state since the program counter cannot always be restored to the instruction following the exception-causing instruction (e.g. when a branch occurs) [Mele89]. For these cases, precise exceptions can be provided by serializing the execution of each floating-point instruction, which results in a significant reduction of performance in many applications.

Chapter Three
Error Detection and Correction Circuits for VLSI Processors

In many applications the performance penalty of system-wide recovery cannot be tolerated, so it is desirable for modules to include mechanisms for rapid correction of most internal errors (i.e., local recovery). In other applications, some system-level actions for recovery are acceptable but high-speed checkers are needed to detect errors as soon as they occur and prevent the spread of erroneous data throughout the system.

Many of the techniques used to detect and correct errors caused by hardware faults rely on a few basic components: encoders, decoders, comparators, and data compression circuitry. In a VLSI processor, coding can provide error detection and correction (EDC) for data kept in the register file and other storage (e.g. PSW, caches, TLB), as well as for data processed by the ALU (e.g. using arithmetic codes [Aviz71b]). For example, single-bit parity can detect an odd number of errors in registers, while Error Correcting Codes (ECC), such as Hamming code, provide error correction capabilities [Rao89]. Check bits must be computed every time storage is modified, and verified whenever storage is accessed. In many modern processors a modification or access of the register file can occur every cycle, thus requiring low latency and high throughput for the circuits generating and verifying check bits. To achieve higher coverage and to detect errors in other modules (not just storage), duplication and comparison can be used [Down64, Tami83] at either the module or chip level. To minimize detection latency, values of internal nodes of modules should be compared each cycle. If most of the internal state is not observable from existing pins, numerous extra pins may be required. In order to reduce the number of extra pins, values of key internal nodes can be ''compressed'' efficiently (on-chip before comparison), into signatures of a few bits, with only a small reduction of coverage [Sedm80, Davi81, McCl85]. Since the comparison is done every cycle, compression and comparison must also be performed with low latency and high throughput.

The modules required to implement the error detection and correction techniques described above, consist mainly of encoders, decoders, comparators, and data compression circuits. These circuits rely extensively on Exclusive OR (XOR) gates. Alternative implementations of multi-input XOR gates are presented in Section 3.1. The different implementations are evaluated with respect to performance, area, and noise margins. The evaluation is performed in the context of the microarchitecture of a VLSI RISC processor where such modules might be used for error detection and correction.

In Section 3.2 we describe and evaluate circuits for implementing Error Correcting Codes (ECC) based on Hamming Code, in which code generation and error correction require multiple parity circuits. Through proper choice of high-speed parity circuits and the specific code to be used (M-code [Cart76, Rao89]) fast correction and check bit generation are achieved.

In Section 3.3 we discuss the comparators and data compression circuitry needed for implementing duplication and comparison. When the two modules whose outputs are being compared are on different chips, compressing the data and sending it off chip for comparison may introduce significant delays in system operation. This potential performance penalty can be greatly reduced (as we will explain in details in subsequent chapters) by using *micro rollback* [Tami88b], which allows detection to be performed in parallel with normal system operation.

3.1. Implementation of Multi-Input XOR Gates

Multi-input XOR gates are key building blocks for many error detection and correction circuits. For example, a single parity bit is generated by XORing all bits in a data word. A single parity codeword is verified by XORing all bits in the codeword. Check bits in error detection and correction codes based on Hamming codes are also generated and verified by computing the parity (XOR) of a subset of the bits in the word [Rao89].

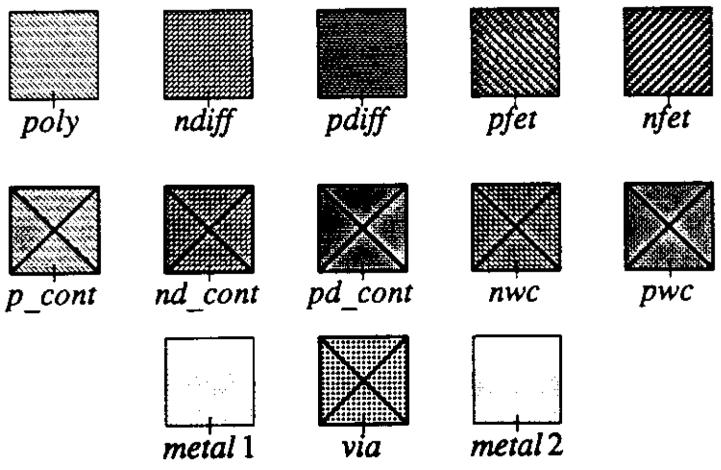


Figure 3.1: Representation of all layers for our layouts.

In this section we describe and evaluate several different implementations of multi-input XOR gates. Our implementation technology is double-metal 2 μ CMOS (MOSIS SCMOS design rules with $\lambda = 1\ \mu$). Figure 3.1 shows how the different layers are represented throughout this thesis. The evaluation criteria are speed, size, and noise margins. The speed is determined by circuit simulation using SPICE. The high noise margin (NM_H) is defined as the difference between the minimum *high* output voltage of the driving gate and the minimum input *high* voltage recognized by the receiving gate [West85]. The low noise margin (NM_L) is defined as the difference between the maximum low output voltage of the driving gate and the maximum input low voltage recognized by the driven gate [West85]. Our circuit simulations show that for static CMOS circuits (with $V_{th} = 0.74\ Volt$): $NM_L = NM_H = 2.1\ Volts$. NM_L is kept at 2.1 *Volts* in all the circuits described in this chapter. NH_H varies depending on the technique used and is thus the value that we report in the discussion of the circuits.

The most appropriate XOR implementation cannot be selected without considering the specific use of the circuit: where on the chip it is connected, pitch matching with other circuits, use for single bit parity or for multiple check bits in an error-correcting code, etc. As a specific example, we discuss the use of multi-input XOR gates for error detection and correction in a VLSI RISC processor. A simplified datapath of a VLSI processor is shown in Figure 3.2. Two registers can be read simultaneously from the register file and their values are transferred over internal buses to the shifter or the ALU. The internal buses are also used to transfer the results of the ALU or shift operation back to the register file [Sher84]. If the register file stores redundant bits for error detection and/or correction, values read from the register file must be sent to checkers, which verify and/or correct the data. Since two values are read simultaneously, two checkers are needed, each connected to one of the internal buses (Figure 3.2).

If the XOR gates are used for generating and checking the parity of data being

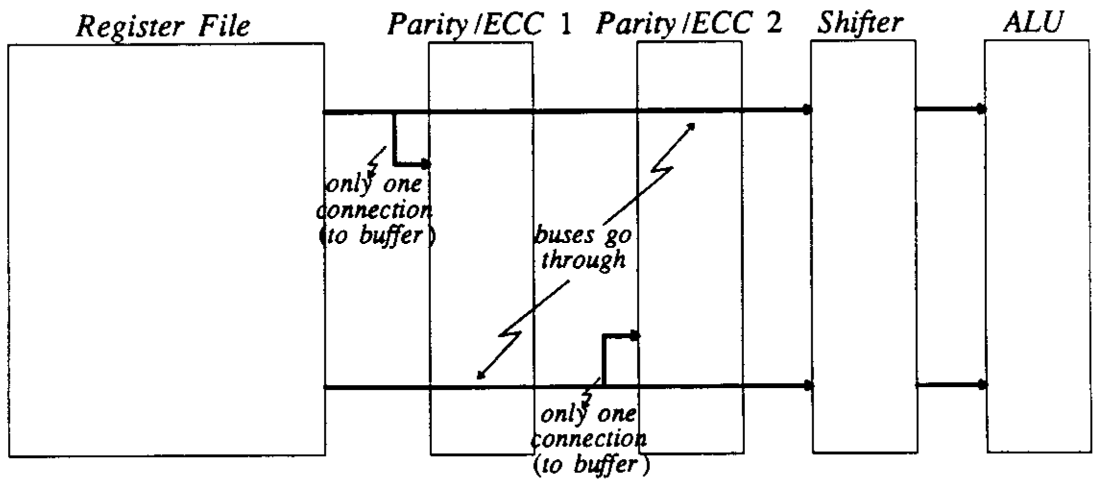
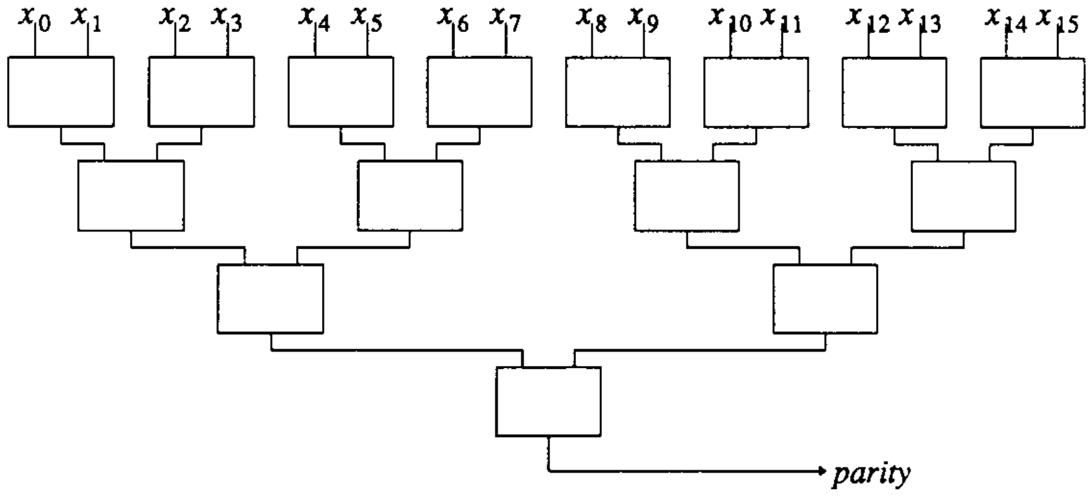


Figure 3.2: A simple processor with a two-port register file. Two bus lines are routed over the parity/ECC circuitry.

transmitted over a bus, the pitch of each cell must match the pitch of the data bus. We will first discuss the performance (delay) and size (chip area) of checkers attached to a two-port register file designed with a pitch of 50 λ. This pitch was chosen to match the pitch of a simple processor datapath. It is the result of a compromise between two conflicting factors: (1) a smaller pitch minimizes the area of the register file, and (2) a larger pitch simplifies the layout of other datapath cells (ALU, shifter, etc) and results in a shorter datapath.

We will describe how the metrics of the parity circuits change when the data bus pitch is reduced to 39λ. The advantage of this pitch is that it allows a smaller implementation of the register file (numbers are given in a latter section). We will also discuss the implementation of these circuits for use with the relatively large datapath pitch (74λ) of the SPUR processor [Lee89]. Given a fixed pitch, the only flexibility is with respect to the size of the cell in the direction of the bus, we henceforth called this size the *stride* of the circuit.

As shown in Figure 3.2, the internal buses are routed over the parity or ECC circuitry. For high performance, these internal buses are implemented in metal, typically second-level metal. The two bus lines over each cell of the parity or ECC circuitry limit the use of second-level metal in these cells, thus making the layout of the cells more difficult. The parity/ECC circuitry can reduce the performance of the datapath due to two sources of added delay: (1) the gate capacitance of the buffers which drive the parity/ECC circuit, and (2) the additional capacitance of the bus lines, which must be lengthened to accommodate the parity/ECC circuits.



x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12} x_{13} x_{14} x_{15}
parity

Figure 3.3: 16-input tree of XORs.

3.1.1. Static Implementation

An M-input XOR gate can be implemented as a tree of 2-input XOR gates (Figure 3.3). The resulting stride of a tree laid out as shown in the figure is: $(\lceil \log_2 M \rceil \times Cell_{stride} + Metal_{line})$. The $Metal_{line}$ term is the stride for routing the parity signal, which is a single metal line, out of the datapath. With SCMOS design rules, $Metal_{line}$ is 7λ (4λ for the line/contact and 3λ for spacing). In a VLSI layout, non-rectangular modules are undesirable since it is difficult to pack them efficiently on the chip and significant chip area is wasted. However, the layout of a binary tree can be compressed to only two rows of $\lfloor M/2 \rfloor$ cells by slightly modifying the design of the basic cell, and by reorganizing the layout as shown in Figure 3.4. In this case, the stride of the tree becomes $2 \times Cell_{stride} + ((\lceil \log_2 M \rceil - 1) \times Metal_{line})$. The second term is the stride required to route wires connecting XOR cells in the second row (routing wires below the cells in Figure 3.4), as well as for routing the parity signal out of the datapath. The layout of the XOR cell allows the first row to be connected to the second row without any extra stride for routing.

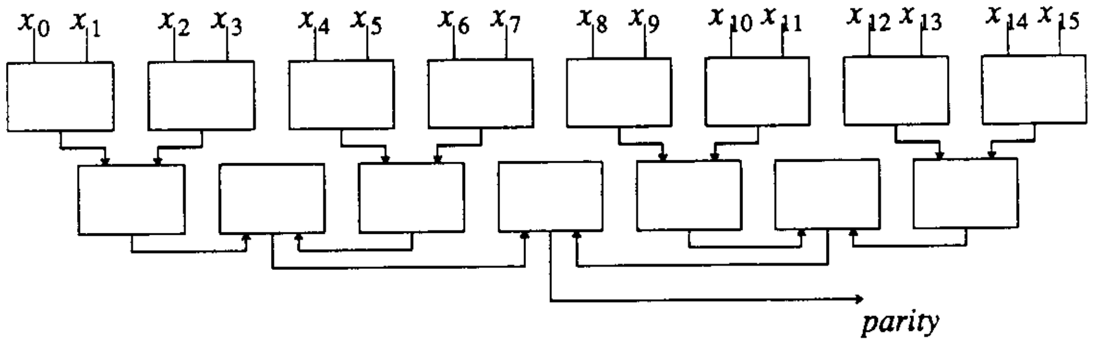

x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12} x_{13} x_{14} x_{15}
parity

Figure 3.4: Compact layout of a 16-input tree of XORs.

A possible static logic implementation of a 2-input XOR gates, is shown in Figure 3.5. The basic 2-input static XOR gate was laid out to match the pitch of 100 λ for every two bits leading to a *stride* for each cell of 23 λ (Figure 3.6). For the complete tree, the stride is twice the stride of the basic cell plus some routing, which adds up to 70λ (compared to 122λ for a straightforward layout of Figure 3.3). A 32-bit parity is generated in 9.25ns. Because of the wide pitch available

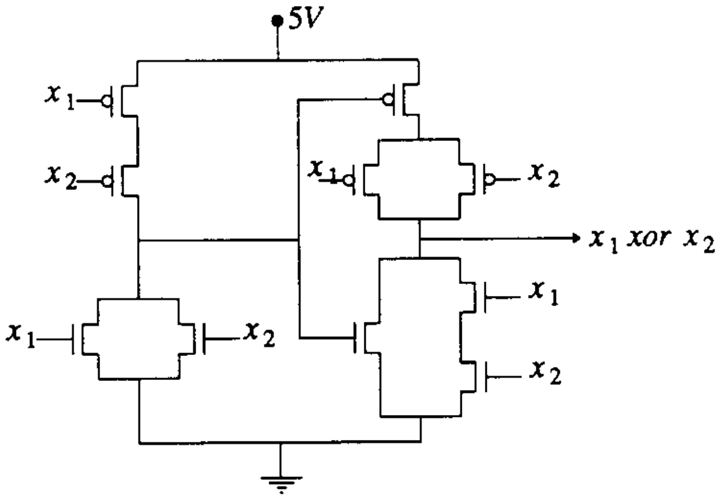


Figure 3.5: Static XOR

for a two-input XOR gate, 100λ, each gate is laid out with practically no second-level metal, simplifying the routing of the buses through the circuit.

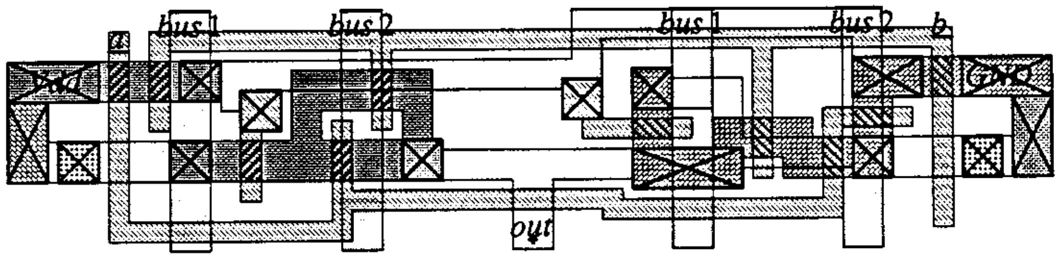


Figure 3.6: Layout of a static XOR gate

3.1.2. Chains of Switching Cells

An M-input XOR gate can be implemented using a chain of M *switching cells* [Siev82], as shown in Figure 3.7. The switching cell (Figure 3.8) consists of four pass gates that have the capability to either interchange the inputs so that $O_0 = I_1$ and $O_1 = I_0$, if $D_i = 1$ or to leave them intact if $D_i = 0$. The inputs I_0 and

I_1 of the leftmost cell are connected to Vdd and GND respectively. For a data word of M bits, with Y *ones*, the logic one signal entering the first cell will be interchanged Y times and will pass through the cells M-Y times. If Y is even then the outputs of the right-most cell will be $O_0 = 1$, $O_1 = 0$, if Y is odd, the outputs will be $O_0 = 0$, $O_1 = 1$.

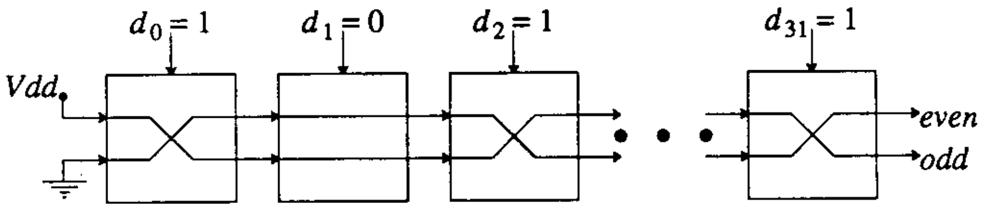


Figure 3.7: A chain of switching cells

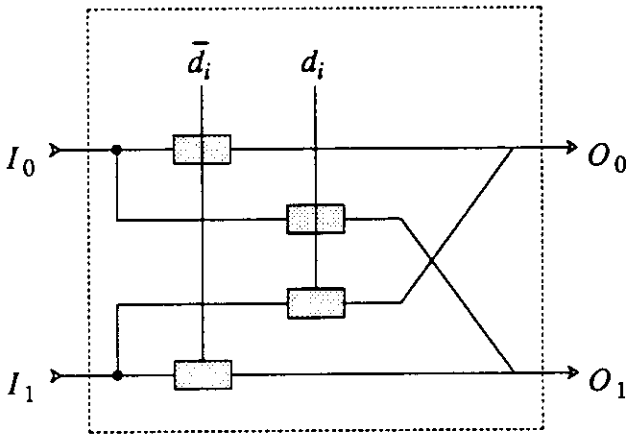


Figure 3.8: A switching cell

Since each switching cell requires both the true and complement values of the data lines $(D_0, D_1, \cdots, D_{31})$, inverters are connected to the bus to provide $\bar{D}_0, \bar{D}_1, \cdots, \bar{D}_{31}$. The stride of the complete parity circuit must include the stride of an inverter, which is around 25λ (depending on the width).

As with a Manchester carry chain adder, performance can be improved significantly by restoring the signal with a buffer every k stages, where typically $k = 4$ [Mead80, Siev82]. Hence, the width of the XOR cells must be significantly smaller than the pitch of the data bus. Four switching cells and a buffer must fit in four times the pitch. Due to this alignment, routing is necessary to bring the value carried by the bus to the XOR gates and back to the data bus, which also contributes to the total stride of the parity circuit (Figure 3.11).

Alternative implementations of multi-input XOR gates using pass gate chains are investigated in the rest of this section.

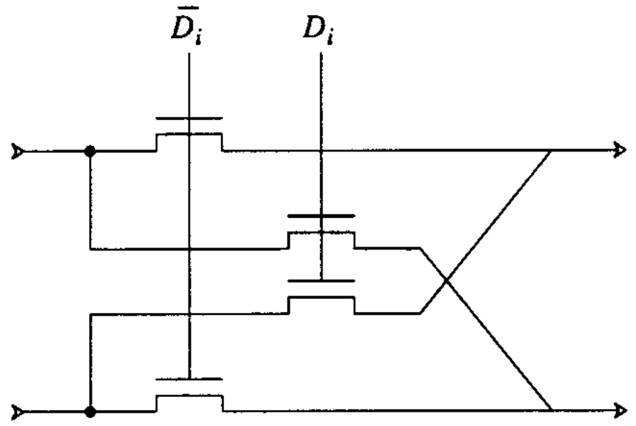


Figure 3.9: A switching cell implemented with N-transistors

3.1.2.1. N-Chain

The simplest implementation of a switching cell uses four N-transistors (Figure 3.9). Four such cells connected serially produce a result (the parity of four bits) with a delay of 1.5ns. For a chain of 32 cells, the delay grows approximately quadratically with the number of cells [Siev82], leading to a total delay of over 60ns to compute the parity.

As discussed above, performance can be improved by connecting a buffer in the chain every four cells, thus obtaining a delay that only grows linearly with the total number of cells. The longest delay for a sub-chain occurs when one rail, previously discharged, must be charged to Vdd. If the buffer is a simple inverter, the total delay for the chain consists of the time to charge the first sub-chain, then discharge the second sub-chain (since the value is inverted), and so on, for the next six sub-chains. The worst case delay for the propagation of the signal through the chain takes 41ns.

Due to the buffer every four cells, the width of the switching cells must be different from the pitch of the data bus. Four switching cells and a buffer fit in a width equal to four times the pitch (200λ). Our layouts show that for a buffer of 52λ and four switching cells with a width of 37λ $((4 \times 37\lambda) + 52\lambda = 200\lambda)$, the stride of the chain is 28λ.

If non-inverting buffers (two inverters for each rail) are used instead of inverting buffers, each sub-chain of four cells can be precharged to Vdd-Vth. With the two leftmost inputs connected to Vdd, since either d_i or $\bar{d_i}$ is asserted for each cell, all segments between the switching cells have a path to Vdd (through at least one N-transistor). Parity is then calculated by selectively discharging one rail through a chain of N-transistors and inverters. After a delay of 43ns for the precharging of such a chain, the discharge can be accomplished in 23ns. Hence, if the time for precharge is available, the actual computation of the parity is faster than with inverting buffers. In order to accommodate the more complex circuitry of the non-inverting buffers (width of 76λ), the width of the switching cell is reduced to 31λ $((4 \times 31\lambda) + 76\lambda = 200\lambda)$. This results in a small increase of the stride of the switching cell from 28λ to 30λ (Figure 3.10). We show in Figure 3.11

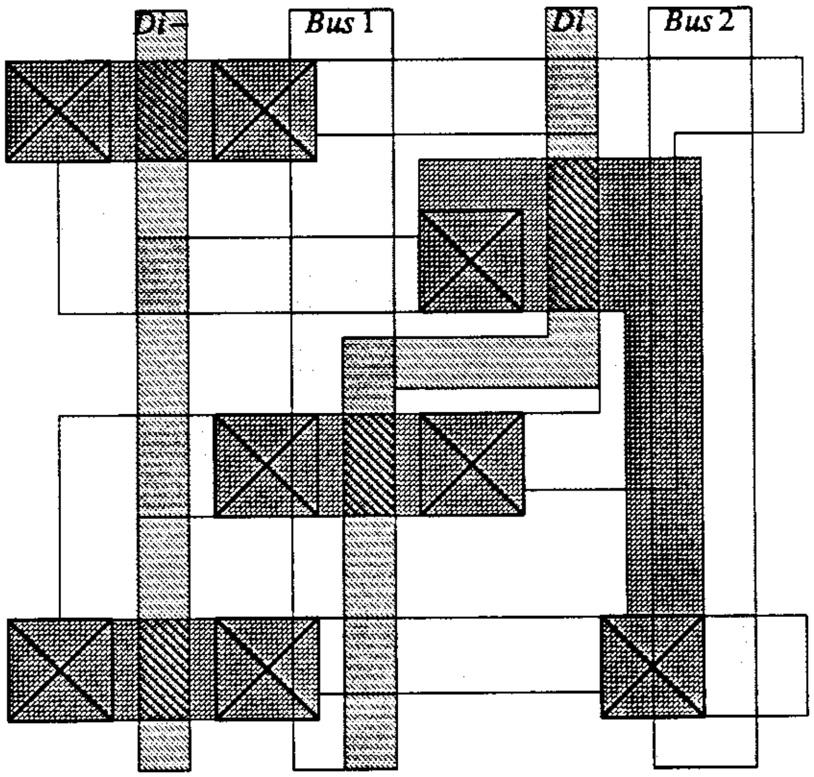


Figure 3.10: Layout of a switching cell implemented with N-transistors

the floorplan of a complete circuitry for a sub-chain of four cells. For the N-chain the total stride accounts to 73λ.

Circuits simulations indicate that, if the chain is implemented in an N-well process, the logic 1 levels at the internal nodes of the chain are degraded from $V_{dd} = 5\ volts$ to $3.55\ volts$, because a logic 1 is passed through N-transistors. This reduces the high noise margin from the normal $NM_H = 2.1 Volts$ to $NM_H = 0.65 Volts$ [West85]. The three methods discussed in the following three subsections improve the noise margins of the internal nodes at the expense of a small increase in area (stride).

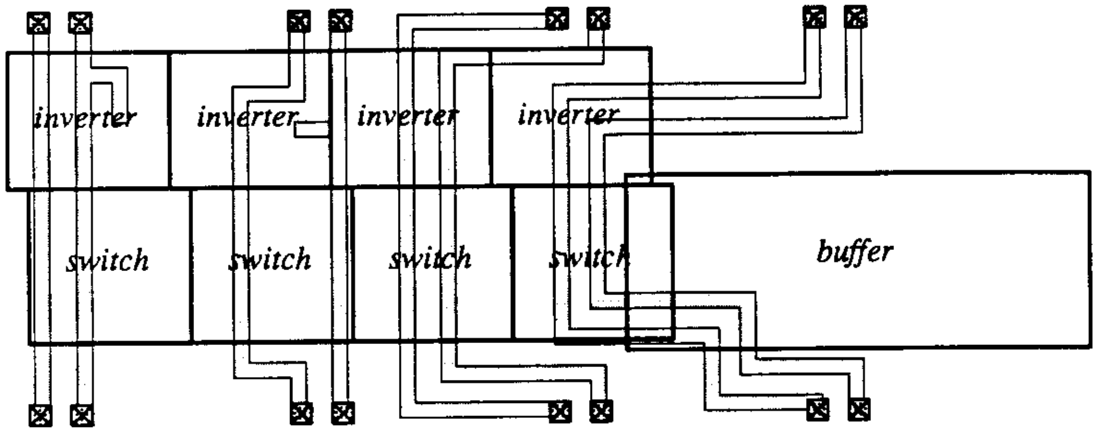


Figure 3.11: Layout of a sub-chain of 4 switching cells (n-chain) with their inverters, a non-inverting buffer and the necessary routing.

3.1.2.2. N-Precharged Chain

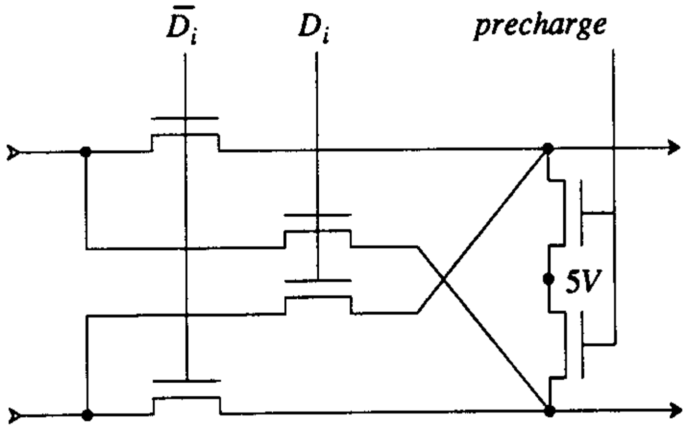


Figure 3.12: N-precharged XOR cell

It is possible to avoid passing a logic one through N-transistors by precharging the entire chain to one and discharging nodes through the N-transistors (Figure 3.12). In order to minimize the size of the cell, precharging is done through N-transistors. The critical path consists of passing a zero through the chain, which takes 23ns. In order to improve noise margins, the voltage of the precharge signal can be raised to $Vdd + Vth$, so that the nodes are charged to a proper logic one voltage level (Vdd). Our simulations show that with a precharge signal of 7 volts, the precharge of the whole chain to Vdd is done quickly in 1.25ns. This supposes the availability of a 7 volt voltage source, a severe constraint for a VLSI circuit. The noise margins are restored to normal levels and the delay for computing a 32-bit parity is 28ns (with double buffers in the chain). The stride of each cell is 32λ (Figure 3.13).

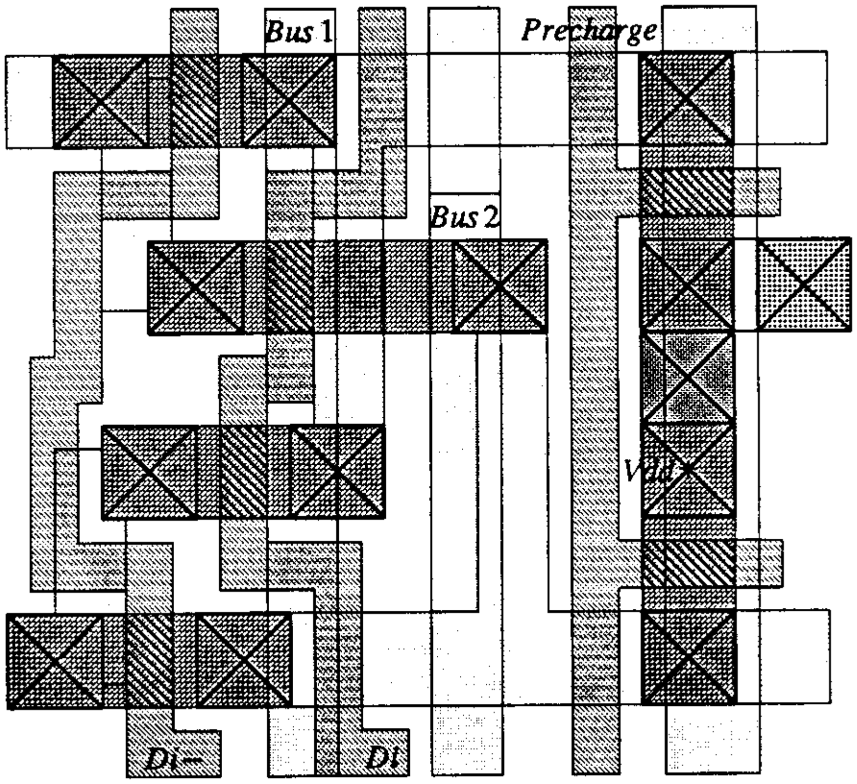


Figure 3.13: Layout of an N-precharged switching cell

Theoretically it is possible to eliminate the need for a second power supply. Bootstrapping can be used to generate this signal from a nominal voltage Vdd [Glas85]. We show an example of a bootstrap circuit in Figure 3.14 [Glas85].

The bootstrap capacitance C_{boot} is used in the figure to bring the voltage of node A to $V_{dd}+V_{threshold}$. That voltage when applied to the gate of transistor M_2 will precharge the output capacitance C_{out} to V_{dd}. A detail explanation of bootstrapping circuits can be found in [Glas85]. Bootstrapping requires a "bootstrap" capacitance that is several times larger than the capacitance of the nodes that are to be charged to the high voltage. For this circuit, the precharge signal is applied to the gates of 64 transistors so the boot capacitor must be more than 100 times larger than the minimum size transistor. To speed up precharging of the large capacitance, multiple stages of bootstrap buffers may be used. Taking all this extra circuitry into account makes bootstrapping impractical for implementing parity.

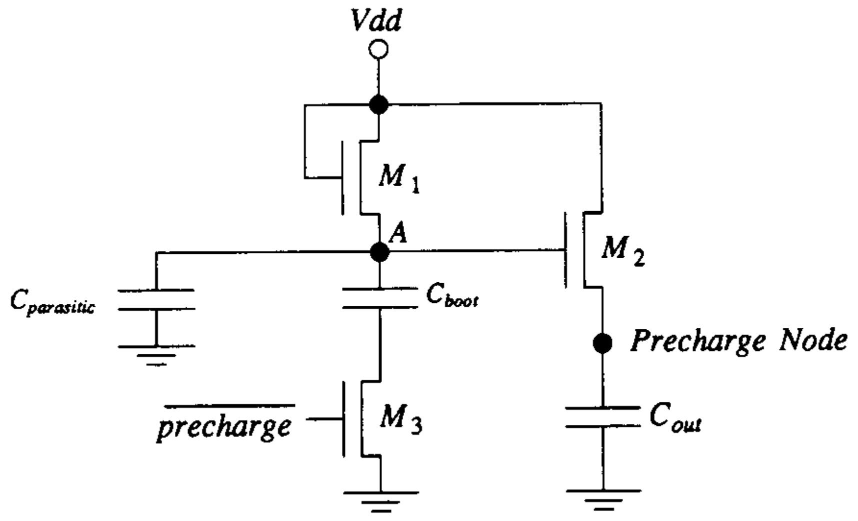


Figure 3.14: Bootstrap circuitry to charge a node to V_{dd} through N-transistors

The addition of N-transistors for precharging, results in small increase of the stride of the switching cell (32λ compared to 30λ for the N-chain). The buffer on the other hand can be made smaller since it can be stretched in the direction of the stride to 32λ. The width of the buffer is thus reduced to 72λ (from 74λ). The stride for the complete parity circuit is 74λ, only 3λ more than the N-chain (the size of the bootstrap circuitry is not included).

3.1.2.3. P-Precharged Chain

The requirement for a higher voltage precharge signal can be eliminated by precharging the chain through P-Transistors (Figure 3.15). Because of the presence of two wells in each segment, the stride of the XOR cell increases to 46λ. The size of this cell is constrained by the need to route vertically two data buses, two signal lines for power (for the precharging) and ground (for well plugs) and one line for the crisscross inherent to the switching cell (when $D_i = 1$ the signals switch rail), all in second level metal. These constraints lead to a cell width of $(2\times3)+(3\times4)+(5\times4) = 38\lambda$ as shown in Figure 3.16. The numbers above represent respectively the data buses width (3λ), the power lines and width of the via contact necessary for the crisscross line (4λ), and the required separation between second level metal lines (4λ). While the stride is increased, the noise margins are normal and the bootstrap circuit is eliminated. The delay to compute the parity through 32 switching cells and 8 double-buffers is 29ns, this following a precharge time of 1.5ns.

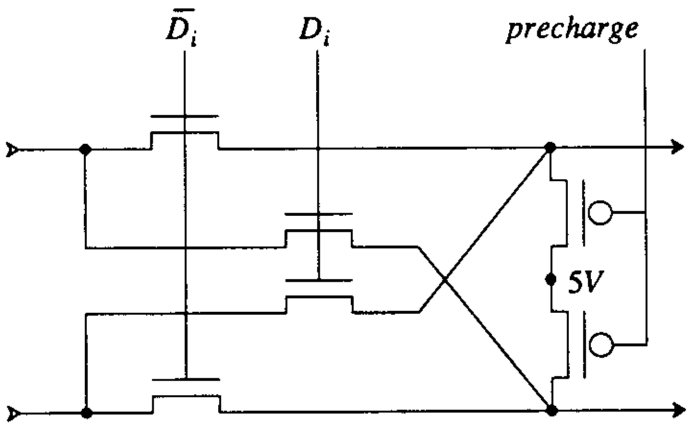


Figure 3.15: P-precharged XOR cell

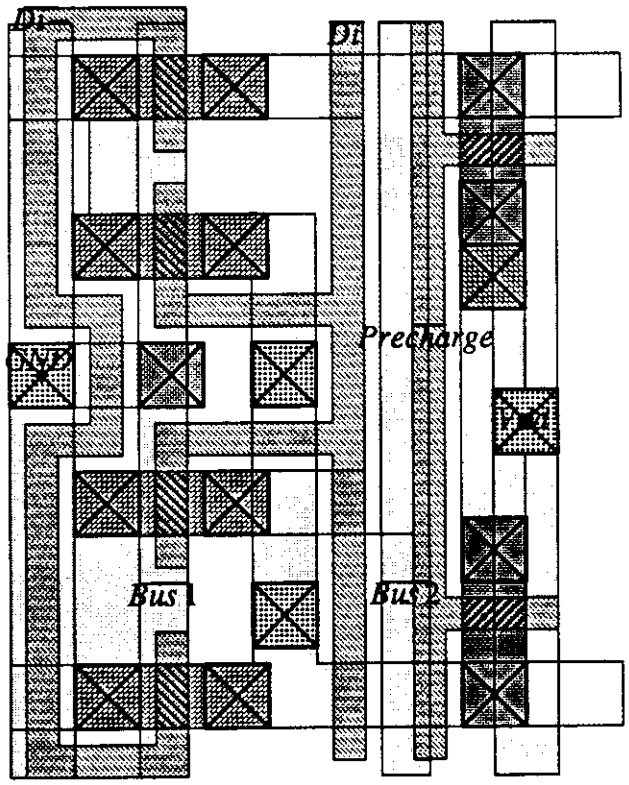


Figure 3.16: Layout of a P-precharged switching cell

3.1.2.4. Dual-chain

The switching cell can be implemented using full transmission gates (Figure 3.17). The noise margins are then maintained at proper levels but the stride of the basic cell, which now contains twice as many transistors and two wells, is 73λ (Figure 3.18). On the other hand, a double buffer can be implemented by "stacking" one buffer on top of the other reducing its width (48λ) and more importantly, use of second-level metal in the cell can be avoided, making the routing of the buses straightforward (using second-level metal). The total stride of the parity circuit is 101λ. The total delay for the chain is longer than the delays of the previous chains, due to the added capacitance at each node. A 32-bit parity is now calculated in 34ns.

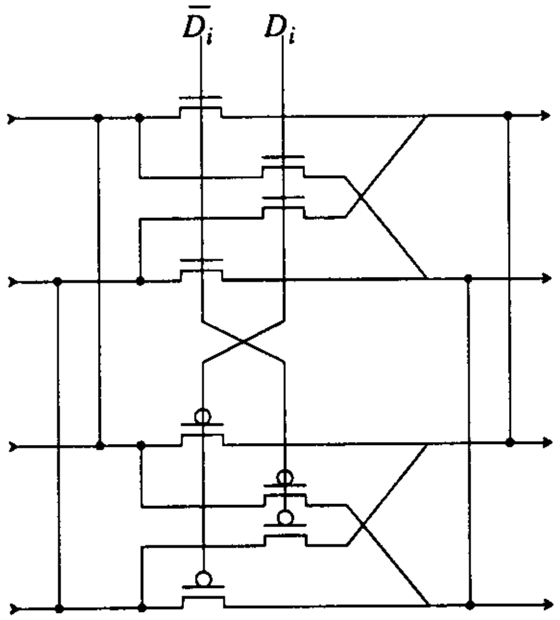


Figure 3.17: Dual-chain XOR cell

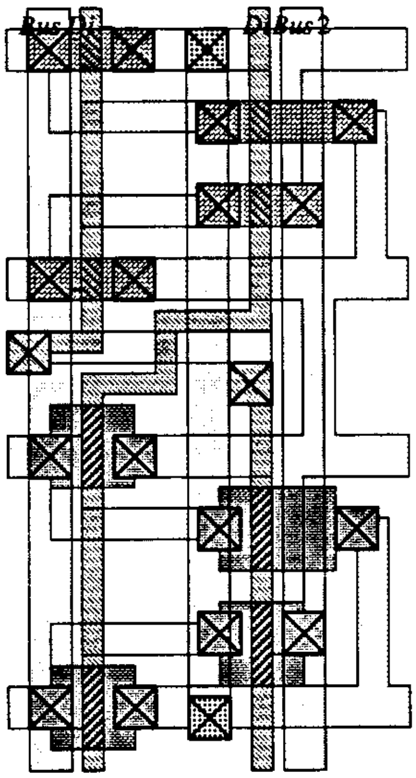


Figure 3.18: Layout of a dual-chain switching cell

3.1.2.5. Switching Cells With Sense Amplifiers

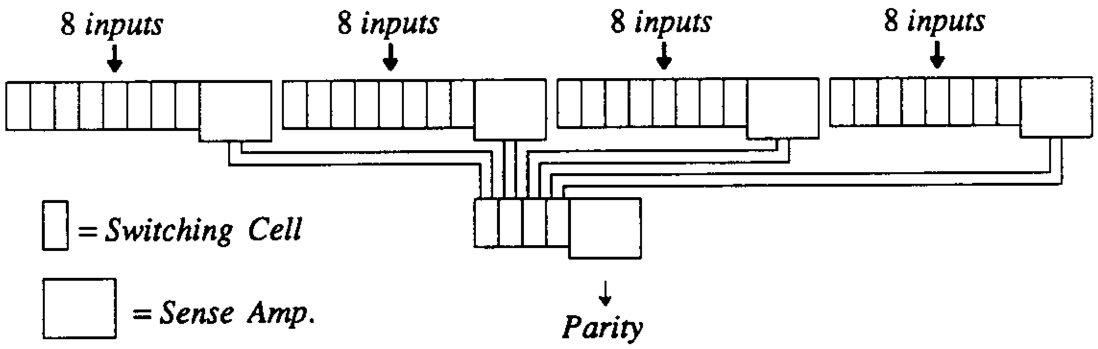


Figure 3.19: Tree of 8-input XOR with sense-amp.

Davis [Davi85] proposed an implementation of a multiple-input XOR gate using a 2-level tree of 8-input XOR gates (Figure 3.19). Each 8-input XOR gate consists of 8 switching cells and a sense amplifier (Figure 3.20). The chain of switching cells is initially discharged by setting the *trigger* signal to 0 volt. When all data input pairs ($D_1, \overline{D}_1, D_2, \overline{D}_2, \ldots, D_{32}, \overline{D}_{32}$) arrive, the trigger signal is set to 5 volts. At this point, the logic one propagates through the chain (from left to right in Figure 3.20) until it reaches the sense amplifier. The $\overline{latch}$ signal is asserted as soon as a voltage difference large enough for the sense amplifier to detect appears at the inputs. That signal connects the sense amplifier to the chain and activates the sense amplifier. The outputs of the sense amplifier (*output* and $\overline{output}$) provide the XOR and XNOR of the incoming data.

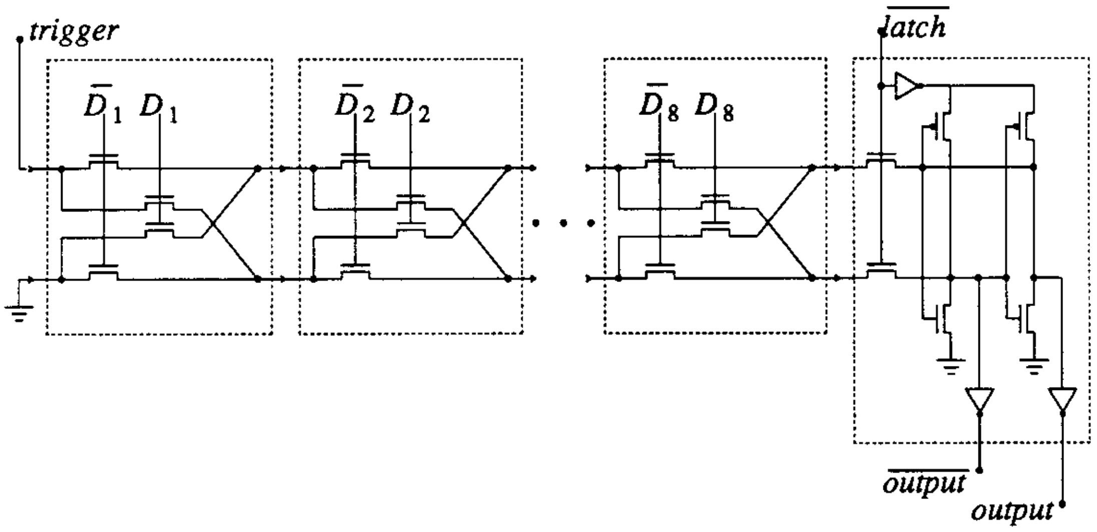


Figure 3.20: A chain of XORs with sense amplifier

The chain is fully discharged in 2.3ns. It takes 3.3ns to propagate the trigger signal through a chain of 8 switching cells. At this point the voltage difference appearing at the inputs of the sense amplifier is around 100 mV. For an additional safety margin, the sense amplifier is activated 0.5ns later ($\overline{latch}$ asserted). The second stage consists of a 4-input XOR gate and takes another 2.5ns. The latch signal of the second stage is asserted 0.5ns later. Total delay for this circuit is 6.8 ns, the fastest among the circuits discussed so far.

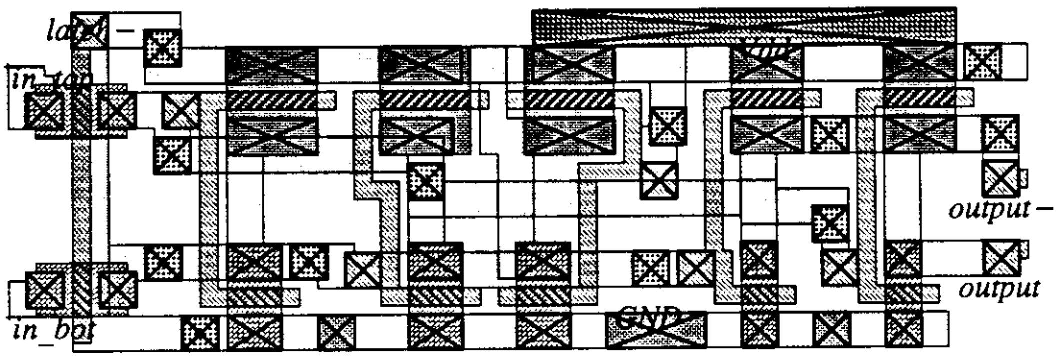


Figure 3.21: Layout of the sense-amplifier

Sense amplifiers speed up the calculation of intermediate results providing fast computation at the expense of large area. The sense amplifier cell contains significantly more logic than the switching cell, and therefore dictates the stride of the chain (Figure 3.21). The sense amplifiers are significantly wider than the non-inverting buffers and they are connected every eight cells instead of every four cells. Hence, the distance between some cells and the corresponding bus lines is larger than for previous chains, requiring more stride for routing. Adding this stride to the stride of the inverter, the sense-amp, and the stride for routing the first level to the second level 4-input XOR gate (which fits partially into the first level), we obtain a total stride for the parity circuit of 142λ.

For completeness, we note that a chain of 32 NMOS switching cells (Figure 3.9) connected to a single sense amplifier produces a parity bit after 31ns. The stride of this circuit consists of 28λ for the switching cell and 27λ for the inverter for a total of 55λ. A period of 28ns must be allowed for discharging the chain before operation.

3.1.2.6. Metrics of the Parity Circuits

The characteristics of the different XOR gate implementations are shown in Table 3.1. The circuits are designed to match the pitch of a dual-port register file (50λ). The table includes the stride and width of the XOR/switching cells, buffers, and sense-amplifiers. Table 3.1 also includes the stride of the complete circuitry, which consists of: an inverter for each input, the switching cell, and the necessary routing (for the static tree the stride consists of two levels of XOR gates and routing between the cells). The worst case circuit delay and the high noise margin (NM_H) of the chain are also shown.

Implementation	XOR/Switch.		Buffer/Sense.		Parity		Noise Mar.	Delay	
	Stride (λ)	Width (λ)	Stride (λ)	Width (λ)	Stride (λ)	Width (λ)	NM_H (Volt)	prech. (ns)	eval. (ns)
• Static tree	23	100	none	none	70	1600	2.1	—	9.3
• Chain + buffers									
N-chain (1 buffer)	28	37	28	52	70	1609	0.65	—	41
N-chain (2 buffers)	30	31	30	76	73	1609	0.65	43	23
N-precharged (5V)	32	35	32	72	76	1604	0.7	2	23
N-precharged* (7V)	32	35	32	72	76	1604	1.95	1.3	28
P-precharged	46	38	46	48	94	1608	2.1	1.5	29
Dual-chain	73	38	73	48	101	1602	2.1	36	34
• Chain + Sense-A.									
1-level	28	50	34	113	55	1713	N/A	28	31
2-level	34	31	34	152	142	1714	N/A	2.3	6.8
* with second voltage source									

Table 3.1: Metrics of parity circuits. Numbers given for "Parity" include the complete circuitry (inverters, XORs, routing, etc.)

3.1.3. Influence of the Data Bus Pitch on the XOR Gate Design

The above discussion of various XOR gate designs is based on a data bus pitch of 50λ. Other designs may have different requirements for the data bus pitch and the number of bus lines that must be routed over/through the cells. In this section we examine qualitatively and quantitatively the influence of the pitch of the data bus on the stride of the circuits. We discuss both smaller and larger pitches. As mentioned earlier, we picked two representative datapath pitches: the pitch of the data bus used in the SPUR project (74λ) [Lee86, Lee89], and the pitch (39λ) of a register file originally designed for minimizing the area. The three register files differ in terms of: (1) area — the SPUR register file being almost twice as large as the 39λ register file, (2) speed — the 50λ register file being slower than the 39λ file because of the longer poly select lines, and slower than the SPUR register file, which uses metal select lines, (3) ease of fitting cells of other modules into the datapath — a larger pitch makes this easier. Table 3.2 shows the characteristics of the three register files. For completeness, we have included the metrics of the 39λ cell and the 50λ cell with metal select lines. For register files with more than two ports, we anticipate that the larger pitch required to fit the extra circuitry for the added ports, will lead to sufficient area for passing additional bus lines over the cells (second-level metal is used almost exclusively for bus lines in our designs).

For the **static** cell, a decrease of the pitch only has a small impact of the stride. We have designed a static XOR gate for the 39λ pitch with a stride of 25λ, only 2λ more then for the 50λ pitch. For a larger pitch, one can see that the original cell is already stretched out in the direction of the pitch (Figure 3.6), to accommodate for a width of 100λ (2 × 50λ). Adding more space in the direction of

Cell	Pitch (λ)	Stride (λ)	Area (λ^2)	Delay (ns)
39λ poly select	39	32	1248	19
39λ metal select	39	42	1638	14
50λ poly select	50	28	1400	22
50λ metal select	50	37	1650	14
SPUR	74	32	2368	13

Table 3.2: Comparison of different register files used to evaluate the influence of the pitch on the parity circuits. Delays are for addressing and reading one of the 64 registers.

the pitch does not contribute to any significant decrease in the stride. Finally, the delay for computing parity is not affected significantly by the slight change in routing capacitances.

For a wide pitch (74λ), the stride of the **N-chain** (30λ) and the **N-precharged chain** (32λ) can be reduced to the same stride as the one-buffer implementation of the N-chain (28λ). This represents a decrease for the stride of 6.7% and 12.5% respectively. Those two cells can also be laid out for a pitch of 39λ without major modifications. The delays of the chains of switching cells for a larger pitch are approximately 2ns longer because of the added capacitance due to longer routing lines between each cell.

For a pitch of 50λ, the **P-precharged** cell has a relatively large stride of 46λ. That is mainly due to the required separation of the wells (in contrast with the N-precharged implementation which also has 6 transistors but a stride of only 32λ). The extra space provided by a pitch of 74λ allows one to compress the cell by separating the wells, leading to a stride of 32λ, a gain of 30% (Figure 3.22). As mentioned earlier, the width of the P-precharged cell (for a pitch of 50λ) is limited by the number of second-level metal lines (Vdd, Gnd, Bus1, Bus2, crisscross line) that are routed vertically. We did not succeed in laying out a cell that would overcome this constraint for a pitch of 39λ.

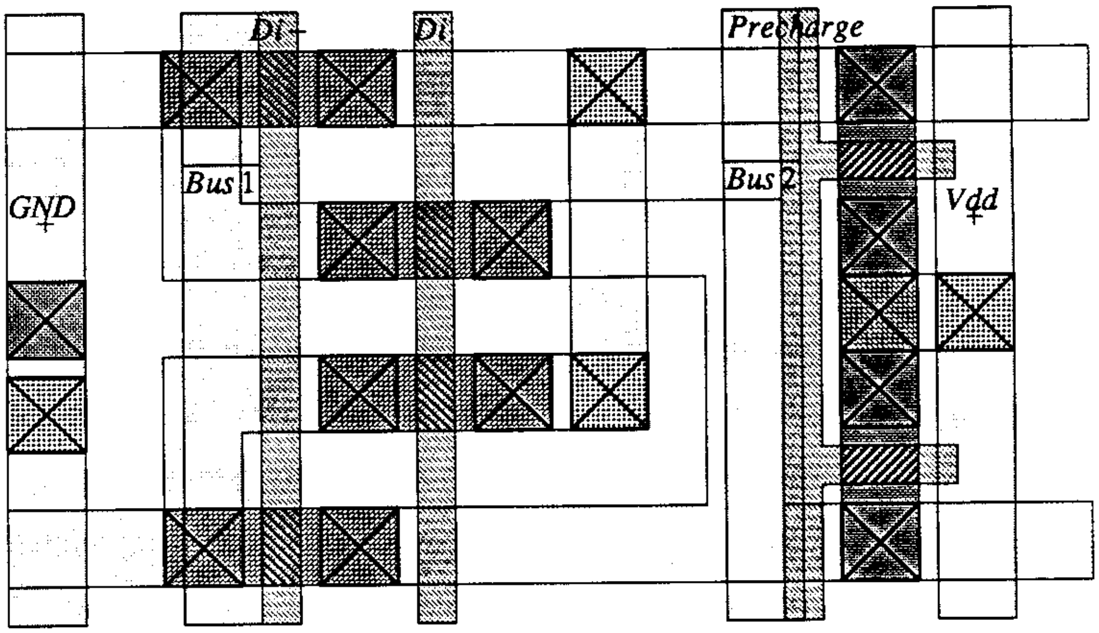


Figure 3.22: P-precharged XOR cell for a pitch of 74λ

The **Dual-chain** can also benefit from a wider pitch. As illustrated in Figure 3.23 the two sub-chains can be placed side-by-side horizontally instead of vertically as we had shown in Figure 3.17. One factor that helps reduce the stride is the sharing of the wells; adjacent cells have their wells interchanged. This brings the stride down to 32λ, a gain of 56%! The routing between the two sub-chains is done using second-level metal, leaving only room for two bus lines to be routed vertically (in the direction of the bus). The layout of this cell for a pitch of 39λ is equivalent to the original layout and the stride is not affected significantly.

The implementation using **Sense-amplifier** would only benefit slightly from a

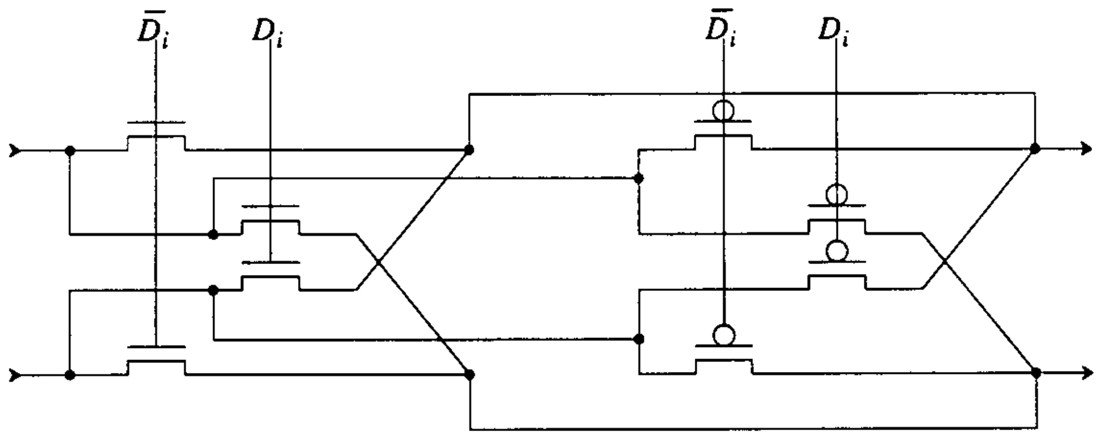


Figure 3.23: Dual chain of switching cells laid out horizontally

larger pitch since both the switching cell and the sense amplifier cannot be compressed significantly. Moreover, most of the overhead in the full parity circuit comes from routing the data bus and the two extra lines required to connect the first stage to the second one. These two overheads are not reduced by a larger pitch. To accommodate a pitch of 39λ, the switching cells can be made smaller (down to a $width = 27\lambda$) but that only leaves 96λ for the width of the sense amplifier. This reduced width leads to an increase of the stride of the sense-amp of 30λ. Our experience with different pitches indicates that sense amplifiers are more appropriate for pitches of at least 46λ.

In conclusion an increase of the pitch leads to comparable strides for the different circuits. This has the effect of giving more freedom to the designer to choose the implementation having either good noise margins or high speed. For a smaller pitch, some implementations are severely restricted and may prove not viable (P-precharged, sense-amplifier). The static implementation offers good compromise regardless of the pitch, while the sense-amplifier implementation presents the best speed but at the expense of large area. As we will see in the next section, the benefits of small but fast chains of switching cells becomes apparent in the context of error detection and correction. In that context, taking into account speed, noise margins, and area, the P-precharged chain may often be the best choice.

3.2. Error Correction Circuitry

The XOR circuits described in the previous section can be used for error detection with a single parity bit. Error correction codes (ECC) [Rao89] can be used for correcting errors locally without resorting to system-wide recovery. In this section we discuss the circuits required to to detect and correct errors in storage elements using error correcting codes based on Hamming Code [Hamm50]. The check bits of these codes are generated and verified using multi-input XOR gates. Each check bit is generated by XORing a different subset of the data bits. When storage is accessed, the same subsets *and* their corresponding check bits are XORed to produce a *syndrome*, which is used to correct some errors and flag others (e.g., multiple bit error) as uncorrectable [Rao89].

This section focuses on circuits which provide single error correction and double error detection (SEC-DED). The measurements presented are based on a SEC-DED code for a 32-bit word. We use a variation of the conventional Hamming code, called Maintenance code (M-code) [Cart76, Rao89]. Both codes require XORing several bits to obtain the syndrome but M-code has the advantage that the maximum number of bits to XOR is smaller, resulting in faster operation. For example, for a 32-bit word, the syndrome is generated by XORing at most 15 bits, while Hamming code as used in [Davi85] requires XORing up to 33 bits (32 data bits and one check bit).

	Data Bits																																Check Bits						
											1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	3	3							
i	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	c_0	c_1	c_2	c_3	c_4	c_5	c_6
g_0	1	1	1	1	1	1	1	1	1	1	1	1	1	1																			1						
g_1	1	1	1	1	1										1	1	1	1	1	1	1	1	1											1					
g_2	1					1	1	1	1						1	1	1	1						1	1	1	1	1							1				
g_3		1				1				1	1	1			1				1	1				1	1	1			1	1	1					1			
g_4			1				1			1			1			1			1			1	1	1			1	1	1	1		1					1		
g_5				1				1			1		1	1			1			1	1	1			1		1		1		1	1						1	
g_6					1				1			1		1				1			1		1			1		1		1	1	1							1

Figure 3.24: M-Code parity check matrix (PCM) for 32-bit Words

The parity check matrix (PCM) for a 32-bit word encoded according to a M-code is shown in Figure 3.24. The PCM shows which data bits are used for the computation of each check bit. Each row corresponds to a check bit. A "1" in a particular column of the PCM indicates that the corresponding bit is in the set of data bits XORed to generate the check bit. One characteristics of the PCM of M-codes is that each column and each row have an odd number of ones [Cart76]. The odd-weight-column property allows the discrimination of even number and odd number of errors and gives better multiple-error detection capability than the even-weight-column code [Rao89]. The ones in each row of the PCM represent the data bits (numbered 0 to 31) that are XORed to generate each check bit $(c_0, c_1, \cdots c_6)$.

The organization of the circuitry required to perform error detection and correction is shown in Figure 3.25. The syndrome generation requires seven rows of multiple-input XOR gates. Hence, minimizing the stride of the cells forming the XOR gate becomes more critical than for single-bit parity. The multi-input XOR

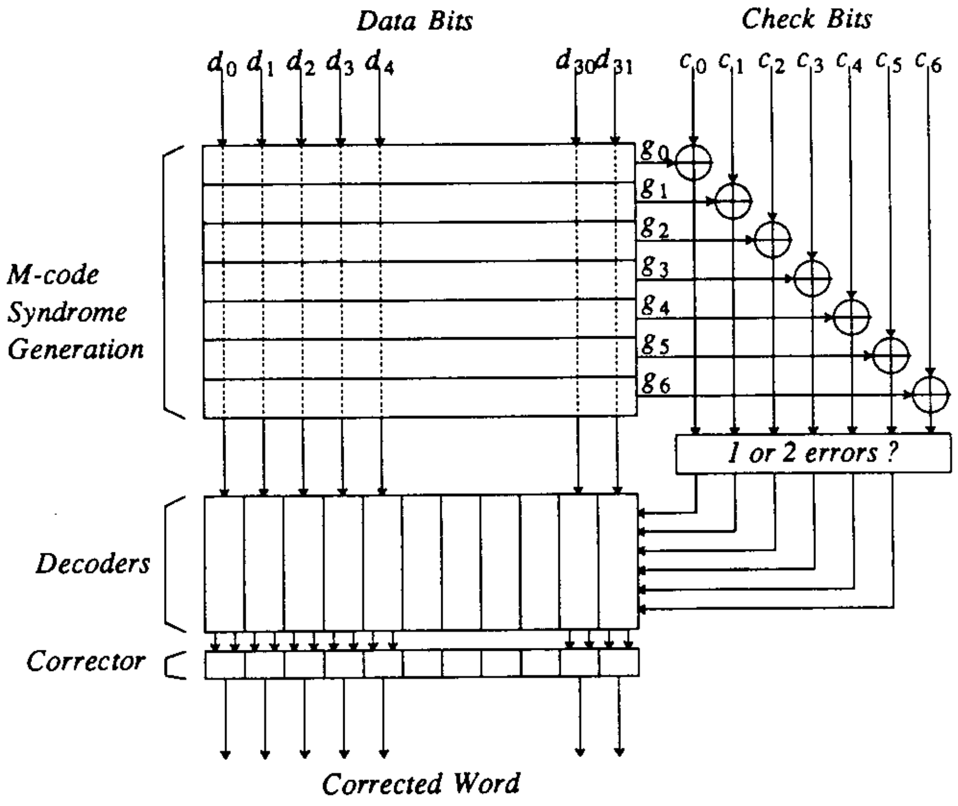


Figure 3.25: Error detection and correction of a 32-bit word.

gates can be implemented in the same way as the parity circuits described in the previous section. The stride and delay of the syndrome generation circuitry for the six different implementations are shown in the top part of Table 3.3. The total stride of the syndrome generation circuitry includes (from top to bottom): routing for pitch adjustment from the data bus to the inverters (pitch adjust in), the inverters, the switching cells, routing for pitch adjustment from the syndrome generator to the data bus (pitch adjust out).

For fast operation and good noise margins at the expense of large area, a static implementation of the multiple input XOR gates can be used for generating the syndrome. If area is the main concern, (disregarding noise margins), N-chain XOR

			Static	N-chain	N-prech. Chain	P-prech. Chain	Dual Chain	Sense Amp.
Syndrome Generation	stride (λ)	pitch adjust. in	2	4	4	8	1	33
		inverters	0	27	28	28	22	27
		Switches (XORs)	483	210	224	322	511	392
		pitch adjust. out	0	12	12	12	5	48
		total	**485**	**253**	**268**	**370**	**539**	**500**
	delay (ns)	precharge	—	17.0	1.0	1.5	16.25	2.5
		evaluate	7.5	10.0	12.8	12.3	16.0	5.0
Error Ident.*	area (λ^2)	**total**	49300					
	delay (ns)	**total**	5.6					
Correction	stride (λ)	decoder	138					
		muxes	60					
		XOR	58					
		total	256					
	delay (ns)	**total**	8					
EDC	stride (λ)	**total**	741	509	524	626	795	756
	delay (ns)	**total**	21.1	23.6	26.4	25.9	29.6	18.6
* outside of the datapath								

Table 3.3: Area and delay of alternative implementations of error detection and correction circuitry (*pitch* $= 50\lambda$).

gates can be used. Precharging with a higher-voltage signal can be added to obtain good noise margins, however there is a significant amount of additional circuitry required for bootstrapping. As with the parity generation, a P-precharged chain offers a good balance between speed, area, and noise margins. The large size of the dual-chain makes it impractical for use in the context of ECC. The sense amplifier implementation can achieve fast operation, at the expense of large area and high complexity for the generation of the sense-amplifier latch signals.

Once the syndrome is generated, it can be used to determine if there is a one-bit error, a double-bit error, or no error. Each data bit is involved in calculating exactly three check bits (Figure 3.24). If there is a single-bit error, three ones will

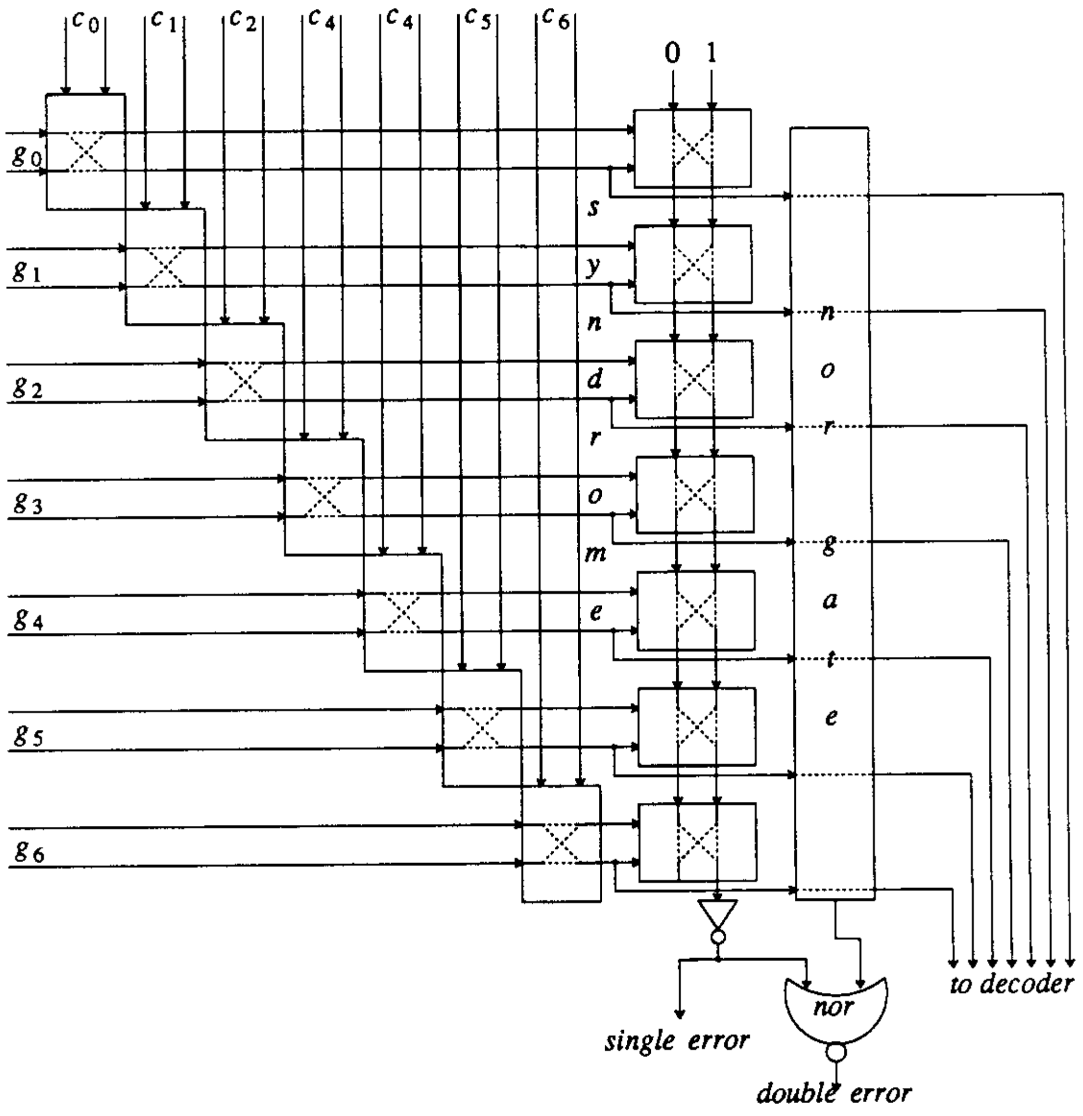


Figure 3.26: Identification of the error.

be generated in the syndrome. For a double bit error, two sets of three bits (among the seven bit syndrome) are affected, always resulting in an even number of ones in the syndrome. When the data word is error-free, the syndrome bits are all zero (since all the check bits match). Figure 3.26 shows the logic used to identify the location of a single-bit error and to detect the presence of two errors. This circuitry

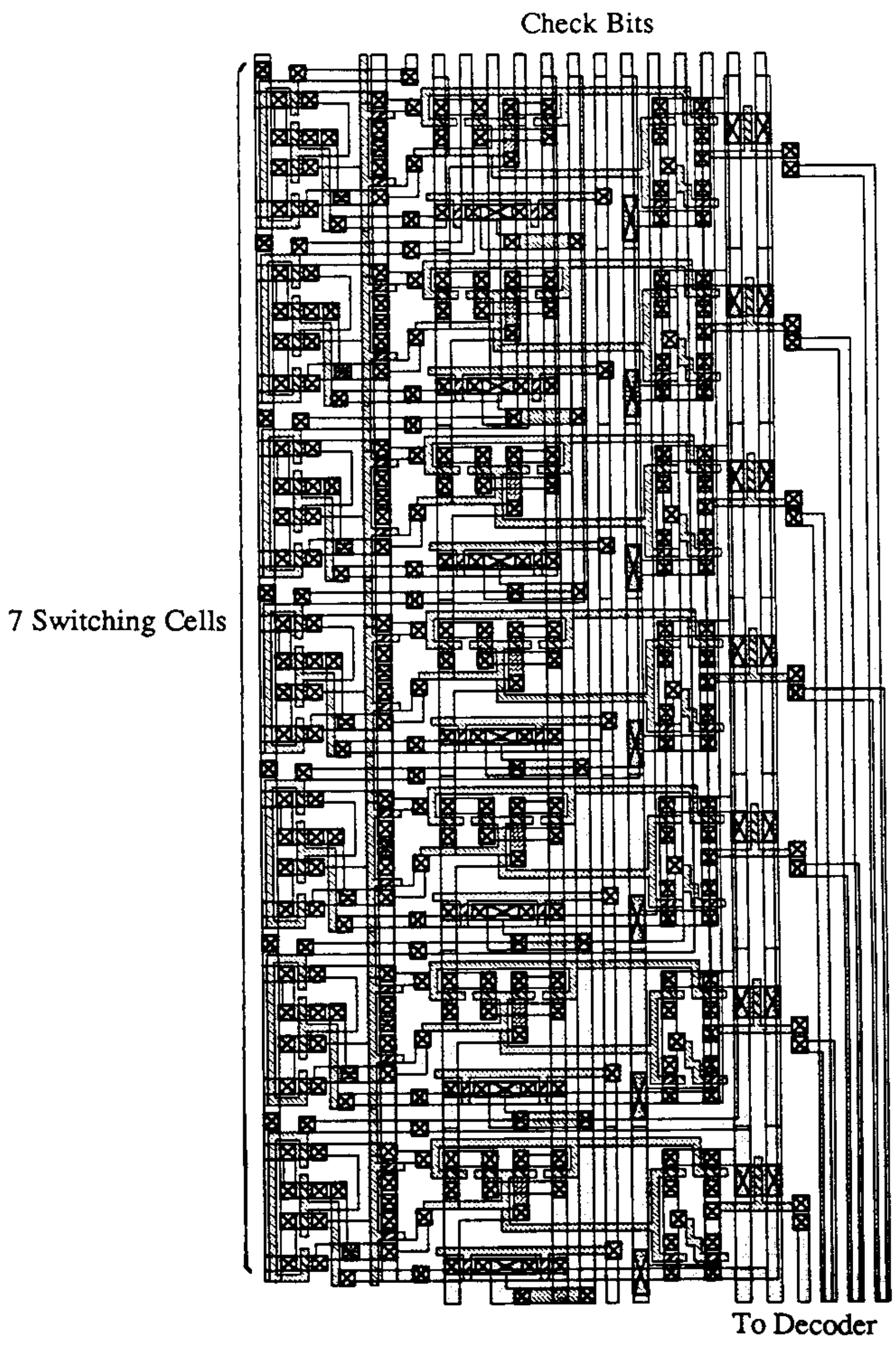


Figure 3.27: Layout of the circuitry required to detect and identify the error.

does not increase the length of the buses since it is connected to the error detection circuitry outside of the datapath. The layout of this circuitry, measuring $49300\lambda^2$, is shown in Figure 3.27. The seven XOR switching cells driven by the check bits $(c_0, c_1, \cdots, c_6)$ have been re-aligned to form a single column resulting in substantial savings in terms of area. To do so, the check bits have been routed over the circuitry using second-level metal. The signals *single error* and *double error* are conditionally asserted 5ns and 5.6ns, respectively, after the syndrome appears. The main component of this delay comes from the 7-input XOR gate that detects if the syndrome has an odd number of ones.

Once a single bit error is identified, the address representing the location of the faulty bit in the data word is routed to a decoder. The outputs of the decoder control XOR gates (the ''corrector'' in Figure 3.25), which invert the appropriate bit. The correction process takes 8ns. Other metrics of the corrector are shown in Table 3.3. In that table, the measurements for the total stride of the ECC circuits are for the circuits which are in the main part of the datapath and thus contribute to increasing the stride of the data bus.

The floorplan of the complete implementation of the error detection and correction circuitry, using P-precharged switching cells, is shown in 3.28. Other possible implementations of the detection circuitry (e.g.. static, dual-chain, etc), will not require major changes in the layout (except for the size of the error detection circuit).

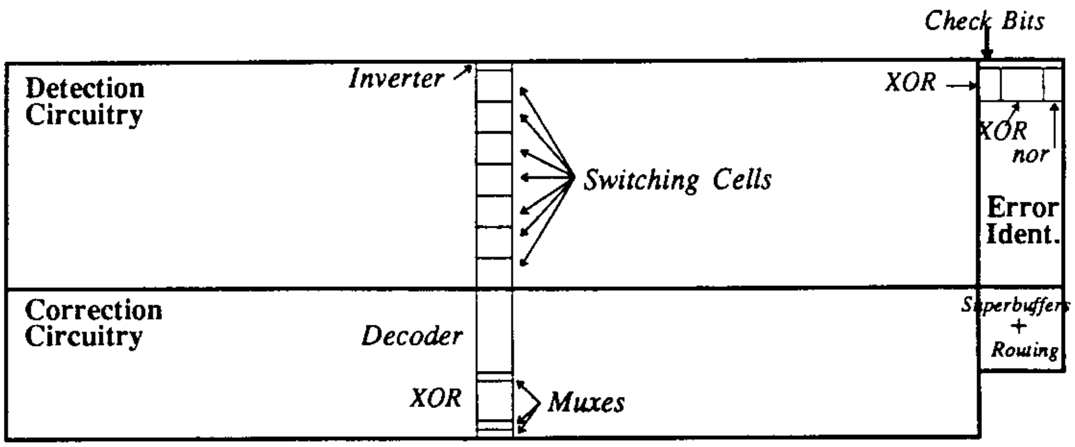


Figure 3.28: Floorplan of the complete error detection and correction circuitry. All modules are drawn to scale.

3.3. Duplication and Comparison

Using duplication and comparison it is possible to achieve high-coverage error detection for all types of modules [Sedm80, Tami83, John87]. Two identical modules process the same information in parallel and some of their output pins are compared every cycle (Figure 3.29). In this section we described the circuits needed for duplication and comparison. These include comparators as well as data compression circuitry necessary to reduce the pin requirements when the two modules whose outputs are compared are on different chips.

3.3.1. Compression

It is often impossible or undesirable to duplicate large VLSI modules, such as processors, on the same chip. Duplication and comparison with such modules requires using at least two chips. Hence, comparison is limited to the information available at the pins. Since many of the results computed by the processor do not immediately appear at the pins, if the comparison is based only on values available at the pins, the system is likely to have long detection latencies. This problem can be solved by including in the compression internal information, such as the output of the ALU, the PSW, and various state registers. The number of bits to compare may thus easily add up to more than one hundred, requiring many extra pins. In order to reduce the number of bits to be transmitted across pins to the comparator, the values on internal nodes can be "compressed," leading to large reduction in pin bandwidth requirements with a small reduction in coverage [Davi81, McCl85].

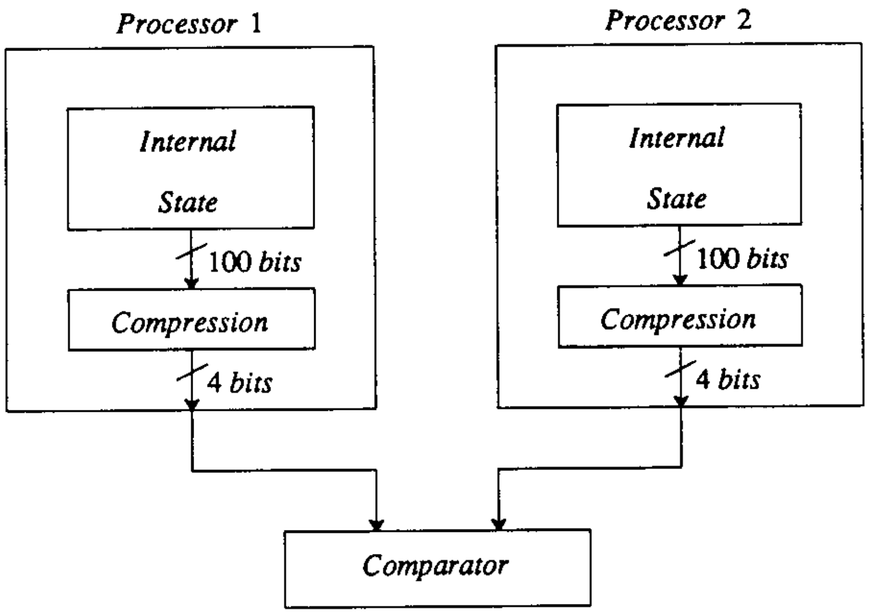
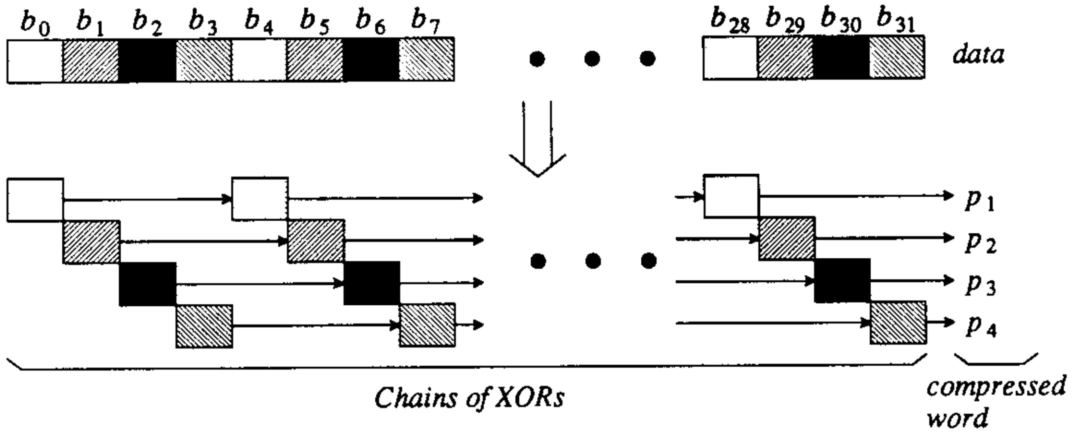


Figure 3.29: Processors Running in Duplex Mode

There are many possible ways to "compress" data [McCl85]. A simple and effective data compression technique is to use several parity bits computed across the data word to be compressed. For example, for a 32-bit word, we can compute a 4-bit "signature" by constructing four interleaved parity chains, each consisting of eight bits from the word. Each chain includes every fourth bit in the word. The implementation of this interleaved parity scheme uses the circuits already described in Section 3.1. A large percentage of errors can be detected by comparing the 4-bit signatures of the two words: any odd number of bit errors and many multiple bit adjacent errors. For random multi-bit errors, 93.75% of the errors will be detected because the signature of the corrupted word has a probability of 1/16 (6.25%) of matching the signature of the correct word (16 different signatures can be obtained from 4 bits).



b_0 b_1 b_2 b_3 b_4 b_5 b_6 b_7
b_{28} b_{29} b_{30} b_{31}
data
p_1
p_2
p_3
p_4
Chains of XORs
compressed word

Figure 3.30: Interlaced parity used for compression

A 4-bit signature of a 32-bit word can be generated in 6ns using four 8-input P-precharged chains (Figure 3.30). Since the chain is small (8 inputs), there is no need to insert a double buffer after the first four cells. In this way, the pitch of each P-precharged cell can be made as wide as the pitch of the data bus (50λ), this results in a stride of 34λ for the switching cell. The four chains can be compressed into one chain where every fourth cell is connected through metal lines. For a pitch of 50 λ we obtained a stride for the complete circuitry of 100 λ, which includes a precharge line, an inverter, a P-precharged cell and the routing between every four cells (Figure 3.31).

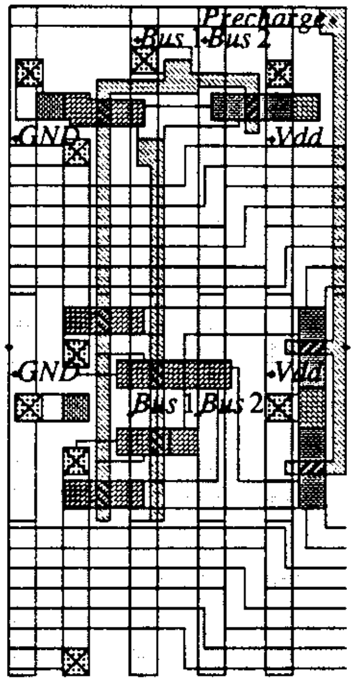


Figure 3.31: A slice of the data compression circuit. From top to bottom: precharge line, inverter, routing of top rail, P-precharged switching cell, routing of bottom rail.

3.3.2. Comparison

A comparator can be implemented using the design shown in Figure 3.32. We have laid out a simple precharged 32-input comparator to match the pitch of the datapath (50λ) and its stride is 49λ. The outcome of the comparison is available 8.3ns after the evaluate signal is asserted.

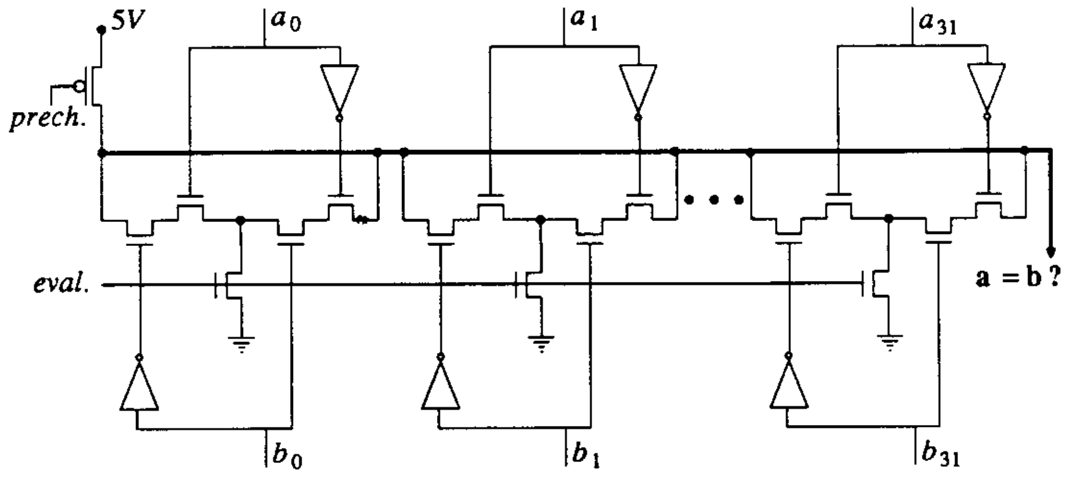


Figure 3.32: A 32-input Comparator

3.4. EDC Circuits and the Micro-Architecture of a Processor

The circuitry added to the processor for error detection and correction takes up valuable chip area and often results in performance degradation. In order to minimize these effects, the parity, ECC, comparator and compression circuits must be as fast and as small as possible. However, as discussed in Section 3.1, these circuits cannot be optimized in isolation from the micro-architecture and implementation of the rest of the chip. For example, chip area is utilized most efficiently if the pitch of the EDC circuitry for the register file matches the pitch of the rest of the datapath, even if this is not the pitch at which the area of the EDC circuitry is minimized. Another example is that it might be necessary to modify the micro-architecture of the processor, adding an extra pipeline stage, in order to reduce the performance penalty of the EDC circuitry.

The use of parity or ECC in the register file of a processor delays both the *read* and the *write* operations. For *writes*, the parity or check bits must be generated and written along with the data. For *reads*, the validity of data words should be established before the data can modify the processor state. If these EDC operations are executed serially, a delay of anywhere from 7ns to 41ns (depending on the choice of implementation) must be added to the time allocated for *reads* and *writes* (Tables 3.1 and 3.3). This would increase the processor cycle time significantly, considering that processors have been designed using the same technology with cycle time of 50ns [Horo87].

For *writes*, it is possible to take advantage of the fact that most pipelined processors have separate pipeline stages for instruction execution and for storing the results in the register file [Kate83, AMD87]. At the end of the cycle during which a result is produced, the result is written into a forward (or bypass) register. This eliminates the need for stretching the cycle time to include the register file write in the same cycle as the computation. Using this property, ECC bits can be generated in *parallel* with the *write* into the forward register. Then in the following cycle, when the EDC bits are ready, they can be written along with the data into the register file.

For *reads* it is desirable to send data directly to the ALU since this operation is often in the critical path. Checking can be done in parallel if the validity of data words can be established before the processor state gets corrupted by ALU operations based on the erroneous data previously sent. If an error is detected, any operation dependent on the *read* must be aborted. The time available to compute EDC bits is thus approximately equal to the time allowed for an ALU computation. Depending on the speed of the ALU, different implementations of the parity/ECC circuits may match this delay. If not, other methods, such as micro rollback can be used to remove this strict limitation on error detection latency (Chapter 4).

Even though EDC operations can be executed in parallel with normal operations, a slight overhead may still occur. As shown in (Figure 3.2), adding EDC circuitry in the datapath results in longer data buses which leads to lower performance even when no error occurs. Our simulation indicates that in the worst case (*stride of ECC* $= 800\lambda$) the extra capacitance adds 1.4ns to a register file *read* which normally takes 22ns. This represents an increase of 6.4%. For some processors [Kane87], a register file read is not in the critical path, so the addition of EDC circuitry may not affect performance during normal operation.

3.5. Summary

Most fault-tolerant systems require that key components, such as VLSI processors, include significant local error detection and correction capabilities. The circuits that provide these capabilities are typically encoders, decoders, comparators, and data compression circuitry. These circuits must provide low latency, high throughput operation in order to be able to perform checks every cycle and prevent erroneous information from propagating throughout the system. Multi-input XOR gates are critical building-blocks for many of these circuits.

We have described several implementations of XOR gates: a tree of static XOR gates, a compact N-chain, two precharged chains and a dual-chain that provide normal noise margins, and a fast implementation based on sense amplifiers. The discussion included major tradeoffs (speed, area, noise margins, pitch matching) in the implementation of circuits for generating parity, for computing ECC check bits, for correcting errors, and for compressing and comparing data.

High coverage error detection can be achieved using duplication and comparison across chip boundaries. In order to reduce error detection latency, values from nodes internal to the chip should be included in the comparison along with the values from external buses. The number of extra pins required for comparison of internal node values can be reduced by ''compressing'' these values into a small signature, which is transmitted across the pins to a comparator. Simple fast circuitry can generate such a signature by computing several parity bits across interleaved subsequences of bits from the internal node values.

Chapter Four
Micro Rollback

The main part of this chapter concerns the implementation of micro rollback for a VLSI RISC processor. Before describing the techniques involved, we give a definition of micro rollback for a single module. The definition is then broadened to simple systems where rollback is accomplished either in real time or virtual time, depending on the modules present in the system. The VLSI implementation of the basic building blocks needed for micro rollback of the processor is discussed. We show how the updated state of the entire processor can be checkpointed after every cycle without replicating all the storage. It is shown that the micro rollback functionality can be added with only a small performance penalty and with a low area penalty relative to the size of the entire chip. We show how the concept of micro rollback can be used throughout a system, discuss the requirements for modules other than the processor, and show how the various modules operate in a multiprocessor system.

4.1. Definition of Micro Rollback

A micro rollback of a module (subsystem) consists of bringing the module back a few cycles to a *state* that it had reached in the past [Tami90]. In order to be able to perform such an operation, it is necessary to save a ''snapshot'' of the state of the subsystem (*checkpoint*) at each cycle boundary. Micro rollback restores the state of a subsystem by overwriting the current state with a ''snapshot'' taken in the past (Fig. 4.1).

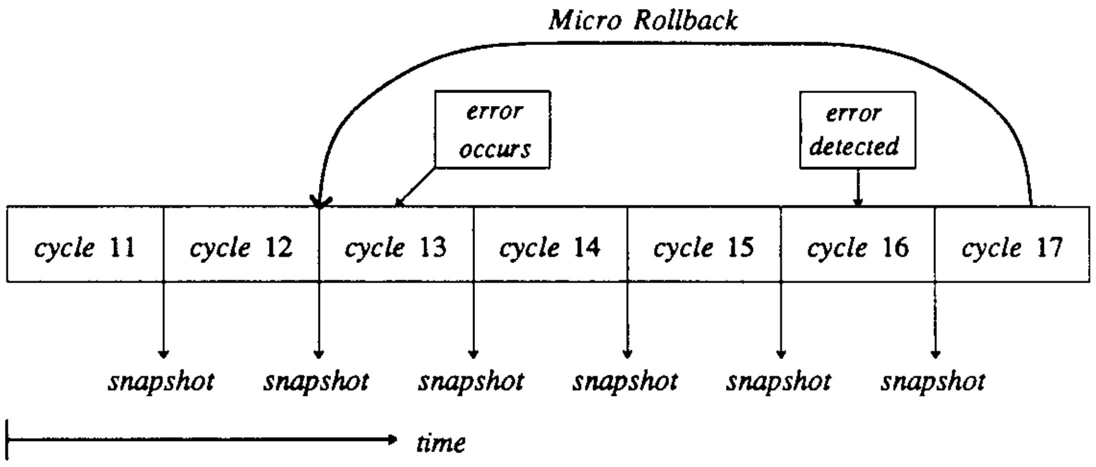


Fig. 4.1. Micro rollback of a module — restoring a saved *snapshot*.

The number of cycles that can be undone — the *rollback distance* — is limited by the number of stored snapshots. One of the design parameters of a system with support for micro rollback is the *rollback range* — the maximum rollback distance that modules in the system must support. The rollback range must be determined based on the latency of the various detection and correction mechanisms in the system as well as the delays in distributing the rollback ''command'' once an error is detected. Specifically, in order to allow detection and correction to be performed in parallel with normal computation, the rollback range must be greater than the worst-case latency of the slowest detection mechanism and its propagation.

The *state* of a module (subsystem) is the contents of all storage elements which carry useful information across cycle boundaries. For example the state of a simple RISC processor is composed of the program counter, the program status word, the instruction register, and the register file. It also includes the contents of some pipeline latches and registers in the state machine which can be changed during the execution of a multicycle instruction. Micro rollback must maintain consistency between the states of all modules in the system [Rand78]. Since instructions also modify external memory (*loads* and *stores*), the state of the memory (cache) must also be checkpointed and rolled back with the processor. We discuss the interaction between the processor and the memory system in a later section and in Chapter 6.

4.1.1. Micro Rollback in Virtual and Real Time

Both the normal progress of a module and rollbacks can be depicted on a virtual time v.s.. real time graph (Figure 4.2). The real time axis (horizontal axis) represents time intervals generated by a clock driving the module. The clock advances unconditionally. The vertical axis, labeled the *virtual time axis*, represents the logical progress accomplished by the module. During normal execution, virtual time advances at the same rate as real time, shown by a diagonal line in Figure 4.2. During a micro rollback, no useful work is accomplished so virtual time remains constant, indicated by an horizontal line on the graph. The length of the horizontal line depends on how much time it takes to execute micro rollback. State restoring is represented by a sudden drop (vertical line) on the graph. After a rollback, processing can resume at normal pace as shown in real time frames 7 and 8.

This representation helps in solving complex situations, such as overlapping rollback signals or simultaneous rollback signals, in systems with multiple rollback sources. Using this representation we will show that a module can roll back according to virtual time or real time, a useful concept for systems composed of

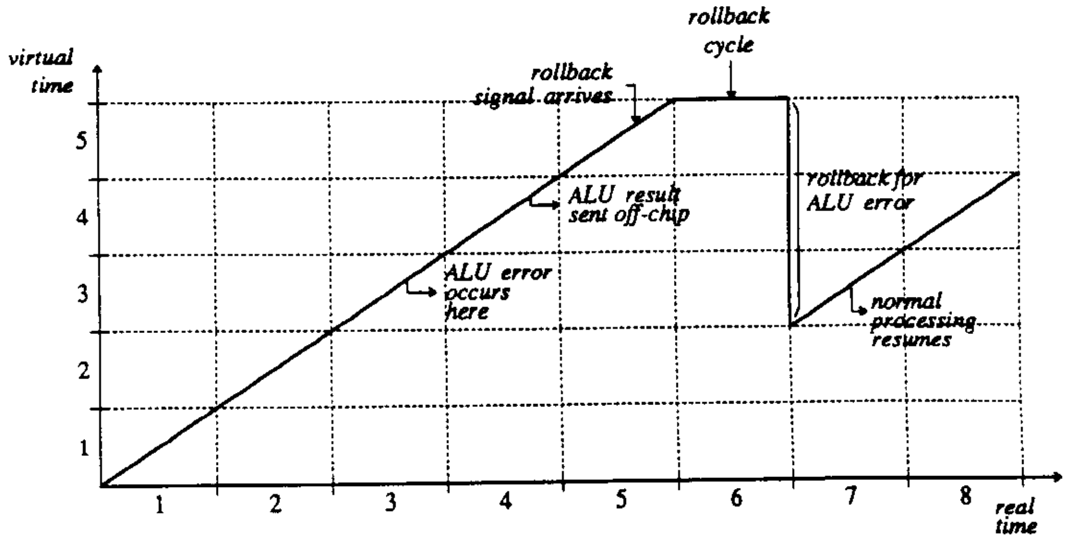


Figure 4.2: Micro rollback in a virtual time vs. real time graph.

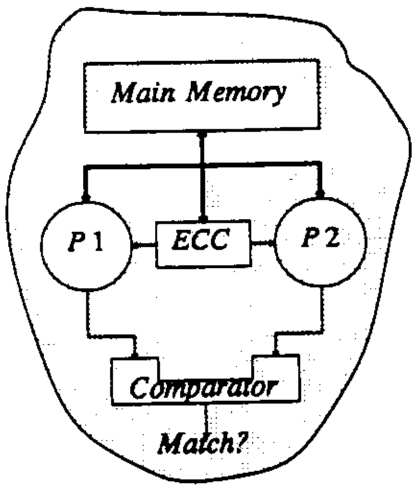


Figure 4.3: Homogeneous system with two checkers. All modules roll back when an error is detected.

modules capable of rollback and other modules which cannot roll back.

In order to define micro rollback, we must first define "normal execution". The normal execution of an ideal module, assuming that no errors or rollbacks occur, can be represented as a sequence of states: $S = (s_0, s_1, \cdots, s_n)$. A real module attempts to perform the same task and the result is a sequence of states: $T = (t_0, t_1, \cdots, t_m)$, where $m \geq n$. Without errors or rollbacks, $m = n$, and $t_i = s_i$ for all i $(0 \leq i \leq n)$.

The state s_α is reached by the ideal module after α cycles. If the real module has either never failed, or failed and was properly recovered, it will reach state s_α at time β, where $\beta \geq \alpha$. Hence, $s_\alpha = t_\beta$. Assume that the first error that occurs after time β, occurs at time $\beta+e$. Hence, $s_{\alpha+e} \neq t_{\beta+e}$. If the error detection latency is d, the error will be reported at time $\beta + e + d$. At this point, a *valid* micro rollback must restore the module to a valid state, i.e., to a state that preceded $t_{\beta+e}$. We assume that rollback is accomplished in one cycle.

- A valid micro rollback of n real cycles, where $n > d$, restores the module to state $t_{\beta+e+d+1-n}$, where $t_{\beta+e+d+1-n} = s_k$ for some k, such that $0 \leq k \leq min(n, \beta+e-1)$.
- A valid micro rollback of n virtual cycles, where $n > d$, restores the module to state $s_{\alpha+e+d+1-n}$.

In relation to Figure 4.2, a *virtual time* rollback of n cycles corresponds to changing the current state to the state n steps lower with respect to the vertical axis (virtual time). A *real time* rollback of n cycles corresponds to changing the current state to the state located n steps to the left with respect to the horizontal axis (real time).

The differences between the two rollback methods (virtual time and real time rollback), are more apparent when we consider an example with two overlapping rollback signals. The system chosen for this example is shown in Figure 4.3. The two processors P1 and P2 operate in duplex mode. Results produced by the ALU of each processor are sent off-chip to a comparator. If a match is found, normal processing continues. If a mismatch is detected, a rollback signal is sent to the other modules in the system, indicating that they must roll back to a point in time prior to the ALU operation that caused the mismatch. In this example, we assume an error detection latency of four cycles for the comparator. We also make the unrealistic assumption that the comparator can be rolled back. The comparator is most likely implemented as a non-pipelined combinational circuit, which cannot roll back.

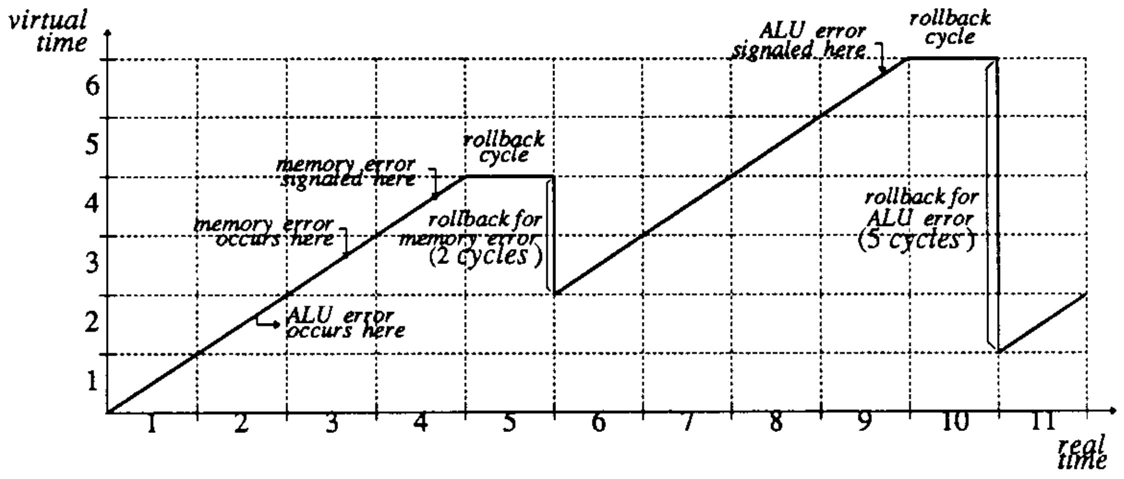


Figure 4.4: Overlapping rollbacks in virtual time

Data words coming from main memory are sent in parallel to the processors and to an ECC (Error Correcting Code) checker. If an error is detected by the ECC unit, a rollback signal is sent to *P1* and *P2*. The error detection latency for a memory error is assumed to be one cycle.

In the example shown in Figure 4.4, an ALU error occurs first, followed by a memory error. After the detection of the memory error, a rollback of **all** modules, including the other checker (comparator), is initiated. Since all modules, including the comparator, roll back a distance of two cycles, the *state* of the system following rollback is identical to the system state at the end of cycle 2 on the real time axis. Hence, as shown on Figure 4.4 (real time frame 9), the ALU error will only be signaled 4 cycles later.

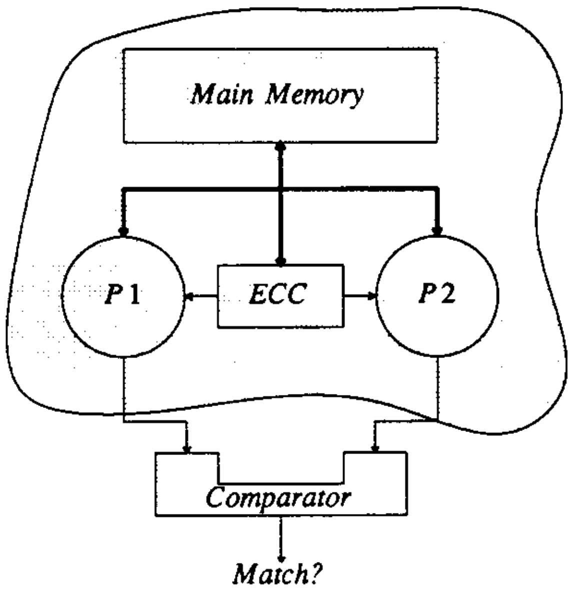


Figure 4.5: Heterogeneous system with two checkers.

The requirements that all modules, including checkers, roll back when an error is detected, complicates the design of the system. Checkers are often combinational circuits and the checking latency may include both the latency of getting the data to the checker and the checking operation itself. Rolling back the checking operation to an arbitrary cycle boundary may be difficult. Fortunately, this is not necessary if rollback is done on the basis of real time.

If checkers cannot roll back, error signals will be generated a fixed number of *real* cycles following the occurrence of the error. This detection latency may be different for different types of errors since it depends on the implementation of the checking mechanism. Hence, the error signal from a checker with latency n is received by the modules exactly n real clock cycles after they receive the erroneous data. Since the purpose of micro rollback is to restore the modules to their state prior to receiving the erroneous information, the optimal operation is a rollback of n real cycles. This should be compared to rollback in virtual time, which would be required if checkers were always rolled back with the rest of the system and their latency would thus be specified in virtual time.

An example of a real time rollback for the system shown in Figure 4.5 is described in Figure 4.6. During the rollback due to the memory error, the comparator is not rolled back. Hence, the ALU error is signaled exactly four real cycles after the ALU error occurs no matter what happens to the rest of the system. When the second rollback signal arrives to the other modules, the five real cycle rollback corresponds to a rollback of only two virtual cycle. This can be seen in Figure 4.6; an horizontal displacement of five real cycles from the end of cycle six brings the system to the state reached after real cycle one.

In order to see how the errors are handled in the two frames of reference, we have drawn the sequence of errors happening in Figure 4.4 on a virtual time axis and the same set of events, represented in Figure 4.6, is drawn on the real time axis. As one can see in Figure 4.7, errors are handled identically but in different time frames.

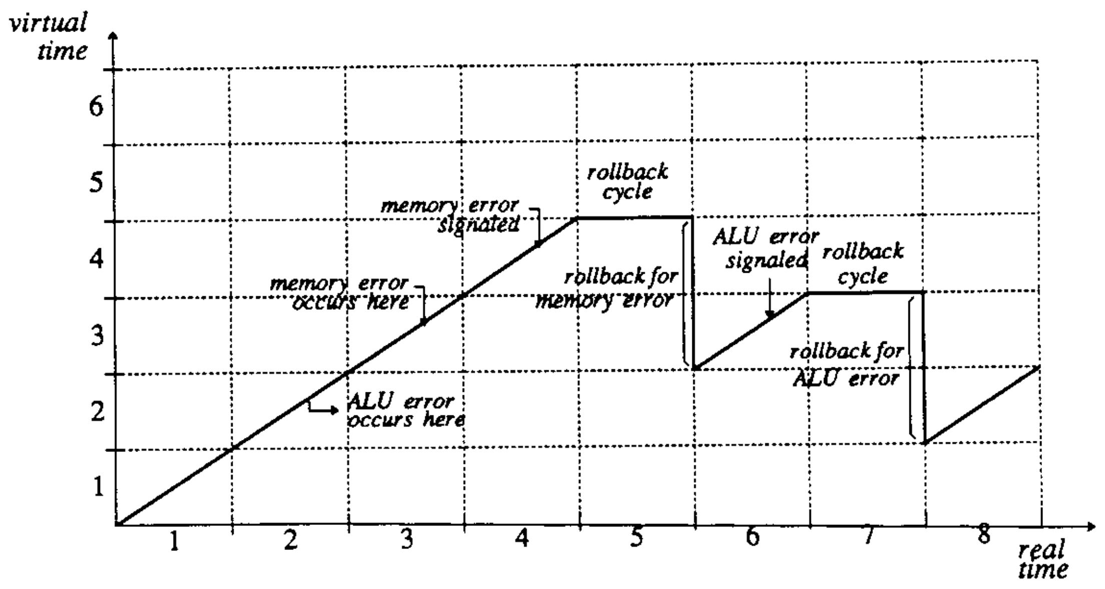
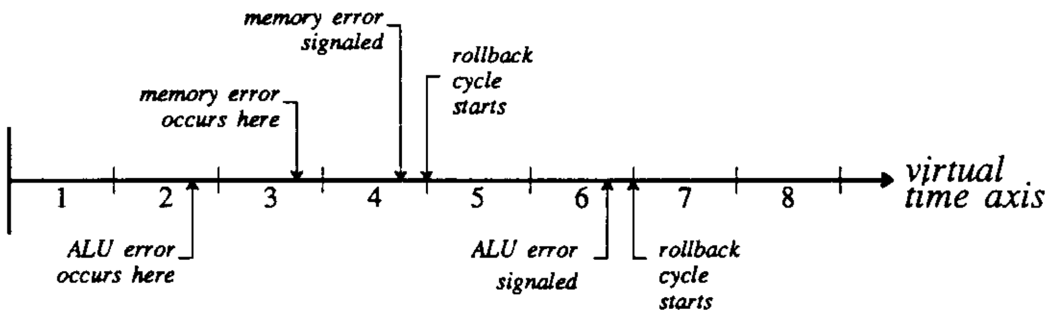


Figure 4.6: Overlapping rollbacks in real time.

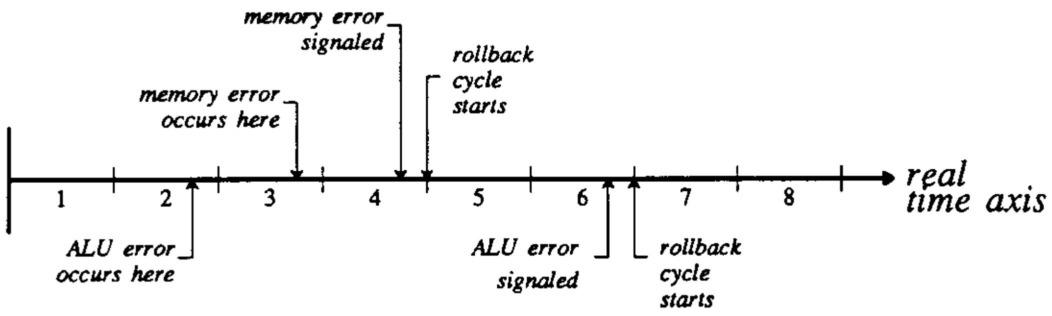
4.2. Support for Micro Rollback in a VLSI RISC Processor

As described in Chapter 1, with micro rollback there is the potential for achieving both high performance and high reliability. However, this potential can only be realized with efficient techniques for supporting micro rollback in the modules of VLSI systems. For example, a possible approach to supporting micro rollback with a rollback range of N cycles in a conventional processor is based on replicating the register file N times. However, this would result in large overhead in both chip area (additional storage cells) and performance (longer buses, larger decoders, etc).

Our research involves the development of techniques at the architecture and microarchitecture level for efficient support for micro rollback. Through full VLSI layouts of the key building blocks, we are able to accurately evaluate the chip area overhead of our techniques. Detailed SPICE simulations of the circuits are used to



(a) Virtual Time Rollback for system in Figure 4.3



(b) Real Time Rollback for system in Figure 4.5

Figure 4.7: Sequence of errors drawn on a real time axis for real time rollback and on a virtual time axis for a virtual time rollback.

determine the performance overhead. Our VLSI implementations are all in CMOS technology, using the MOSIS scalable design rules (SCMOS). The SPICE simulations are based on circuits extracted from the layout, assuming a 2 μ ($\lambda = 1.0$) process (as in Chapter 3).

As a concrete example, we have designed and implementing a VLSI processor capable of micro rollback. We have chosen the Berkeley RISC II processor [Patt82, Kate83] and determine the area overhead and performance penalty for adding to it the ability to perform micro rollback. In later chapters, we will describe methods for adding micro rollback to more complex processors. The process of saving the state of a RISC processor and the method used to roll back in one cycle are described below. The state to be saved and restored is located in the register file and the individual state registers.

4.2.1. Micro Rollback of the Register File

At every cycle, a *write* into the register file may be performed. As discussed above, the state of the register file can be preserved for N cycles by replicating it N times (for example, using shift registers) [Hwu87]. This technique results in high area and performance overhead due to replication of the logic, longer buses, and increased attached circuitry. As we will show in Section 4.2.4, this technique is more appropriate for single state registers, for which a more efficient technique is not available. Instead of replicating storage, we use the approach of delaying commitment of state modifications [Smit88, Hwu87]. The proposed method uses a *delayed write buffer* — DWB, which minimizes the extra hardware needed, and still allows a rollback of up to N cycles to be executed in a single cycle [Tami88b, Tami90].

4.2.1.1. High-level Description

Whenever the processor writes data to one of its registers, the address of the destination register as well as the data to be written, is stored in an N-word first-in-first-out queue (FIFO), which we call a *delayed write buffer* — DWB (Figure 4.8). The DWB delays each *write* by N cycles before it is finally written into the register file. During every cycle, a new entry is made in the right-most cell of the DWB. If a write occurs, the data is entered and the DWB position is marked *valid*. If no write occurs, the DWB word is reserved but marked as *invalid*. During every

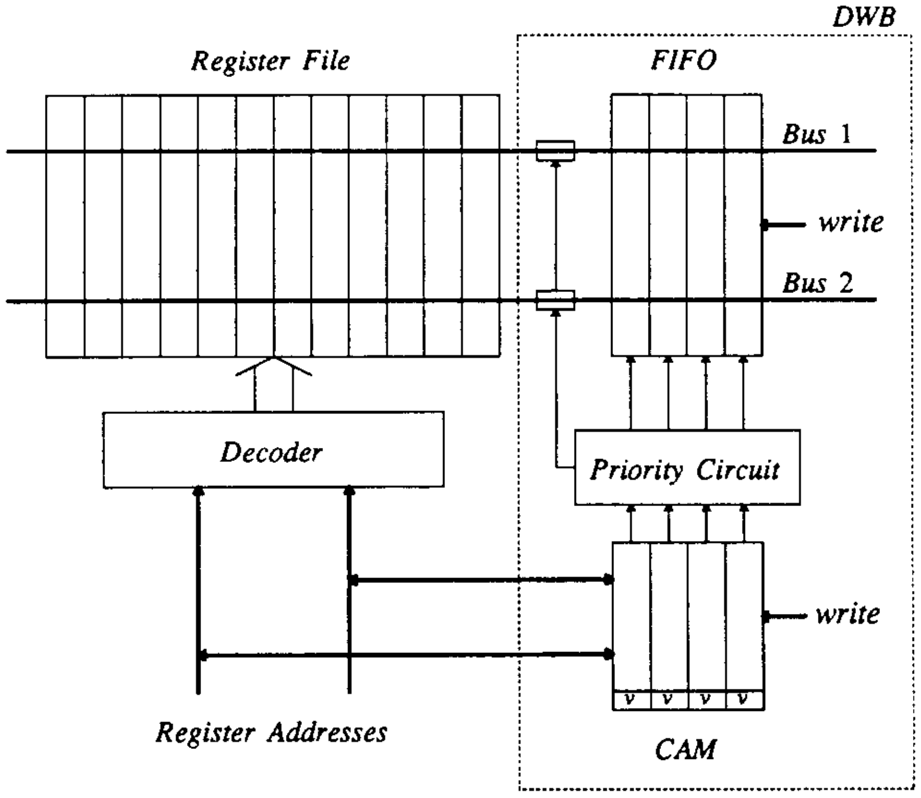


Fig. 4.8. A register file with support for micro rollback.

cycle, the oldest (left-most) entry in the DWB is written to its corresponding address in the register file if its valid bit is set, and discarded otherwise.

In order to roll back p cycles ($1 \leq p \leq N$), the last p entries in the DWB (the right-most p entries in Figure 4.8) are invalidated by clearing the valid bits (no data transfers are needed).

The register address corresponding to the data shifted in during a *write* is held in a content-addressable memory (CAM). During the register read phase of every instruction, the register addresses of the two operands are compared with the addresses of the registers stored in the DWB. If there is a match and the valid bit in the CAM is set, the data of the matching register is gated on the corresponding internal data bus. If there is more than one match for a particular operand, a priority circuit is used to provide the most recent version available in the DWB. This corresponds to the rightmost valid register in the FIFO in Figure 4.8.

An important feature of this design is that during both register *read* and register *write* operations the register file and the DWB operate in parallel without significant conflicts for the use of the internal buses (which would lead to additional delays).

4.2.1.2. Implementation and Interfacing of the Register File and DWB

As in RISC II, the datapath includes a large register file consisting of 128 32-bit registers, organized in eight overlapping register banks [Kate83]. The RAM cell used in this register file is a two-port cell which allows two simultaneous *reads* and one *write* during a processor cycle. Both buses are precharged prior to a *read*.

The top section of the DWB contains the data to be written into the register file, while the bottom part contains the register addresses of the corresponding data. The data part is both a FIFO queue and a RAM. The FIFO queue functionality is implemented using shift registers which can also be accessed as a RAM. In this way, an entry can be selected by the bottom part of the DWB and its value driven onto the bus. The bottom part is a FIFO which is also a CAM. Each FIFO/CAM cell consists of a one-bit static shift-register cell as well as circuitry for associative lookup. Since an instruction may require two operands, two lookups in the CAM can be performed simultaneously (see Figure 4.9). As a result of the associative lookup, one or more ''match lines'' are asserted and the most recent one is determined by a priority circuit (Figure 4.10) and used to address the top section of

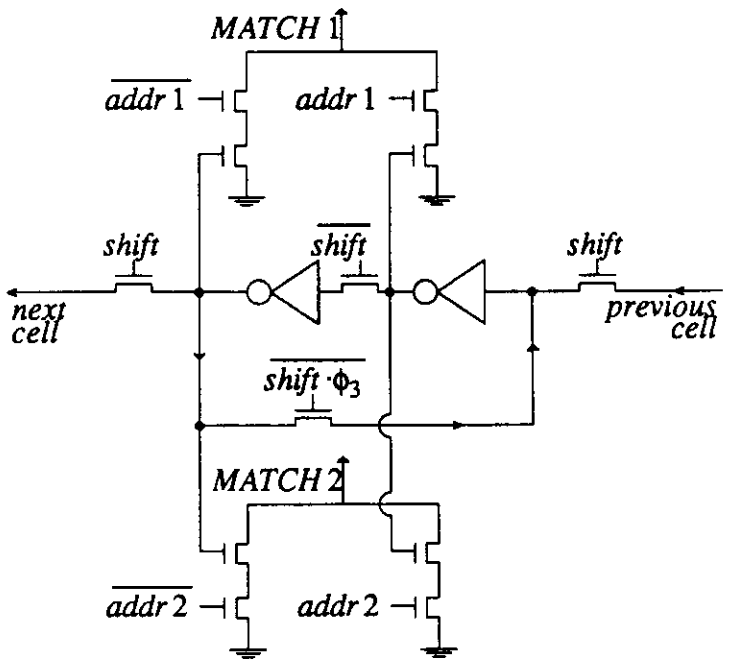


Figure 4.9: A FIFO/CAM cell for the delayed-write buffer.

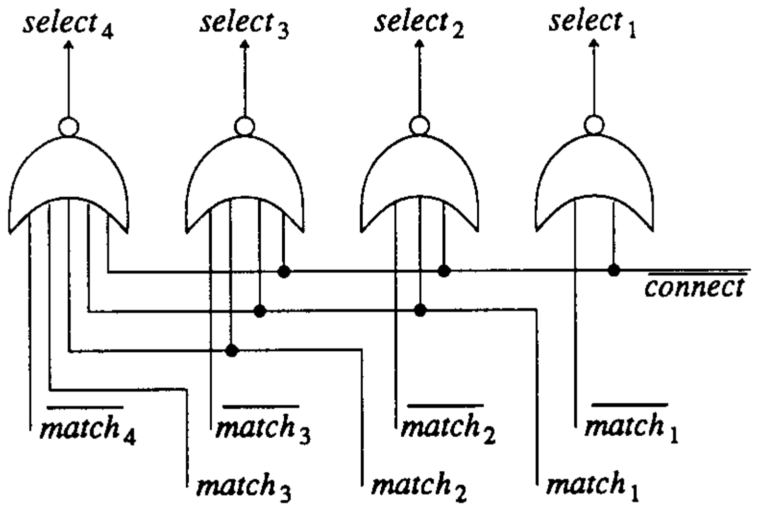


Figure 4.10: Priority Circuit for the DWB.

the DWB. Figure 4.11 shows the layout of the CAM cell.

The register file data bus is connected, through switching logic (Figure 4.12)

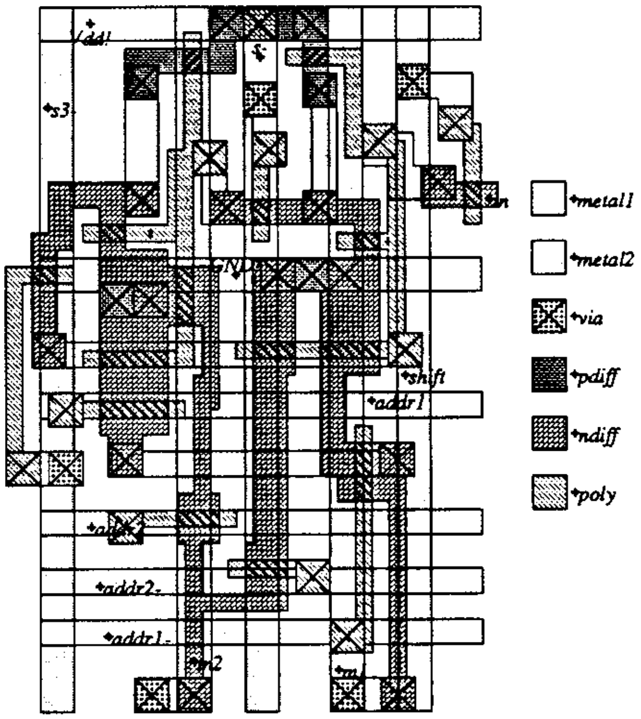


Figure 4.11: Layout of the FIFO/CAM cell.

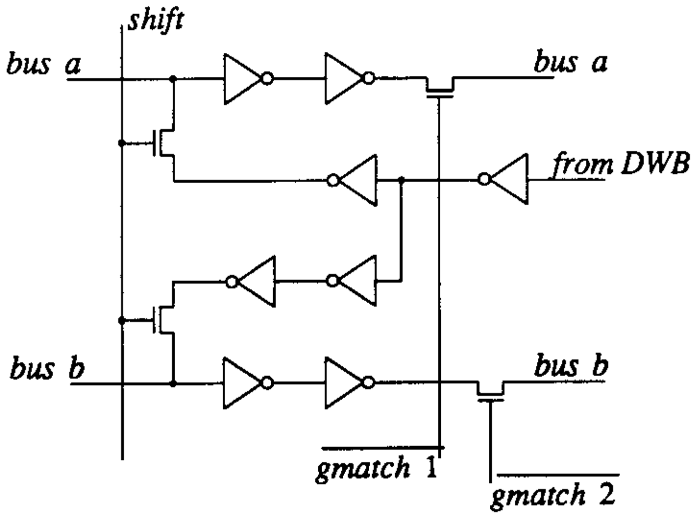


Figure 4.12: Switching logic between the register file and the DWB.

to the DWB. Precharging the bus and selecting the proper register is done in parallel for the register file and the DWB. The outcome of the lookup in the CAM determines whether the buses are connected or disconnected. If there is no match, the register file provides the data, while the DWB takes over if there is a match.

4.2.1.3. Alternative Implementation of the FIFO/CAM

The FIFO/CAM cell shown in Figure 4.9 is a critical factor in determining the area and performance overhead of the DWB. In order to determine the area and delay of different transistor implementations of a FIFO/CAM cell, we have laid out and simulated a CAM composed of four entries of 7-bit tags for three different transistor circuits (Figure 4.13). Only the transistors required for the comparison are shown in the figure. The shift-memory cell is similar to the one in Figure 4.9. The FIFO/CAM cells were laid out to match the stride (92λ) of the FIFO/DATA cells used in the MIRROR processor [Tami91]. For this configuration, cell (a) is 73% larger than cell (c) even though it has only one more transistor. This is due to the extra metal interconnects needed for cell (a). For other strides, results may be different but cell (c) should still be smaller than cell (a) or (b). Cell (b) and cell (c) are faster than cell (a) mainly because the discharge of the match line is done through a single N-transistor, compare to two N-transistors for cell (a). The gate of the discharge transistor in cell (c) (the transistor connected to the match line) is driven slightly faster than the discharge transistor in cell (b). As a result, cell (c) is slightly faster then cell (b). Cell (c) is thus is a good choice for a CAM with a size comparable to ours (4×7).

If the height of the CAM cell is critical and if the width of the cell (in the

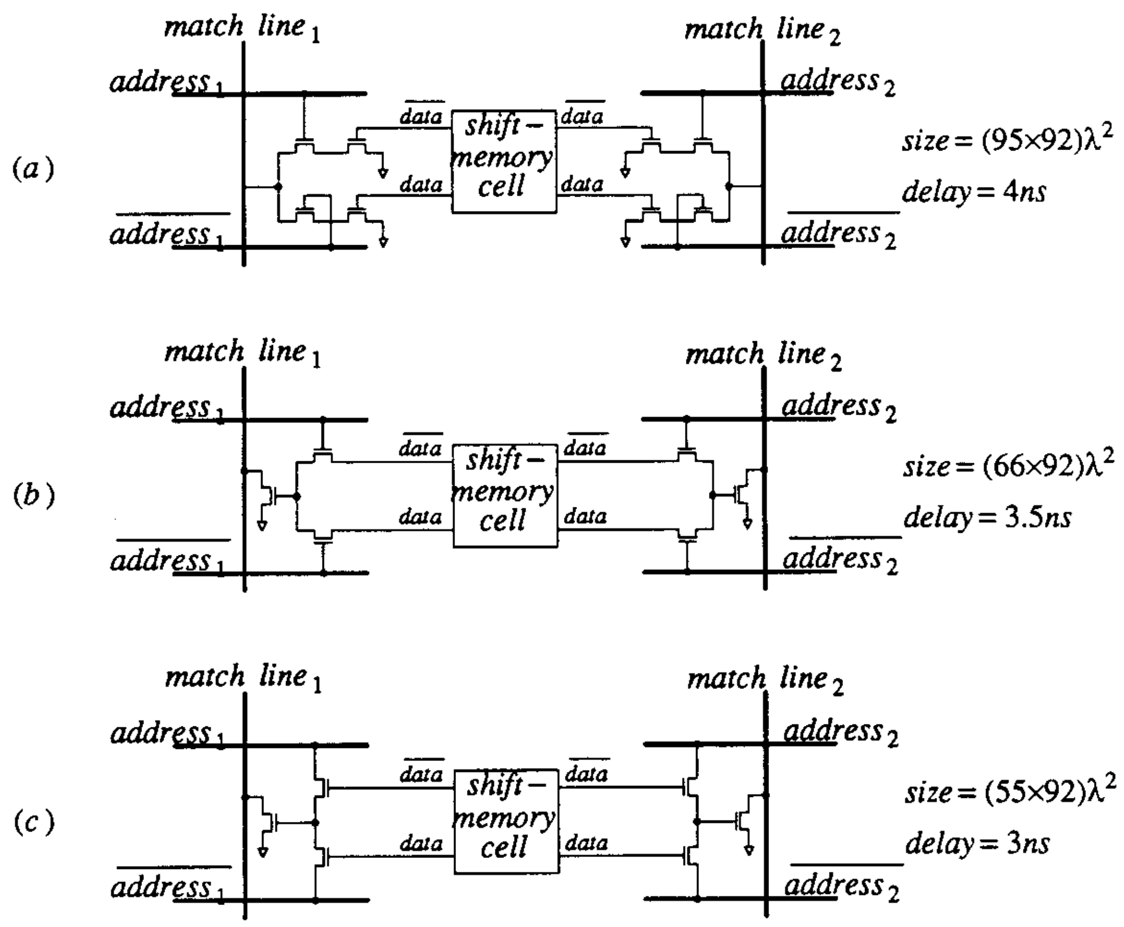


Figure 4.13: Comparison of three different transistor circuits for CAM cells. The cells are laid out to match a stride of 92λ. The delays are for a CAM of four 7-bit tags.

horizontal dimension) is not constrained, the cell described in Figure 4.14 can be used [Chow89]. In this cell, only two address lines go through the cell (compared to four in the previous implementations). This reduces the number of second-level metal lines going horizontally through the cell to four (Vdd, GND, $\overline{address}_1$, $\overline{address}_2$), which decreases the minimum height of the cell to 26λ (32λ for the lines minus 6λ for sharing GND or Vdd). This design requires two vertical lines to be connected to each port of each cell forming the tag. One line is conditionally discharged through P-transistors while the other one is conditionally discharged through N-transistors. The lines are combined through an inverter and a NOR gate to form a *match line*. The addition of two vertical lines and the replacement of N-transistors by P-transistors (requiring separate wells for the matching circuitry), leads to a cell with a large width. In our context, where the cells must fit in 92λ, the design of this cell is impractical.

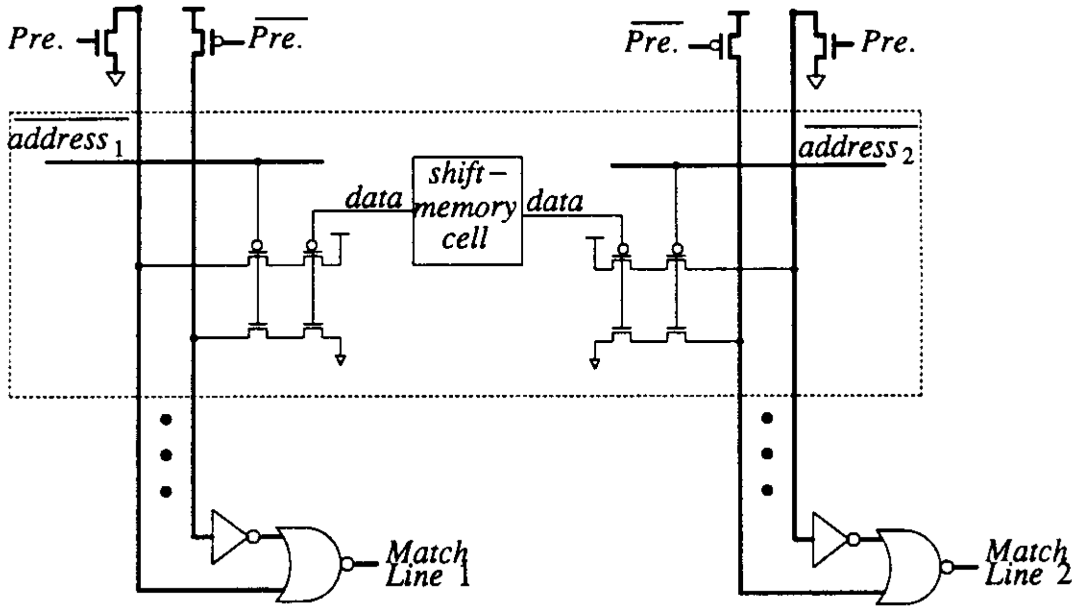


Figure 4.14: CAM cell optimized for reducing the vertical dimension. Only two address lines are routed horizontally.

4.2.1.4. Pipeline Organization

In RISC II, instructions are executed in a three-stage pipeline: instruction fetch, execute (includes reading from the register file), and write (store the result in the register file) [Kate83]. The separate stage for writing to the register file is needed because writing a value to a large register file is a time consuming task and the separate stage for it facilitates most efficient use of processor resources. In order to prevent pipeline interlocks, RISC II uses *internal forwarding*: the result of the operation is stored in a temporary forward register at the end of the second pipeline stage and, if necessary, it is immediately (in the next cycle) forwarded to the next instruction. In the third stage of the pipeline the value from the forward register is written to the register file.

With our register file organization, writing the result can be done quickly since it is never written directly into the large register file. Instead, the result is written to the first (right-most in Figure 4.8) entry in the DWB. The entire DWB serves a function similar to that of the forward register in RISC II. If we consider an instruction to be complete only when its result is written into the real register file, and given a DWB with n stages, a processor using our register file organization has an $n + 2$ stage pipeline: instruction fetch, execute, $n - 1$ stages of advancing through the DWB, and writing the result into the real register file.

4.2.1.5. Performance and Area Overheads for Micro Rollback Support in the Register File

The *read* and *write* delays when accessing a large register file are often critical factors in determining the overall processor timing. In a RISC processor, where the pipelining and control are very simple, the register file is likely to be the largest single module [Kate83]. In this subsection we present the details of the timing of the RISC processor with micro rollback support, focusing on the timing of the register file. The performance penalty and area overhead for micro rollback support in the register file are discussed.

4.2.1.6. Timing

Processor timing is based on a four-phase clock. The internal buses are precharged during ϕ_1 (phase 1), the registers are read during ϕ_2, the ALU operates and writes the results into the DWB during ϕ_3 and ϕ_4. During ϕ_3 and ϕ_4 the DWB is also shifted and the oldest entry is written into the register file [Tami88a].

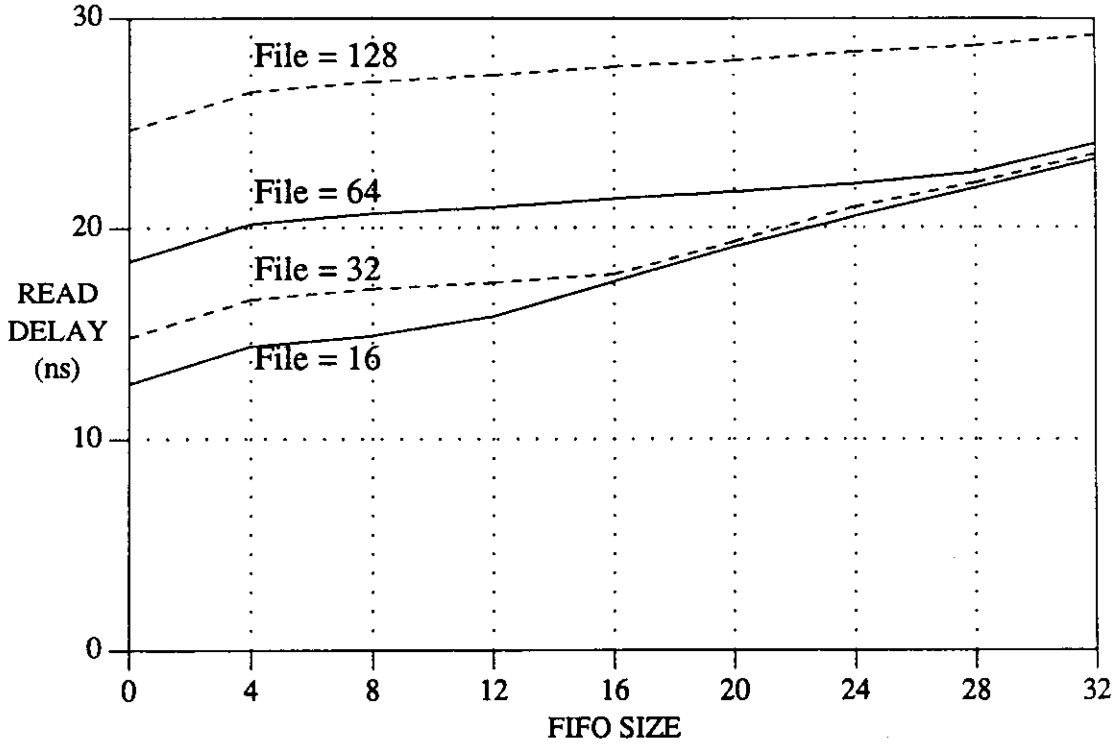


Fig. 4.15. The performance overhead of the DWB — read delay for various sizes of the register file and the DWB.

To determine the performance penalty for micro rollback support in the register file, we have produced a complete VLSI layout of the register file and have simulated the operation of these circuits, using SPICE, for several register file and DWB sizes. The results of our simulations are shown in Figure 4.15. The overhead introduced for micro rollback support can be determined by comparing the read delay with a DWB of the size of interest to the read delay with a DWB of size 0 (no micro rollback support). For example, a file of 128 registers with a DWB of 8 registers has a *read* delay of 27 ns (vs. 24.7 ns for a standard register file).

The *read* delay can originate from two different sources:

(1) when there is no match in the DWB: the address is sent to the register file, the chosen register discharges the file bus lines, a superbuffer connected to the register file bus discharges the DWB bus lines.

(2) when there is a match in the DWB: the address is looked up in the CAM, a match is detected, the priority circuit determines which register to select, and the chosen FIFO register discharges the FIFO bus lines.

For a large register file, the critical path will most likely be path (1) (for a small DWB). For a small register file with a large DWB, the main data bus is discharged before the buses are connected. In this case the critical path is path (2).

4.2.1.7. Area Overhead

Using the same combination of register file and DWB sizes, the area overhead required for micro rollback support in the register file is shown in Figure 4.16. Note that this overhead includes all the circuitry involved with the DWB: the storage cells in the DWB, the CAM cells, the priority circuit, the control, etc. The layout of a 64 entry register file with a DWB of depth four is shown in Figure 4.17.

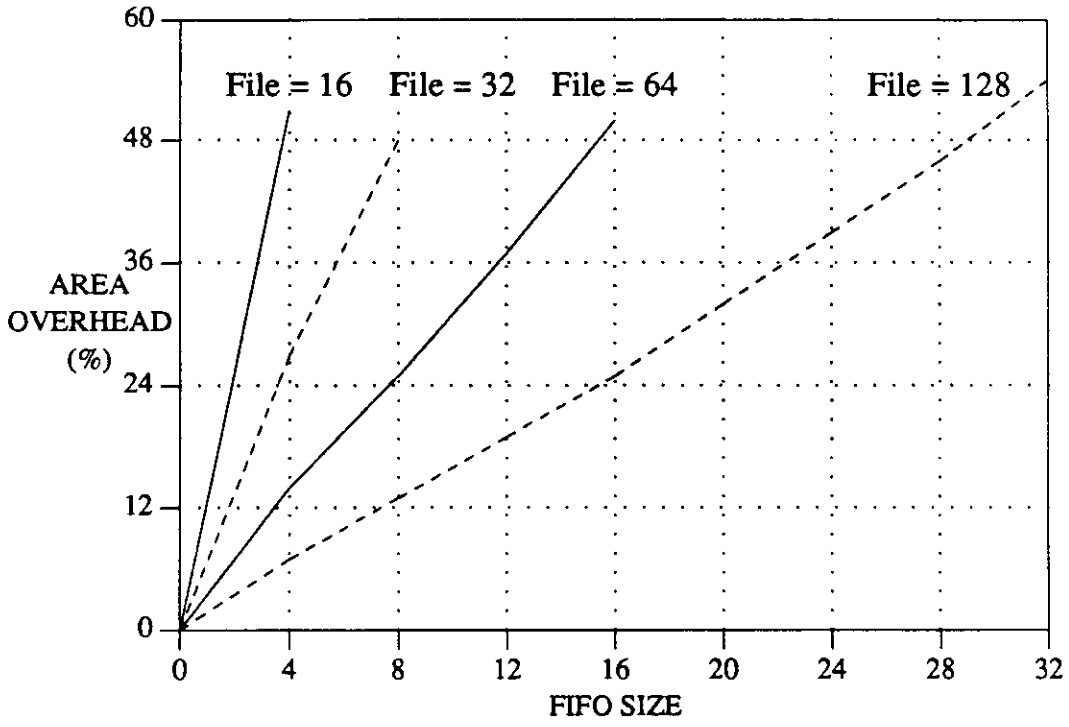


Fig. 4.16. The relative area overhead of the DWB.

4.2.2. Comparison of the DWB with other Methods

Precise interrupts (exceptions) in processors with multiple functional units require undoing changes made by instructions which were issued after the interrupted instruction but caused state changes, out of order, before the exception was detected [Smit88]. As mentioned in Section 2.4, the techniques used to implement micro rollback are similar to techniques used to support precise interrupts. In this subsection, we describe some of the mechanisms used to implement precise interrupts [Smit88, Hwu87] and compare them to the proposed technique for micro rollback.

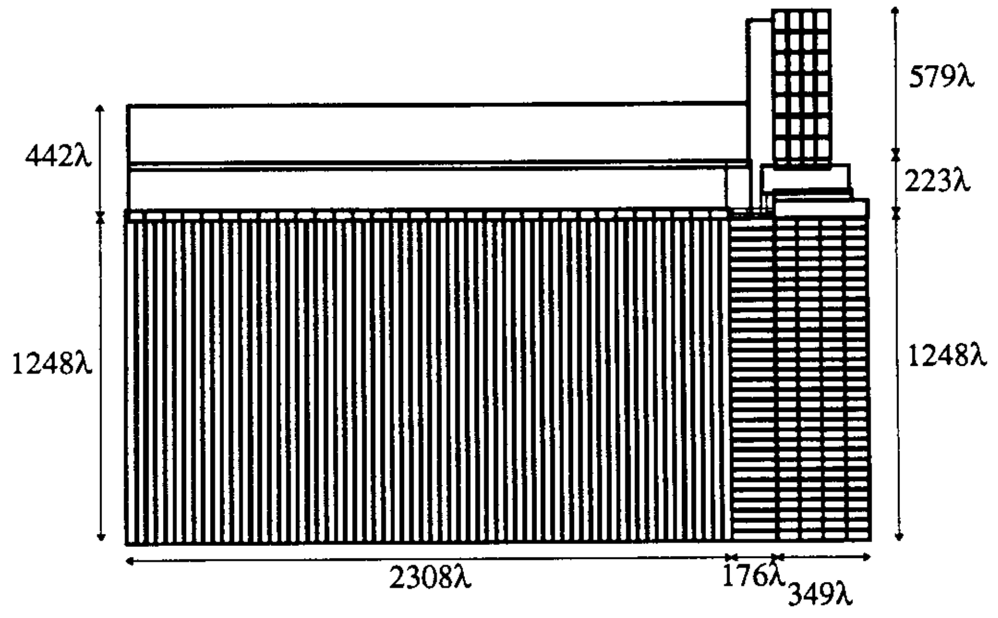


Figure 4.17: Layout of a register file containing 64 registers and a DWB of four registers.

Reorder Buffer

Figure 4.18 shows the organization of a reorder buffer (RB) along with a result shift register (RSR) [Smit88]. The result shift register is used to (a) reserve the result bus (b) route the results of the functional units to the proper entry in the reorder buffer. It is managed as a FIFO queue but information can be inserted out-of-order. When an instruction is issued, the functional unit it specifies along with a tag pointing to the next available entry in the "reorder buffer" (indicated by the tail pointer) are stored in the RSR. Data are stored in the RSR at position p (p entries away from the head of the RSR) where p is the latency of the functional unit specified.

The destination register of the instruction being issued along with the current PC are stored in the RB at the location pointed to by the tail pointer. When an

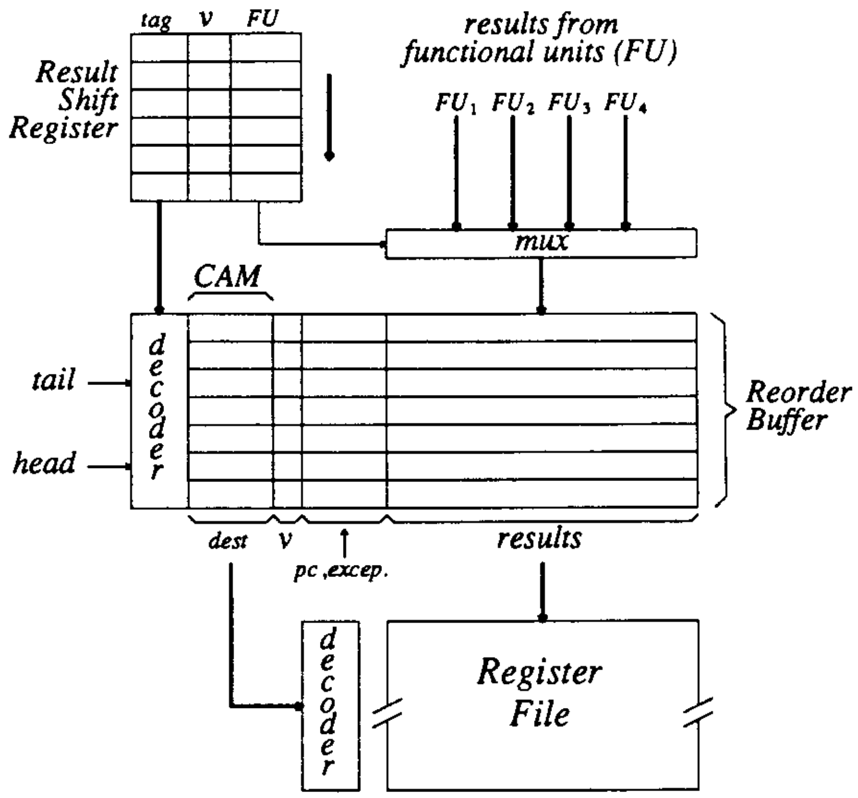


Figure 4.18: Reorder buffer with result shift register.

entry reaches the end of the RSR the computation result from the functional unit along with the exception bits are routed to the reorder buffer at the location pointed to by the tag. The reorder buffer gathers results produced out-of-order and sends them to the register file in order. Bypass circuits are provided in the reorder buffer so that *reads* to the register file always return the most update value. Exceptions are handled not when they are produced by the functional units but when they reach the head of the RB. If an exception is detected, instructions following the instruction specified by the head pointer in the RB are invalidated. Precise interrupts can thus be achieved (the PC is also available from the head of the RB). In [Smit88] it is stated that the circuitry required for implementing the reorder buffer is conceptually simple but that there is a great deal of it.

Micro rollback is inherently simpler than mechanisms that provide precise interrupts using a reorder buffer — there is no need to keep track of instruction boundaries since the rollback event is transparent to the software. One demonstration of this simplicity is that the DWB is simpler than the RB. Since the RB must put in order results that are produced out-of-order, it cannot be implemented as a simple FIFO queue. Instead, there is a need for a circular buffer, implemented as a RAM (random access memory) with tail and head pointers. Two extra buses and associated decoders are required for the RB. An extra write bus across all RB entries is used for writing into any entry. An extra read bus is used for transferring values to the permanent register file. With the DWB, all writes are done to one end of the queue and transfers to the register file are done from the other end. Furthermore, extra hardware is needed with the RB for manipulating the head and tail pointers.

History Buffer

In order to avoid the bypass circuitry of the reorder buffer, a *history* buffer (Figure 4.19) is proposed in [Smit88]. A similar mechanism, called the *backward difference*, is proposed in [Hwu87] for out-of-order issuing units. When an instruction is issued, its destination register value and register number are written into the history buffer. *Writes* to the register file are sent directly to the register file without having to go through a reorder buffer. Following an exception, the information provided by the the history buffer is sufficient to restore the contents of registers that may have been modified out-of-order. The history buffer has two

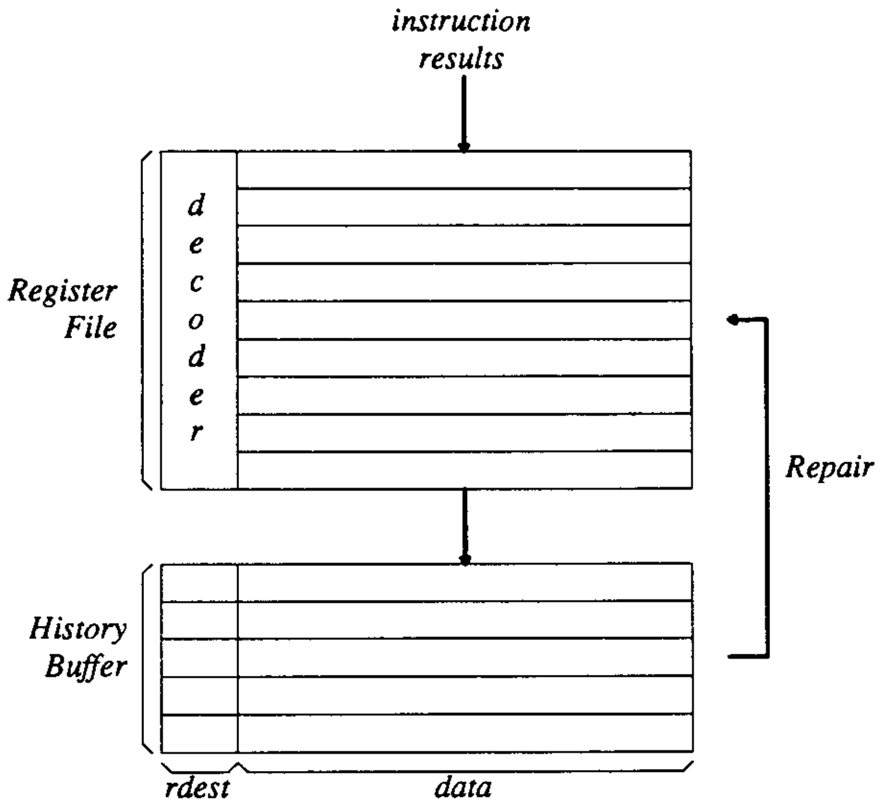


Figure 4.19: History Buffer.

significant disadvantages. First, an extra port in the register file is required so that superseded values can be placed into the history buffer. This increases the size of the register file and would most likely increase access time. Secondly, a repair (rollback in our case) requires several cycles to transfer data from the history buffer to the register file. For real time systems operating in an hostile environment where the error rate is high, this may not be acceptable.

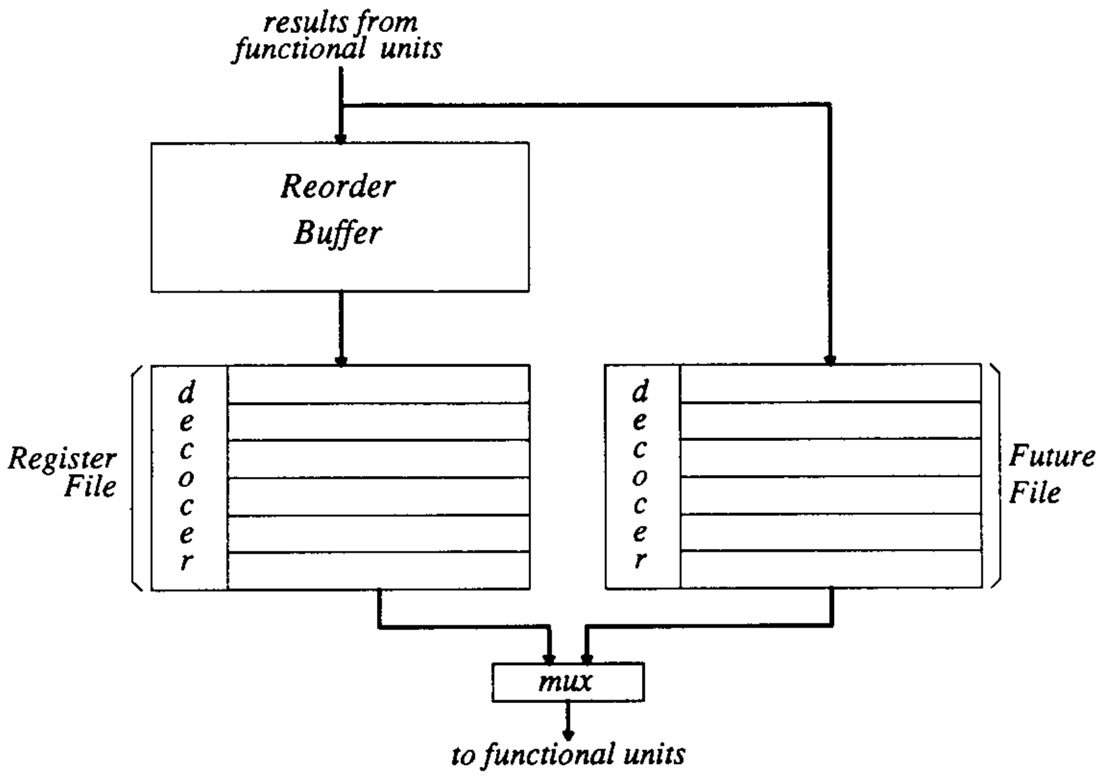


Figure 4.20: Future File.

Future File

In the *future file* scheme (Figure 4.20) proposed by Smith and Pleszkun [Smit88], two register files are required: one is accessed directly by the processor during *reads* and *writes*, the other is connected to a reorder buffer without reading logic. In this way, one register file provides good performance while the other one, which is transparent to normal execution, provides the correct state for rollback to a precise instruction boundary.

The authors mention that the future file presents some advantages for machines that implement interrupts via an "exchange" [Smit88]. This advantage is obtained at the expense of the large overhead introduced by adding an extra register file. In a VLSI environment, the requirement of writing to two register files simultaneously may lead to additional delay due to the way that the datapath is organized (normally a sequence composed of the register file, shifter, ALU, etc.). The problem of multi-cycle recovery is also present with the future file.

Copy technique

This technique is useful for maintaining fast access time to a group of registers (such as the register file), without increasing the bandwidth to those registers. For each register in the processor, there is a set of c backup registers which preserve the state of c checkpoints in the *active window* [Hwu87]. The backup registers are organized as a stack so that the *current register* is pushed onto the stack during a checkpoint, and popped when a repair is needed. The main disadvantage of this method is the huge area overhead required (in the order of $c+1$ times greater than a single register). According to the authors, the technique should still be attractive for an on-chip register file, a claim that may be correct only for very small register files. Current high-performance processors have a large register file, usually between 32 [Kane87, Lee89a] and 128 [Brow90] registers, which makes this technique impractical to implement. The article claims that performance is not affected, but no actual implementation of the copy technique (which requires longer buses) is presented.

Forward Difference

This method is a variation of the Reorder Buffer Method described in [Smit88]. The forward difference mechanism also has some similarities with the DWB method. For example, the forward difference can be used to delay writing of changes made by instructions executed speculatively (e.g. by predicting a conditional branch) [Hwu87]. Fast repair is possible with the forward difference by using methods similar to what is used for micro rollback. Hwu and Patt [Hwu87] do not provide implementation details, so a detailed comparison cannot be presented.

4.2.3. Detection and Correction of Errors in the Register File

A large register file, build out of memory cells, is likely to contain most of the state for simple VLSI processors. Hence, the register file is the module most likely to fail (especially due to transient faults). Since a transient fault may change the value of any of the bits in the register file, rolling the system back a few cycles will not automatically correct the erroneous data. Hence, if an error occurs in one of the registers in the file, it is necessary to have redundancy in order to recover. One approach is based on the use of a duplex processor from which valid copies of damaged information may be recovered. An alternative approach is to use error correcting codes in the register file. In both cases micro rollback prevents checking or correction delays from slowing down normal operation.

4.2.3.1. Duplex Processors

Two processors execute the same instructions in lock-step. A separable error-detecting code (e.g. parity) is used in the register file of each processor so that it will be possible to determine which processor has the valid data. If a processor detects an error in a value read from the register file, it initiates a micro rollback in both processors to the cycle prior to the register file access. The correct data is then copied from the correct processor to the one that detected the error and normal operation is resumed [Tami91].

4.2.3.2. Use of Error Correcting Codes

If some separable error correcting code (e.g. Hamming) is used in the register file, it is possible to recover from errors in the register file without the use of a duplex processor. The code bits are generated when a word is transferred to the DWB. During *reads* from the register file, the correction circuitry detects and corrects them appropriately. Since it is undesirable to lengthen the critical timing path, words read out of the register file are captured by latches and checked in parallel with their processing by the ALU. If an error is detected, processing is stopped, the word is then corrected and stored in the register file. A micro rollback is then initiated in order to restore the state of the processor to the cycle prior to the one when the register file read was performed.

4.2.4. Micro Rollback of Individual State Registers

In various locations on the chip there are individual registers that contain part of the processor state. These include the program counter (PC), instruction register (IR), program status word (PSW), pipeline latches, etc. Since these registers may be modified every cycle, support for micro rollback of up to N cycles means that for each register there must be N "backup registers" provided to save the state for the previous N cycles. We present two methods for rolling back individual state registers; the first one is based on a FIFO memory, while the second one is based on a small RAM. Both methods achieve micro rollback of a variable number of cycles in a single cycle.

With the FIFO method, a state register, labeled *current* in Figure 4.21, is backed up by a set of N registers which are organized as a FIFO. The FIFO registers are transparent to the processor in the sense that all *reads* and *writes* are performed using the current register. At the end of each cycle, the content of the current state register is copied into the FIFO which pushes down all the other registers. In order to roll back C cycles, the first C registers in the FIFO are invalidated, then the first register marked "valid" in the FIFO, identified through a priority circuit, is copied into the current register. Normal processing can resume with the next cycle.

The RAM method uses a pointer to "next available backup register" in a small RAM to create a circular buffer (Figure 4.22). The content of the current register is saved every cycle into the RAM and the pointer is then incremented. In order to roll back C cycles, C is subtracted from the current value of the pointer (modulus the number of elements in the RAM) and the result is used as an address

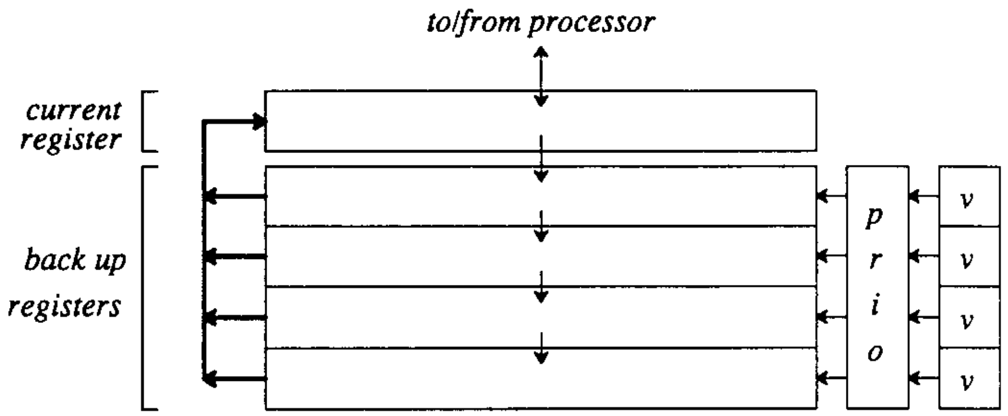


Fig. 4.21. The FIFO method for micro rollback of individual state registers.

of a register in the RAM which is copied to the *current register*. The input to the address decoder of the RAM is the same throughout the chip, allowing a single centralized implementation of the counter and subtraction logic.

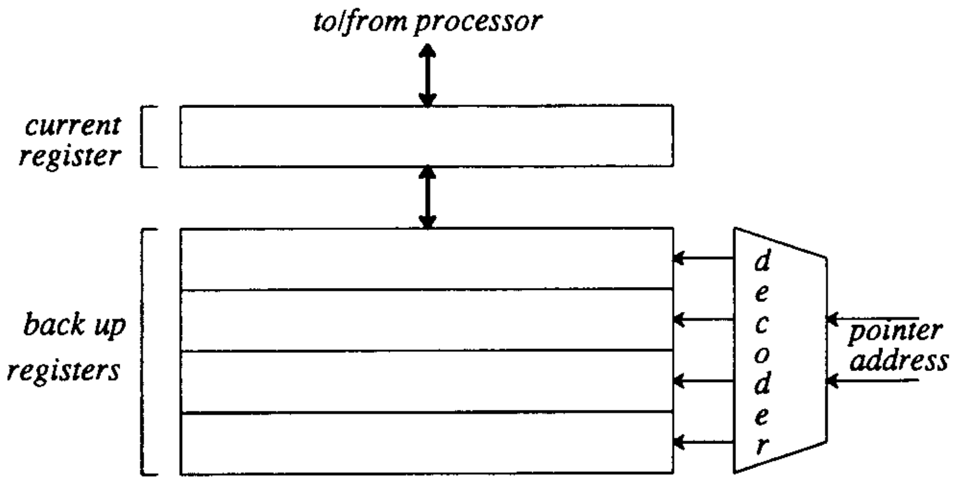


Fig. 4.22. The RAM method for micro rollback of individual state registers.

Table 4.1 shows the sizes of the components and the *stride* (dimension of the circuit parallel to the bus) of the data arrays for the two methods. The difference in

Method	data cell	data array	control area	total area

FIFO	Height = 39λ
	Width = 54λ
		Stride = 382λ
		Area = 477,500λ^2
			43,166λ^2	520,666λ^2

RAM	Height = 39λ
	Width = 34λ
		Stride = 271λ
		Area = 316,250λ^2
			22,517λ^2	338,767λ^2

Table 4.1: A comparison of the area overhead of the two methods for micro rollback of an individual state register by up to six cycles.

size is due to the greater complexity of the data cell for the FIFO (shift and read). There is no extra delay added to the processor since the extra logic is not connected serially to the path followed by the buses; only the "current register" interacts with the rest of the system. On the other hand, the area for each state register must be increased. For example, in order to implement the capability of rolling back four cycles using the RAM method, the area must be increased by a factor of 5.8.

4.2.5. Micro Rollback for the PC Unit

In the Berkeley RISC II processor there are three registers used for storing the next, current, and last values of the program counter [Kate83]. In the original design these registers are organized as a small FIFO and the following transfers occur during each cycle:

```
new_value → next_pc → pc → last_pc → discarded
```

Micro rollback of the PC unit could be supported by treating the three registers as individual state registers. However, as discussed in the previous subsection, this can result in significant area overhead. A more efficient implementation takes advantage of the FIFO organization of the PC unit and uses the FIFO method described in the previous subsection [Lian90]. The basic organization of the new

PC unit is shown in Figure 4.23.

Upon a rollback of C cycles, the C leftmost registers are invalidated. A special-purpose circuit, called ''PC Mapper'' is then used to select a register as follow:

next_pc:	first valid bit
pc:	second valid bit
next_pc:	third valid bit

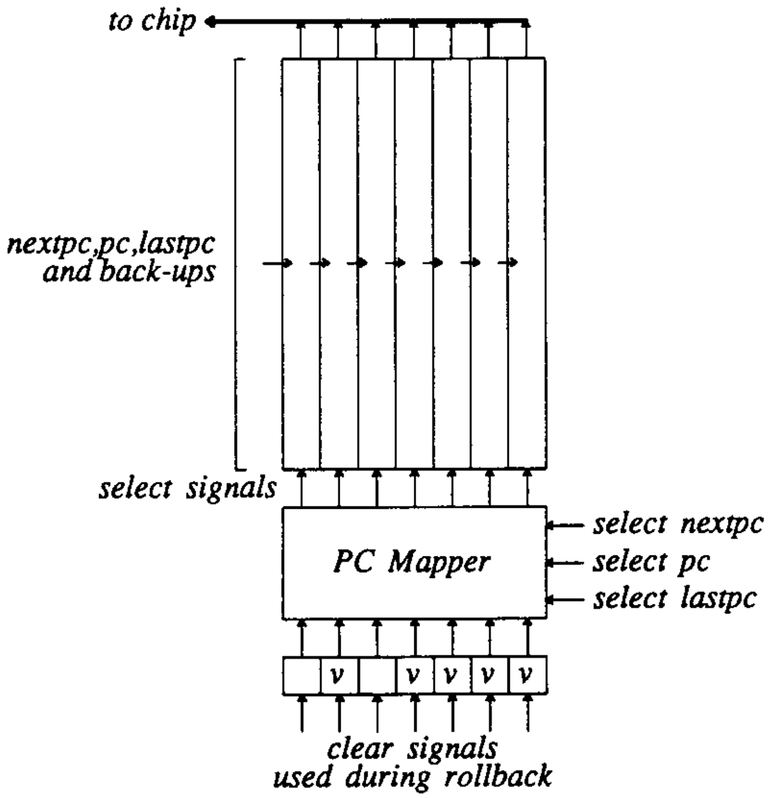


Figure 4.23: Micro rollback of the PC unit.

The PC mapper (Figure 4.24) is implemented using circuitry similar to the Invalidate Write Counter that we will describe in details in the next section

(Section 4.2.6). The layout of the complete PC unit with a rollback range of four cycles is only 2.8 times larger than the layout of the PC unit used for a system without micro rollback. This compares favorably to an overhead factor of 5.8 that would be achieved if the RAM method from the previous subsection was used for each PC register individually.

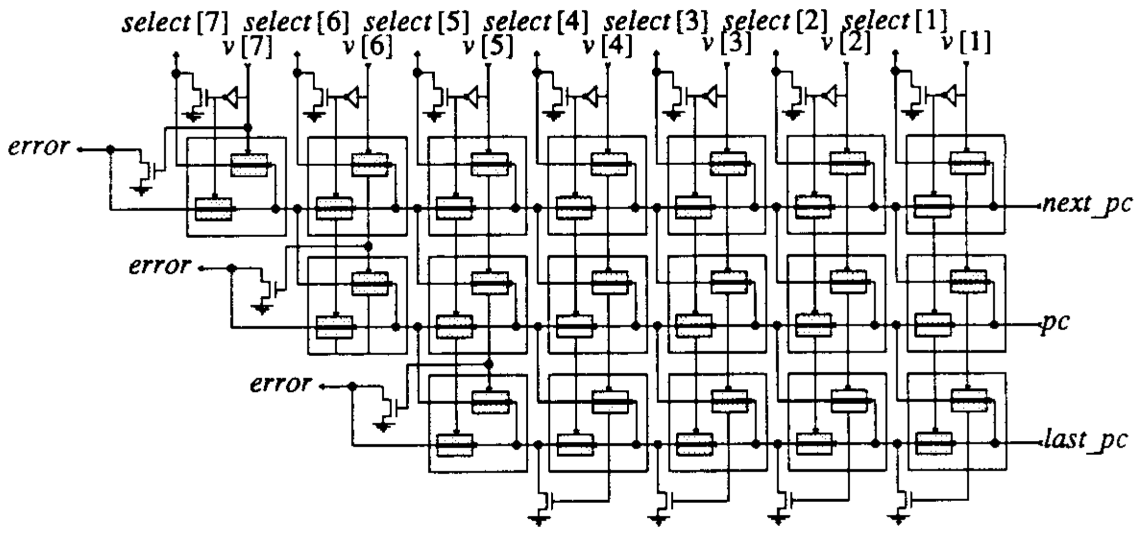


Figure 4.24: Mapper circuitry for the PC unit.

4.2.5.1. Error Detection for State Registers

A potential problem with rolling back the state registers occurs if micro rollback is initiated and the version of the individual register read from the backup registers is erroneous (due to a transient fault that occurred after the value was stored in the backup registers). Detection of such errors can be supported by adding error detection bits to the value of the register when it is stored in the backup memory. During a rollback, the contents of the appropriate backup register goes through the parity checker and is stored into the current state register. If an error is detected, a micro rollback on one additional cycle can be initiated, thus recovering from this potentially serious error.

4.2.6. A Delayed-Write Buffer for Infrequently Modified Registers

The method used to roll back the register file, described in Section 4.2.1, is based on the fact that *writes* into the register file can occur every cycle. For example many RISC processors complete an instruction every cycle [Kate83, Kane87]. Because it is possible to have N *writes* during N cycles, a Delayed-Write Buffer of depth N is necessary.

In some modules, *writes* to the register file may occur at a lower rate than once per cycle, so a smaller DWB may be sufficient. For example in the Motorola 68881 most instructions take at least 30 cycles to execute [Moto85a]. It is unnecessary to dedicate a DWB of N registers if it is known that during N cycles at most one *write* can occur so there is at most one value to invalidate in the entire DWB at any point in time. If a rollback of N cycles can "undo" at most M *writes*, only M registers are needed in the DWB. Considering that floating-point registers are 80-bits wide, the gain in area can be considerable.

A DWB for a system in which at most three *writes* can occur during five consecutive cycles ($N = 5$, $M = 3$) is shown in Figure 4.25. The part containing the data and the register addresses is similar to the full DWB, except that it now contains M registers instead of N. The control section is significantly more complex. It includes three major new components: a Write Monitor (WM), an Invalidate Write Counter (IWC), an Invalidate Write Mapper (IWM), and some logic for the shifting of the FIFO buffer.

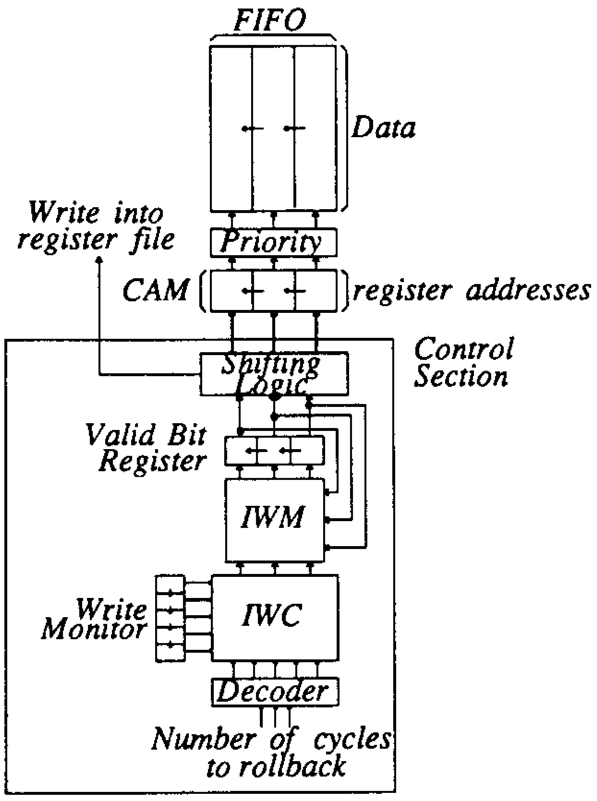


Figure 4.25: DWB for infrequently modified registers.

The Write Monitor keeps track of all the *writes* executed by the module; a one is shifted in whenever a *write* is executed, while a zero is shifted in otherwise. The number of bits in the Write Monitor is the maximum number of cycles that the register file may have to roll back (in this case, $N = 5$). The number of ones in the Write Monitor should never exceed M (in this case, $M = 3$). If it does, an error will be signaled.

The Invalidate Write Counter (see Figure 4.26) determines how many *writes* have been executed in the past C cycles ($C \leq N$) based on the contents of the WM. The inputs to the circuit use negative logic. Internally, the lines carrying the number of cycles to roll back are precharged to GND in the first phase phase. In

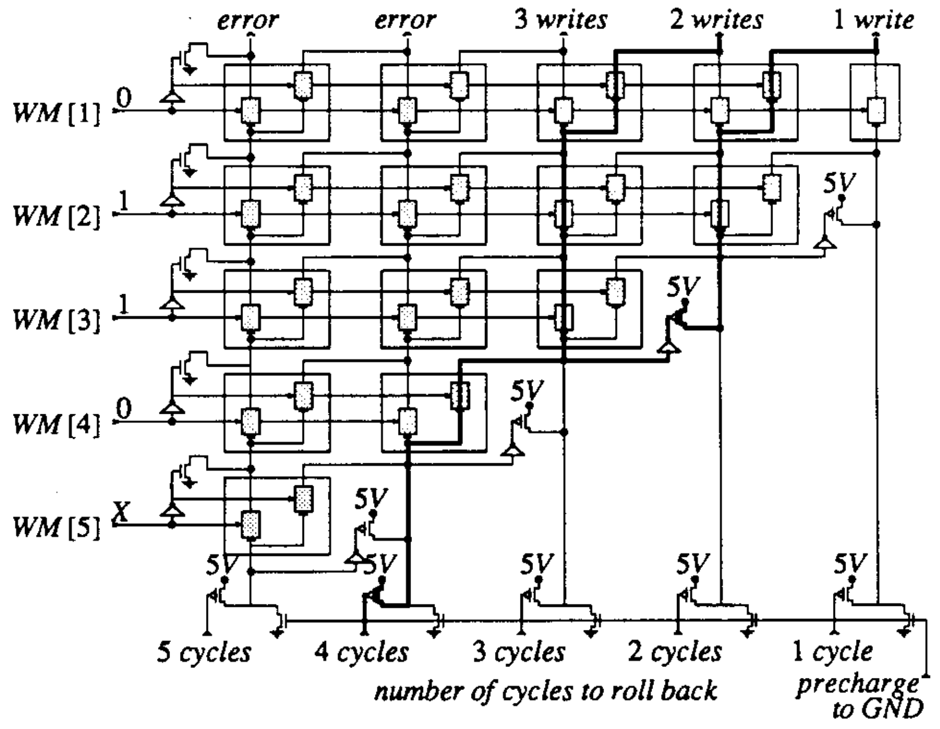


Figure 4.26: Invalidate write counter (IWC).

the second phase, only one input is zero so all but one of the lines are effectively disconnected from the inputs. The output of the circuit indicates that W *writes* are to be invalidated. In order to simplify the rest of the control circuitry, the circuit also indicates that $W-1$, $W-2, \cdots, 1$ *writes* should be invalidated. An error indication is signaled if the inputs result in a request to invalidate more than M writes. The basic cell for the IWC is a simple demultiplexer implemented with full transmission gates (see Figure 4.27). Its input is connected to output 1 when *select* = 1 and output 0 when *select* = 0. In Figure 4.26 we have shown a path created when the contents of the Write Monitor is [X0110] and a rollback of 4 cycles is requested. As shown in the figure, 2 *writes* (and 1 *write*) must be invalidated. The fifth entry in the Write Monitor does not modify the output because it occurred more than 4 cycles ago.

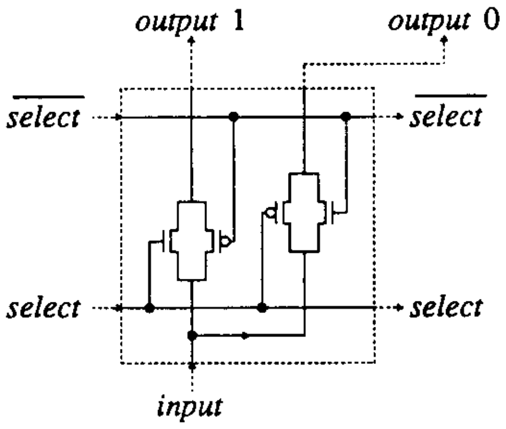


Figure 4.27: Basic cell for Invalidate Write Counter.

The Invalidate Write Mapper is used to identify which locations in the Valid Bit Register must be invalidated. The input is the number of *writes* to undo, while the outputs are asserted when the corresponding bits in the Valid Bit Register are to be cleared (Figure 4.28).

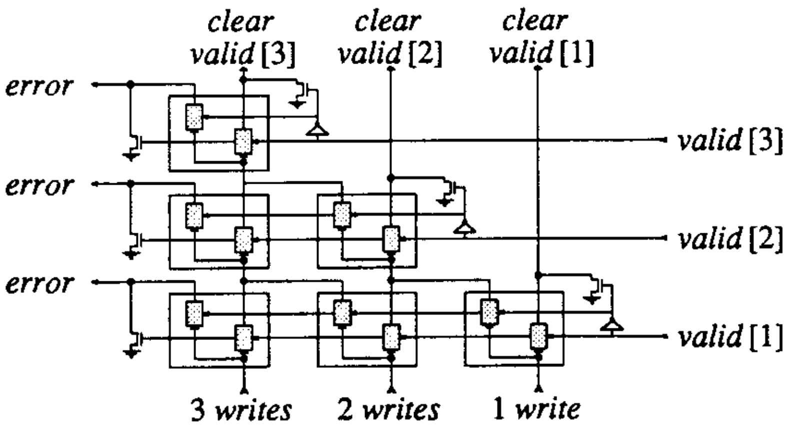


Figure 4.28: Invalidate Write Mapper (IWP).

The circuitry that controls the shifting in the DWB is shown in Figure 4.29. The leftmost register is shifted out to the register file only if the "oldest" bit in the

Write Monitor is one. The other registers shift to the left only if there is room, otherwise they remain idle.

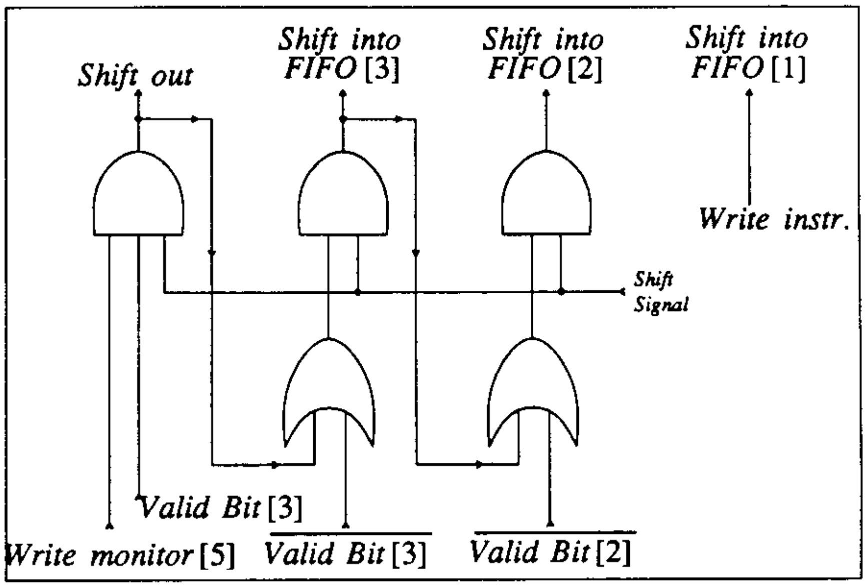


Figure 4.29: Control circuitry for shifting the DWB.

We have produced a layout of a complete Delayed Write Buffer similar as the one shown in Figure 4.25 ($N = 5$ and $M = 3$). The area of the circuit, including both the control and the all the storage, is 778640 λ^2, which is approximately 20% of the area of a dual-port register file with 64 32-bit registers. The control circuitry accounts for 13% of the total area for the DWB. We have determined the area of a DWB for several values of N (the maximum number of cycles that can be rolled back) and M (the maximum number of writes that can be undone). These results are summarized in Table 4.2 which indicates the ratio of the area of an "optimized" DWB (optimized for a small number of *writes*) over the area of a full version with $M = N$. For example if a module operates in such a way that its register file is never modified more than once every 4 cycles and the module may have to roll back up to 8 cycles, the table indicates that for $N = 8$ and $M = 2$ the optimized DWB takes only 34% of the area taken by a full DWB with 8 registers.

		Delayed Write Buffer Area (λ^2)			
		Generalized DWB			Full
		M=2	M=3	M=4	M=N
N (Maximum number of cycles to rollback)	4	527380 (58%)	758030 (84%)	988680 (110%)	900200 (100%)
	5	547040 (49%)	778640 (69%)	1010240 (90%)	1125250 (100%)
	8	617420 (34%)	851870 (47%)	1086320 (60%)	1800400 (100%)
	16	888700 (25%)	1130750 (31%)	1372800 (38%)	3600800 (100%)

Table 4.2: The Areas of "Optimized" and "Full" DWBs.

We have simulated the DWB circuitry using SPICE. The critical path is through the decoder, the IWC, the IWM, and the Valid Bit register, resulting in a delay of approximately 13ns. This fast operation allows single cycle rollback.

4.3. System Issues in the use of Micro rollback

In addition to the processor, micro rollback can be used effectively with other modules of the system. Parallel error checking and delayed error signals can be implemented using techniques similar to those used in the processor. The decision to use these techniques instead of serial checks is based on the potential performance improvement and the hardware overhead involved. In general, whenever data can be received or transmitted before it is checked, a DWB, such as the one described in Section 4.2, may be used to delay permanent modification of critical state until the results of the check are received.

4.3.1. Micro Rollback and Cache Memory

Most modern processors use caches to increase performance by decreasing the effective memory access time. The introduction of serial checking in the path between the processor and the cache introduces a severe degradation of the benefits gained by the cache. In this case, parallel checking means a significant performance gain.

As with the register file, a DWB can be used to support micro rollback in the cache. During every *load* instruction there is a lookup in the CAM to ensure that the processor will obtain the most recent value stored to the specified address. If the address is not found in the DWB, data comes either directly from the cache on a *hit*, or from main memory on a *miss*. On a hit, no critical state in the cache is affected so no actions need to be taken to undo the *load*. On a miss, a line in the cache has to be replaced by data fetched from main memory. During a micro rollback, there is no need to restore the previous contents of the cache line — the worst effect of an erroneous load is that a line has been fetched unnecessarily.

A *store* instruction is handled in the same way as a *write* to the register file (Figure 4.30). The *store* is delayed for N cycles in a DWB. The rightmost entry in the DWB is marked *valid* when a *store* is initiated. The last n entries in the DWB are marked *invalid* during a rollback of n cycles. Unlike the DWB for the register file, where both a register *read* and a register *write* could occur during the same cycle, a *load* and a *store* from/to the cache cannot be executed in parallel. This causes a problem when a valid *store* comes out of the DWB and a *load* is requested. Priority must be established between the two requests. Two possible solutions for this problem are: (1) the *load* is blocked until an invalid *store* reaches

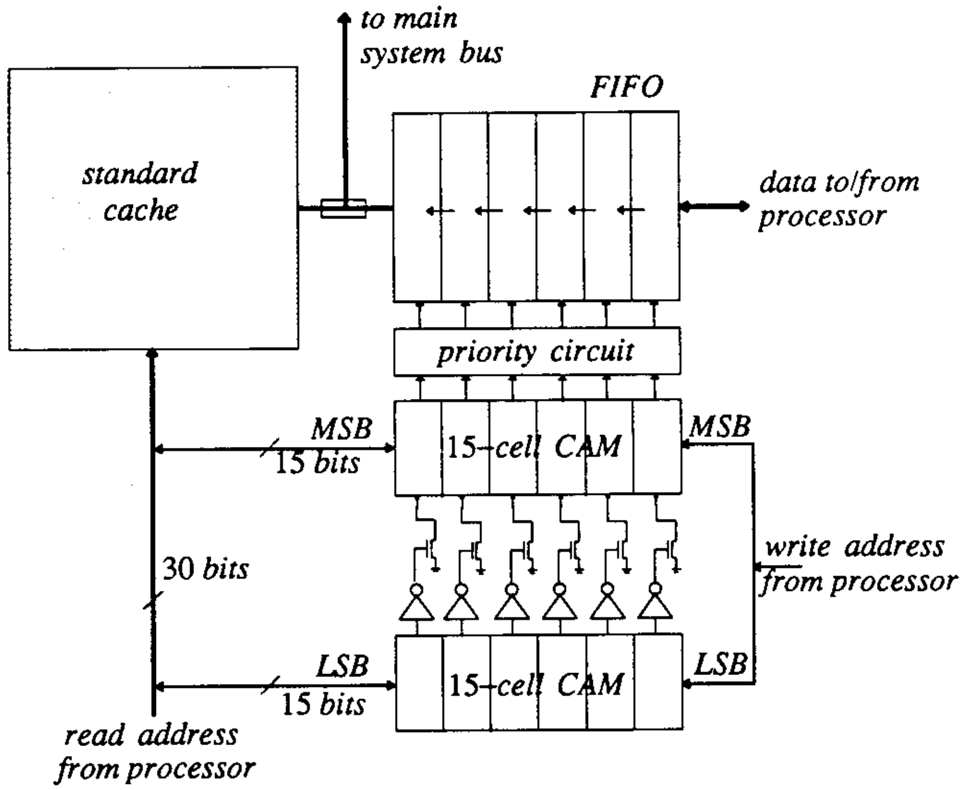


Fig. 4.30. A cache memory with support for micro rollback.

the end of the FIFO, at which time the *load* can be executed, or (2) the *stores* are blocked and priority is given to the *load.* The first solution is easier to implement but degrades performance. The required circuitry for the second solution, which does not affect performance, is described in Chapter 6.

It should be noted that the interaction between the cache and main memory occurs after the data has gone through the DWB. Thus, either write-back or write-through caches may be used and in both cases there is no need to undo a *write* to main memory (checking of the data is complete before the write to main memory occurs).

As shown in Figure 4.30, the lookup in the CAM of the cache DWB requires a comparison of thirty address bits. In order to speed up that comparison, the comparators are partitioned into two segments and thus achieve 12 ns operation. The lookup in the DWB is performed in parallel with the lookup in the cache. Since the access time of the cache in current VLSI systems is typically longer than 12 ns, the lookup time in the DWB should not degrade system performance.

4.3.2. Micro Rollback and Main Memory

In many systems it may not be desirable to include main memory in micro rollbacks. One of the problems with performing micro rollback in all modules of a large multi-module system is that it requires synchronous operation. For micro rollback of up to N cycles, each module must buffer up to N "versions" of its state and be capable of precise roll back of a specified number of cycles. This is difficult to do if the entire system is not completely synchronous. Many high-performance buses have asynchronous protocols so it may be difficult to coordinate micro rollbacks of the processor and main memory.

If the frequency of interactions (communication) between two particular modules in a system is small, the receiving module can simply wait for the error checks to be completed before using the data. Under these conditions the effect of waiting on performance is small and there is no justification for the extra hardware and added system complexity for micro rollback support. This argument often holds for main memory and I/O interface units in systems in which the processor uses a cache. In such systems the processor "rarely" accesses main memory. In addition, since communication across the system bus typically occurs in multiple word blocks, performing the checks serially, in a pipeline fashion, only delays the first word of the block and causes minimal performance degradation.

4.4. Micro Rollback in a Multiprocessor Environment

So far we have demonstrated how micro rollbacks work in a single processor system. We now describe how micro rollbacks can be used with a shared-memory multiprocessor system. The architectural model considered is the following: a collection of processors, each with its own private cache, are connected to each other as well as to main memory through a single shared system bus.

When a *write* is executed by a processor, the local cache and the rest of the system are normally not aware of it until the data exits the FIFO and is written into the local cache (Figure 4.30). Any actions required by the cache coherency protocol then takes place as with normal caches.

In a multiprocessor system where each processor has a local cache there is a problem of maintaining identical views of the logically shared memory from all the processors [Cens78]. Specifically, in order for the caches to be transparent to the software, the system is often required to be *memory coherent*, i.e., the memory system is required to ensure that "the value returned on a *load* is always the value given by the latest *store* instruction with the same address" [Cens78]. A multiprocessor system in which DWBs are used with the caches is *not* memory coherent. Specifically, a *load* executed by one processor cannot return the latest value written to the address by a *store* from another processor until the value stored "propagates" to the head of the DWB and is written to the cache (Figure 4.31).

In a multiprocessor system that is not memory coherent it is desirable to maintain the weaker condition of *sequential consistency* [Lamp79]. Sequential

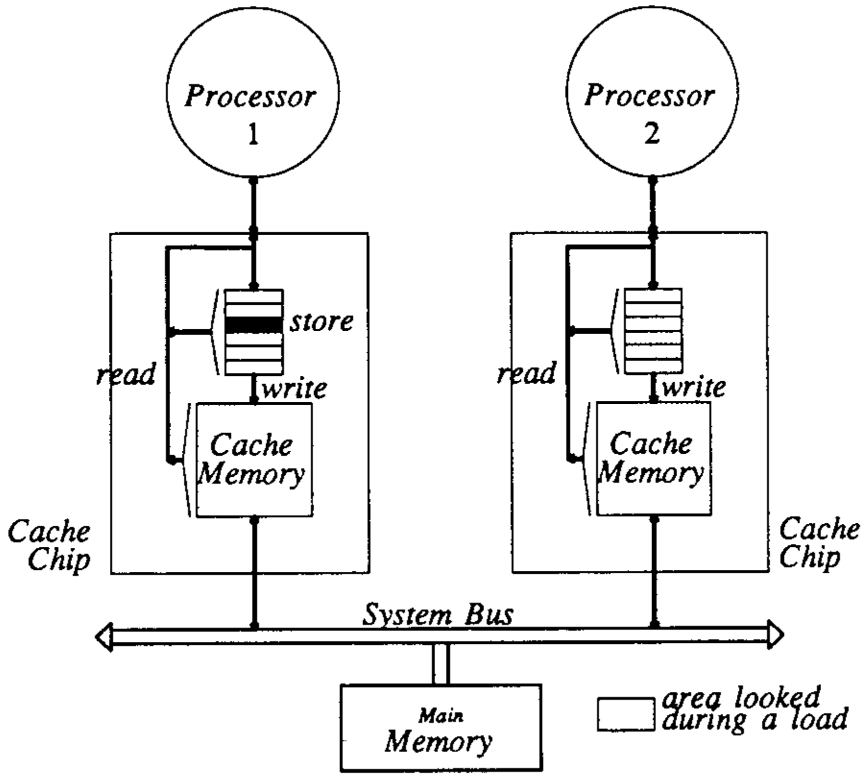


Figure 4.31: A recent *store* made by processor 1 is still in the DWB when processor 2 performs a *load*. The area looked at during a *load* does not include the DWBs of other processors.

consistency requires that "the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." Unfortunately, a multiprocessor system that uses DWBs is *not* sequentially consistent. The problem is illustrated by Lamport's [Lamp79] example of a mutual exclusion protocol:

```
Process 1: a := 1 ;
        if  b = 0  then critical section ;
             a := 0
        else  . . .  fi
Process 2: b := 1 ;
        if  a = 0  then critical section ;
             b := 0
        else  . . .  fi
```

With the DWBs, Process 1 can set a:=1 at the same time that Process 2 sets b:=1. They can then both reach their *if* statements before the *stores* setting *a* and *b* have time to propagate to their respective caches. Sequential consistency is violated since the result of the execution is as though the sequence of operations is:

```
Process 1:  if  b = 0  then critical section ;
Process 2:  if  a = 0  then critical section ;
Process 1:  a := 1 ;
Process 2:  b := 1 ;
```

Without modifications to the scheme shown in Figure 4.31, a multiprocessor in which the processors use the DWBs with their local caches is a very limited system. Synchronization through atomic instructions, such as *test-and-set*, can also become a problem. For a *test-and-set* instruction, the *set* is made right after the *test* during the same bus transaction. Using a conventional processor with the cache described in Figure 4.30, the *test* consists of reading the variable from the cache, while the *set* involves a *write* to the FIFO. A problem occurs if processor P_1 performs a *test-and-set* and the *write* is still in its FIFO when processor P_2

performs another *test-and-set* on the same variable. Processor P_2 will not be able to observe the *set* by P_1 and both processors will enter the critical section concurrently. As shown above, because the system is also not sequentially consistent, other mutual exclusion protocols that work on conventional multiprocessors may not work on a multiprocessor that uses the DWBs.

In order to allow synchronization in a multiprocessor with the DWBs, the cache controller must be modified. The simplest modification is to allow cache blocks to be locked during test-and-set operations. Specifically, a block containing a semaphore should be locked from the time it is accessed, until the time that it is modified. For a DWB of depth N, this means that the block is inaccessible during N cycles plus the time it takes to execute the *test-and-set* instruction. An alternative solution is for the cache controller to look up DWB entries as well as cache entries for each transaction it observes on the bus. If there is a bus transaction which ''hits'' on the DWB, it is blocked until either the corresponding data emerges from the DWB and is stored in the cache or the local node performs a micro rollback and the data is marked invalid. While this latter solution requires more hardware (support for *two* simultaneous lookups in the DWB — one from the processor and one from the cache controller), it has the advantage of maintaining sequential consistency, thus allowing parallel programs to work as expected.

4.5. Summary and Conclusions

In this chapter, we have described a mechanism, called *micro rollback*, which allows checking to proceed in parallel with intermodule communication by supporting fast rollback of a few cycles when a delayed error signal arrives. Micro rollback is a powerful technique that facilitates the implementation of high-performance VLSI systems which are also highly fault-tolerant. It allows a variety of concurrent error detection and correction techniques to be used with minimal performance penalty.

We have presented a systematic way to design VLSI computer modules that can roll back and restore the state which existed when the error occurred. Specifically, the implementation of micro rollback in simple synchronous systems involves replication of small isolated registers and the use of full *delayed-write buffers* (DWBs) for storing recent state changes to large register files. The efficiency of micro rollback has been demonstrated by showing the low area and performance overheads it incurs when added to a simple register file. For example a typical processor with a register file of 64 registers and a rollback range of four, has an area overhead for the register file of 14% and the processor cycle is stretched by 5%.

We have described a mechanism for reducing the size of the DWB for micro rollback of register files which are not modified every cycle. Based on CMOS VLSI layouts of the DWB and its control circuits, it was demonstrated that in some realistic situations our new design can result in savings of more than fifty percent of chip area compared to the previously used ''full'' DWB.

When applied to a VLSI RISC processor, the micro rollback technique is characterized by extremely low performance overhead and a modest area overhead compared to the area of the entire processor. We have shown how micro rollback can be used in a complete system with a memory hierarchy and we have discussed problems encountered in multiprocessor systems. The various subsystems presented in this chapter are representative of many common modules in wide variety of VLSI systems and are thus new critical building blocks which are likely to have wide application in future systems that will combine fault-tolerance and high performance.

Chapter Five

Micro Rollback for High-Performance Multi-Module VLSI Systems

Current VLSI computer systems consist of many complex chips in addition to a CPU chip — for example, floating point coprocessors, memory management units, communication coprocessors, etc. In this chapter we show how micro rollback can be implemented in such complex heterogeneous systems consisting of a variety of modules which may interact asynchronously [Trem89b].

The techniques discussed so far for micro rollback are based on system-wide rollback of a specified number of cycles. In a synchronous system, when one module rolls back, other connected modules roll back the same number of cycles in order to maintain a *consistent state* [Tami88b, Rand78]. Maintaining a consistent state is not as simple in a system with several modules which operate with different clocks and interact using an asynchronous protocol (for example, the processor and coprocessors in a system based on the Motorola 68020 [Moto85b]). Specifically, once one module rolls back, the number of cycles that other modules should roll back depends on recent interactions between the modules. This is a special case of the general problem of recovery in distributed systems [Rand78], except for the requirement that the entire operation should be performed by hardware in only one or two cycles. In this chapter, we present techniques for meetings these requirements. These techniques are based on using inter-module interactions as a basis for synchronization when coordinating roll back of multiple modules to a consistent global state. This is accomplished with special-purpose circuits that translate between inter-module interactions and internal clock cycles of each module. We present the design of these circuits and their evaluation based on VLSI layouts and extensive simulation.

5.1. Micro Rollback of Multi-Module System

In a system that consists of several modules, a rollback signal initiated by a module may affect other modules connected to it. Following a rollback of one of the modules, its state may be inconsistent with the state of other modules. If at time T module M_1 is rolled back t time units, its new state is consistent with the state of another module M_2 if, and only if, one of the following conditions is met:

(a) there were no interactions between M_1 and M_2 in the interval [T-t,T], <u>or</u>

(b) M_2 is rolled back to its state prior to any interactions with M_1 during the interval [T-t,T].

In case (a) there is no need to roll back module M_2. Since M_2 has not interacted with M_1 since time $T - t$, if the states of the two modules were consistent before M_1 was rolled back, they remain consistent following the rollback without requiring further action by M_2. In case (b) both M_1 and M_2 must be rolled back. To determine which case applies as well as the ''distance'' that M_2 may have to roll back, interactions between modules must be monitored. An *interaction* or a *transaction* is any transfer of information between two modules, such as data transfer, control signals, etc. In this section we describe how we monitor transactions and how we maintain consistency throughout the system when a rollback occurs.

In a synchronous system, all inter-module interactions are synchronous with a common clock. If one module, M_1, rolls back C cycles, the simplest way to maintain consistency in the system is to roll back all other modules C cycles plus the rollback signal propagation latency (Figure 5.1). This implies that some modules unnecessarily roll back even if they have not interacted with module M_1 in the past C cycles. In some cases, performance can be improved if modules are rolled back selectively, depending on their recent interaction with M_1. This method will be described in the context of asynchronous systems.

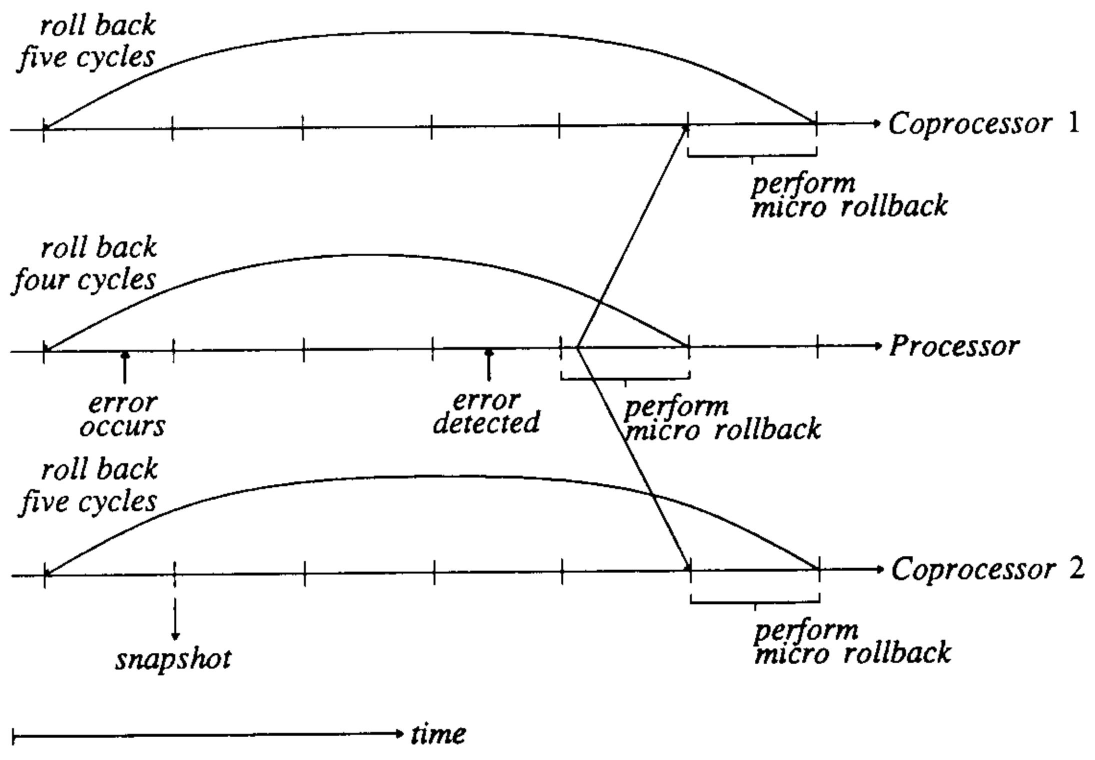


Figure 5.1: Micro rollback in a synchronous system: the module which receives the error signal is rolled back by a distance equal to the error detection latency. Adjacent modules roll back by the same distance plus the rollback signal propagation latency.

Many systems consist of modules that operate with different clocks and interact asynchronously. For example the Motorola 68020 processor [Moto85b] can operate at 25MHz together with a Motorola 68881 [Moto85a] floating-point unit operating with a 16.7MHz clock. Without a common clock, rollback in an asynchronous system cannot be coordinated based on the number of cycles to roll back (Figure 5.2). If two modules, M_1 and M_2, each roll back C cycles of their internal clock, their states following rollbacks may be *inconsistent*. If module M_1 rolls back C_1 cycles internally and during the last C_1 cycles it has participated in T transactions with another module M_2, then module M_2 must roll back to the state it had prior to the last T transactions with M_1. For M_2, T transactions may correspond to a different number of internal cycles, C_2.

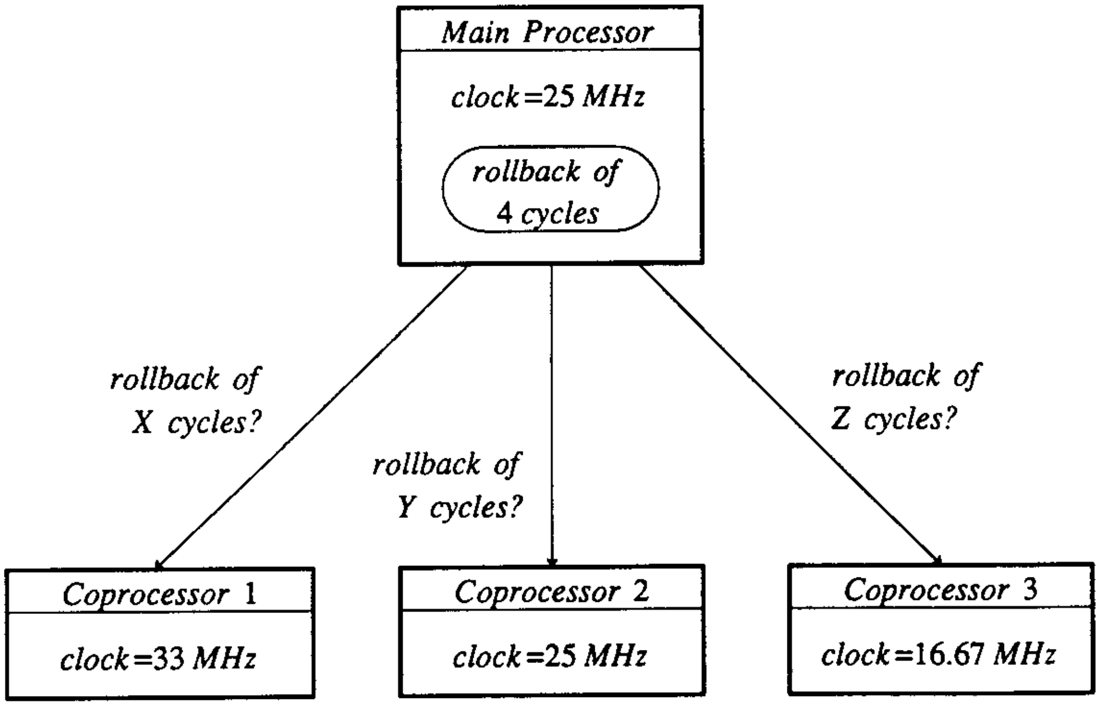


Figure 5.2: Micro rollback in an asynchronous system. No common clock $\rightarrow$ rollbacks cannot be synchronized through cycles.

In order to roll back to a consistent system state, the rollback of all the modules must be coordinated. This coordination is based on intermodule transactions. The module that initiates a rollback of a specified number of internal cycles must translate this number to the number of transactions that have occurred during that time with other modules and send this number of transactions to the other modules. In order to participate in a rollback initiated by other modules, each module must be able to receive the number of transactions to rollback and translate them to internal cycles. We have designed a circuit for performing the mapping between internal cycles and transactions. We call this circuit a transactions-to-cycles/cycles-to-transactions *transducer* (Figure 5.3). A module connected to several different modules through dedicated communication ports and links requires a transducer for *each* connection (Figure 5.4).

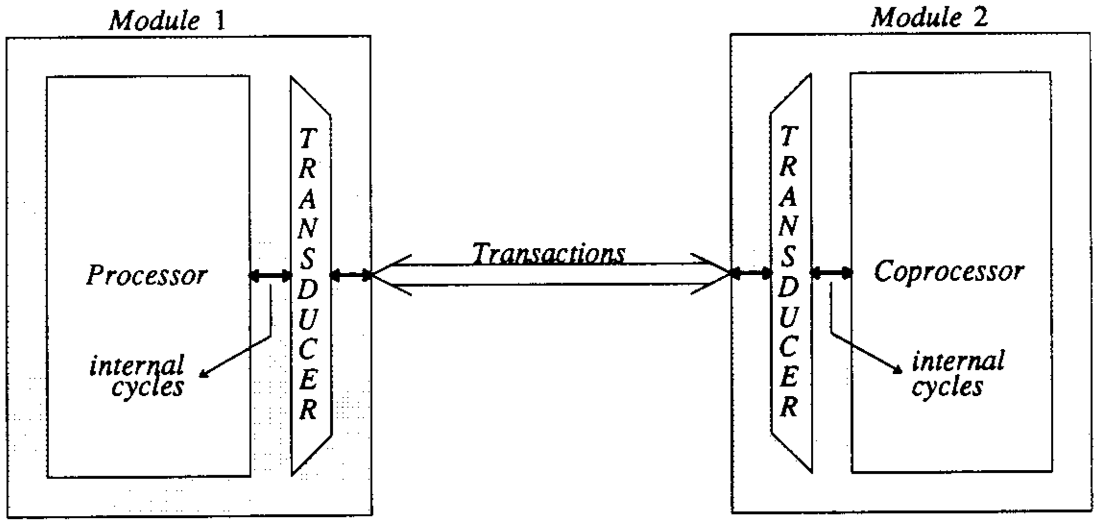


Figure 5.3: Synchronization through transducers.

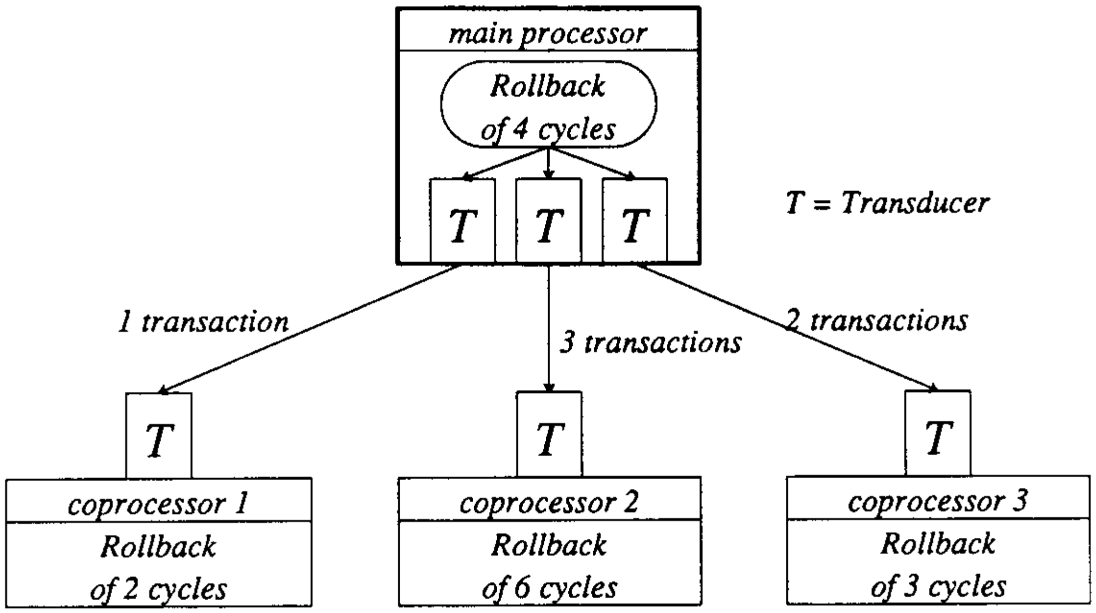


Figure 5.4: Transducers in an asynchronous multi-module system.

5.2. Cycles-to-Transactions and Transactions-to-Cycles Transducer

For a module M_1, a transducer (see Figure 5.5) performs three basic functions:

(1) It keeps track of transactions with another module M_2.

(2) When a rollback signal is generated internally, indicating that M_1 must roll back C_1 cycles, the transducer determines how many transactions (T) M_1 has performed with M_2, and sends that number to M_2.

(3) Upon receiving a rollback signal from M_2, the transducer converts the incoming number of transactions T to an internal number of cycles C_2. A micro rollback of C_2 internal cycles is then initiated.

The shift register labeled *Transaction Monitor* in Figure 5.5, is used to keep track of transactions with another module. During each cycle, a 1 is shifted in if a transaction occurs, a 0 is shifted in otherwise (no transaction). When a rollback of

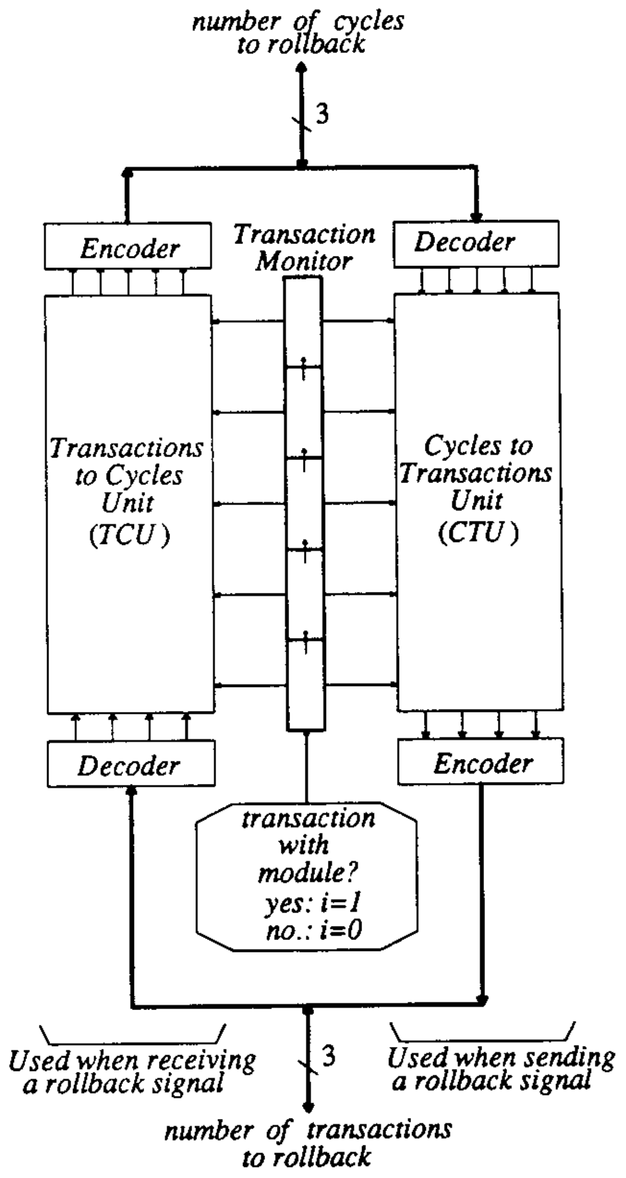

number of cycles to rollback
3
Encoder
Transaction Monitor
Decoder
Transactions to Cycles Unit (TCU)
Cycles to Transactions Unit (CTU)
Decoder
Encoder
transaction with module? yes: i=1 no.: i=0
Used when receiving a rollback signal
3
Used when sending a rollback signal
number of transactions to rollback

Figure 5.5: A transactions-to-cycles/cycles-to-transactions transducer.

C internal cycles is performed by a module, the first (bottom) C entries in the Transaction Monitor are cleared.

Two special circuits, a "Cycles-to-Transactions Unit" (CTU) and a "Transactions-to-Cycles Unit" (TCU), are used in a transducer to perform the

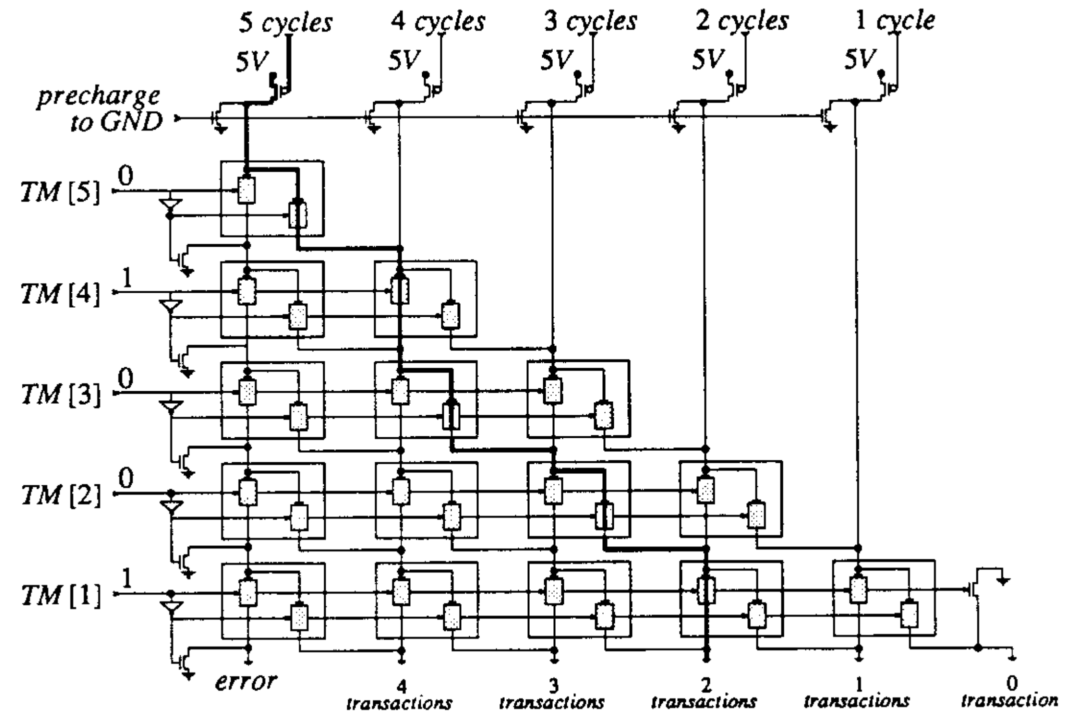


Figure 5.6: A cycles to transactions unit (CTU).

necessary translations. A CTU which converts a number of cycles ranging from 1 to 5 to a number of transactions ranging from 1 to 4, is shown in Figure 5.6. This circuit is similar to the Invalidate Write Counter described in Chapter 4. Instead of monitoring *writes*, it monitors transactions. The inputs are: a) the number of cycles to roll back — N, and b) the contents of TM, the shift register monitoring transactions. The output is the number of transactions that have occurred during the last N cycles, i.e., the number of transactions that should be "undone." The thick line in Figure 5.6 represents the connection established when a rollback of 5 cycles occurs and 2 transactions are "stored" in the Transaction Monitor. An error is signaled if the CTU determines that more than 4 transactions have to be undone. The maximum number of transactions possible for a given number of cycles depends on the module itself and on the protocol used to communicate with other modules.

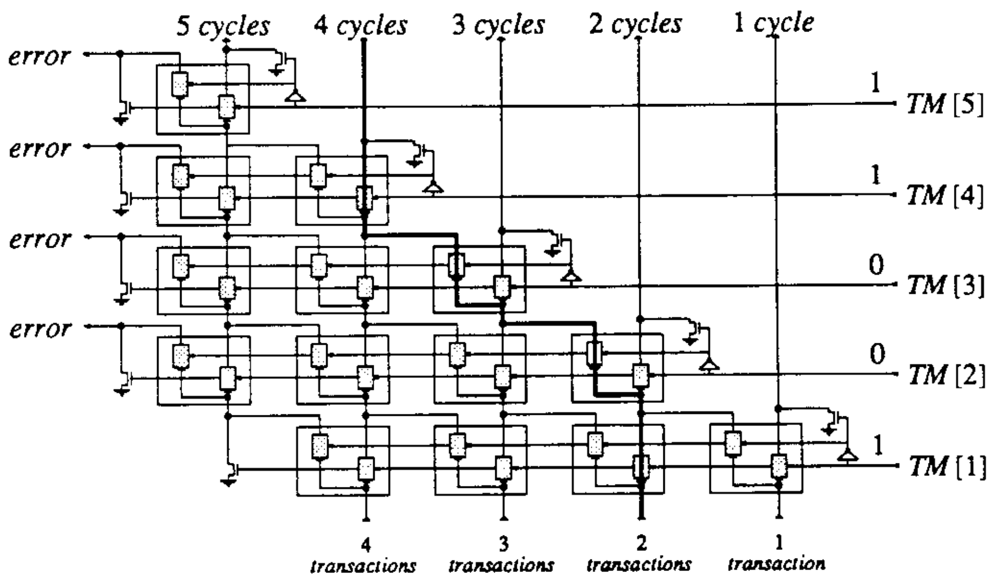


Figure 5.7: A transactions to cycles unit (TCU).

A TCU, which is dual to the CTU described above, is shown in Figure 5.7. The inputs are: a) the number of transactions to roll back and b) the contents of the transactions monitor. The output is the number of cycles to roll back. In Figure 5.7 the thick line represents a rollback of 2 transactions requiring a rollback of 4 cycles internally. In the case where the requested number of transactions to roll back is larger than the number of transactions contained in the Transaction Monitor, an error signal is sent to the control unit.

We have laid out the complete circuitry of the bidirectional transducer (Figure 5.8) and it measures 164000 λ^2, which represents 4% of the area of a dual-port register file with 64 32-bit registers, or 22% of a simple ALU. The time required to convert cycles to transactions (and vice versa) is about 10ns, which makes a single-cycle rollback possible even for fast modules.

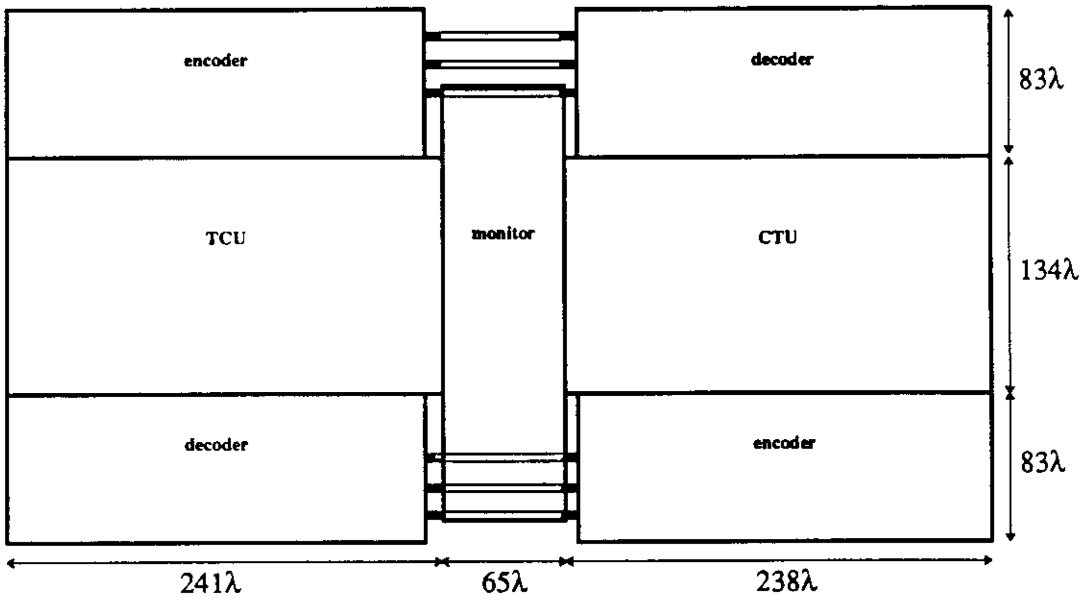


Figure 5.8: Layout of a transducer (not showing basic cells).

5.3. Micro Rollback in Bus-Connected Systems

Periodic checkpointing of process states and roll back to a previous state when an error is detected is a common technique for error recovery in distributed systems [Rand78]. If each process is checkpointed independently, rolling back one process may require rolling back a second process further in time which, in turn, may cause a third process to roll back, etc. leading to an uncontrolled *domino effect* [Rand78]. In the worst case, this can result in all processes in the system rolling back to their state when the system is initialized.

In the systems discussed so far, the modules are interconnected in a tree topology, where there is a unique path between every pair of nodes (Figure 5.2, Figure 5.3). In the context of micro rollback, which is done at the level of hardware modules, the domino effect cannot occur in such system. However, if the modules are connected in an arbitrary topology, where there are several independent communication paths between pairs of modules, the domino effect could, potentially, occur. Since the range of rollback is severely limited (a few cycles), this can make recovery impossible.

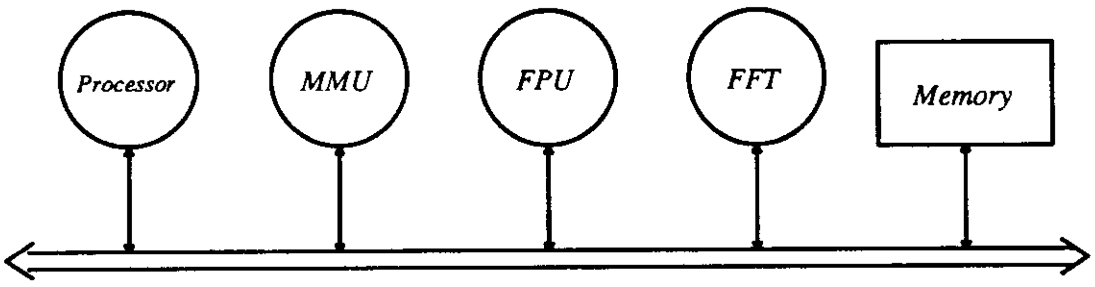


Figure 5.9: Bus-based multi-module system.

At first glance, it appears that the domino effect can be a problem when micro rollback is used in common bus-connected systems [AMD87, Moto85b] (Figure 5.9). For example, in the system shown in Figure 5.9, the following situation could occur:

- a rollback signal is initiated in the main processor which rolls back C cycles.
- the main processor sends a number of transactions to roll back to the FPU.
- the FPU rolls back and its transducer determines that the MMU must also roll back because it interacted with the FPU during the last few cycles.
- the MMU rolls back, and its transducer sends a number of transactions to roll back to the main processor, which, in turn, is now required to roll back more than C cycles.

In a system where all modules are interconnected via a common bus, this problem can be solved by using bus transactions as a common logical clock [Lamp78]. Bus transactions can be monitored by all the modules in the system and used for synchronization. There are two possible techniques for using bus transactions to achieve a consistent state following rollback:

1) Each module has one transducer which monitors generic bus transactions (Figure 5.10). Whenever a module detects an internal rollback signal, it converts it to a number of bus transactions, and puts it on the bus. All the other modules read this number of bus transactions, convert it to an internal number of cycles, and roll back. The disadvantage of this method is that it generates unnecessary rollbacks. Modules may roll back a certain number of system bus transactions even if they have not had any interactions with the rest of the system.

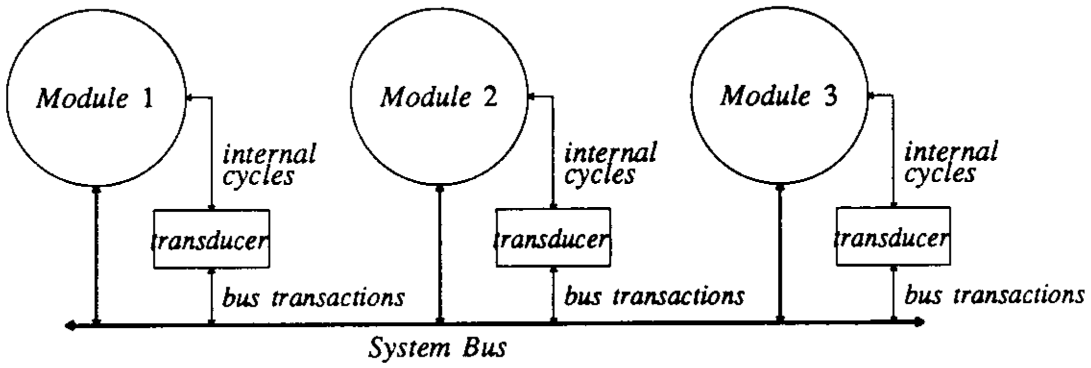


Figure 5.10: Elimination of the domino effect using a transducer to convert between bus transactions and internal cycles.

2) Each module has two transducers (Figure 5.11). The shift register (monitor) in the bottom transducer, shifts every time there is a bus transaction on the system bus. A "1" is shifted in if the bus transaction belongs to the module (*private* bus transaction). A "0" is shifted in otherwise. The shift register in the top transducer shifts every internal clock cycle. A "1" is shifted in if there was a bus transaction

involving the module during the particular cycle. A "0" is shifted in otherwise. If a rollback signal is detected, the following conversion occurs:

Generic Bus Transactions $\rightarrow$ Private Bus Trans. $\rightarrow$ Internal Cycles

In this way modules roll back only if necessary, but require twice the amount of hardware. The delays are also doubled which may make the implementation more critical for modules operating at high frequencies.

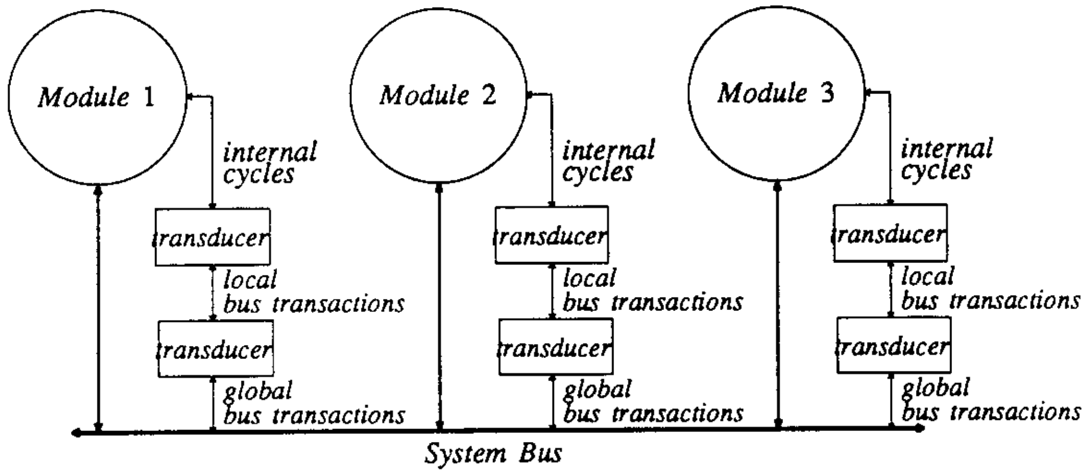


Figure 5.11: Elimination of the domino effect using two transducers to convert between generic bus transactions, private bus transactions, and internal cycles.

5.4. Summary and Conclusions

In a synchronous system, where all modules share a common clock and communicate via synchronous links, maintaining consistency following rollback is simple: all modules roll back the same number of cycles. In this chapter we have shown that micro rollback can also be implemented in heterogeneous asynchronous systems, where different modules operate with different clocks and communicate over asynchronous links. The key to implementing micro rollback in this type of system is a simple circuit that translates between the number of local cycles to roll back and the number of inter-module transactions. We have presented details of the implementation of this circuit as well as techniques for applying a variation of it in bus-connected systems composed of a processor and several asynchronous coprocessors.

Chapter Six
Interactions with Modules Outside of the Micro Rollback Domain

In the previous chapter, it was shown how micro rollback can be implemented in a multi-module system where each module by itself is capable of micro rollback. A set of modules always rolling back together to maintain a consistent state is called a *micro rollback domain* (Chapter 4).

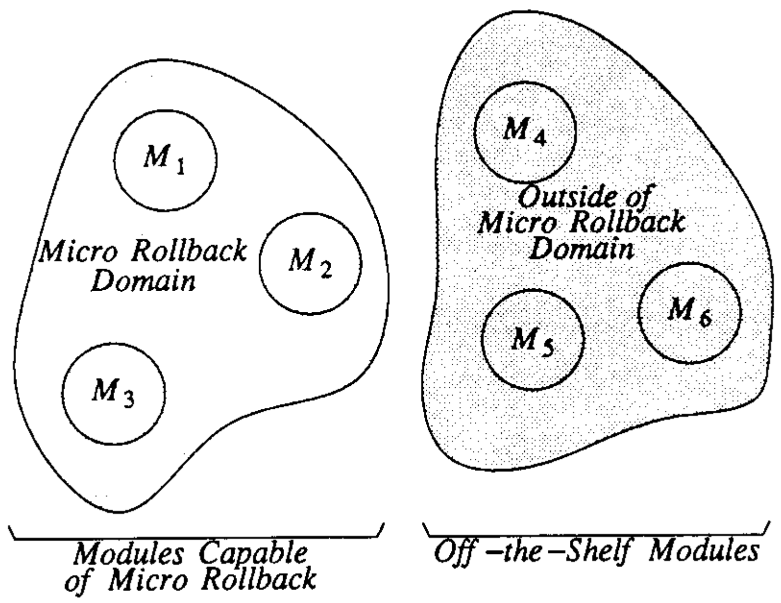


Figure 6.1: Classification of modules in a system.

In many cases it is useful to include in a single system some modules which are capable of micro rollback and some that are not. We call such a configuration a *hybrid system*. In a hybrid system, the modules which are incapable of micro rollback must remain outside the rollback domain. The interactions between the rollback domain and modules which are not capable of micro rollback (the ''outside world'') is the topic of this chapter. We will show that standard modules can communicate with the rollback domain through a dedicated interface unit. We focus on the design of simple interface units for memory systems and floating-point coprocessors.

6.1. Hybrid Systems

A fault-tolerant system based on micro rollback may benefit from the high-level of performance provided by specialized modules implemented for standard systems. For example, for some applications performance can be increased through the use of dedicated coprocessors for floating-point operations [Inte87, Moto85a, Rowe88, Mont90]. Complex ''intelligent'' cache chips [Mele89, Mati89] can improve the performance of the memory system to match the requirements of modern microprocessors. Other applications which may benefit from dedicated coprocessors include digital signal processing (e.g. FFT), graphics, pattern recognition, etc.

Adding micro rollback to these complex chips can be achieved by using the methods described in the previous chapters. These methods involve the modification of several units (leading to new layouts and new masks) which may require several man-years to produce. Even though these methods were developed with the goal of minimizing the area overhead and the performance degradation, the critical path may still be altered, requiring a reduction of the frequency at which the system (including all modules) operates. If the chip is to be produced for specific applications (low volume), or if a short design cycle is a priority, or if the system requirements demand that the performance of other modules does not decrease, the process of adding micro rollback to a complex chip in the system may not be a viable solution.

With the recent advances in custom and semi-custom design, a hybrid system residing on a single chip is foreseeable. A floating-point unit (FPU) could be part of a standard cell package (the MIPS 3010 uses only 75000 transistors) and could be combined with a previously designed processor capable of rollback, such as the MIRROR processor [Tami91]. For such cases, it is desirable to design general interface units which allow a single-chip hybrid system to be built rapidly without modifying the cells themselves (e.g. the floating-point multiplier pipeline).

The external inputs and outputs of a system are often inherently incapable of rollback. Sensors, data acquisition modules, commands and controls sent to physical systems, are a few examples of modules where rollback is difficult or even impossible to achieve. In these cases, a mechanism for interfacing the physical world with a high performance fault-tolerant system based on micro rollback must be developed.

Another possible motivation for building a hybrid system is the use of some modules that are implemented in a different technology, which is less susceptible to faults, and as a result have a much lower rate of errors than the modules that are capable of rollback. For instance, modules could be implemented with larger feature size, or they could be embedded in shielded packaging, or a technology more resistant to radiation (e.g. GaAs) could be used for their fabrication. With a low rate of errors, system requirements may be met despite the use of slow recovery mechanisms, such as system-level recovery [Rand78]. Some modules in the system (e.g. those implemented in GaAs) may have a much lower failure rate than other modules. It may be acceptable to use slow recovery mechanisms for errors caused by the failure of these ''resilient'' components, while faster recovery mechanisms will be necessary for errors caused by modules with a high failure rate. As an example, we consider a hybrid system, with GaAs modules, which are not capable of micro rollback, and CMOS modules, which are. All modules process data at their inputs as soon as it arrives, without waiting for checker outputs. For the GaAs modules, an error signal implies that their local state may be corrupt, so system-level recovery must be initiated. For the CMOS modules, an error indication triggers a micro rollback which restores them to a valid state, thus avoiding the need for system-level recovery.

6.2. Rollback Domain Interface Unit

Since state changes in modules outside the micro rollback domain cannot be undone, direct communication from modules in the rollback domain to the ''outside world'' cannot be achieved. In order to allow any communication to occur, a dedicated hardware module, called a *rollback domain interface unit* (RDIU), must be used to interface modules inside the rollback domain to modules outside the rollback domain (Figure 6.2).

The general case of establishing communication between the two domains can be made by treating that each transfer as an arbitrary transaction. By delaying all communication (through a buffer) from the rollback domain to the outside domain, it is possible to ensure that data or controls reaching the modules not capable of rollback are committed and do not need to be rolled back (Figure 6.3). In the other direction, i.e. from the outside domain to the rollback domain, communication does

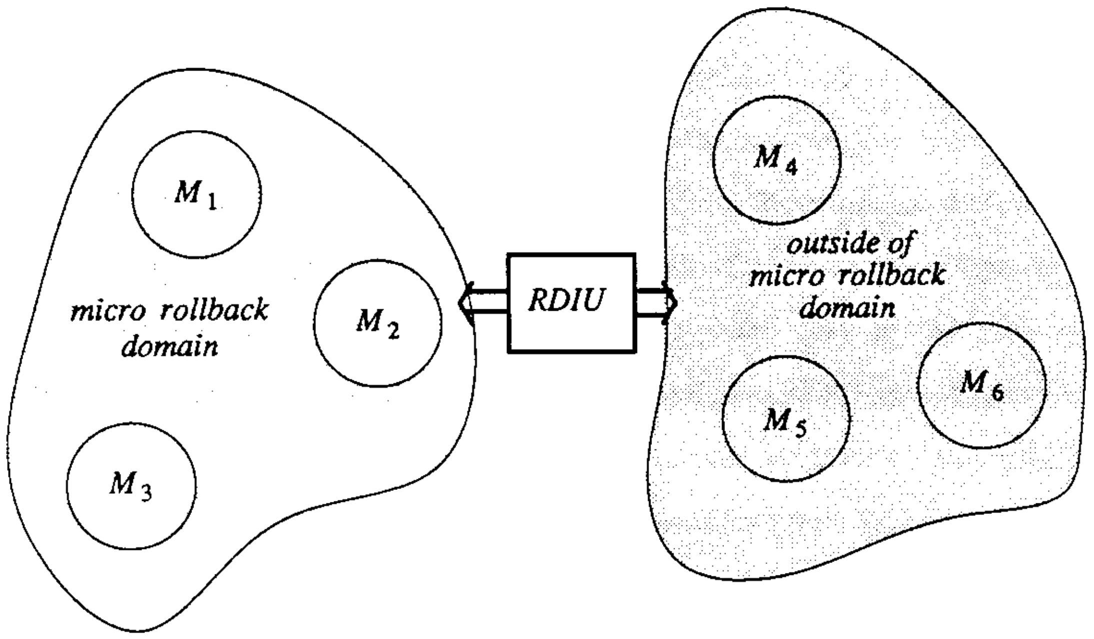


Figure 6.2: Insertion of an RDIU in the communication path. Both domains exchange data and control signals through the RDIU.

not need to be buffered since even if erroneous data reach a module, it is possible to undo any damage by rolling back the module. On the other hand, data produced by a module outside the rollback domain may need to be "replayed" following a rollback. This capability must also be provided by the RDIU interfacing the two domains ("replay memory" in Figure 6.3).

Delaying all communication in one direction may result in severe performance degradation. Based on the semantics of the communication between two modules, it is possible to optimize the interface so that only state damaging interactions are buffered. To better understand how an RDIU can be selective in the messages that are to be buffered and the ones that are not, we consider two important examples: the processor-memory interface (Section 6.3) and the

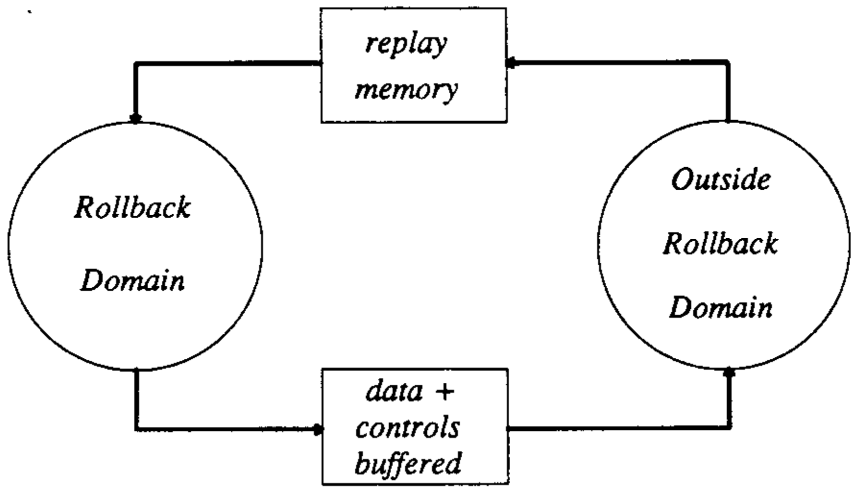


Figure 6.3: General RDIU allowing communication between the two domains. Communication is buffered in one direction, while exchanges are "logged" for possible replay in the other direction.

processor-coprocessor interface (Section 6.4 and Section 6.5).

6.3. Processor with Cache/Memory System Outside the Micro Rollback Domain

High-performance cache/memory modules providing high data transfer rates, efficient addressing modes, and wide data transfers, have been developed in order to increase the performance of computer systems. A system based on micro rollback can benefit from using these modules if, for the reasons mentioned in the previous section, proper interfacing is established. In this section, we investigate the design of specific RDIUs for interfacing a micro rollback capable processor (MRCP) with a variety of cache/memory systems. As we described in Chapter 4, depending on the cache/memory configuration it may not be necessary to roll back main memory. In the following discussion we look at the case where the chosen cache/memory system needs to be rolled back. For the sake of clarity, we refer to the cache/memory subsystem as the ''memory''. We first look at a simple one-cycle memory, going to a more complex burst-access variable latency memory system with out-of-order responses.

6.3.1. Processor and Single Cycle Memory

In the simple system depicted in Figure 6.4, a processor accesses memory through *load* and *store* instructions in a single bus cycle. The memory consists of standard chips without rollback capability. Direct communication cannot be allowed since erroneous *stores* can corrupt the memory. To allow interaction between the modules, an RDIU is required in the communication path between the memory and the processor.

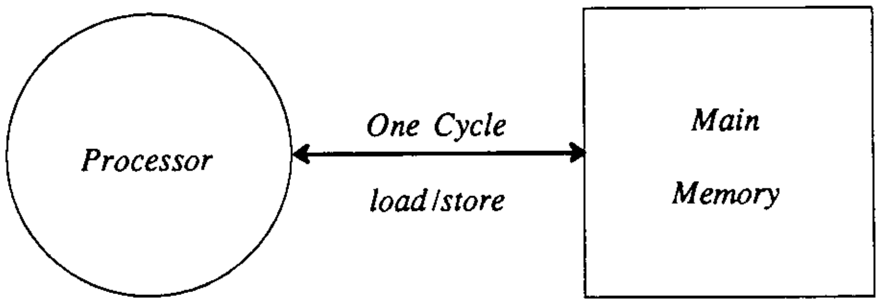


Figure 6.4: Single bus cycle communication between processor — memory

In order to make the RDIU transparent to the MRCP and the memory, the original interface between the modules must be maintained (Figure 6.5). While interacting with the processor, the RDIU behaves like the memory subsystem. For example, if the original interface specifies that the processor must be stalled if the memory system is not ready, the RDIU must send the same control signals to the

MRCP if the memory sends a busy signal to the RDIU. No modifications should be required to the protocol already established by the processor.

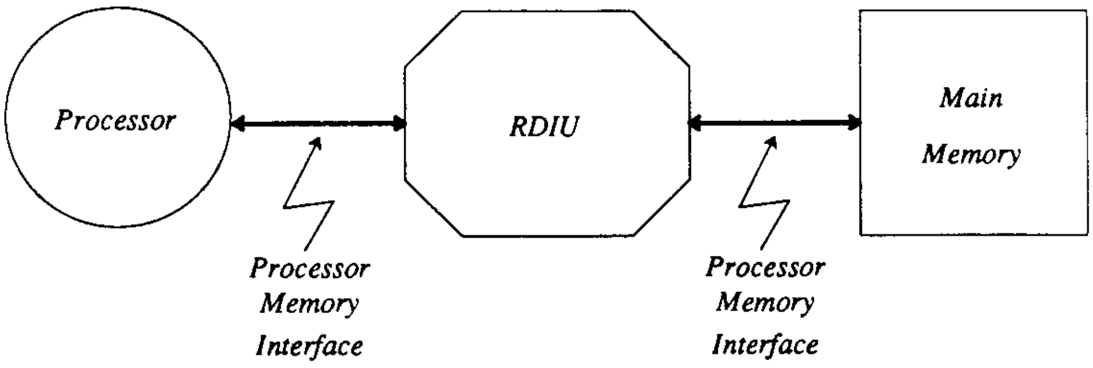


Figure 6.5: RDIU inserted in between a processor and main memory.

General RDIU for Single Cycle Memory

The replay memory needed for the general RDIU, described in Section 6.2 and shown in Figure 6.3, is not needed for a single cycle memory. With a memory system responding to processor requests during the same cycle as they are issued, only one active request is active at a time. The memory system starts processing a request and finishes it before starting a new one. Following a rollback, there are no outstanding requests. The exchanges that occurred during the past *n* cycles are replayed by the memory itself upon request from the processor.

In the other direction, that is from the processor to the memory, data and control signals generated by the processor during *load* and *store* instructions are not "committed". The general RDIU is used to delay these interactions until they are out of range of a potential rollback. Data and control can be sent "safely" to memory only after being delayed in a buffer until they pass the maximum rollback distance (the rollback range). The following operations happen in the RDIU during a *store* or a *load* (Figure 6.6):

— the address, control signals (*loads* and *stores*) and data (for *stores*) are shifted in the FIFO and the entry is marked valid.

— every cycle entries in the FIFO are shifted.

— valid entries reaching the end of the FIFO are sent to the memory. Invalid entries are discarded.

The main disadvantage of using a general RDIU for the processor-memory interface, is the overhead introduced by delaying *loads* for N cycles. A module may be idle waiting for data that would normally be available, but is being delayed in the FIFO. As shown below, this overhead is large, making the use of a general RDIU impractical.

Starting with a system S_1 with no RDIU, we define the performance measure τ as being the overhead *per cycle* that results if an RDIU is inserted in the communication path between the processor and the memory. The parameters upon which τ depends are:

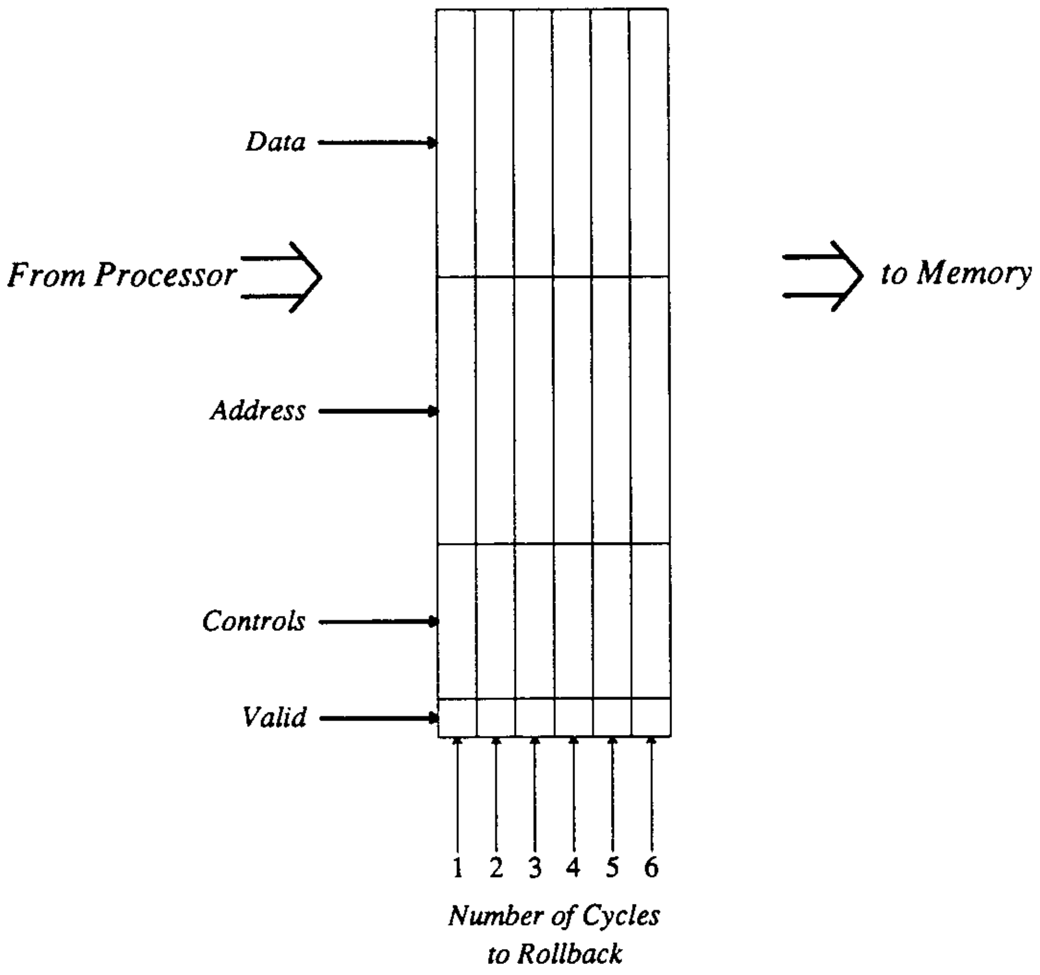


Figure 6.6: The general RDIU delays all interactions between the two domains

N	length of the FIFO
f	percentage of *load* instructions
s	average number of instructions that can be scheduled during the *load* latency

If we assume that one instruction per cycle can be scheduled, and that s non load-dependent instructions can be scheduled during the N "extra" cycles following a *load*, then the penalty is reduced to $(N - s)$. From this and the load frequency f we find:

$$\tau = f \times (N - s).$$

We consider a system where the FIFO of the RDIU must hold data for up to four cycles, $(N = 4)$. For a typical RISC processor *loads* represent about 20% of the instructions mix [Henn90]. From compiler technology, it is known that following a *load*, delay slots can be filled as follow [Gros83]:

first delay slot:	70%
second delay slot:	30%
third delay slot:	10%

Which means that on the average out of 10 *loads*, 7 will have one slot filled, 3 will have one more slot filled, and 1 will have an extra slot filled. On the average

$$0.7 + 0.3 + 0.1 = 1.1$$

useful instructions can be scheduled while waiting for a *load*. From these numbers we obtain:

$$\begin{aligned} N &= 4 \\ \mathrm{f} &= 0.2 \\ \mathrm{s} &= 1.1 \end{aligned}$$

which gives:

$$\tau = 0.58$$

Delaying *loads* through an RDIU with a FIFO of length four, would thus cause a considerable overhead of 0.58 cycle per cycle. For example if S_1 execute a program in 1000 cycles, a system with a DB of depth four would take an extra 580 cycles (1000×0.58) to execute (for a total of 1580 cycles). Thus, while the general RDIU is simple and maintains the original order for the transactions between the processor and memory, it also results in significant performance degradation.

Fortunately, the semantics of the memory can be used to reduce the performance penalty of the RDIU. The rest of this section deals with the design of a special purpose RDIU called a delayed store buffer (DSB) which eliminates this degradation.

Delayed Store Buffer

Since a *load* does not modify the state of memory (as opposed to a *store* which overwrites a memory location) it can be executed even if it is not *committed*. The memory will be consistent with the rest of the system even if *loads* are not ''undone'' (on the memory side) during a rollback. Thus, load requests can be sent directly to memory and the overhead of the RDIU can be eliminated.

Using a scheme similar to the DWB of the register file in Chapter 4, it is possible to provide for *loads* access to the most recent updates performed by the uncommitted *stores*. This hardware is the DSB. Sending *loads* directly to memory minimizes the load latency but introduces the problem of memory bus contention between *loads* and previously issued *stores* (see Section 4.3.1).

Contention for the memory bus occurs as a result of the following situation:

the first entry of the FIFO in the DSB is valid and is ready to be shifted out onto the bus to be written to the memory, while, simultaneously, a *load* is initiated by the processor. We describe two solutions to this problem:

Priority to stores: priority is given to the valid data in the DSB, *loads* simply block and wait for a "bubble" in the FIFO (a non valid entry) so that a load request to memory can be made. This solution is simple in the sense that the hardware required is minimum but the original performance may not be maintained if collisions occur. In the worst case, a *load* is blocked for *N* cycles until *N* valid *stores* are sent from the DSB to memory. Saving and loading a register file (through multiple *stores* and *loads*) during a context switch is an example of when blocking can happen. During normal execution, blocking depends on the probability of having a *load N* cycles after a *store*. This situation does not occur often since for typical integer programs *loads* represent up to 20% of the instruction mix, while *stores* represent around 10% [Henn90].

Priority to loads: to maintain the same level of performance as a system without an RDIU, *loads* must have priority on accessing the bus. Specifically, if there is a *store* ready to be sent to memory and a *load* is requested from the processor, the DSB remains idle (does not shift) and the *load* proceeds as in a normal system. Even though the FIFO is not shifted, its contents are now one cycle closer to *commit time* which should be reflected in the RDIU. Logic that keeps track of how long data have been in the DSB must be provided. The following section explains how this mapping is accomplished in hardware.

Mapping Between a Number of Cycles to Roll Back and DSB Invalidate Signals

If N is the maximum number of cycles that the processor can roll back, then we must keep track of what happened in the previous N cycles and map a number of *cycles* $n \leq N$ to a number of *invalidates* $(i \leq N)$ (Figure 6.7). This kind of mapping is similar to the mapping of a number of cycles to invalidate to a number of *writes* to invalidate in the optimized design of the register file (Section 4.2). The only difference is that in the RDIU, whenever there is a *shift*, the whole FIFO shifts, compared to individual shifting of the registers in the DWB. Individual shifting was necessary in the optimized register file in order to minimize the number of entries in the FIFO. The size of the FIFO was based on the maximum number of *writes* that could occur during N cycles. For the one cycle memory, N *stores* can occur during N cycles, which dictates a FIFO of N registers. A global shift of the FIFO results in simpler mapping (than with individual shifts) but the performance is slightly lower since data coming from the processor are blocked if the first entry of the FIFO is occupied, even if the FIFO does not contain N valid entries. Individual shifting of entries in the RDIU, similar to the shifting for the optimized DWB, would allow less blocking than with global shifts. The task of locating valid entries in the FIFO (done by the invalid write mapper (IWM) in the optimized register file), is eliminated for this RDIU. The block diagram of the modified delay buffer with the circuitry for the mapping is shown in Figure 6.8.

A "1" is shifted in the *shift monitor* (SM) in Figure 6.8 whenever the FIFO

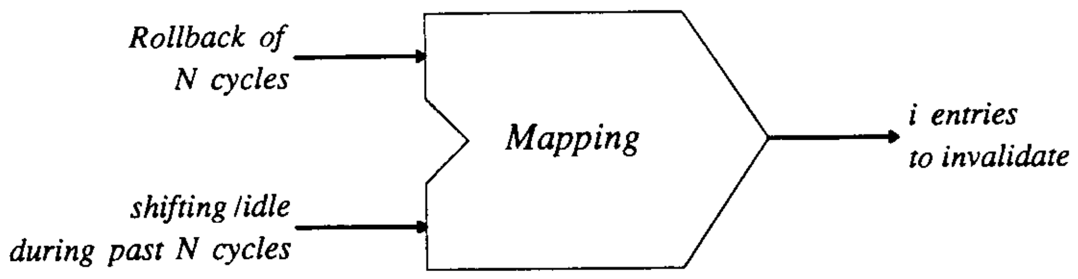


Figure 6.7: Mapping between *n* cycles and *i* invalidates.

shifts. Otherwise a "0" is shifted in. The *N* cells in the shift monitor (where *N* is the rollback range) shift every cycle. The FIFO shifts every cycle unless there is a collision with a *load* being issued by the processor. The *Mapper* (Figure 6.9), a circuit similar to the Invalid Write Counter (IWC in Chapter 4), converts a number of cycles to roll back to individual invalidate signals for the entries in the FIFO. We show in Figure 6.9 a Mapper for a delay buffer for which the FIFO was blocked for two consecutive cycles due to conflicts with *loads*. A rollback of four cycles in this example will be routed through the transmission gates of the Mapper to two invalidate signals ($invalidate_1$ and $invalidate_2$). This shows that the two idle cycles have been acknowledged by the RDIU.

6.3.2. Processor and Memory with Wait States

In a system where the latency for memory accesses is determined by the memory rather than the (faster) processor, *wait states* are introduced between the processor request and the data transfer from memory. This is accomplished with simple *handshaking* between memory modules and the processor. As long as the memory notifies the processor that the data is not ready, the processor is in a *stall* mode (for simple systems).

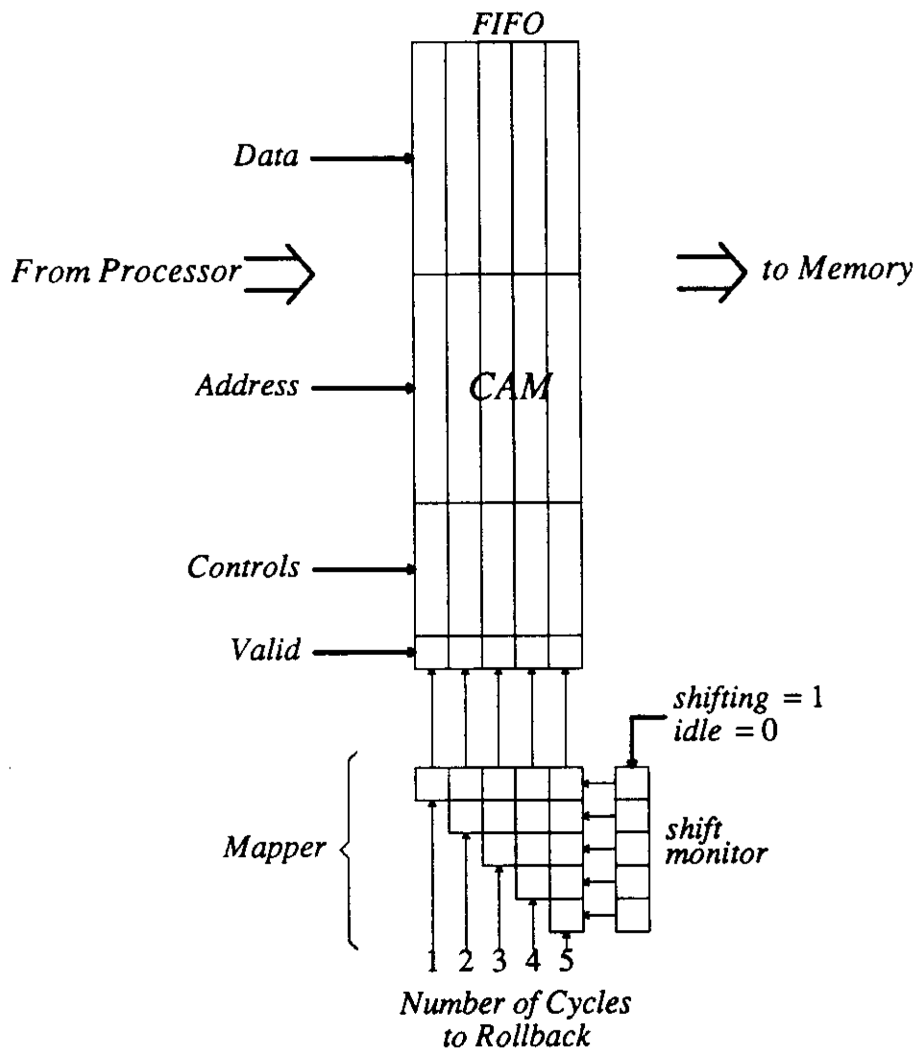


Figure 6.8: A delayed store buffer serving as the RDIU for single cycle memory.

When an RDIU is introduced between the processor and the memory, the memory interacts directly with the RDIU instead of the processor. If the memory is busy, the FIFO of the RDIU is stalled, preventing data in the FIFO from being sent out to memory. During that time, the RDIU blocks any incoming *stores* from the processor. Even though the FIFO is not shifted, the data contained in the RDIU moves closer to the commit time during each cycle (wait state or not). This

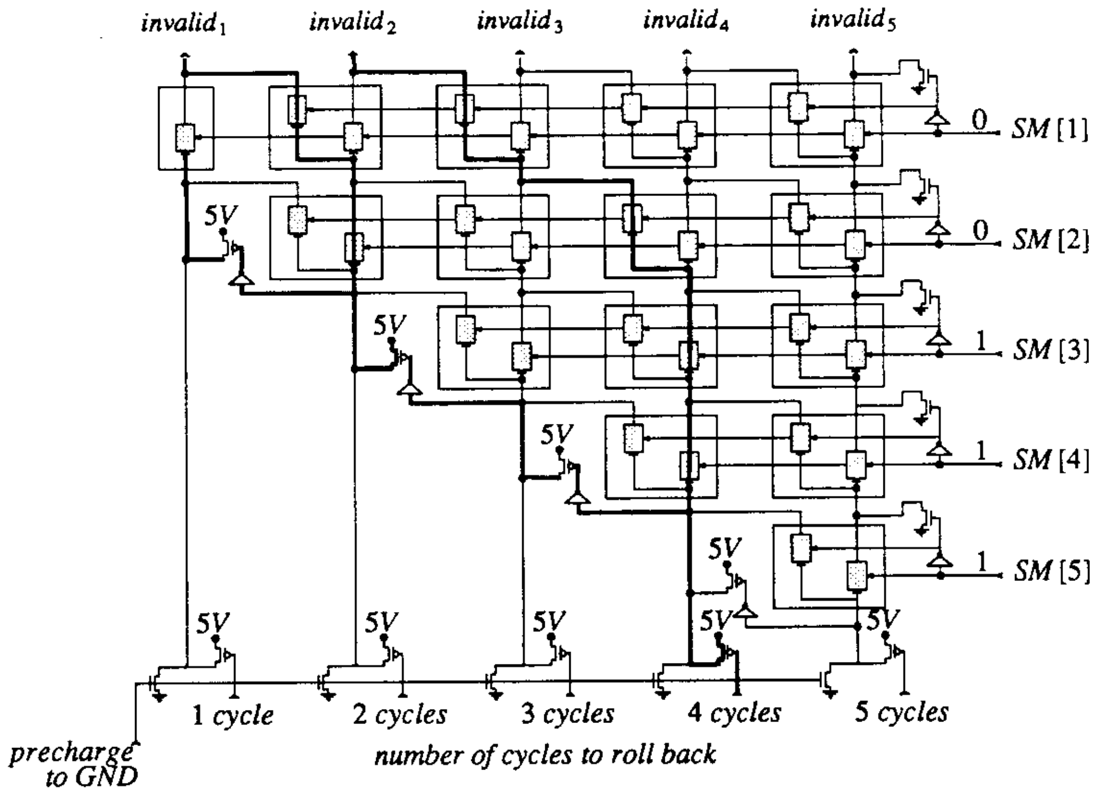


Figure 6.9: Store mapper. In this case a rollback of 4 cycles invalidates only the first two entries in the buffer.

situation is similar to the one described in the previous section where a collision between a *load* and a *store* resulted in stalling the FIFO of the RDIU but also kept track of how long data has been in the FIFO. Thus, the RDIU structure previously described in in Figure 6.8, can be used for memory systems with wait states.

As with the simple memory with no wait states, a replay memory is not needed since the processor blocks until access is complete and the memory handles processor requests sequentially in a pipelined manner (there is never more than one outstanding request).

6.3.3. Processor and Memory with Burst Access Mode

Burst access mode allows rapid loading or storing of consecutive blocks of words in memory. A single address $[A]$ is sent to memory and the rest of the words in the block $[A+1, A+2, ..., A+n]$ are accessed through simple handshaking between the processor and the memory. Burst access mode can result in higher memory bandwidth and reduced load on the address bus. It also offers the possibility, once the starting address is sent, to use the address bus as a supplement to the data bus thus achieving higher bandwidth through wider data transfers [AMD87].

For a system which provides burst access modes, the insertion of an RDIU requires the following for maintaining correct operations for *stores* and *loads*:

Stores:

When the processor requests a burst mode access, it is granted by the RDIU and the communication starts between these two modules. All requests are buffered for N cycles. After the first request goes through the FIFO (after a delay of N cycles), the RDIU requests a burst access to the memory. The control signals that the processor initially sends to the RDIU indicating a burst access must be stored along with the address and the data in the FIFO of the RDIU. A CAM similar to the one described earlier guarantees that the latest update of a variable is available for subsequent *loads*.

Loads:

In previous systems we have described a scheme where *loads* have priority over *stores* whenever there is a conflict between the output of the FIFO of the RDIU and the processor. This is not the case with burst accesses. If a burst *store* is initiated by the RDIU to the main memory, it is not interrupted by a *load* from the processor. This simplifies the logic involved and does not require complex operations for stopping the burst access, execute the *load*, and resume the *store*. The *load* simply blocks for n cycles ($n \leq N$) until the burst *store* terminates. This operation decreases performance slightly when a *load* follows a burst *store* too closely. This can occur for example during a context switch when the whole register file is saved through many consecutive *stores* followed by many consecutive *loads* to bring the new variables into the register file. To alleviate this problem, non-load instructions can be inserted between the series of *stores* and *loads*.

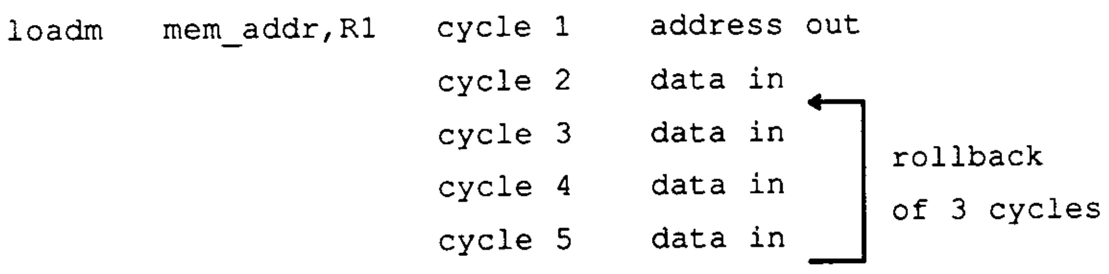


Figure 6.10: Rollback during a burst load (similar for burst store).

If a rollback occurs in the middle of a burst load/store, e.g. a few cycles after the address has been sent (Figure 6.10), the processor is not able to regenerate the current address since during a burst access only controls signaling consecutive addresses are sent. In order to be able to continue the burst *load/store*, the address must be regenerated and the proper control signals must be sent by the RDIU. Hence, the RDIU must keep track of the addresses used for each access, including each word access which is part of a block burst access. For each *load/store* requested by the processor, the corresponding address is stored in a FIFO in the RDIU (Figure 6.11). A burst access is detected by the RDIU and instead of getting the address from the processor, it is generated on-chip by incrementing the previous address by the amount originally specified by the processor at the beginning of the burst protocol. The result is stored in the FIFO. After a rollback, the RDIU must determine if it is in the middle of a burst load/store so that if an increment signal is received from the processor, it will be interpreted as a continuation of a burst load/store.

Addresses entering the FIFO of the RDIU are eventually shifted out of the FIFO when they reach commit time. At that point they become out of reach of a rollback and can thus be discarded. During a rollback, the FIFO containing the addresses in the RDIU keeps shifting since the addresses are effectively getting ''older'' (closer to commit time).

The replay memory is not required for the same reasons that we enumerated when describing for the previous memory systems (processor blocking until access is complete, and sequential, non-pipelined accesses).

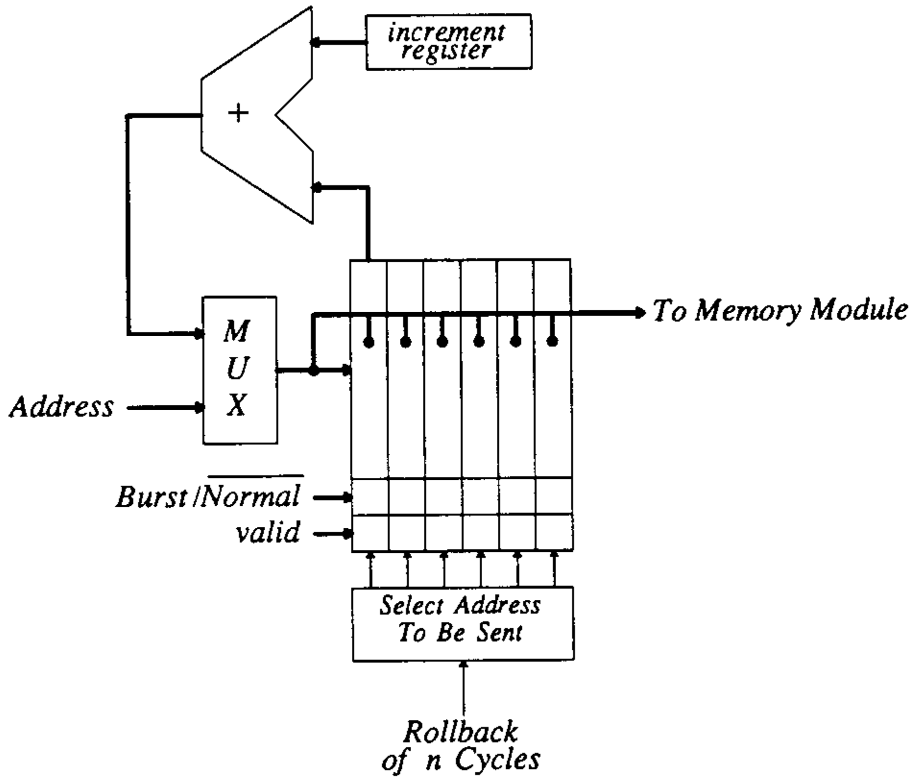


Figure 6.11: Logic required to handle rollbacks occurring during a burst load.

6.3.4. Processor and Memory with Pipelined Accesses

In order to increase the bandwidth between the processor and memory modules, accesses to memory can be pipelined. The process of receiving the address, decoding the address, accessing the memory array, fetching the data, and sending the data back to the processor can be pipelined by inserting latches between distinct segments (Figure 6.12). In this way, a request to a multi-cycle latency memory system does not hold the bus waiting for the access to complete (other requests can be made). As an example we mentioned the memory interface of the Intel i860 processor [Perr89]. The bus interface unit can send up to three memory requests before getting data associated with the first request back from memory. With pipelined access, high throughput can be achieved despite long latencies in the memory subsystem.

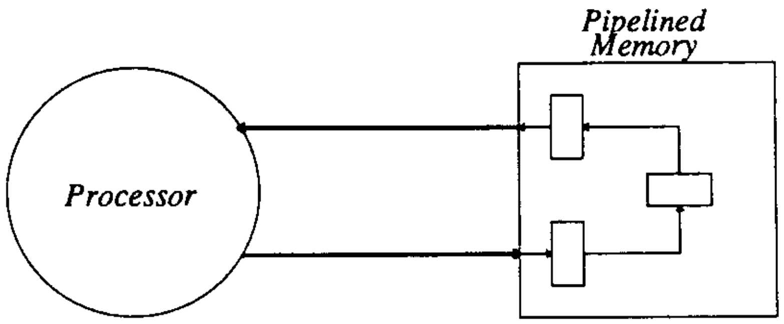


Figure 6.12: Pipelined memory subsystem

Because the memory pipeline resides outside the micro rollback domain, a processor rollback does not propagate to the memory modules and cannot ''undo'' the previous memory requests. This introduces a problem when normal execution resumes after a rollback since responses to *previous* memory requests may collide with *new* memory requests. For correct operation, the first few data items returning from memory should be discarded.

With pipelined memory, a processor may send a new *load* request to the memory while it accepts data requested p cycles ago (where p is the depth of the memory pipeline). Thus, a rollback of n cycles, besides undoing *loads* initiated during the past n cycles, also undoes the ''accept data'' operations tied to *loads* issued beyond n cycles (the rollback distance). Data transfers which are the result of loads initiated more than n cycles prior to rollback are valid. Hence, following rollback, the data must be resent to the processor.

We propose three schemes for supporting pipelined accesses and micro rollback. These schemes are suited for three alternative pipelined memory designs. The first system implements in-order fixed latency *loads*, the second one has variable latency *loads*, while the third one can process *loads* out-of-order. We concentrate on the *load* operation which is more complicated than the *store*. *Stores* can be handled by the same hardware as described previously (data are buffered and entries are invalidated during a rollback). When an invalid *store* reaches the end of the FIFO, the pipeline advances but no request is sent to memory.

In-Order Fixed Latency Loads:

When *loads* return in-order after a fixed number of cycles (through synchronous operations), a shift register of length L, where L is the latency of a *load*, is used to keep track of pending *loads* (Figure 6.13). During each cycle, a 1 is shifted in the load monitor (LM) if a *load* is requested by the processor, otherwise a 0 is shifted in. When a the data arrives from the memory modules, it is sent directly to the processor only if the rightmost entry in the shift register is 1 (valid). Otherwise the data is discarded. A rollback of n cycles invalidates the n most recent (leftmost) entries in the load monitor, so that the *loads* associated with those bits get discarded once they arrive from memory (Figure 6.13).

The replay memory shown in Figure 6.13 saves all data sent to the processor during the past N cycles. The *replay register* (under the replay memory in Figure 6.13) is used to indicate which entries in the replay memory are valid. The input of the replay register is connected to the output of the load monitor, thus forming a longer shift register. If a rollback of n cycles occurs, the pointer pointing to the rightmost entry in the load monitor is moved "up" n positions into the replay

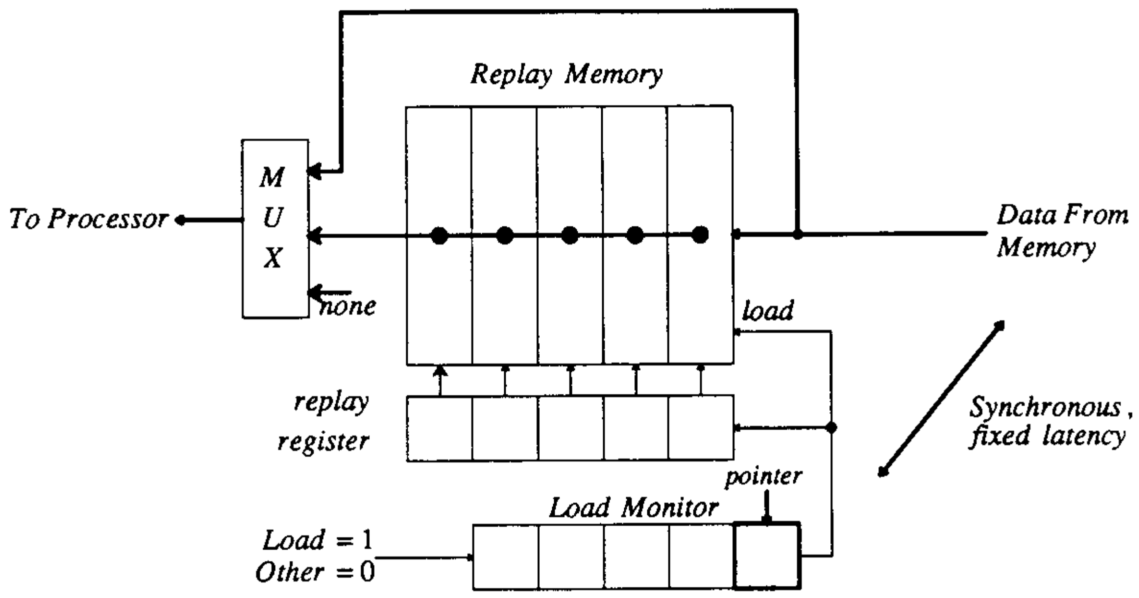


Figure 6.13: An RDIU for pipelined memory with in order fixed latency loads.

register. If the content of the replay register pointed to by the pointer is "1", then the data saved in the replay memory is re-sent to the processor.

During a rollback, the memory subsystem, being outside of the rollback domain, is unaware of the rollback and may keep sending data. Thus, in order to be able to replay this data when normal execution resumes, the incoming data is shifted into the replay memory while the pointer points to the same entry in the replay register.

When a "0", indicating a rollback, reaches the rightmost entry of the load monitor, the replay memory does not shift. Instead, the pointer is shifted one entry "to the right" in the replay register. This has two effects, (i) by not shifting the replay memory, incoming data is discarded (which is required by the rollback), (ii) by moving the pointer, data that has arrived while the pointer is in the replay

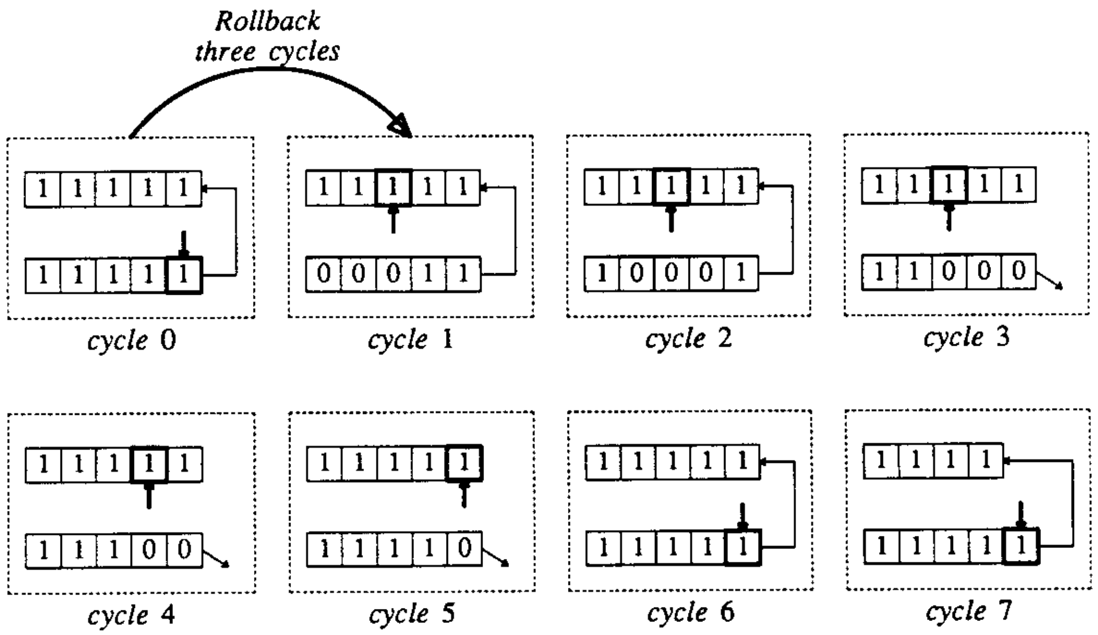


Figure 6.14: Contents of the load monitor before and after a three cycle rollback.

register is sent to the processor. After *n* "0s" reach the rightmost position, the pointer is back to its original position. Figure 6.14 shows an example of a three cycle rollback occurring after several pipeline accesses have been issued. The state of the shift registers after the rollback is shown in cycle 1. For the next 3 cycles (cycles 1, 2, and 3), data from the replay memory is re-sent to the processor. During cycle 4 and 5 new data that have just entered the replay memory are sent to the processor. Cycle 6 and 7 show the normal situation where the data arrives directly from memory and is sent at once to the processor.

In-Order Variable Latency Loads:

In the previous scheme, the length of the load monitor was dictated by the fixed latency of the *loads*. Valid signals stored in the load monitor reached the end of the shift register precisely at the same time as the data from memory arrived. For asynchronous protocols, *loads* can have different latencies. Since the width of the load monitor must accommodate various load latencies, the bit that corresponds to a load response is not necessarily the rightmost bit in the load monitor. Special circuitry must be provided so that the rightmost <u>valid</u> bit in the load monitor (instead of the rightmost bit) is used to control the destination of the data coming from memory. A load monitor of width *W* will allow up to *W* consecutive pending *loads*, assuming that the *loads* were issued in consecutive cycles.

The circuitry shown in Figure 6.15 allows pipelined accesses with variable latencies to be rolled back. During normal operation, a ''1'' is shifted in both the load monitor and the discard register (DR) if a *load* is present. Otherwise, a ''0'' is shifted in. Without any rollbacks, the discard register has the same contents as the load monitor (when the contents of a location in the load monitor is ''0'', the corresponding bit in the DR is never accessed, in order to simplify controls it is also set to ''0''). Bits in the load monitor indicate when load requests have arrived. Bits in the discard register reflects the validity of those entries based on rollback signals. When a rollback occurs, the *n* leftmost entries in the discard register are set to ''0''. When data comes back from memory, along with a *data ready* signal, the priority circuit identifies the location of the rightmost valid bit in the load monitor. That entry is then used to select the corresponding entry in the discard register. Data is either routed to the processor (discard register entry = 1) or

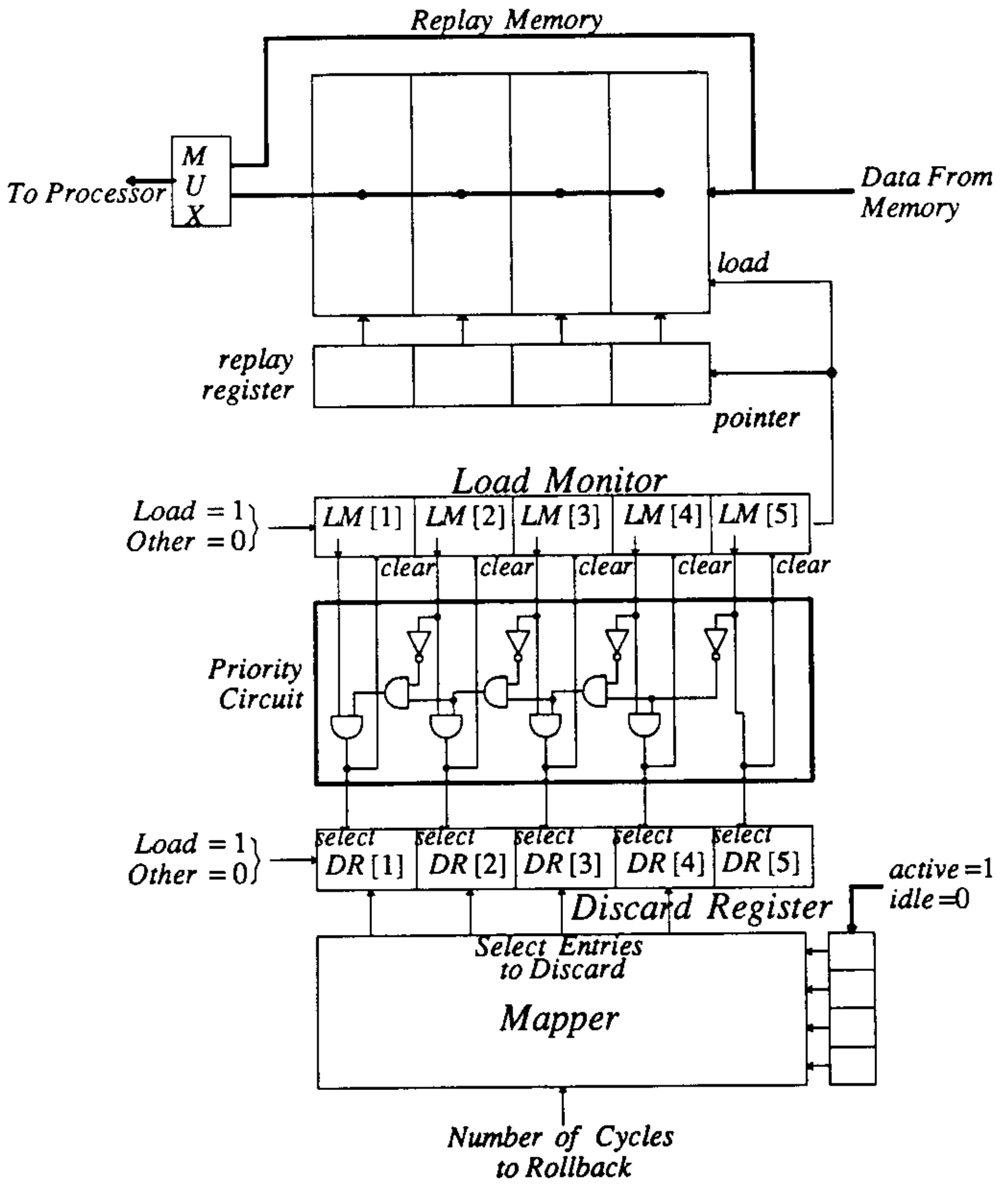


Figure 6.15: An RDIU for pipelined memory with in-order variable latency loads.

discarded (entry = 0). The rightmost valid bit in the load monitor is reset to "0" (*clear* signal in Figure 6.15) using a delayed version of the signal used to select a DR entry. If a "1" reaches the end of the load monitor and the request has not yet been provided by the memory, the load monitor stops shifting. Every cycle the contents of the load monitor move closer to commit time regardless of whether or not they shift. This is acknowledged by shifting a "0" into the shift register of a mapper similar to the one described in Section 6.3.1. The replay memory is handled similarly to the one used for the fixed latency pipelined memory.

Out-of-Order Loads:

Some memory systems allow requests from a processor to be processed out of order. In such systems, a tag is associated with each data item so that when the request is returned by the memory system, the processor is able to identify where the data belongs. An RDIU for such a system is represented in Figure 6.16. The only difference with the scheme for variable latency loads concerns the matching of the incoming data with entries in the load monitor. When a response to a *load* arrives from the memory, the address tag is compared with entries in the RDIU. If a match is found, the data is routed/discarded depending on the corresponding value in the discard register.

A straightforward implementation of the RDIU shown in Figure 6.16 with a data bus and address bus of 32 bits, would lead to a long narrow "band" of circuitry. In order to obtain better proportions, the CAM part of the RDIU is "folded" in half, reducing the height-to-width ratio of the full RDIU to around seven (from around fourteen). Folding is accomplished without adding a layer for routing buses over the circuitry since the address bus must go through both the CAM and the replay memory (Figure 6.17). The dimensions of the components appearing on the layout of Figure 6.17 are mostly based on cells used for designs

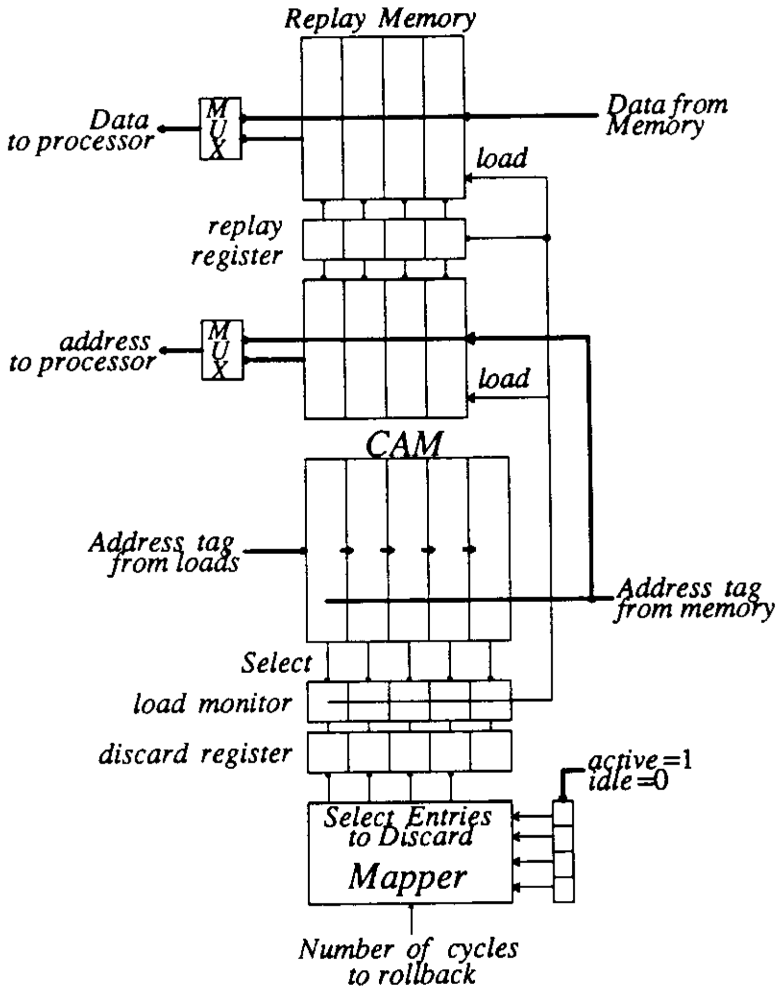


Figure 6.16: RDIU for out-of-order loads. The multiplexors (*MUX*) select the input coming directly from memory during normal operation or the input coming from the replay memory following a rollback.

described in details in Chapter 4.

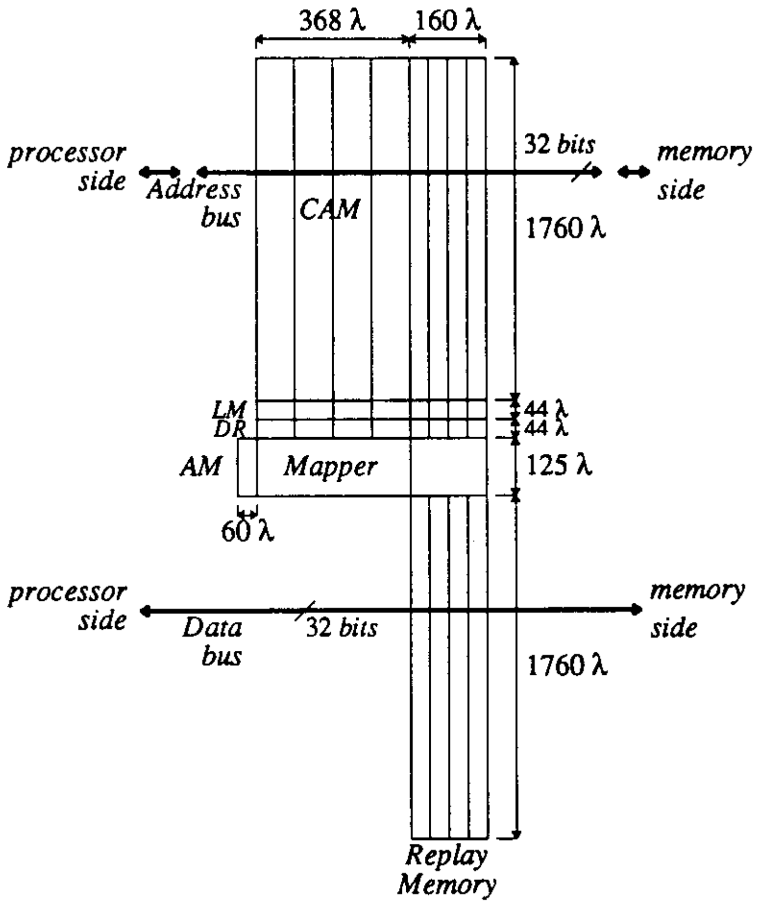


Figure 6.17: Floorplan of an RDIU for out-of-order loads. Some of the components are drawn larger in order to make the figure more readable.

6.3.5. General RDIU for a Processor-Memory System

We showed in the preceding section, that high-performance can be maintained in a system with modules inside and outside of the micro rollback domain only if the RDIU is tailored to the protocol used in that system. Through different designs we demonstrated that by adding features to the RDIU it was possible to operate with only a slight loss of performance. To better understand the function of a complex RDIU and to generalize its use in a processor — memory system, we show in Figure 6.18, a block diagram of an RDIU that can handle simple requests, burst accesses, pipelined accesses, and out-of-order requests. Each individual feature presented in the preceding schemes is included.

6.4. Loosely-Coupled Coprocessors and Micro Rollback

In this section we examine how off-the-shelf loosely-coupled coprocessors can be inserted into a rollback system. The term loosely-coupled is used to describe coprocessors that depend on the processor for initially receiving instructions and operands, but are capable of operating independently from the processor once execution has started. Generally, loosely-coupled coprocessors do not have direct access to the instruction bus, instead the processor assumes the task of fetching, decoding and sending related instructions to the coprocessor. Memory operands are also the responsibility of the processor and these are transferred to the coprocessor through a form of handshaking. The pipeline of a loosely-coupled coprocessor is completely independent from the processor pipeline so that once a coprocessor instruction is started, complete parallelism can occur between the two units. Generally, exceptions for loosely coupled coprocessors are not reported to the processor at the moment they occur, but only when another coprocessor instruction is executed, this is referred to as a *deferred trap* [Sun91].

As a concrete example, we look at the Motorola 68881 [Moto85a] (the Intel 80387 is just as representative). Even though loosely-coupled coprocessors do not represent the current state-of-the art in coprocessor interfaces, they provide a good example of how an RDIU can be inserted in a system so that modules inside and outside the micro rollback domain can interact. In the next section we will discuss

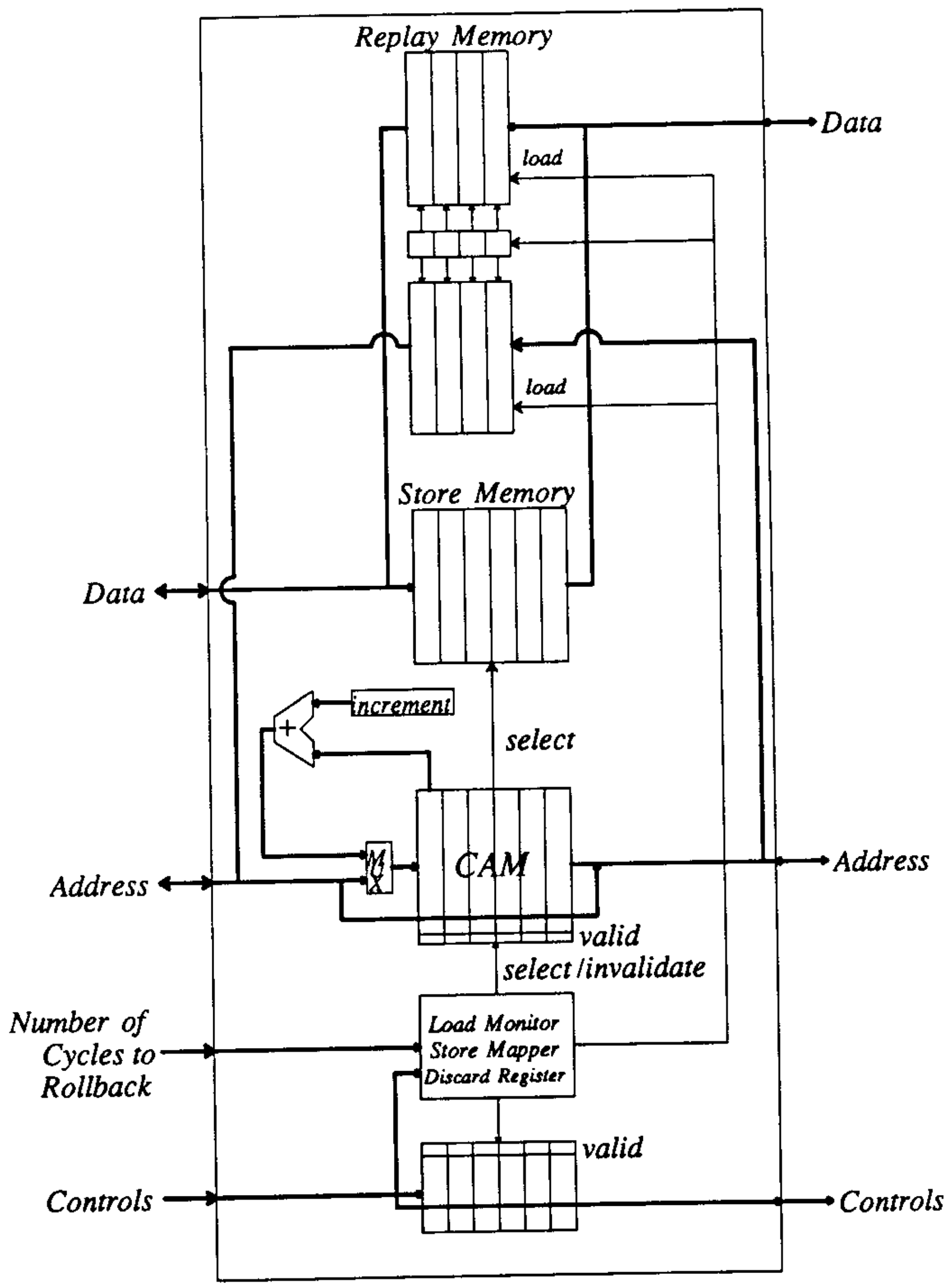


Figure 6.18: Block diagram of a complex RDIU handling simple requests, burst accesses, pipelined accesses, and out-of-order requests.

tightly-coupled coprocessors, such as the MIPS R3010 [Kane87], which are more representative of current high performance coprocessors.

The Motorola 68881 is a "passive" coprocessor in the sense that all floating-point instructions and their operands are fetched by the main processor. The execution part of the instructions is completely done by the 68881 and is independent from the execution of "normal" processor instructions. This allows almost complete overlap between the two processing units once the coprocessor instruction is started. An interesting feature of the 68881 is the asynchronous protocol used with the processor, most likely a Motorola 68020, which allows different clock frequencies to be used for the two modules.

In the context of micro rollback, controls and data cannot be sent directly from the processor to the coprocessor since they could change the state of the coprocessor and upon rollback of the processor, could not be restored to a correct state. An RDIU is thus introduced in the communication path between the processor and the coprocessor (hereafter referred to as P and C). A delay buffer RDIU such as the one described in Section 6.3.1 and shown in Figure 6.6 can be used. All information flowing from P to C including floating-point instructions, data operands, and controls signals, is delayed by N cycles. In this way, once an instruction exits the FIFO of the RDIU, it is committed and can be executed "safely" by the coprocessor. For instance, if the main processor fetches a floating-point ADD, it is sent immediately to the RDIU where it is delayed for N cycles and then sent to the coprocessor where it is executed at once.

The approach described above is general and does not consider the semantics of the coprocessor instructions. By looking at the exchanges occurring between P

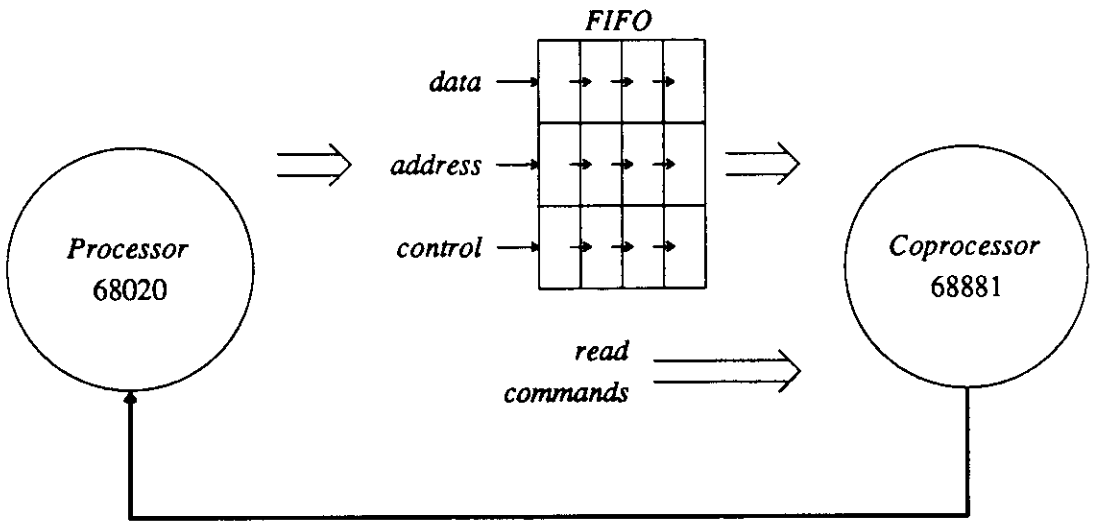


Figure 6.19: The N-Deep FIFO delays data, addresses, and control signals send by the processor to the coprocessor.

and C, during coprocessors instructions it is possible to exploit the fact that some commands sent by P to C do not cause state changes in C and can bypass the delay buffer. The commands sent by P that do not change the state of C are mostly request to read one of the coprocessor's interface registers (CIR). For example, during all floating-point arithmetic instructions and during floating-point register moves, the 68881, after receiving the initial controls, will request the current program counter (PC) from the processor. The request by C is made by setting a bit in one of its control register. Soon after sending the initial controls, P reads the *response register* on C and detects that the PC must be transferred. The latter command (read of the response register), does not modify the state of C and can be issued directly after the initial commands. A modified delay buffer, which routes *read* requests directly to C without delaying them, constitute an efficient RDIU which improves performance in respect to a pure delay buffer (Figure 6.19). *Read* commands sent directly to the coprocessor are processed sequentially (one at the time) and are not pipelined. Complicated transfers from C to P are decomposed into several one-cycle read commands. Thus, a replay memory is not needed since any processor rollback that ''undoes'' a read command, would request that read command again when processing resumes. If a coprocessor-processor system has multiple cycle read commands, the RDIU must reject any ''unwanted'' data, similarly to the RDIU described for the pipelined memory in Section 6.3.4.

The insertion of the FIFO in the communication path between P and C, introduces a delay of N cycles for each state changing interaction. This penalty translates directly into a system overhead of N cycles times the number of interactions that each coprocessor instruction has. The possible overlap between the coprocessor instructions and the main processor instructions occurs only after the *execution* phase of a coprocessor instruction has begun. It is important to keep in mind that floating-point operations executed by the MC68881, take anywhere from 30 to more that 1000 cycles to be completed. Thus an overhead of 4, 8 or 12 cycles per floating-pint instruction may not be that significant.

6.5. Tightly Coupled Coprocessors

Coprocessors referred to as ''tightly coupled coprocessors'' are added into a system by forming a seamless integration of their instruction set with the processor. By duplicating some of the pipeline stages, such as instruction decode, tightly coupled coprocessors are able to access instructions and data much faster than loosely coupled coprocessors. The handshaking protocol occurring between a processor and a loosely coupled coprocessor can be eliminated resulting in levels of performance at least an order of magnitude greater than the loosely-coupled coprocessors described in the previous section [Rowe88]. On the other hand, their greater interaction with the processor renders the task of integrating them into a micro rollback system more challenging. As a concrete example we look at a system composed of a version of the MIPS R3000 RISC processor which can roll back (refer to the R3000mr hereafter), and the standard MIPS R3010 floating-point accelerator [Kane87], which cannot rollback. The R3010 is representative of tightly coupled coprocessors such as the IBM RS/6000 [Groh90] which receives instructions directly from the instruction cache and decodes them separately, and the Intel i860 [Sit89] which decodes floating-point instructions in parallel with integer instructions.

The close coupling between the R3000 and R3010 as well as the visibility of the FPA pipeline to the processor, make the R3010 a "complicated" candidate which led us to investigate several alternative implementations of RDIUs. Even though we focus on the MIPS R3010, the ideas presented in this section are general and can be applied to other coprocessors.

6.5.1. The R3010 Coprocessor Operation

The instruction pipeline of the R3010 parallels the one of the R3000mr main processor. The floating-point accelerator (FPA) continuously monitors the instruction stream. If a floating-point instruction is detected by the FPA, it enters the 6-stage pipeline, and transfers data (and exception signals) synchronously with the processor. Otherwise, e.g. during normal instructions, the FPA simply ignores the instruction. Independently of the incoming instruction, the FPA pipeline advances in order to achieve synchronization with the R3000mr.

Uncommitted instructions cannot be sent directly to the R3010 since they could cause modifications of the internal state from which the R3010 could not recover. The same applies to data items that are moved to the FPA (floating-point loads from memory and floating-point moves from the CPU to the FPA). Some buffering between the instruction stream, the data bus, and the R3010 must be present so that only committed instructions and committed data reach the coprocessor.

6.5.2. Simple Solution: Intercept Individual FPA Instructions

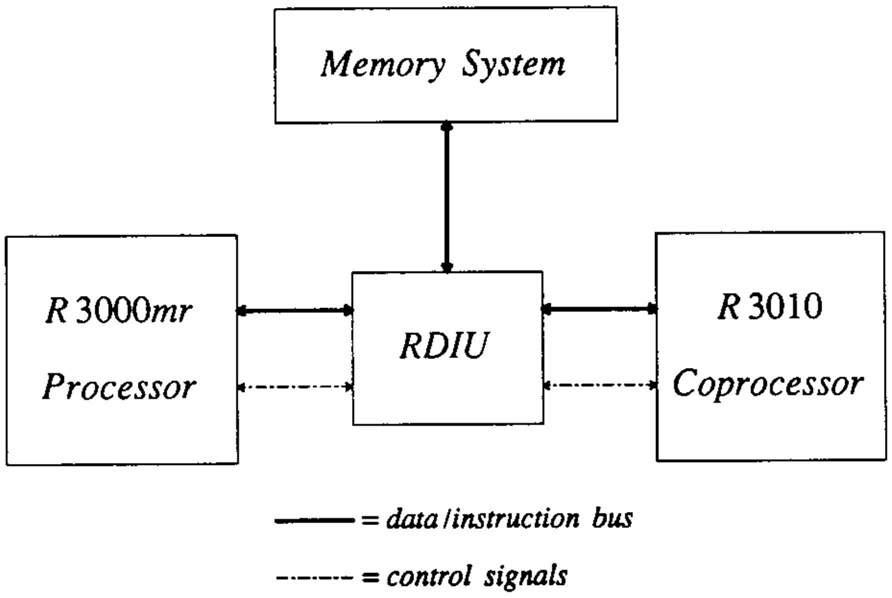


Figure 6.20: The RDIU intercepts all incoming instructions before they reach the R3000 and R3010.

The first RDIU design that we propose for the R3000mr-R3010 system consists of a module that intercepts incoming instructions before they reach the processor and the coprocessor (Figure 6.20).

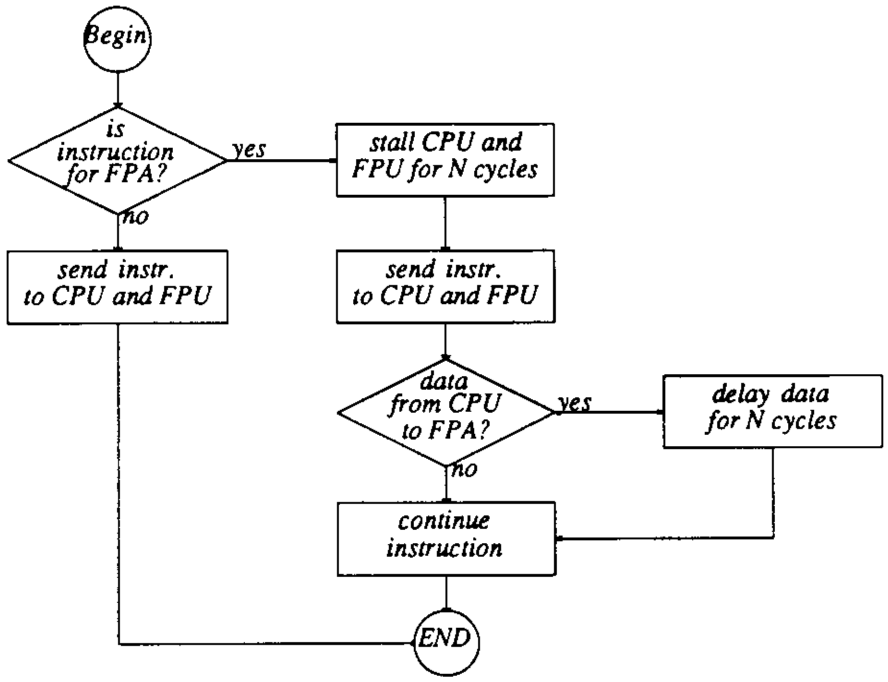


Figure 6.21: Role of the RDIU in the simple "intercept FPA instructions" solution.

The RDIU can determine quickly if the instruction requires FPA intervention by decoding the three most significant bits of the incoming instruction (I_{31}, I_{30}, I_{29}). This can be implemented with three 2-input NAND gates $(I_{31}' I_{30} + I_{30}' I_{29})$. If the instruction is a normal instruction, it is sent immediately to the R3000mr and the R3010. The R3000mr proceeds with the execution of the instruction while the R3010 simply discards it. If the instruction is an FPA instruction, the CPU and the FPA are stalled for N cycles, where N represents the rollback *range*. If data is to be exchanged from the CPU to the FPA, it must also be delayed until it becomes committed. This is accomplished by buffering data through the RDIU. The flowchart in Figure 6.21 describes this process.

Since only committed instructions and data enter the FPA, there is no need to

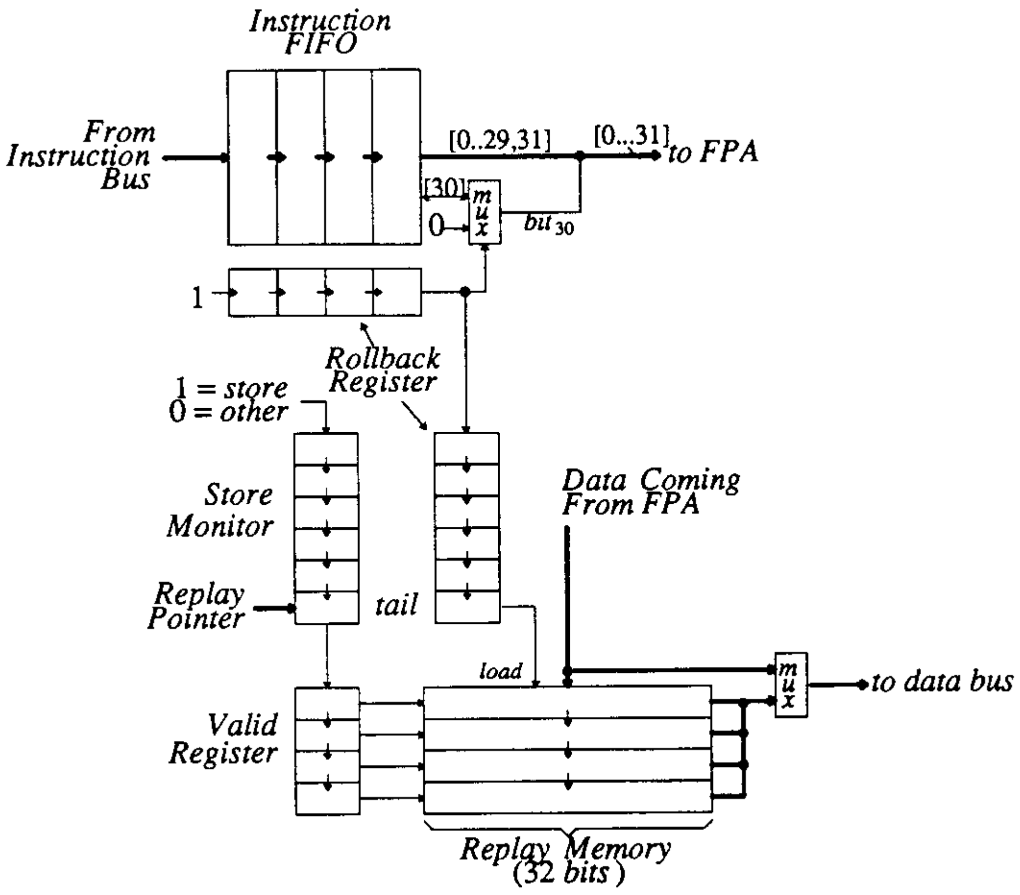


Figure 6.22: Circuitry of the RDIU replaying the previous data movements coming out of the R3010.

"undo" operations in the R3010. The state of the R3010 does not need to be modified upon receiving a rollback signal (this is a requirement for this system since we are using a standard R3010). If data movements from the FPA to the CPU or to memory (floating-point *moves* and *stores*) occurred during the past cycles covered by the rollback distance, there is a need to replay the data produced by the FPA.

The circuitry shown in Figure 6.22 supports both the delaying of information from the instruction memory to the FPA and the replaying of data sent in the last four cycles by the FPA. The rollback register is used to keep track of which instructions are valid. It is composed of two parts: the first one has four entries, (based on the rollback range) and is attached to the instruction FIFO, the second part has six entries corresponding to the number of pipeline stages in the R3010. During normal execution, the rollback register is filled with ones. When a rollback occurs, the last n entries, where n is the rollback distance, are invalidated (set to zero). The instruction FIFO delays all instructions coming from the instruction bus for N cycles. The instruction is then either sent to the FPA or a non-FPA instruction is sent if a rollback canceled this instruction. This is accomplished by replacing bit 30 of the instruction by a ''0'' which indicates a non-FPA instruction (based on the contents of the rollback register). A non-FPA instruction is ignored by the R3010.

The store monitor has six entries, each one corresponds to a stage in the pipeline of the R3010. A ''1'' is stored in the store monitor when the RDIU detects that the instruction being sent to the FPA will cause data to come out of the FPA. The *replay memory* is a small FIFO that stores any data that comes out of the FPA for N cycles (in this case $N = 4$). If a rollback of n cycles occurs, the *replay pointer*, located at the tail of the store monitor, is moved n positions into the *valid register* (associated with the replay memory), thus pointing to data that was produced n cycles ago. The data item pointed to by the replay pointer is put onto the data bus and sent out to the CPU. The rest of the data that were produced by the FPA are replayed one by one by letting the FPA execute and by continuously shifting the store monitor, the replay memory, and the valid register (and the rollback register). The replay pointer is brought back to its ''normal'' position (the tail of the store monitor) one position at the time when ''invalid'' entries appear at the tail of the rollback register. During that time the replay memory does not shift.

Figure 6.23 shows the contents of the rollback register, the store monitor, and the valid register before and after a rollback of three cycles. In this figure we show a general case where several consecutive floating-point stores are pipelined one after the other (one per cycle).

The initial state of the FPA is shown in *cycle 0* in Figure 6.23. The contents of the valid register (VR) shows that the FPA has sent out data for each of the past four cycles. The store monitor shows that the pipeline of the R3010 is filled with instructions generating data movement out of the R3010. The ones in the rollback register indicates that all entries are valid. After a rollback of three cycles, the first three entries in the rollback register are invalidated. The contents of the replay memory is accessed by moving the replay pointer three positions into the FIFO (shown in *cycle 1* in Figure 6.23). During the next seven cycles (2,3,4,5,6,7, and 8), the pointer is kept at the same position while the R3010 pipeline keeps producing committed results which are shifted in the replay memory. When a zero (rollback entry) reaches the tail of the rollback register, the output of the store monitor is discarded and the rollback pointer is moved up one position in the replay memory. This allows the FPA to catch up with the CPU by ''squashing'' the rollback cycles one by one. The last invalid entry in the rollback register, shown in *cycle 10* in Figure 6.23, causes the replay pointer to move back to the tail of the store monitor, which is its assigned position during normal execution. In this way rollback has been accomplished in a transparent way for both the CPU and the FPA.

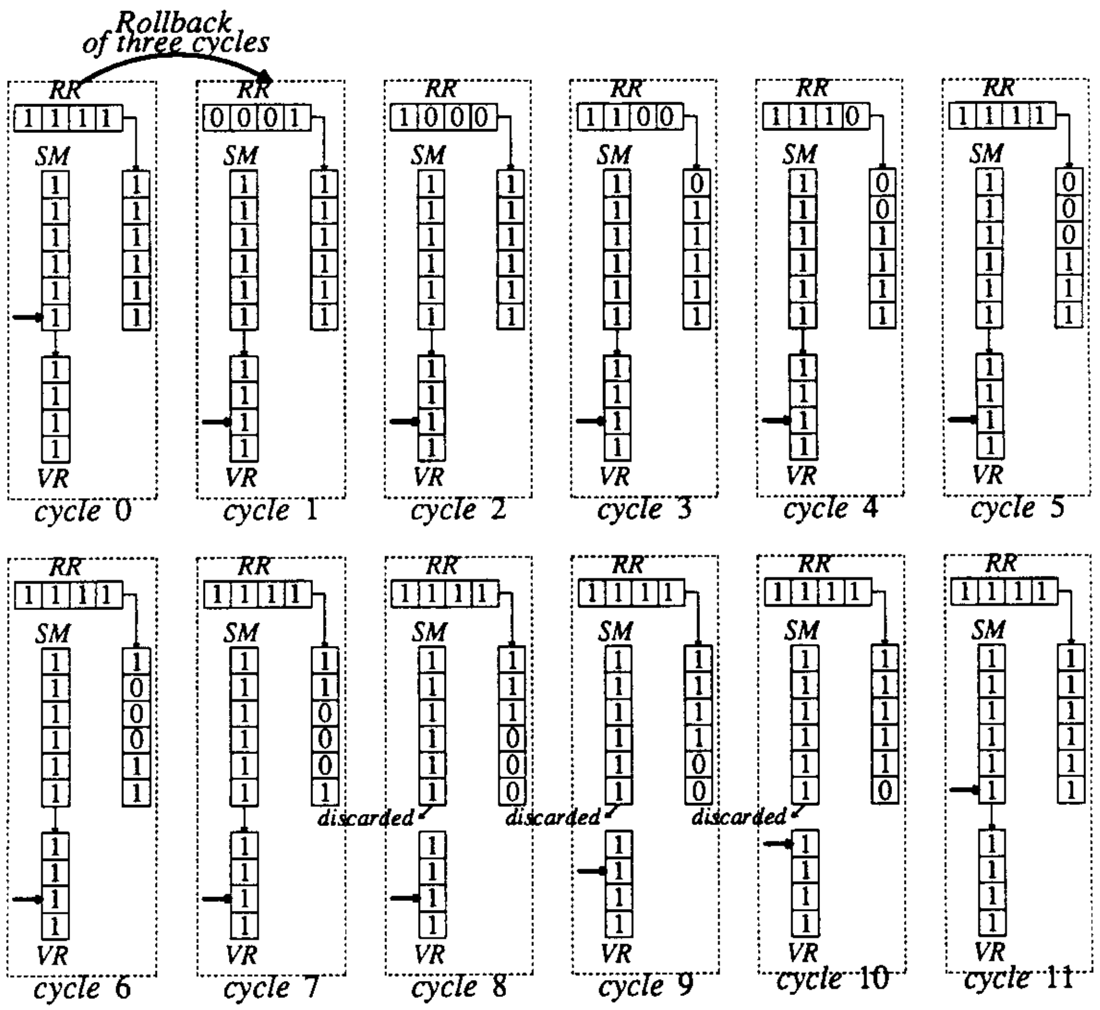


Figure 6.23: Contents of the rollback register, the store monitor, and the valid register following a rollback. We assume that several floating-point stores can be pipelined consecutively.

The performance degradation due to the insertion of the RDIU in the system is reflected by two components. First, the processor cycle is slowed down since an instruction fetch must now go through the RDIU. The operations performed by the RDIU during normal instruction fetches are minimal (2 levels of gates), but once added to the time it takes for an instruction to go through the pins, it becomes non negligible. This problem can be alleviated by sending instructions directly to the

CPU and generating a *synchronization rollback* of one cycle if the instruction is later identified to be an FPA instruction. We will discuss this method in more details in Section 6.5.2. The second source of performance degradation is due to the N-cycle delay added to each FPA instruction. Moreover, the instructions generating a movement of data from the CPU or memory to the FPA must account for another N-cycle delay. This overhead can be described as follow:

$$overhead = P \times ((f_{fpa} \times N) + (f_{fpa} \times f_{dmi} \times N))$$

Where the overhead is the total number of extra cycles required to run the process. P is the total number of instructions executed for a process, f_{fpa} is the fraction of floating-point instructions, f_{dmi} is the fraction of floating-point operations causing data movement into the R3010. An example of the overhead can be obtained from numbers published in [Cmel91]. For the program SPICE, $P = 21569$, $f_{fpa} = 18\%$, and $f_{dmi} = 50\%$, which gives an overhead of 23295 cycles for $N = 4$, or around 86% considering a cpi (cycles per instruction) of 1.25 (the program runs almost twice as slow).

The components required to implement the RDIU are similar to the ones described in previous sections (FIFO, Mapper, etc). The only constraint on the design regards the speed at which an instruction can enter the RDIU, be identified as a normal instruction and be sent out to the CPU. Unless the synchronization rollback method is used, this delay is added to the processor cycle time since fetching a word from the instruction cache represents a critical path. The rest of the logic does not interfere with normal operation and is only active during rollback. Operations such as moving pointers, shifting FIFOs, and mapping an input signal are simple and easily fit in the full clock period allowed for them to execute.

The small amount of logic necessary to implement the RDIU as well as "loose" timing restrictions for the rollback circuitry, make this chip a good candidate for semi-custom or gate array implementation.

6.5.3. More Efficient Solution: Non-damaging FPA Instructions Sent Directly

In this section we present an improvement over the previous scheme, which is achieved at the expense of a adding a small amount of circuitry in the RDIU.

Of the seventeen FPA instructions, five do not modify the state of the R3010 upon completion. These five instructions include *moves* from the FPA (data registers and control register), *stores* from the FPA, and *branches* based on a previously calculated condition. Even though the branches have an impact on the CPU and on the flow of instructions (which can be rolled back), they do not affect the state of the FPA. Those five *non-damaging* FPA instructions, can be issued directly to the FPA without waiting until they become committed. The flowchart reflecting this change is shown in Figure 6.24.

The flowchart is similar to the one shown in Figure 6.21, except that further decoding is done to find if the instruction will modify the state of the FPA. The extra hardware required for implementing this scheme consists of simple logic decoding five bits from the opcode of incoming instructions. This logic is comparable to the logic used to detect floating-point instructions, and can be implemented with two-level logic.

The performance of this scheme is improved since non-damaging FPA instructions are not delayed. The overhead becomes:

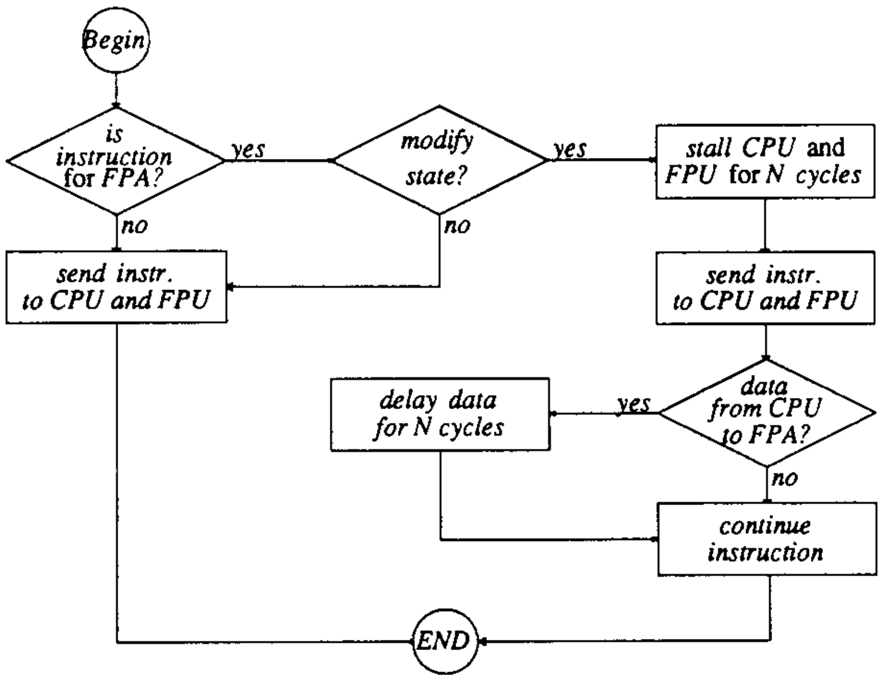


Figure 6.24: Flowchart representing an optimization for instructions that do not modify the state of the FPA.

$$overhead = P \times (((f_{fpa} \times (1 - f_{nd}) \times N) + (f_{fpa} \times f_{dmi} \times N))$$

where f_{nd} represents the percentage of floating-point instructions that do not cause a change of state upon completion in the FPA.

For the same example (SPICE) we obtain $f_{nd} = 22\%$ which lead to a reduced overhead of around 74% (vs. 86% for the simple scheme).

6.5.4. Solution for High-performance: Instruction Prefetch Buffer

The close interaction between the R3000mr and the R3010 requires that upon the detection of a state-modifying FPA instruction both units be stalled for N cycles. When the CPU is stalled, the instruction fetching mechanism is disabled and the flow of incoming instructions stops until execution resumes. When a new instruction is fetched, the same procedure occurs, i.e. the instruction is partially decoded to see if it is an FPA instruction, in which case it is delayed for N cycles. For consecutive FPA instructions this process can be pipelined so that several FPA instructions can be waiting sequentially until they become committed.

The idea of pipelining the ''waiting time'' of incoming FPA instructions requires a continuous flow of fetched instructions. Since this cannot be accomplished by the CPU (which is stalled from the moment an FPA instruction is detected), a simple instruction prefetch buffer (IPB) is inserted in the system. The IPB is used mainly to continue the fetching of instructions even when the CPU is stalled due to the detection of an FPA instruction. If a sequence of FPA instructions is fetched and queued in a pipelined manner in the IPB, the overhead of N cycles will be incurred for only the first FPA instruction. This method can lead to a significant gain in performance if instructions can be fetched correctly without processor intervention.

In order to keep the IPB simple (no need to redesign the whole CPU), subsequent instructions can be assumed to be the sequential continuation of the current instruction stream. When the delayed instructions are released to the CPU and FPA, a different flow of instructions may result since previous conditions were not available when the instruction fetches occur. In that case, instructions in the prefetch buffer have to be invalidated and a new instruction stream needs to be fetched. The sequence of operations controlling the flow of an incoming instruction is shown in Figure 6.25.

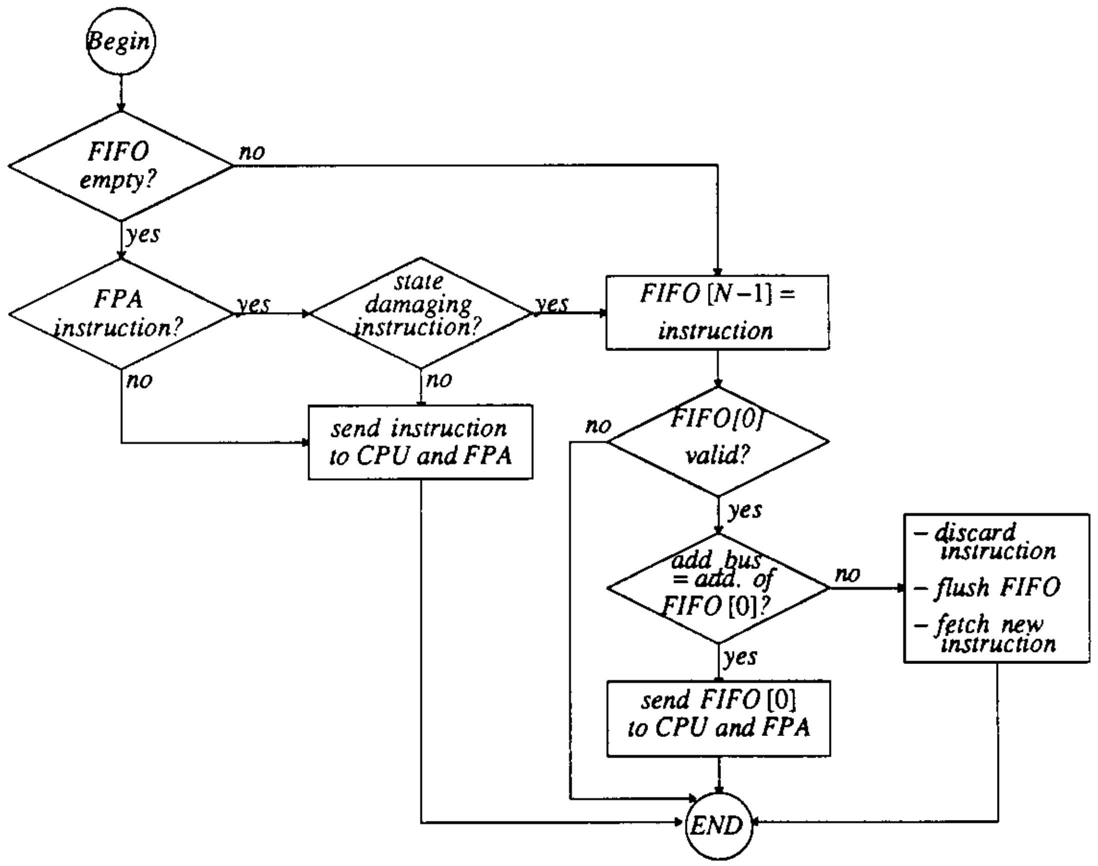


Figure 6.25: Flowchart of the progress of an incoming instruction. A continuous flow of instructions is assumed.

The circuitry required to replay the data sent out by the R3010 upon rollback is similar to the one described in figure 6.22. When a rollback of n cycles occurs, the last n instructions in the FIFO are invalidated. Entries in the rollback register are also invalidated appropriately. The replay memory is accessed so that previous results are sent back to the CPU. The CPU itself is rolled back.

The improvement in performance due to this method occur when at least two floating-point instructions appear consecutively in the code. In the best case, a sequence of several FPA instructions would execute just as fast as a normal system without rollback, except for the startup delay of N cycles. Adjustment to the compiler should be made in order to favor grouping of FPA instructions so that the overhead becomes minimal. A dynamic measurement of the instruction flow would be required to obtain accurate numbers representing the performance gain over the previous method.

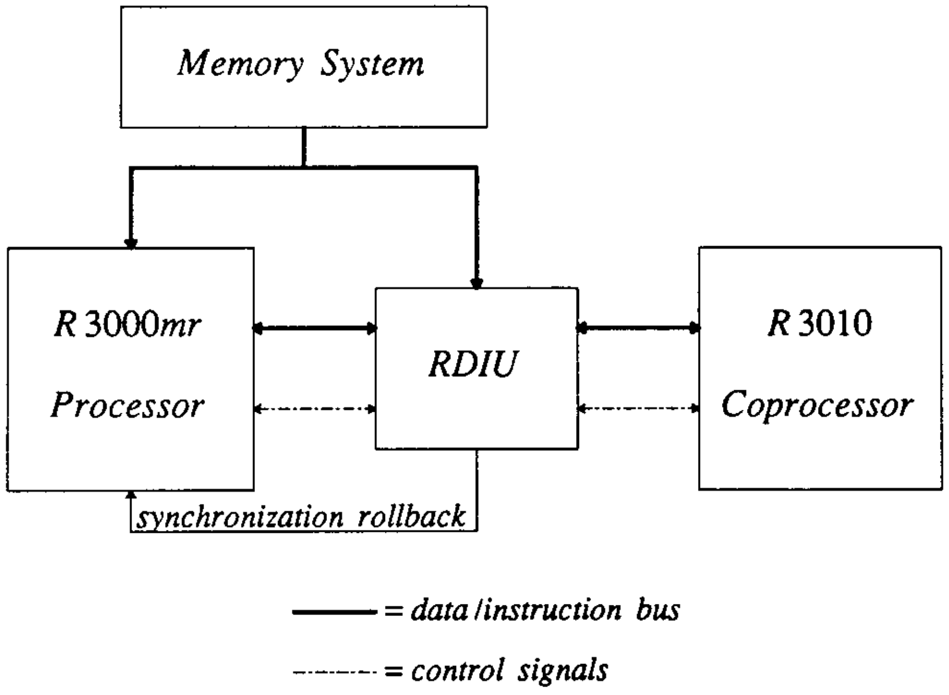


Figure 6.26: By removing the RDIU from the instruction fetch path, it is possible to maintain the same processor cycle time. A "synchronization" rollback is necessary to synchronize the CPU and FPA upon the detection of an FPA instruction.

It is possible to improve the protocol so that instructions can be sent directly to the R3000mr without having to go through the RDIU. This execution of instructions becomes speculative in the sense that it is guessed that these instructions are not of the FPA type. In the case of normal R3000 instructions, the instructions reach the CPU without added delay and are executed just like in a system without micro rollback (Figure 6.26). For FPA instructions we let the processor execute for a few cycles even after the RDIU has detected that the instruction which just entered the queue is an FPA instruction. In this way instruction fetch continues (controlled by the R3000) without the intervention of the RDIU. If control signals are normally exchanged between the processor and the FPA when an instruction goes through the first few pipeline stages, the RDIU must be capable of duplicating these same signals. In this way the processor will proceed as if no RDIU is present. This may lead to an increase in the complexity of the RDIU. The capability to send a rollback signal of N cycles to the CPU must be added to the RDIU. This rollback signal differs from a normal rollback signal in the sense that it does not affect the FIFO of the RDIU since its purpose is just for synchronization and not due to error detection. The other advantage due to this method is the better utilization of the instruction fetching unit of the R3000mr. By letting the CPU execute and pursue instruction fetching, it is possible to eliminate the fetching logic previously required for the RDIU.

6.6. Conclusion

The main purpose of this chapter was to demonstrate the capability, through the use of a rollback domain interface unit (RDIU), of integrating off-the shelf modules into a micro rollback system. This added ''feature'' allows a system to take advantage of standard high-performance modules that are developed for applications or environments different than the ones micro rollback is targeted for.

We have concentrated on modules which interact directly and frequently with a processor capable of micro rollback (of the kind described in Chapter 4), namely memory subsystems and floating-point coprocessors.

The memory subsystem, a dedicated high-performance cache, was shown to be a good example of a module which can be integrated in the system without high cost and without significant performance degradation. A single-cycle cache memory requires hardware similar to a large register file so that recent changes can be held up until they become committed. The problem of collision between *load* and previous *stores*, nonexistent for register files, but unavoidable without dual port caches, was solved by adding a Mapper which lets data ''age'' in a delay buffer.

We have described an RDIU for a complex memory subsystem capable of burst accesses, pipeline accesses, and out-of-order requests. The key idea is to keep enough information in the RDIU to allow quick recovery upon the reception of a rollback signal. Buffering of damaging transaction is also required but did not cause trouble because of the very nature of the memory subsystem (data is not processed, it is merely stored).

Coprocessors, loosely or tightly coupled, were shown to suffer some performance degradation due to the delay that must be imposed to state-damaging instructions before they reach the coprocessors. Nonetheless, it was shown that the amount of hardware required to implement an RDIU for these coprocessors was small and would allow full use of the coprocessors' instructions. The introduction of floating-point coprocessors led to a gain of up to two order of magnitudes [Inte87] in conventional systems which makes them very attractive for micro rollback system even when, in the worst case, a penalty of N cycles is added to each exchange between the processor and the coprocessor.

Our choice of complex modules with complex interface protocol, has led to the development of general techniques that we believe can be applied to a variety of off-the-shelves modules. Through the use of RDIUs, the applicability of micro rollback is extended beyond custom designed modules and allows a broader range of applications to benefit from parallel error detection.

Chapter Seven

Combination of Micro Rollback with Speed-Up Techniques for Uniprocessors

In previous chapters we have shown how micro rollback can be used to minimize error detection overhead in terms of area and delays. Serial delays are eliminated through the use of parallel error detection. Area overhead is reduced by having the possibility to use slower (longer latency), smaller modules. In this chapter we take our approach one step further. We integrate micro rollback with speed-up techniques developed for increasing the performance uniprocessors.

In high performance processors, it is desirable for normal operation to be based on the optimistic assumption that rare events do not occur. This leads to the elimination of delays that are introduced by checking for those events in conventional designs. For example, arithmetic exceptions are not expected to occur at a high frequency in normal programs. Careful design of the pipeline can remove the delay caused by checking for exception so that results in normal operation are produced more quickly. Upon the detection of an arithmetic exception, the program can be stopped and brought back to a point before the exception-causing instruction so that proper handling can proceed (typically, a trap is triggered) [Smit88, Hwu87]. The removal of exception logic from normal execution may allow the clock cycle to be shortened or pipeline stages to be eliminated. If exceptions do not occur often, significant gains in throughput can thus be achieved.

There are fundamental similarities between the handling of errors due to hardware faults and the handling of events such as arithmetic exceptions, page faults, cache misses, and TLB misses. All of these events are infrequent and require the hardware to deviate from its normal operation (e.g. fetch a cache block or jump to a trap handling routine). If the hardware is designed to optimize performance for the normal case, the handling of the rare events may require undoing state changes that were incorrectly performed. Undoing incorrect state changes is precisely the task of micro rollback in handling errors caused by hardware faults. Thus, efficient high performance implementations can be achieved by combining the hardware for micro rollback and for handling the other rare events.

In order to achieve high performance, processors rely on techniques such as register renaming [Logr72, Kell75], out-of-order issuing, execution, and completion of instructions [Smit88], and speculative execution of instructions [Hwu87, Smit90]. Register renaming, as we will describe in the next section, relies on more than one instance of a data register to eliminate dependency between instructions. The extra registers needed for register renaming can be used for micro rollback, thus leading to an efficient implementation of both techniques. Out-of-order execution and speculative execution of an instruction stream (beyond a conditional branch) requires dedicated hardware to support precise interrupts and branch repair. In Section 7.2 and 7.3, we describe how micro rollback can be combined with this hardware to form a *standby reorder buffer* which is a flexible mechanism for delaying commitment of changes to the processor state.

7.1. Micro Rollback and Register Renaming

In order to avoid register storage conflicts in a processor with several functional units capable of producing out-or-order results, *register renaming* can be used [Logr72, Kell75]. An instruction can be issued even though its target register would normally overwrite the source register of an instruction issued previously but which hasn't used the register yet. For example, in the following code:

```
div     r3,r2,r1
add     r5,r4,r3
load    r3,&mem_address
```

The `div` instruction is issued and takes several cycles to produce a result. The `add` instruction is also issued but cannot execute since one of its source register is not ready (`r3`). The `load` instruction could normally not be issued since it would overwrite the content of `r3` before it has been used by the `add` instruction (write after read (WAR) hazard [Kell75]). However using register renaming the logical register `r3` associated with the `load` is renamed to a different physical register. In this way the load can proceed, loading new data into a physical register which will then be available for subsequent instructions accessing `r3`. In the meantime the previous physical register containing the data referred to by `r3` is kept intact so that it can be used by the `add` instruction as soon as the `divide` instruction terminates. After the source `r3` has been used by the `add`, the physical register representing the previous `r3` can be released so that it can in turn be renamed as a physical register for another logical register. In a similar way, register renaming can be used to eliminate write after write (WAW) hazards.

In some cases the actions associated with register renaming can be accomplished by an optimizing compiler. One of the limitation of static register allocation and scheduling (done at compile time) is tied to the fact that the latency of the functional units may be variable. For example: (a) an early-exit multiplier producing results with different latencies depending on the operands, (b) a load pipe with variable latencies due to cache hits/misses, and (c) a variable latency divider [Will91]. Moreover, as we shall see later, register renaming can be used to logically separate two functional units sharing a pool of registers, without having complex circuitry for keeping track of the current usage of the registers for both units. For example, even though the fixed-point unit of the IBM RS/6000 executes the floating-point loads, it is decoupled from the floating-point register file and from the registers currently being used by the FPU by using hardware register renaming. Static register allocation would be ineffective in this case since the units were purposely designed so that they could operate independently and offer the flexibility to operate ''out-of-synch'' at the instruction level (one unit can be ahead of the other one by a few cycles). Finally, hardware register renaming has the advantage (over a static scheme) of ''adding'' physical registers to an architecture without increasing the bit field allocated in the instruction word to specify the logical registers.

Hardware register renaming, for a register file of 32 registers, has been shown to improve performance by about 15%-21% compared to a scheme without renaming [John89]. However, it should be noted that those numbers were obtained with a compiler optimized to minimize memory traffic (minimizing the number of loads/stores) by assigning a maximum number of local variables to on-chip registers. Johnson mentions that for a processor with a large number of registers and/or with a compiler optimized to assign different instances of a local variable to different registers, the results would be closer to what is obtained through hardware register renaming. On the other hand, the basic architecture for which the numbers above were extracted has fixed latencies for all its functional units, thus reducing the advantage that hardware renaming has over the static method.

Register renaming allows parallel execution of instructions through the use of extra registers which keep ''old'' data temporarily until data are no more needed. This technique is similar to micro rollback where data are also kept temporarily until they are committed. To accomplish micro rollback, the register renaming technique can be modified so that the old contents of registers can be kept ''alive'' for a few cycles before being discarded. This efficient use of resources for adding micro rollback can contribute to a low performance overhead.

Since a full cycle is often dedicated for the register renaming [Groh90], there is a possibility for adding micro rollback to the processor without increasing the processor cycle. Furthermore, by using the existing hardware it is possible to add micro rollback with only a small area overhead.

7.1.1. Micro Rollback of a Simple Processor Using Register Renaming

In this section we describe how a simple processor with a single functional unit can be rolled back using register renaming. The advantages and disadvantages of this method will be compared to the delayed-write buffer method described in Chapter 4.

Description

Even though the following method is general, for the sake of clarity we describe the hardware required to roll back a register file of 64 registers from one to four cycles (rollback range $N = 4$). Four physical registers are added to the register file so that up to four temporary values can be kept at all times. Since a maximum of one write per cycle occurs, this is sufficient to hold temporary data for up to four cycles. These extra registers are used to save copies of registers that would normally be overwritten by uncommitted data (during a *write*).

Every cycle, logical register addresses (logical tags) are mapped to their most recent physical register assignment (physical tag). Source registers logical tags are renamed through a mapping table consisting of 64 rows of 7-bit tags (Figure 7.1). Since all logical tags go through the same process (accessing the mapping table), we consider all tags to be renamed even if a logical tag has the same "binary" value as its corresponding physical tag. This can occur for instance when a logical tag has kept the same assignment since initialization.. This is only dependent on initializing the mapping table with matching tags at power up and is not a requirement for this method to work.

The mapping table is implemented as a RAM with two decoders to provide the throughput necessary for two source operands to be renamed per cycle. In order to check for assignments resulting from previous instructions that are not yet committed, a small CAM (4-deep) is also scanned in parallel to see if there is a more recent assignment. If so, a match signal will select the correct assignment (Figure 7.1). The CAM with the priority circuit and the logic attached to the logical and physical tags is simply a delayed-write buffer (DWB) for the mapping table.

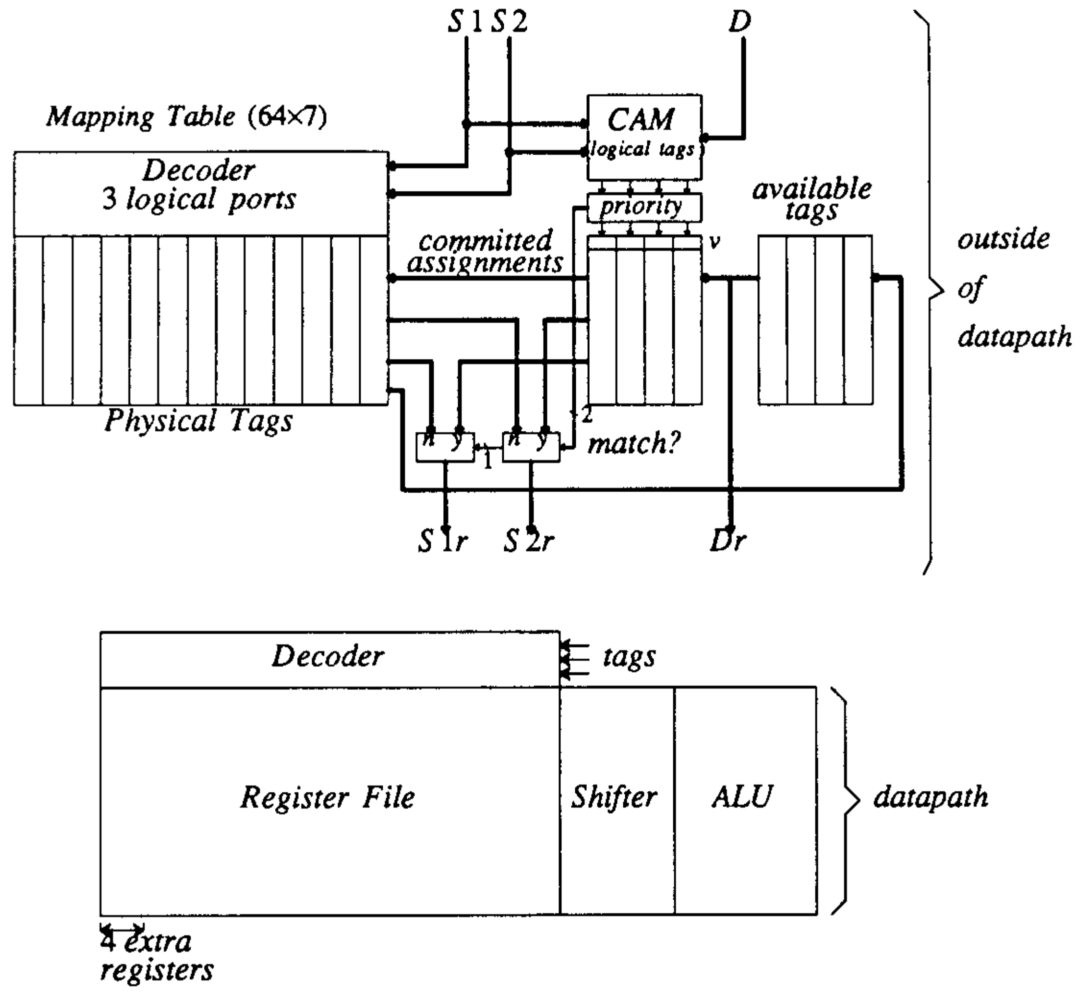


Figure 7.1: Micro rollback of a simple processor using register renaming. Tag logic is outside of the datapath. Four extra registers are added to the register file. The pitch of the datapath remains the same. The diagram is not drawn to scale.

A destination address causes a new renaming of a logical tag. That is accomplished by storing in the CAM the logical tag along with the first available physical tag. After four cycles, at which point the write associated with the destination register (e.g. during an ALU operation) becomes committed, the assignment is transferred to the mapping table. The old physical tag is returned to a list of available tags so that it can be reused. Since *reads* and *writes* occur during different phases of the processor cycle, the decoders for the mapping table can be shared by the source tags and the destination tags (similarly to the decoders in the register file).

The implementation described above contains 72 locations where tags can be stored. Since there are only 68 physical tags, there are always four unused locations. A better implementation can be made by eliminating the ''available tags FIFO'' shown in Figure 7.1. This can be accomplished by invalidating entries in the CAM upon initialization and by providing a feedback path from the output of the FIFO/CAM back to the input (Figure 7.2). A physical tag is always made available for a new assignment. Every cycle, depending on the valid bit of the leftmost entry in the FIFO, either (a) the bit is equal to ''1'': a write is made into the mapping table in which case the physical tag that used to correspond to the logical tag being remapped is made available, or (b) the bit is equal to ''0'': the physical tag corresponding to the leftmost entry in the FIFO is returned to the head of the FIFO through the feedback path. Invalid entries in the FIFO occur since a destination register is not necessarily modified every cycle.

A rollback consists of invalidating the last *n* tag assignments. The writes made during the last *n* cycles are thus disregarded and the data preserved in the register file is made available through the old tag assignment which is still in the mapping table.

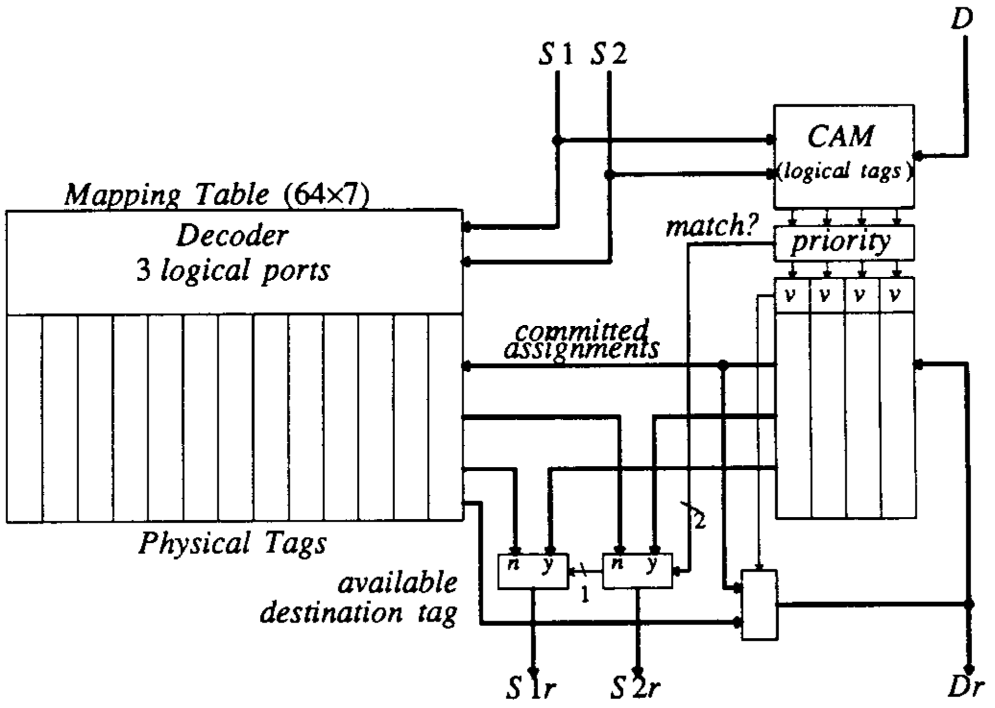


Figure 7.2: Circuitry with minimum storage for tags.

Implementation

A preliminary layout of the logic necessary for implementing the scheme described in Figure 7.2 is shown in Figure 7.3. Each one of the components shown in the figure has been laid out previously [Tami90b, Tami90a]. Formulas based on the dimensions of the basic cells used in our previous designs can be derived to obtain the dimensions of all modules in Figure 7.3. The formulas for two read ports/one write port (2R/1W) cells were obtained directly from the layout of the Mirror Processor [Tami91]. The height of the register array is based on the databus pitch (39λ) and on the overhead for precharging the select lines (26λ). The width of the array includes the array itself, based on a stride of 32λ for the basic memory cell (two read ports and one write port time multiplexed), plus four wide power

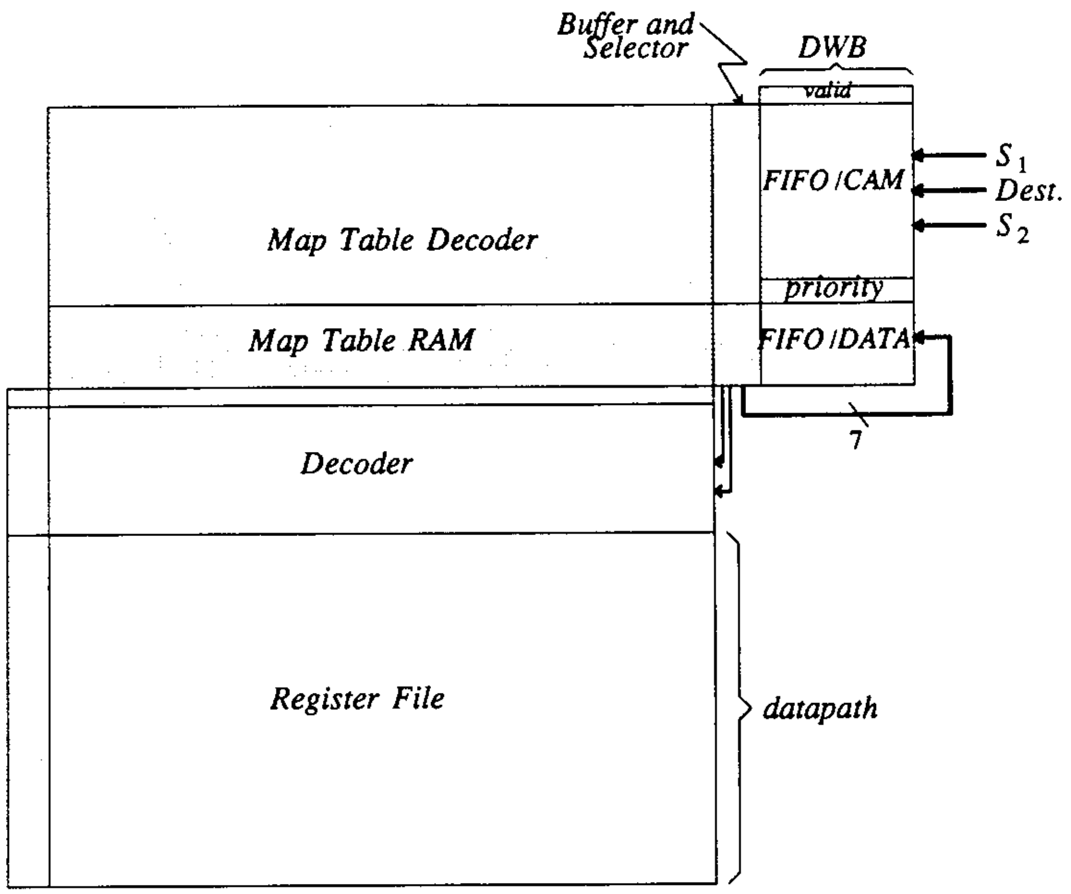


Figure 7.3: Preliminary layout of the logic required to implement micro rollback through tag manipulation. The gray area represents the overhead due to micro rollback.

lines and additional circuits to gate the data line onto the databus. The height of the decoder is based on the pitch of the basic decoder cell (48λ) and by the superbuffers driving the select lines through the array. If a DWB is present then the pitch of the decode cells in the decoder is dictated by the pitch of the CAM in the DWB (pitch of 103λ). Also, the decoder must be stretched in the vertical direction (moved up in Figure 7.3) so that the priority circuit can be inserted between the CAM part and the data part of the DWB. This is shown in Table 7.1

where 253λ (as opposed to 157λ) is added to compute the height of the decoder. The width of the decoder matches the register array and is not affected by the presence of a DWB. The height of the DWB includes (1) the data part, which matches the pitch of the register array, (2) the CAM part, which has a pitch of 103λ, and (3) the priority circuit, valid bits and buffers (403λ). The width of the DWB includes connecting circuitry with the register file, N FIFO/DATA cells (stride of 92λ), and buffers.

		2R/1W ports	1R/1W ports
Register File Array	Height (λ) Width (λ)	$(D \times 39) + 26$ $(R \times 32) + 269$	$(D \times 39) + 26$ $(R \times 32) + 206$
Decoder (no DWB)	Height (λ) Width (λ)	$(A \times 48) + 157$ $(R \times 32) + 269$	$(A \times 32) + 127$ $(R \times 32) + 206$
Decoder (DWB)	Height (λ) Width (λ)	$(A \times 103) + 253$ $(R \times 32) + 269$	$(A \times 87) + 213$ $(R \times 32) + 206$
DWB	Height (λ) Width (λ)	$(D \times 39) + (A \times 103) + 403$ $(N \times 92) + 149$	$(D \times 39) + (A \times 87) + 320$ $(N \times 72) + 75$

D = number of data bits
R = number of registers
A = number of address bits
N = rollback range			

Table 7.1: Formulas for computing the area for register file arrays, decoders, and Delayed Write Buffers (DWB). Results are given for register files and DWBs with one or two read ports and one write port.

The formulas shown in Table 7.1 are based on layouts with matching pitch and stride for adjacent cells. This facilitates the layout greatly and is representative of what CAD tools providing auto-placement and routing would generate. The numbers used for the formulas are obtained from the layout of a complete processor [Lian90] which includes a register file and a DWB. This provides an environment representative of what one would obtained for a layout based on the same cells.

In a different context, where shrinking of individual components is highly desirable, or where CAD tools do not provide pitch matching between adjacent cells, our formulas may not be representative of how the same modules would be laid out. For instance, the height of a decoder laid out to match the pitch of the CAM cells in the DWB is almost twice as large as a normal decoder (871λ vs 445λ for 6 address lines). In a context where local optimization is emphasized (Chapter 4), or where the set of constraints is different, the decoder may be laid out with minimum height (different pitch) and connected to the DWB through extensive routing (for pitch matching). Since extensive routing may be difficult to evaluate, we have chosen in this chapter, to match the pitch of adjacent cell to facilitate the computation of the area overhead for a variety of modules. This will lead to comprehensive comparison between different methods, which is our main objective.

In order to calculate the overhead caused by the register renaming method, we first find the parameters describing the sub-modules according to the formulas of Table 7.1. As mentioned earlier, as a concrete example, we consider a register file of 64 32-bit registers and a rollback range of four cycles.

The datapath is basically left intact except for the 4 extra registers added at one end of the register file. The decoder for the original register file is enlarged to accommodate for the decoding of four more registers via one extra bit. The height of the decoder is calculated from the formula with ''no DWB'' since there isn't any DWB attached to the register file. Hence, the complete register file can be characterized by $D = 32, R = 68$, and $A = 7$.

The map table is a register file (even though it stores *addresses*) with three read ports and one write port. Three different tags need to be read from the mapping table: the physical tags of the two sources registers and the physical tag which is about to be overwritten by the leftmost entry in the FIFO (through the write port). The latter is needed to make a physical tag available for the incoming logical destination tag.

Adding an extra read port to a 2-read/1-write register file results in an increase of approximately 25% in the area of the RAM array [Muld91]. Hence, the register file array area calculated according to Table 7.1 needs to be multiplied by 1.25. For the decoder, adding a port leads to an increase in the height of the decoder. The new height for a normal three-port decoder (without an adjacent DWB) is $Height = (A \times 72) + 231$. But since the decoder for the map table is adjacent to a DWB, its pitch is dictated by the pitch of the CAM cell which is (from Table 7.1): $Height = (A \times 103) + 253$. This formula leads to a larger dimension and is the one we use for the decoder. Finally, the width of the data words in the map table is equal to the number of bits in the physical tags, which is seven for this example. The map table is thus characterized by $D = 7$, $R = 64$, and $A = 6$ and by multiplying the area of the register file array by 1.25 (extra port).

The DWB for the map table needs to contain four entries (rollback range) of six address bits representing the logical tags and seven data bits for the physical tags. It is characterized by: $A = 6, D = 7$, and $N = 4$.

The area taken by each component is shown in Table 7.2. In this table we show the area of a register file for a processor where there is (a) no rollback, (b) rollback implemented using the DWB method covered in Chapter 4, and (c)

		Register File		Map Table		DWB	Total
		Array	Decoder	Array	Decoder		
No Rollback	Param.	$D=32, R=64$	$R=64, A=6$				
	Height (λ)	$(32 \times 39) + 26$ 1,274	$(6 \times 48) + 157$ 445				
	Width (λ)	$(64 \times 32) + 269$ 2,317	$(64 \times 32) + 269$ 2,317				
	Area (λ^2)	2,951,858	1,031,065				3,982,923
DWB Method	Param.	$D=32, R=64$	$R=64, A=6$			$D=32, A=6, N=4$	
	Height (λ)	$(32 \times 39) + 26$ 1,274	$(6 \times 103) + 253$ 871			$(32 \times 39)+(6 \times 103)+403$ 2,269	
	Width (λ)	$(64 \times 32) + 269$ 2,317	$(64 \times 32) + 269$ 2,317			$(4 \times 92) + 149$ 517	
	Area (λ^2)	2,951,858	2,018,107			1,173,073	6,143,038
	Overhead	0%	96%			100%	54%
Renaming Method	Param.	$D=32, R=68$	$R=68, A=7$	$D=7, R=64$	$R=64, A=6$	$D=7, A=6, N=4$	
	Height (λ)	$(32 \times 39) + 26$ 1,274	$(7 \times 48) + 157$ 493	$(7 \times 39) + 26$ 299	$(6 \times 103) + 253$ 871	$(7 \times 39)+(6 \times 103)+403$ 1,294	
	Width (λ)	$(68 \times 32) + 269$ 2,445	$(68 \times 32) + 269$ 2,445	$(64 \times 32) + 269$ 2,317	$(64 \times 32) + 269$ 2,317	$(4 \times 92) + 149$ 517	
	Area (λ^2)	3,114,930	1,205,385	(×1.25)865,978	2,018,107	668,998	7,873,398
	Overhead	5.5%	17%	100%	100%	100%	98%

Table 7.2: Overhead for adding a rollback of range four to a 64×32 bit register file for both the DWB method and the register renaming method. The numbers are extracted from layouts of the Mirror processor. The parameters are: $D =$ number of data bits, $R =$ number of registers, $A =$ number of address lines, and $N =$ rollback range.

rollback implemented using register renaming. For a register file of 64 32-bit registers and a rollback distance of 4, the DWB method presents an overhead of 54% compared to an overhead of 98% for the register renaming method.

Advantages

One advantage of the register renaming method over the DWB method concerns removal of the logic attached to the datapath. This logic increases the capacitance of the bus lines and lengthens the processor cycle time. Indeed, the added delay with the DWB method comes mainly from having longer buses due to the FIFO/CAM (which contains data and register tags) which is connected in between the register file and the shifter/ALU. Buffers located in between the register file and the DWB also contribute to a small increase in the processor cycle. With the register renaming method, the datapath goes through a single uniform structure forming the register file. By removing the DWB, the FIFO/DATA cells are deleted from the datapath. This is important considering that the FIFO/DATA cell of the DWB is more complex than a simple register file cell, but must still fit into the same datapath pitch. For the DWB method, *N* columns (*N* is the rollback range) of this complex cell contribute to a significant increase of the datapath stride. Alternatively, in order to reduce the length of the data path, the pitch could be increased, but this would result in a larger area for the datapath.

With the DWB method it is not clear how sense amplifiers could be integrated to speed up reading of operands. One possibility is to insert two sets of sense amplifiers, one for each of the logically separated data parts (register file and DWB), this, however, would add a significant stride to the datapath. Register renaming, on the other hand adds a few registers to the register file, but does not modify the underlying logic. The only impact of this technique on the timing would be a possible longer time required before the latch signal cuts off the sense amplifiers from the bus of the register file (going from 64 to 68 registers). The decoder of the main register file is smaller than the decoder obtained with the DWB method since it does not have to match the stride of the complex FIFO/CAM cell present in the DWB. Another advantage obtained through this method is that the extra logic required is logically and physically separated from the datapath. This means that a high performance datapath would not require modifications except for adding *N* registers to the register file. Of course the decoder requires small modifications especially if one more bit needs to be decoded, but should would not disturb the timing of the datapath.

The principal advantage of this method is more apparent in the context of a multiple functional unit processor with a superscalar architecture. In that case, micro rollback and out-of-order execution are merged together sharing the same circuitry, diminishing overhead in terms of area and delays. We will describe this situation in the next section.

Disadvantages

The benefits described above come at a large price. The delay to perform register mapping is large and is likely to be on the critical path of the processor. The time to access the register mapping table is serialized with the access to the decoders and the discharge of the data lines, leading to a longer processor cycle. On the other hand, as we will see in a later section, some processors allow a full clock cycle for register renaming, a length of time that should be sufficient to handle our extra logic.

The area occupied by the mapping table is very large mainly because of the large decoder which must decode logical addresses. Compared to the DWB method, the area overhead is much larger, 98% compared to 54% for the DWB implementation. The register renaming method is also more complicated to initialize. Initial tags must be loaded into the CAM, which contributes to a few extra cycles for initialization. Finally, it is not clear how register renaming can be integrated with register windows. In the current context, because of those disadvantages, the DWB method is a logical choice over the register renaming method.

Observations

With the DWB method, a buffer is used to store uncommitted data. Similarly, register renaming uses a buffer to store uncommitted tag assignments. During a rollback, buffers containing uncommitted modifications are selectively invalidated, effectively undoing changes that should not have occurred.

Following a rollback, the register file associated with the DWB method contains valid data while the register file of a processor using register renaming may contain invalid data in some registers. With register renaming, rollback is accomplished by ''moving'' the micro rollback logic at a different level. Upon a rollback, uncommitted data is made unavailable through invalidation of tag assignments. The data part of the register file is left intact while the address part is made more complicated.

Error Detection for the Register Renaming Logic

The logic needed to implement micro rollback using register renaming mainly consists of memory cells (CAM, RAM, shift registers, etc.). Similarly to the register file and the DWB (Chapter 4), errors occurring in this added logic can be detected by generating a parity bit for each pair of logical and physical tags. The parity bit is generated from the bits forming both tags and the valid bit. The parity bit propagates through the DWB along with the tags and is stored in the mapping table after N cycles. The parity bit is recomputed and XORed with the stored parity when the physical source tags are read from the mapping table or from the DWB. A mismatch generates a rollback and requires a processor transfer of the tag assignment from the fault-free processor.

The error detection/correction hardware is significantly larger for the register renaming method than for the DWB method. An extra parity generator/checker is needed and a different protocol is required to transfer tags among processors.

7.1.2. Micro Rollback and Register Renaming for a Superscalar Processor

In the previous section we have described how the register file of a processor could be rolled back using register renaming. As mentioned before, the original goal of register renaming was to reduce storage conflicts for processors with several functional units [Logr72, Kell75]. In this section we will describe how we can make use of the existing hardware dedicated to register renaming to accomplish micro rollback. As a concrete example, we will base our discussion on the scheme chosen for the IBM America project [Groh90], which is basically the same scheme that was later implemented for the RS/6000 [Grov90].

Description

In order to allow (1) greater parallelism and (2) synchronization of the fixed-point unit (FXU) and the functional units of the floating-point unit (FPU), register renaming is used for the source and destination registers of the FPU. The FPU can execute instructions with three source registers and one destination register. The 5-bit field for each register address corresponds to one-of-32 logical registers in the register file.

When an instruction enters the *rename stage* of the floating-point pipeline, the three logical source register tags S_1, S_2, and S_3 (Figure 7.4), are mapped to 6-bit physical register tags through the map table. The destination register tag is renamed, i.e. a new physical tag is assigned to its logical tag. The new tag is obtained from the free list (FL) and the old tag is kept in the pending target return queue (PTRQ). Both queues (FL and PTRQ) are maintained through a set of pointers. Tags in the PTRQ are released by previous arithmetic instructions which signal that data contained in those registers have been used and do not need to be kept ''alive'' any longer. Before returning to the free list, a comparison of the tag is made with pending *stores* (''='' in the pending store queue of Figure 7.4). If there is no match, the tag is sent to the free list. If a match occurs, the give back (GB) bit is set and the tag (contained in the ''T'' field) will be released only when the store completes. There are eight (six for the RS/6000) extra physical registers provided, making the depth of each queue equal to eight. A lengthy explanation of the renaming method can be found in [Groh90]

The current IBM RS/6000 implementation of the floating-point unit allows two instructions to be renamed per cycle, indicated by the two registers R_0 and R_1

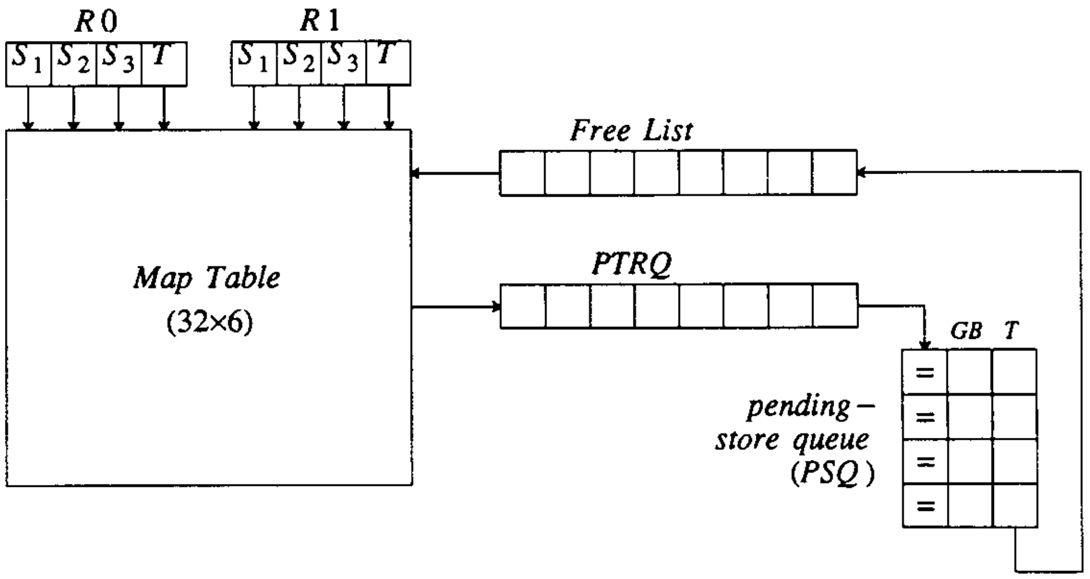


Figure 7.4: Register renaming scheme for the IBM RS/6000

pointing to the map table in Figure 7.4. For the sake of simplicity, we describe the hardware as if only one instruction enters the rename stage per cycle.

Adding micro rollback to the register file means holding "overwritten" data long enough in temporary registers so that if a rollback occurs it is possible to restore old values. This action is similar to what is accomplished by register renaming by saving old register values until previous operations complete. There are some subtle distinctions between the two sets of requirements which we will uncover in the following paragraphs.

The mapping table, the different queues, as well as the pointers maintaining those queues represent part of the processor state dedicated to register renaming and thus need to be rolled back. These sub-modules operate synchronously and can thus be rolled back individually using methods described previously. This would lead to a large overhead and would not take advantage of the specific functions accomplished by each sub-module. For example if a module (e.g. map table), containing only committed data, sends data to an adjacent module (e.g. the PTRQ), this data does not need to be buffered through a DWB before reaching the adjacent module. An efficient method avoiding superfluous logic is described in the following paragraphs.

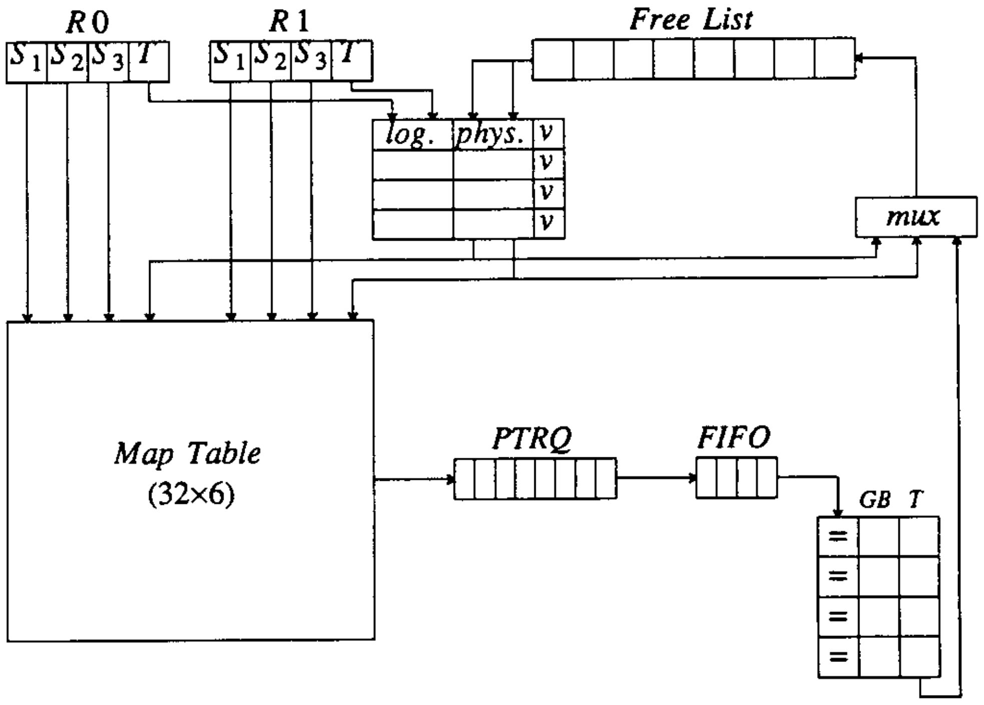


Figure 7.5: New structure for register renaming and micro rollback for the IBM RS/6000 processor architecture.

The map table, which can be implemented as a small RAM, represents the largest state of the register renaming control structure. To prevent uncommitted data from corrupting the map table, a small buffer is inserted in the path between the target register tag and the map table (Figure 7.5). Any new tag mapping is first stored in the buffer. Only after four cycles (four being the rollback range), is the assignment stored into the table. When a rollback occurs, the tag assignments stored in the buffer are invalidated so that when they reach the end of the buffer they are returned to the free list instead of being stored in the map table.

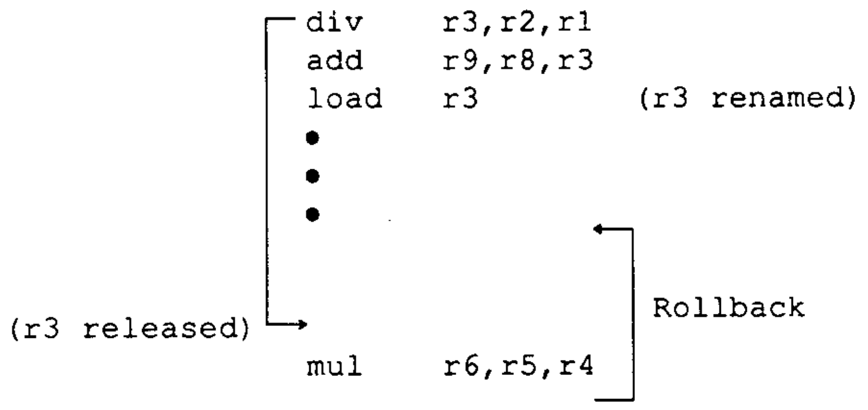


Figure 7.6: Register tag released after a long latency. This example shows the need to "unrelease" a tag during a rollback.

When a tag assignment enters the map table, it pushes the old tag value out to the PTRQ. Since any action performed on the map table is committed, the tag that comes out is also committed, which means that the PTRQ also contains committed data and its data does not need to be rolled back. On the other hand, actions performed on the PTRQ may not be committed. For example let's consider the code shown in Figure 7.6. The `div` instruction is issued and takes several cycles to produce a result. The `add` instruction cannot execute since its data depends on the result of the previous instruction. The `load` following the `add` creates a new assignment for `r3` and the old assignment is pushed into PTRQ waiting to be released by the `add`. When the `div` terminates, the `add` fetches `r3` (using the physical address saved at issue time) and start executing. At that time the old assignment for `r3` is released. If a rollback occurs shortly after `r3` is released, it is necessary to "unrelease" it so that when the `add` re-executes, it has the proper register to release. This is accomplished by adding a small buffer and by managing the queue slightly differently (Figure 7.7). Released tags are buffered in a small FIFO (four entries) before they enter the free list. This has the effect of delaying the action of releasing tags by four cycles, allowing time for a subsequent rollback to invalidate the last *n* released tags. With eight extra physical tags available for register renaming [Groh90], a PTRQ of length twelve will guarantee that at least four entries separate the head pointer from the (wrap-around) tail pointer. Those four entries act as a temporary buffer for pre-released tags. Upon a rollback, the pointers are brought back to their previous value, pointing to tags available at that time.

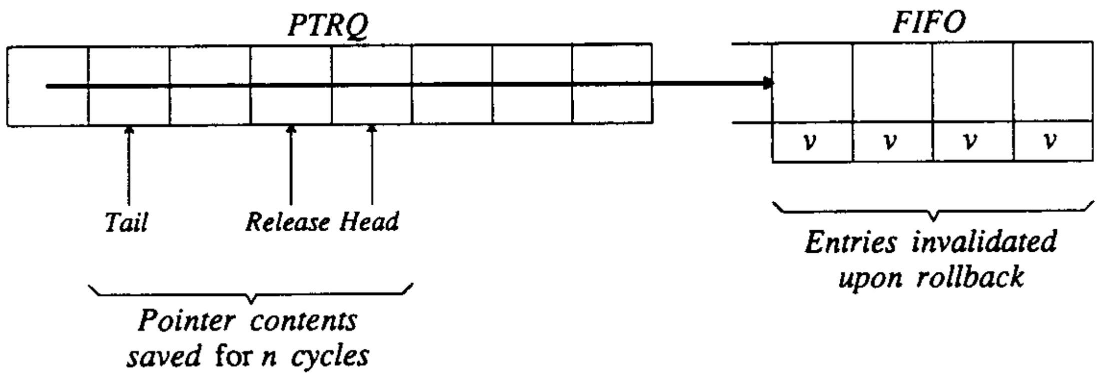

PTRQ
FIFO
v
v
v
v
Tail
Release Head
Entries invalidated upon rollback
Pointer contents saved for n cycles

Figure 7.7: New Pending Target Return Queue (PTRQ). A small FIFO is added and the PTRQ is made longer so that tags can be "unreleased".

In this section we showed how micro rollback could be added using a register renaming technique (a) to a simple processor and (b) to a processor with a few functional units. Different tradeoffs showed that the area needed for implementing the register renaming technique for a small processor, is much larger than with the DWB. Unless the few advantages tied to the register renaming method (enumerated earlier) are critical, the DWB is a preferred choice. For case (b), the processor already uses register renaming for performance reasons, so adding micro rollback through register renaming instead of with a DWB, becomes relatively inexpensive. The main changes to the existing hardware include:

— assign a new physical tag for all instructions with a destination field (done in the America project, but only *loads* are renamed in the RS/6000). On a more complex processor, this is already implemented.

— add a small buffer in front of the map table

— modify the PTRQ

— rollback pointers

We have concentrated our efforts on the register renaming circuits since they can be combined effectively with micro rollback. Other parts of the processor (coprocessor) would require additional circuits similar to the ones described in Chapter 4. An accurate evaluation of the area overhead will be given in a later section (for a more complex processor).

Error Detection for the RS/6000 Register Renaming Logic

Errors occurring in the logic needed for register renaming and micro rollback can be detected by using parity and state compression/comparison. A parity bit can be attached to a pair of tags when it is written into the DWB and when it is later stored into the map table. This parity bit is checked whenever a tag pair is read from the map table or from the DWB. For the physical tags in the PTRQ, the FIFO (extension of the PTRQ), the pending store queue (PSQ), and the free list (FL), a parity bit can be used in conjunction with collecting the state of the tags at a few key places (entering and exiting queues, and in the PSQ).

7.2. Integrating Support for Micro Rollback and Precise Interrupts

In order to achieve high throughput, most modern CPUs consist of several pipelined functional units connected together through a complex instruction issuing unit. Some supercomputers, such as the CRAY-1, have several functional units but are limited to issuing one instruction per cycle [Cray77]. Other processors such as the SIMP processor [Mura89], the Intel i960CA [Inte89], the Intel i860 [Kohn89], and the IBM RS/6000 [Grov90], can issue multiple instructions per cycle to various functional units. In this section we consider processors of the first type discussed above (Figure 7.8). As in [Smit88], we assume that the process state consists of the program counter, the register file and main memory. The instruction issue unit provides a bandwidth of one instruction per cycle if there are no register interlock conflicts and no result bus contention.

As shown in Figure 7.8, different functional units (one of which is a load/store pipe) have different latencies leading to out-of-order completion. Smith and Pleszkun [Smit88] proposed a method for implementing *precise interrupts* on such a processor.

An interrupt is *precise* if, when it is processed, all instructions before the instruction causing the interrupt have terminated and none of the subsequent instructions have modified the state of the processor [Smit88]. Precise interrupts force a machine to preserve the view of a machine which executes one instruction at the time, finishing each one before processing the next one. Generally, for an interrupt to be precise, the program counter is restored to point to the interrupt causing instruction. Interrupts, as defined in the article by Smith and Pleszkun, include traps resulting from exception conditions (illegal opcode, divide by zero,

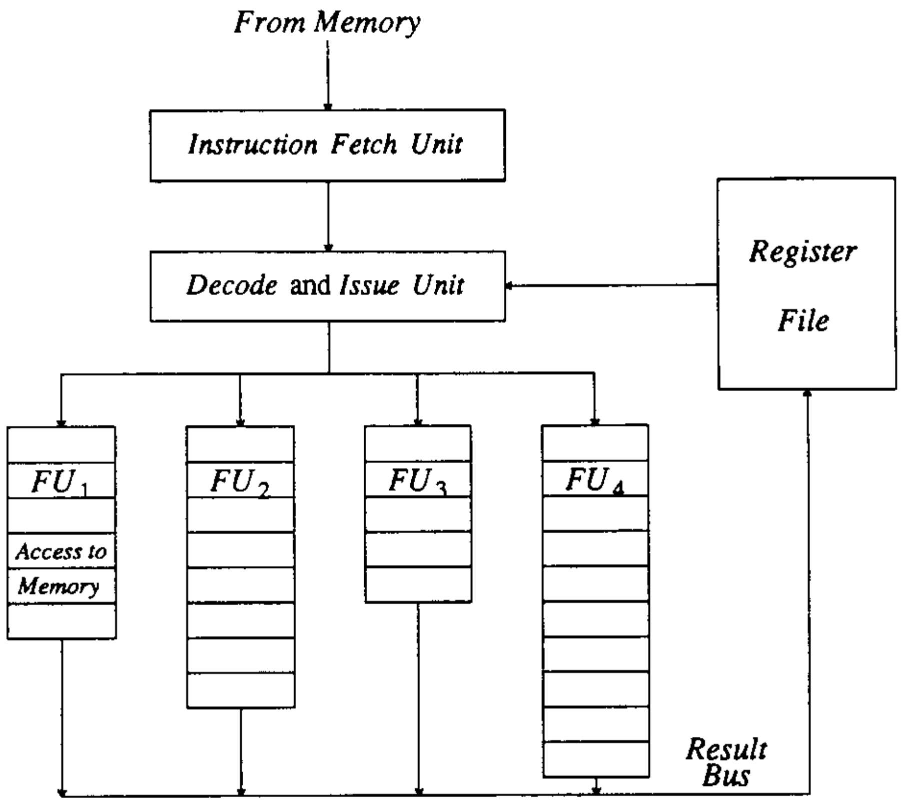


Figure 7.8: Processor with several pipelined functional units with a single instruction issuing unit and a single result bus.

overflow, etc) and external interrupts which are caused by sources outside of the current process (e.g. I/O interrupts).

In order to provide precise interrupt, Smith and Pleszkun proposed to add a result shift register (RSR) and a reorder buffer (RB) to the processor (Figure 7.9). The result shift register provides a way to reserve in advance the result bus based on the latency of the operation in the decode phase. It also saves the address of the location in the reorder buffer where the result will be routed. When an instruction is decoded, an attempt is made to insert a new entry in the result shift register. If the desired location is already occupied by a valid entry, the issuing unit simply

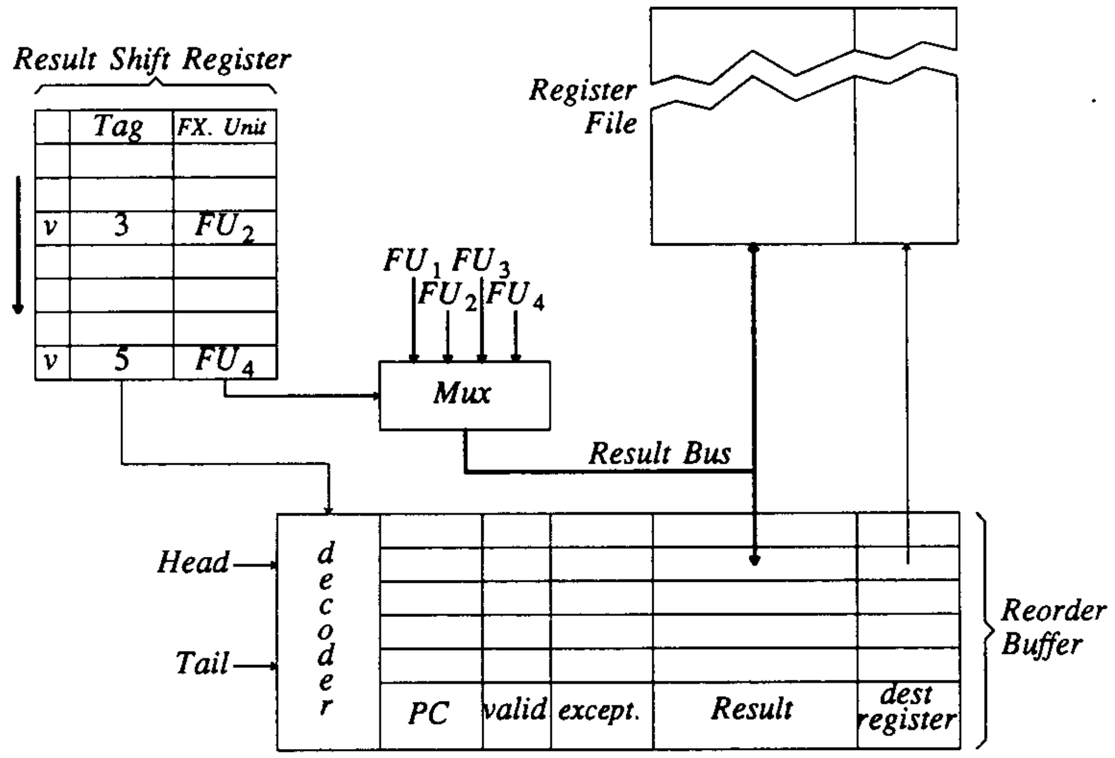


Figure 7.9: Result shift register and associated reorder buffer.

waits one cycle for the FIFO to shift. The reorder buffer (RB) allows results to be made available to subsequent instructions as soon as they are produced by the functional units. The results are temporarily stored in a queue and are committed to the register file in program order (Figure 7.9). When an instruction is decoded, information corresponding to its operation is written in the reorder buffer entry indicated by the tail pointer. The head pointer indicates which entry is going to be transferred to the register file next, as soon as the corresponding result is produced. In order to make the latest results available to subsequent instructions, bypass circuits are connected to the reorder buffer (not shown on the figure).

7.2.1. Rollback of the RSR and RB Using Multiple DWBs

In this section, we show how micro rollback can be added to a CPU with support for precise interrupts using DWBs to delay commitment of changes to the RSR and RB.

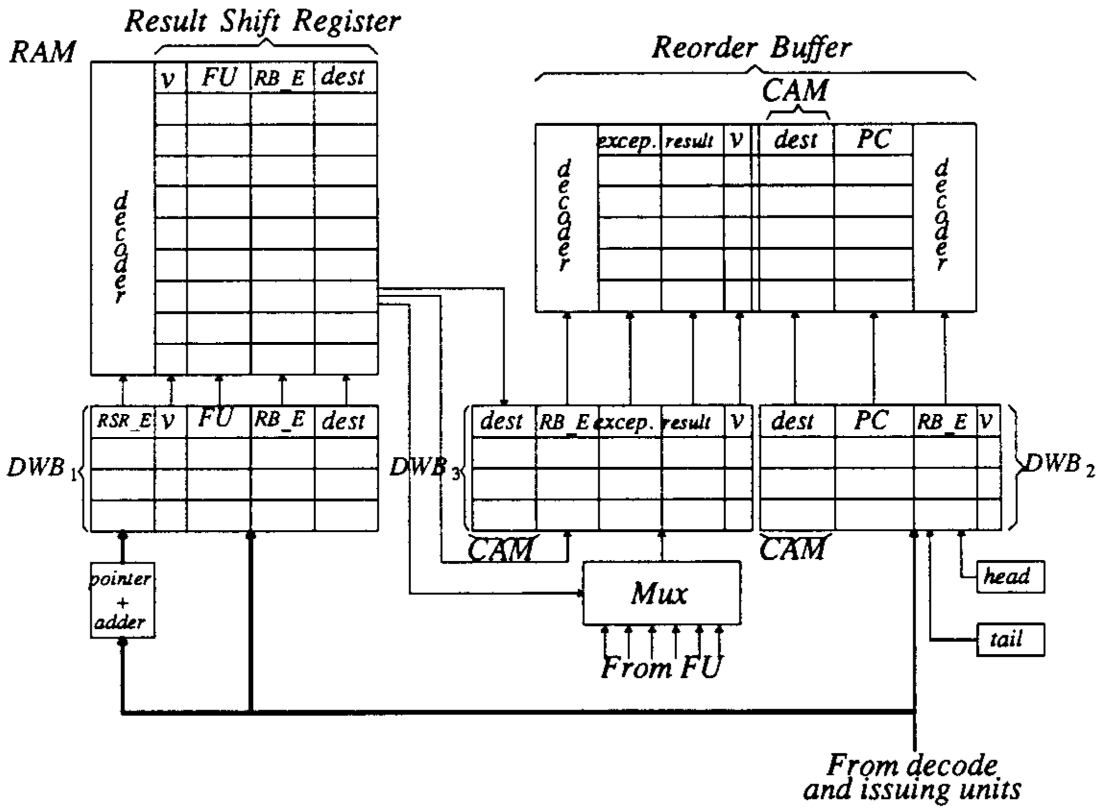


Figure 7.10: Micro rollback added to a multiple functional unit CPU.

The result shift register is a complex structure in which entries are inserted out of order and shifted every cycle. To facilitate micro rollback, the shift register capability of the RSR is implemented as a RAM with a pointer indicating the head of the buffer (Figure 7.10). Shifting is simulated by incrementing the pointer every cycle so that it points to the next location in the RSR. The address of a new entry in the RSR is determined by adding the latency of the functional unit specified by the current instruction to the pointer. This forms the associative field identified as *RSR_E* is DWB_1 (Figure 7.10). The bypass circuitry provided by DWB_1 is required for the RSR since the pointer may access a location in the RAM whose most recent contents are in the data part of DWB_1. Each RSR entry includes a valid bit (v), a tag identifying the functional unit (*FU*), a tag identifying a location in the reorder buffer where the result is going to be forwarded (*RB_E*), and the destination register number (*dest*). As we will explain in the next paragraph, the reorder buffer is split in two independent parts which requires a duplication of the *dest* field in order to be able to do an associative lookup to provide the most recent update of a register.

As proposed in [Smit88], each entry in the reorder buffer is modified once during issue time and once when the result is produced by the functional unit. Micro rollback is added to the reorder buffer by logically splitting it into two units and adding two DWBs (DWB_2, and DWB_3 in Figure 7.10) to delay commitment of all modifications.

The components of the RB entry which are modified at issue time are the destination register number and the PC (*dest* and *PC* in DWB_2). These fields are buffered in DWB_2 along with the value of the tail pointer at issue time (*RB_E* in DWB_2). The *dest* field is stored in a CAM with bypass circuits in order to detect RAW hazards if one of the N subsequent instructions specify *dest* as a source operand.

The second part of the reorder buffer includes the result and exception conditions produced by the functional units. As soon as results exit the functional units, they are routed to the reorder buffer. The results and exceptions may not be committed and must thus be buffered through a DWB (DWB_3 in Figure 7.10). Results residing in DWB_3 are the latest updates of the data registers and must be accessible to subsequent instructions. The CAM part of DWB_3 contains the *dest* field associated with the result. This field is made available by carrying it through the RSR and storing it in DWB_3 at the same time as the results. An alternative to carrying the *dest* tag through the RSR would be to retrieve it from the reorder buffer when storing the result in DWB_3. This would require an extra read port and the corresponding buses in the reorder buffer. Upon exiting DWB_3, the result, the exception bits, and the valid bit are transferred to the reorder buffer.

A rollback consists of invalidating entries in the three DWBs and rolling back the pointer of the RSR and the tail pointer of the RB. The head pointer of the RB is not rolled back since it is advanced only when committed data exit the RB.

To summarize, we describe the complete path followed by an instruction. When issued, the functional unit and the destination register specified by the instruction are stored in DWB_1. One entry in the RSR (*RSR_E*) and one entry in the RB (*RB_E*) are reserved and the corresponding entry numbers are stored in DWB_1. At the same time, the *dest* fields of the instruction, the PC and the tail pointer value, are stored in DWB_2. After N cycles (we assume for this example that the latency of the functional unit is greater than N), data is stored in the reserved slots of the RSR (from DWB_1) and the RB (from DWB_2). When the result is produced by the functional unit, it is routed along with the exception bits to DWB_3 by the *FU* field of the RSR. The *dest* field is transferred from the RSR to DWB_3. After N cycles, the result and exception bits are transferred to the reserved location in the RB. When the head pointer reaches the reserved location, the result is sent to the register file conditionally on the value of the exception bits. Subsequent instructions check for RAW hazards by scanning the *dest* field in DWB_2, DWB_3 and the reorder buffer.

In order to evaluate the area overhead resulting from adding micro rollback, we consider the RSR to be a register file with one read port and one write port, while the RB is a combination of a register file with two read ports and one write port and a CAM (comparable to a DWB). From Table 7.1 in the previous section, we can calculate the area overhead for register files modified to accommodate micro rollback (matching the decoder pitch with the CAM cells of the DWBs). The numbers for the one read port/one write port (1R/1W) in Table 7.1 were obtained from modifications of the 2R/1W basic cells. The area taken by the three added DWBs is computed in a similar way. This area, once added to the extra area required for the modifications of the RSR and RB gives us an approximation of the total area overhead. Some of the area needed for the extra control signals is taken into account since the formulas are based on a real design which has control lines running across the circuitry. The resulting floorplan of the RSR and RB with the DWBs should be similar to the floorplan of a similar chip without rollback capability since we position the DWBs in front of the RSR and RB.

For the RSR the parameters are:

$$
\begin{aligned}
\text{N} &= \text{rollback range} \\
\text{R} &= \text{maximum number of pipeline stages in the FUs} \\
\text{A} &= \left\lceil \log_2(\textit{maximum number of pipeline stages}) \right. \\
\text{D} &= \left\lceil \log_2(\textit{number of functional units}) + \right. \\
&\quad \left\lceil \log_2(\textit{number of entries in the RB}) + \right. \\
&\quad \left\lceil \log_2(\textit{register file size}) \right.
\end{aligned}
$$

For a CPU with a register file of 32×64 bits (floating-point data), with six functional units with pipelines as long as 10 stages, with 6 possible entries in the RB, and for a rollback range of four, we obtain for the original result shift register: N=4, A=4, D=6, R=10. For the modified RSR the parameters are: N=4, A=4, D=11, R=10. Table 7.3 shows the area overhead for each individual component. The table shows that the area dedicated to the RSR is increased by 250%.

		New Parameters	Original Design			Modified Design			DWB	Overhead	
			array	decoder	total	array	decoder	total	total	(λ^2)	(%)
RSR	total	N=4,A=4 D=11,R=10	136,760	134,130	270,890	239,330	297,086	534,412	368,333	675,894	250
RB	left	N=4,A=3 D_1=69,R=6	1,252,537	138,761	1,391,298	1,252,537	138,761	1,391,298	1,926,342	1,926,342	138
	right	N=4,A=5 D_2=35,R=6	1,446,163	88,754	1,534,917	1,446,163	88,754	1,534,917	1,180,311	1,180,311	77
	total	N=4,A=3 D=106,R=6	2,698,700	227,515	2,926,215	2,174,620	227,515	2,926,215	3,106,653	3,106,653	106
Sum	Total		2,835,460	361,645	3,197,105	2,413,950	522,601	3,460,631	3,519,021	3,782,547	118

Table 7.3: Calculation of the area overhead for adding three DWBs and modifying the result shift register and reorder buffer.

The RB is decomposed in two sections. The first section is connected to the

RSR while the second one is connected to the issuing unit. For the first part (on the left side in Figure 7.10 and labeled "left" in Table 7.3), we have the following parameters:

N	=	rollback range
R	=	number of entries in the RB
A	=	$\lceil \log_2(\textit{number of entries in the RB})$
D_1	=	(bits of results) + (bits for exceptions)

This gives N=4, R=6, A=3, and D=69 which results in an overhead of 138%. The "right" part of the reorder buffer is more complex in the sense that one part is similar to a register file while the other part is a CAM with a priority circuit similar to the one used in a DWB. In Table 7.3 the area for the CAM is accounted with the area for the register array. For this part of the RB, the parameters are:

N	=	rollback range	
R	=	number of entries in the RB	
A_1	=	$\lceil \log_2(\textit{number of entries in the RB})$	[for the decoder]
A_2	=	$\lceil \log_2(\textit{register file size})$	[for the CAM]
D_2	=	$\textit{PC width}$	[for the RB]
D_3	=	$\lceil \log_2(\textit{number of entries in the RB})$ + $(\textit{PCwidth})$	[for DWB_2]

For the same processor with its original register file of 32 entries of 64 bits, a PC of 32 bits and 5 bits dedicated for exceptions, this gives: N=4, A_1=3, A_2=5, D_2=32,

D_3=35, and R=6. Table 7.3 shows that the area overhead for implementing micro rollback for both the RB and the RSR, results in a 118% increase in area.

In the next section, we will describe another method which attempts to take advantage of the features of the structures being rolled back as well as the way that they are interconnected in order to reduce the area overhead.

7.2.2. Rollback of RSR and RB Using Marked Entries

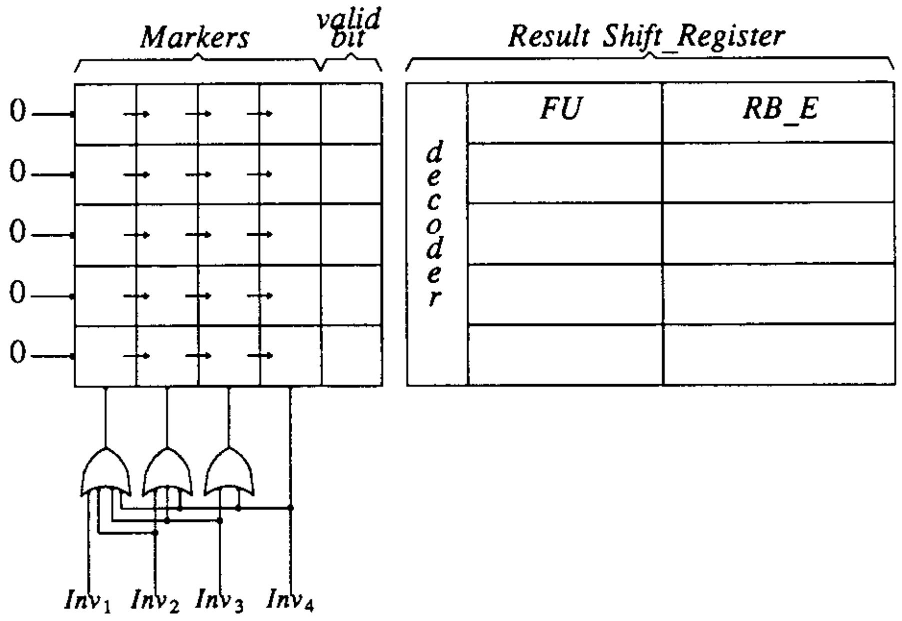


Figure 7.11: Four bit mark used to identified entries (for possible invalidation) modified in the past four cycle.

Since new entries in the RSR are only written to invalid locations, valuable contents cannot be overwritten. This means that upon a rollback, there is no need to restore the previous contents of recently modified RSR registers. The register is simply invalidated so that it does not generate unwanted actions when it reaches the head of the buffer (the other purpose of the invalidation is to make entries available for upcoming instructions). To provide rollback capability, entries in the result shift register are marked using an N-bit shift-register. Upon entry in the RSR, a "1" is shifted in the leftmost cell (Figure 7.11). Each cycle, a zero is shifted in. After four cycles, the tag contains (0000) indicating that the valid entry in the RSR is committed.

A rollback of the entries conditionally clears the valid bit based on the value of the mark bits for each entry (Figure 7.12). The mark bits are implemented through shift register cells. The "clear valid bit" and "clear mark bits" signals are generated through a wired-OR of four AND-gates implemented with pass-transistor logic. Only one of the AND-gates can be "on" at any time since only one "1" circulates through the shift register. The "clear mark bits" signal clears the appropriate bits during the next phase. The valid bit is a simple one-bit memory cell. The fields *FU* (functional unit number) and *RB_E* (reorder buffer entry) in the RSR are used in the same manner as for the RSR described in Figure 7.9.

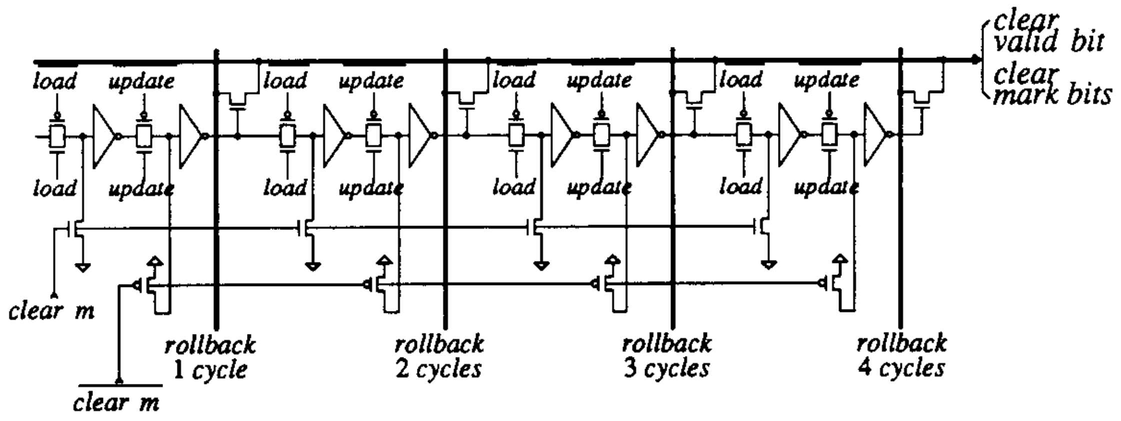


Figure 7.12: Mark bits and the clear valid bit signal for the RSR.

A rollback must also "replay" the few entries that might have reached the head of the result shift register during the past *n* cycles. This is done by moving

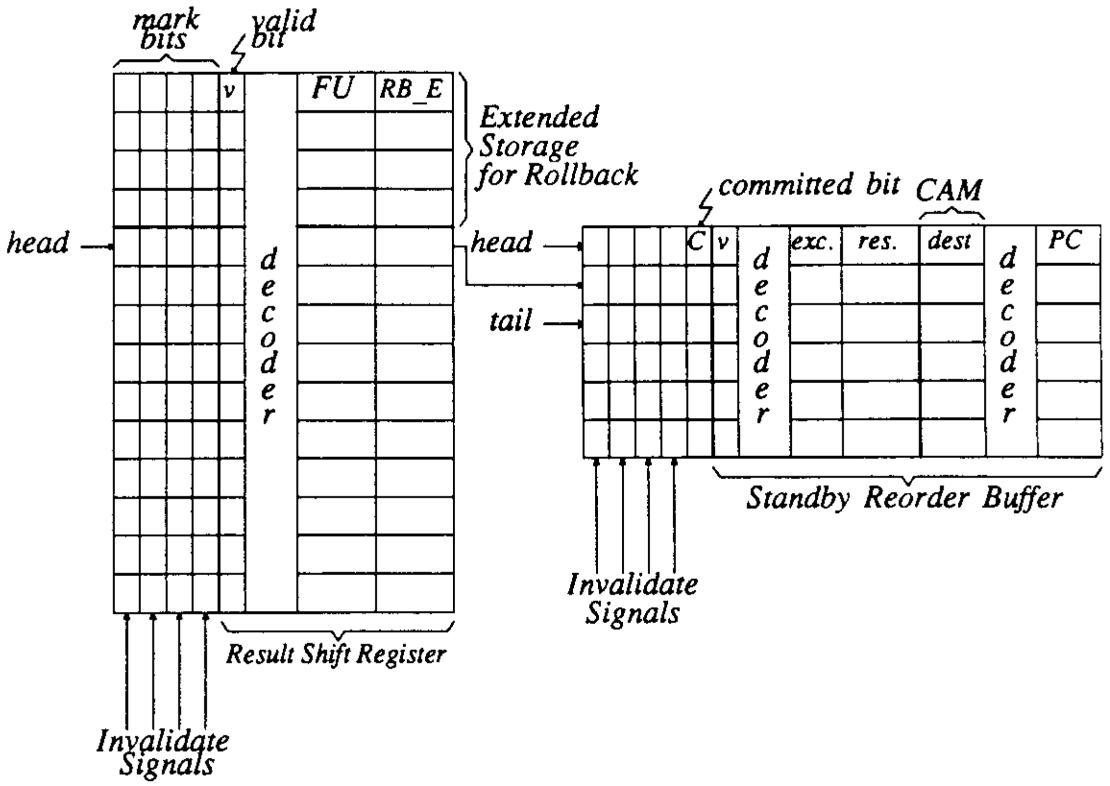


Figure 7.13: Marked result shift register and standby reorder buffer. Each structure is implemented as a RAM with the associate decoder (*dec*).

the head pointer "up" n positions in the RSR (Figure 7.13). In order to guarantee that there is a "buffer area" in the RSR (which is implemented as a circular buffer), the number of entries (originally K) in the RSR is increased by N. K is the number of stages of the longest pipeline connected to the RSR. Since an entry cannot be more than $K-1$ locations away from the head pointer, by making the depth of the RSR $K+N$, each entry remains intact for N cycles after it has been released by the head pointer.

The structure that we propose for combining rollback to the reorder buffer, is called a *standby reorder buffer* (SRB) (Figure 7.13). The name comes from the fact that entries wait in the buffer in ''standby'' until they become committed. As with the result shift register, new entries in the standby reorder buffer do not overwrite previous valid entries. Hence there is no need to preserve old values for rollback recovery. As for the reorder buffer proposed in [Smit88], the standby reorder buffer is managed using pointers. The head pointer indicates the next result to be transferred to the register file. The tail pointer indicates the next available entry in the RB.

The SRB is modified twice. The first time, the destination register (*dest*) and the program counter (*PC*) are stored in program order by the issuing unit at the entry pointed to by the tail pointer. The second time, the result (*res.*), the exception bits (*exc.*), the valid bit (v), and the mark bits (described later), are stored when the functional unit completes the operation, to the entry pointed to by the *RB_E* field in the result shift register. Two logical decoders are shown on Figure 7.13 since they are accessed by different address buses and select different fields in the SRB. Depending on the timing between the issuing unit and the RSR for a specific implementation, it may be possible to combine the two decoders into one by adding a control signal selecting which ''half'' of the SRB should be modified.

All in-order modifications made between time {t} and {t+n} are entered between the positions occupied by the tail pointer at time {t} and the position occupied at time {t+n}. By rolling back the tail pointer to its position at time {t}, all modifications made in the past n cycles will be eliminated. As we will explain later, the head pointer is moved only when the data it contains is committed, hence it does not need to be rolled back (a head pointer move is always committed).

A mechanism based on mark bits and a ''committed bit'' is used to provide rollback for random modification in the SRB. Whenever a result becomes available it is routed to the reorder buffer and ''marked'' as recently modified, i.e. the leftmost bit in the small shift register is set to one. Every cycle a ''0'' is shifted in the register. After four cycles, the mark bits are all zeroes (0000) and the committed bit is set to ''1''. The committed bit remains ''1'' until it is cleared by the control logic when the result is transferred to the register file. The transistor diagram for the last mark bit and its control signals is shown in Figure 7.14.

An entry pointed to by the head pointer is not sent to the register file until the committed bit is set. This guarantees that an uncommitted *change* will stay at least four cycles in the standby reorder buffer so that it can be intercepted before modifying the state of the CPU permanently. Rollback is accomplished in the same way as for the result shift register. If there is a match between the invalidate lines and the mark bits (both being one), a ''clear valid bit'' and a ''clear mark bits'' signal is initiated.

Implementation

The area overhead for adding micro rollback to the RSR and for the SRB can be calculated using the formulas shown in Table 7.1. This will give us the area for all the logic except for the mark bits. A layout of the mark bit cell indicates a size of of $88\lambda \times 32\lambda$. The committed bit and the valid bit are included in the register array. A small adder is also needed to compute the address of an entry in the RSR based on the the head pointer and the latency of the functional unit. The adder can be implemented with four simple full adder cells (for a RSR of up to 16 entries) to form a four-bit serial adder.

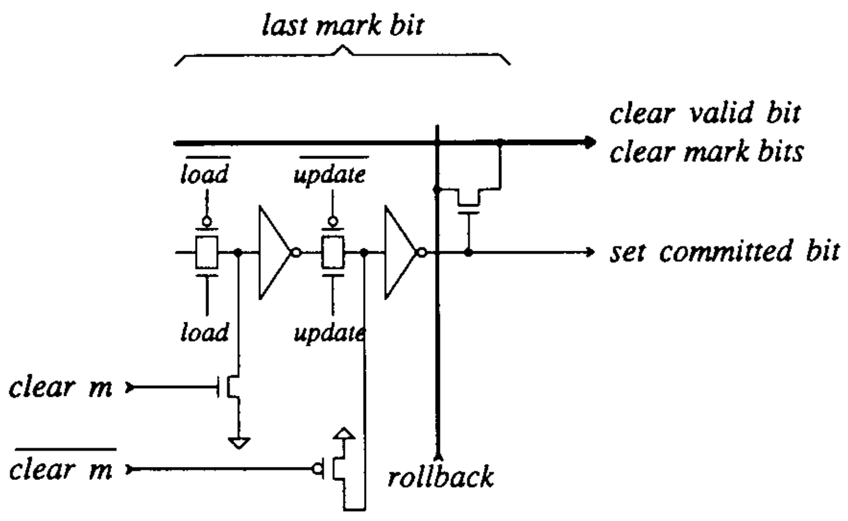


Figure 7.14: Transistor diagram of the last mark bit.

The modified RSR needs to be expanded by N registers vertically, and N mark bits must be appended to its width. This leads to a large increase in the area dedicated to the RSR since N is of the same order of magnitude as D (D = number of data bits in the RAM). The results of the computation are shown in Table 7.4. From the table, ignoring the area for the four-bit serial adder, we obtain an area overhead of 78% for the result shift register.

The standby reorder buffer is not extended in its number of entries from the original reorder buffer. This is due to the fact that entries are maintained in the buffer until they become committed. One could argue that to maintain similar performance we have to add N entries so that the issuing unit does not stall, but this is not entirely true. Since results entered the SRB somewhere between the head and tail pointer, they stay in the buffer for n cycles ($1 =< n =< R$). When the result and exception bits reach the head of the buffer they may already be committed and be sent to the register immediately without stalling the issuing unit.

RSR	parameters	array	decoder	mark bits	total	overhead	
		area (λ^2)	area (λ^2)	area (λ^2)	area (λ^2)	area (λ^2)	(%)
Original Design	N=4,A=4 D=7,R=10	157,274	134,130	—	291,404	—	—
Modified Design	N=4,A=4 D=7,R=14	195,546	166,770	157,696	520,012	228,608	78

Table 7.4: Area overhead for adding micro rollback to the result shift register using mark bits (four-bit serial adder not included). N = rollback range, A = number of address lines in the RAM, D = number of columns in the RAM, R = number of rows in the RAM.

In the worst case, the RB is full and its head pointer points to an entry that must wait four cycles before being transferred to the register file. The average case lies somewhere in between those two cases. A detailed simulation of the processor and of this structure is required to quantify the exact performance degradation. Since we do not have a simulator we show in Table 7.5 the area overhead for the best case (same number of entries as the RB) and for the worst case (SRB with four more entries). The area overhead is 3% for the best case and 19% for the worst case.

		Reorder Buffer					Rollback Logic			Overhead
		array		decoder	CAM	total	mark	commit	total	(%)
		1 port	2 ports	(two)	(DWB)		bits	bit		
Best Case	par.	R=6,D=37	R=6,D=65	R=6,A=3	N=6,A=5	—	R=6,N=4	—	—	—
	area	584,662	1,180,621	253,924	643,518	2,662,725	67,584	7,488	75,072	3
Worst Case	par.	R=10,D=37	R=10,D=65	R=10,A=3	N=10,A=5	—	R=10,N=4	—	—	—
	area	772,694	1,508,429	117,298	643,518	3,041,939	112,640	12,480	125,120	19

Table 7.5: Area overhead for adding micro rollback to the reorder buffer, forming a standby reorder buffer (SRB).

By adding the numbers (area) obtained from Table 7.4 and Table 7.5 we obtain the total overhead for adding micro rollback to both the RSR and the RB. The results are 10% for the best case and 25% for the worst case. Those numbers compare favorably to the numbers obtained for the method with three DWBs (118% overhead).

7.3. Combination of Rollback with Several Speed-Up Techniques

In the preceding sections of this chapter, we considered how micro rollback can be implemented in conjunction with two mechanisms used in high-performance uniprocessors: register renaming and support for precise interrupts in superscalar processors.

In this section we describe other techniques used to improve performance of advanced processors. Specifically we discuss a processor with:

— multiple functional units

— out-of-order execution/completion

— restart after exception (precise interrupts)

— restart after mispredicted branch

— register renaming

— micro rollback

Multiple functional units are necessary for achieving a high degree of parallelism beyond what pipelining alone can achieve. Out-of-order execution and completion of instructions eliminate some of the delays associated with functional units conflicts and data dependencies. As described in the previous sections, precise interrupts are required to guarantee proper restart after exceptions. Single cycle restart after a mispredicted branch is essential for quickly correcting the state of a processor following execution of a wrongly predicted conditional branch [Hwu87]. Register renaming removes anti-dependencies (WAR hazards) and output dependencies (WAW hazards) for the register file. Micro rollback is required to undo state changes which were made based on errors caused by hardware faults. We show how these techniques can be implemented using a standby reorder buffer (SRB).

Precise Interrupts

With the SRB, restart from an interrupt is accomplished as described in [Smit88]. Exceptions are not dealt with when they occur but only when they reach the head of the standby reorder buffer (Figure 7.15). At that time results are forwarded to the register file or cause an interrupt depending on the exception bits stored with that entry.

Branch Repair

The occurrence of conditional branches in the instruction stream reduces the ability of a processor to fetch instructions ahead of time. By introducing dependence between the result of an instruction and the next instructions to be fetched, execution is delayed and NOPs enter the pipeline.

In previous work on hardware support for precise interrupts [Sohi87, Hwu87], and on support for "boosted" instructions [Smit90], it has been proposed to executed one of two paths of a branch (taken or not taken) conditionally until a branch condition is resolved. If the path was guessed right, execution proceeds, otherwise all state changes performed by instructions following the branches must be undone. Once again a parallel between increasing performance and micro rollback exists. In one case, assumptions are made on the outcome of a branch, in the other, assumptions are made regarding the correctness of the data/instructions. Those assumptions are made first on valuable statistics showing that branches can be predicted correctly anywhere from 60% to 93% of the time [Lee84], and secondly on the fact that errors do not occur frequently.

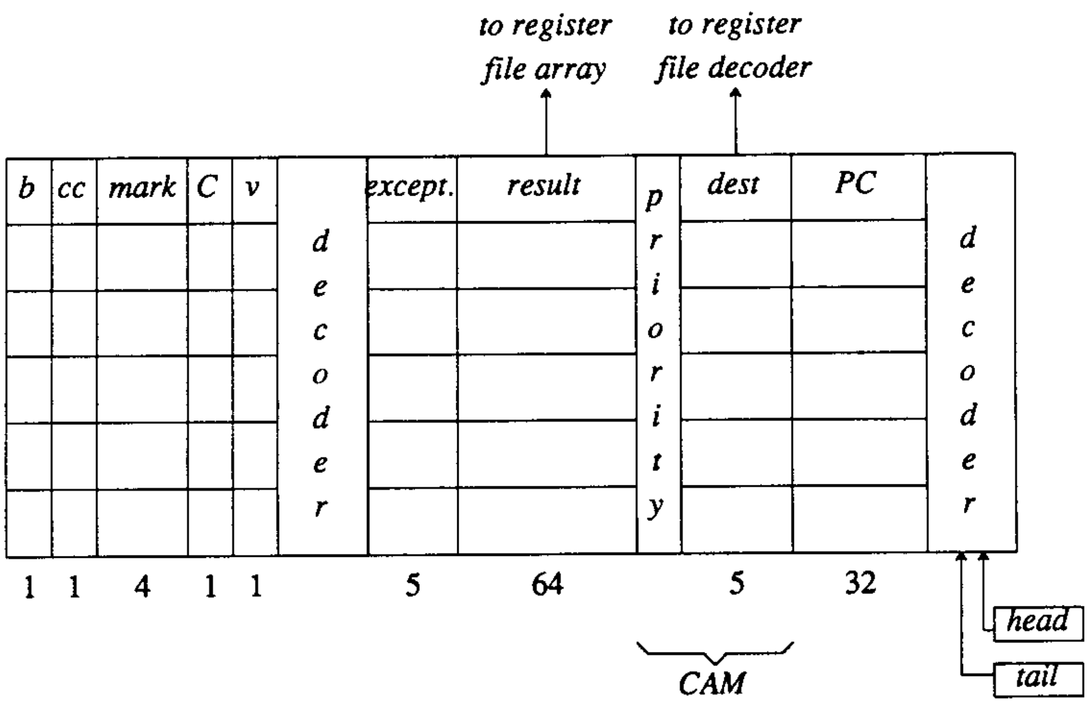


Figure 7.15: Standby reorder buffer (SRB).

With the addition of minimal circuitry to the SRB shown in Figure 7.13, we show that it is relatively simple to recover from mispredicted branches. Whenever a branch is encountered in the instruction stream, it is entered in the standby reorder buffer (the "b" bit is set to "1") even if it does not affect the contents of a destination register (Figure 7.15). The predicted outcome of the condition is also stored in the SRB at the location indicated by CC. On some machine, the target of a comparison is a destination register; in this case the predicted condition code bit are stored in the result field of the SRB. In our case we assume that there is a one bit condition code which is carried by the SRB upon the prediction of a path. When the condition is resolved, a few pipeline stages later, it is sent to the reorder buffer at the branch entry. The outcome of the condition is then compared with the one that was assumed. If they match, the head pointer moves down purging the branch instruction; full throughput is achieved. If the conditions do not match, all instructions that follow the branch, i.e. all instructions placed after the branch in the reorder buffer are invalidated. Alternatively as proposed in [John89], instructions can be canceled only when an instruction (branch) reaches the head of the buffer. In this way, the design of the cancellation circuit is simpler since backing-up the state consists of invalidating the whole reorder buffer instead of selectively invalidating entries.

Register Renaming

The processor model proposed by Smith [Smit88] (Figure 7.8) stalls when one of the source operands of the instruction in the issuing unit is not ready. This situation occurs when one of the source operands matches the destination field of one of the entries in the reorder buffer and the result is not yet valid. The processor also stalls for one cycle if there is a conflict in the RSR (i.e the result bus can not be allocated at the desired time).

Higher throughput can be achieved if out-of-order issuing combined with register renaming is allowed. This can be done by a scoreboarding mechanism [Thor70] or through reservation stations [Toma67] or using a centralized unit (register update unit) [Sohi87]. Many other papers have been published on instruction issuing units [Sohi90, Acos86, Arya85]. Our goal is not to expand on that subject, but to combine micro rollback with a technique representative of what can be encountered in high performance processors. A scheme based on a standby reorder buffer and reservation stations is representative of proven techniques [Toma67, John89] and should be indicative of how micro rollback can be combined with other models described in the aforementioned papers.

We assume that functional units can accept stalled instructions while monitoring the latest progress of all the functional units. For this model, register renaming is accomplished through the SRB. When instructions are issued, they search the SRB in parallel with the register file for the latest update. If there is a match in the SRB, the data is forwarded. If the result is not yet available, the tag corresponding to the SRB entry is forwarded so that when it becomes available the functional unit can proceed. If another instruction with the same destination register enters the SRB, it occupies a more recent entry and all subsequent instructions referring to this logical register will obtain the value from this entry in the SRB since a priority circuit selects the most recent value (Figure 7.15). The register has thus been renamed.

Micro Rollback

Rollback for the SRB is similar to the scheme described in Section 7.2.2. When a rollback occurs, the mark bits are checked to determine how many cycles have the entries been present in the SRB. If the ''stand-by time'' is less than or equal to *n* (the rollback distance), the result and the exception bits are invalidated. The other fields are not modified. The tail pointer is also rolled back which causes entries between time {t} and {t+n} to be discarded.

Since new entries in the SRB do not overwrite potentially useful information (i.e. information that we may try to retrieve), such entries can be written directly into the SRB; there is no need to delay the writes. This has the benefit of removing one level of bypass logic, or forwarding logic, that would be necessary if a DWB were to be used in front of the SRB.

Error Detection for the Standby Reorder Buffer

Errors occurring in the SRB can be detected in the same way as for the DWB method (Chapter 4) and the register renaming method (Section 7.1). A parity bit is generated by XORing the data being written in the SRB and is stored along with the data. When there is a need for accessing the data, the parity is regenerated and compared with the stored one. Since entries in the SRB are modified twice (at issue time and after computation time), two parity bits can be computed based on the separate fields.

7.4. Rollback of Pipelined Functional Units with Long Latencies

In the previous sections we have described how to implement micro rollback in a CPU with multiple functional units. We concentrated our efforts on the hardware connected to the functional units; specifically the result shift register and the reorder buffer. The functional units themselves contain several latches which are part of the processor state, and must also be rolled back. Rolling back generic functional units with long latencies, is the topic of this section.

Most high-end processors have several pipelined functional units with varying latencies. For example in the CRAY-1, latencies range from two cycles up to fourteen cycles [Cray77]. The pipes are composed of combinational logic separated by latches forming individual pipeline stages. The latches in the pipelined functional unit can be rolled back using the methods described in Chapter 4 for individual state registers. This has the disadvantage of adding a significant amount of logic into the datapath of the functional unit. Each latch would be replicated *N* times (where *N* is the rollback range) leading to over 50 additional latches for some pipes. Adding logic to a highly optimized circuit translates into longer delays and lower performance. Moreover adding control signals to the pipes would also complicate the design of these complex circuits.

Performance degradation can be minimized if the functional units are not modified for micro rollback. Instead, the functional units can be handled as independent circuits which cannot be rolled back (see Chapter 6). Since a functional unit interacts with other units only through its inputs and outputs, the circuitry attached to the ends of the pipelines (inputs and outputs) must be used to implement rollback.

The proposed circuitry is similar to the circuitry used in Chapter 6 to implement rollback of tightly coupled coprocessors. For each pipeline there is a shift register of length S, where S is the number of stages in the pipeline (Figure 7.16). This shift register hereafter called the *pipeline usage monitor* (PUM). Whenever an instruction is dispatched to a functional unit with its source operands available, a ''1'' is shifted in the PUM. A ''0'' is shifted in if no operation is initiated. The PUM shifts every cycle. In this way, a ''1'' in the PUM follows the corresponding operands in the pipe. When a result reaches the end of the pipe, it is routed through the result bus to the reorder buffer only if a ''1'' appears at the end of the PUM. Otherwise it is discarded. For a rollback of n cycles, the n most recent entries in the PUM are set to ''0'' so that results corresponding to those bits get discarded a few cycles later, when they reach the end of the pipeline. In this way any operations that have entered the pipelined *after* the error occurred, are invalidated. This is necessary since operations could enter the pipeline with corrupted operands produced by previous erroneous operations. Note that if a structure similar to the reorder buffer described in the previous sections is used, its most recent entries are also invalidated since those results should not be available for instructions that will be replayed.

Since the functional units are not rolled back internally, results produced by valid instructions during the past n cycles must be ''replayed'' (see Section 6.5). Units connected to the functional units will thus get data just as if the pipeline were rolled back internally. The replay capability is provided by a small *replay memory* (Figure 7.16), which stores all valid results coming out of the pipeline, and a *replay pointer* (RB) which points to the the location which provides the data for the result

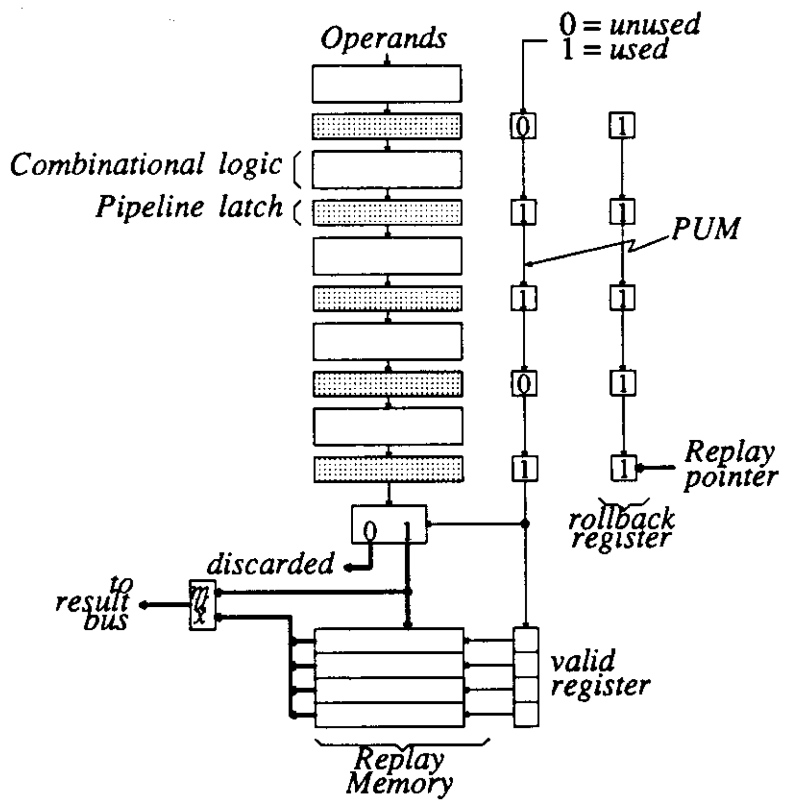


Figure 7.16: Pipeline usage monitor (PUM) associated with each functional unit in the CPU. Rollback of the pipe as a whole is simulated.

bus. During normal operation the replay pointer points to the last latch in the pipeline. The *rollback register* (RR) is a shift register that keeps track of which instructions in a functional unit will be canceled due to micro rollback. It is initially filled with "1" indicating that no rollback has occurred. Every cycle, during normal operation, a "1" is shifted in the RR. Upon a rollback of *n* cycles, the *n* most recent entries of the rollback register are set to "0". The replay pointer is moved down *n* positions so that it points to a replay memory and a *valid register* (VR) entry. The recovery procedure starts by re-sending valid results from the replay memory at the location pointed to by the rollback pointer. The pointer moves up only when a ''0'' appears at the end of the rollback register indicating a bubble in the pipeline. Eventually the pointer is moved back into its normal position since the number of bubbles in the RR equals the number of positions that the pointer was moved. This is shown in Figure 7.17. The pipeline does not need to be stalled and full throughput is achieved. There is no contention for the result bus since the result shift register is also rolled back to allocate the resource.

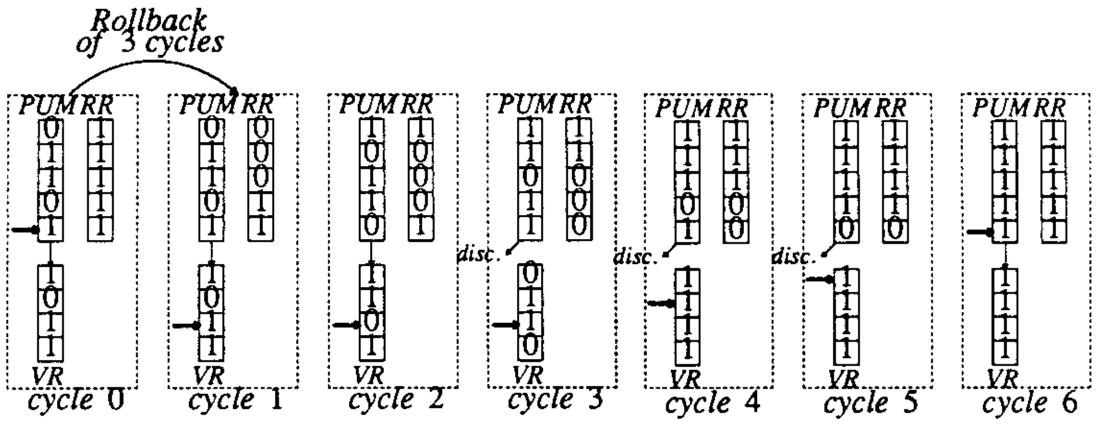


Figure 7.17: Example of a three cycle rollback for a 5-deep pipeline.

7.4.1. Elimination of the Result Shift Register

The presence of a PUM for each functional unit suggests the elimination of the result shift register (RSR) by distributing some of the logic to the functional units. This would eliminate unnecessary stalls in the issue unit due to conflicts in reserving the result bus in advance (through the RSR). Arbitration for the result bus can be done when results are produced, by assigning fixed priorities to the different functional units. A similar mechanism is used in the Motorola 88100 to allocate the ''write-back'' slot to one of its functional units [Mele89]. For the Motorola 88100 priority is given to single-cycle instructions (most fixed-point instructions), then to the integer multiplier, the floating-point multiplier, the floating-point adder, and finally to the load unit. To prevent long stalls of low priority instructions, a high priority can be given to units which have requested the write-back slot more than one cycle ago. Johnson claims that the added circuitry to implement the "priority to old instructions" scheme is minimal [John89].

The *RB_E* tags, previously in the RSR (Figure 7.13), are used to route a result to the reserved location in the reorder buffer. By eliminating the RSR, *RB_E* can be distributed to the functional units. The PUM (pipe usage monitor) of each pipeline can be made wider to accommodate this added tag. Because of the possibility of contention for the result bus, a pipeline may have to stall and send a busy signal to the issuing unit. Even when the pipeline is stalled, the intermediate stages continues to "move" towards commitment every cycle. This must be acknowledged by a mapper similar to the one described in Section 6.3.1.

7.5. Conclusion

In this chapter we investigated schemes for integrating micro rollback with mechanisms normally associated with speedup techniques for high performance processors.

We developed a method for evaluating the area overhead of several designs that we proposed. The method is based on accurate measurements of layouts of a complete chip and on the fact that most components can easily be scaled. The calculations based on this method gave us clear-cut ideas of which methods are expensive in terms of area and also indicated how much area should be dedicated to the extra circuitry required for micro rollback. Layouts of the several designs would have given more accurate results but would not have changed the conclusions that we reached.

We have shown that *register renaming* can be used to implement micro rollback in a simple processor. Even though the overhead was large compared to the DWB method, the method is attractive for processors with hardware register renaming already implemented on-chip for performance reasons.

We introduced a structure called a *standby reorder buffer* which allows several speedup techniques such as out-of-order execution, branch repair, and register renaming to coexist with micro rollback. Mark bits were added to the standby reorder buffer to provide micro rollback. The SRB was shown to be substantially smaller than an equivalent scheme using DWBs to roll back the structures (RSR and RB).

The goal of this chapter was twofold. First, it was shown that for existing high-performance processors, there are efficient ways for adding micro rollback to the architecture. Second, by investigating ways of combining a variety of techniques enhancing parallelism for a CPU with micro rollback, we hope to demonstrate to computer architects that it is possible to design powerful fault-tolerant processors using micro rollback as the main method for parallel error detection and fast error recovery.

Chapter Eight
Summary

This dissertation has addressed the problem of achieving a high level of fault tolerance for a computer system while maintaining high performance. The target systems operate in hostile environments, where a high rate of errors is likely. Due to the real-time constraints of the applications, long interruptions in service are not acceptable, adding to the system requirements that recovery from errors must be achieved quickly.

Traditional methods for satisfying the above requirements lead to costly replication of the hardware and/or reduced performance. Performance degradation results from implementing concurrent error detection and correction with circuits operating in series with intermodule communication.

A system based on micro rollback circumvents these problems by allowing error checking to be performed in parallel with inter-module communication. The optimistic assumption that a system operates without faults ''most of the time'' allows the removal of checkers from the communication paths between modules. This leads to faster inter-module communication and/or shorter pipelines, contributing to improved performance. Micro rollback allows a module to begin processing all inputs immediately upon arrival, despite the possibility that they are erroneous. Upon the eventual detection of an error, micro rollback provides fast restoration of the system to a valid state.

The theoretical value of micro rollback could be undermined if inefficient implementations of the method are used. Simple replication of the storage for the complete state of a module would lead to a large area overhead and reduced performance. There are two sources for the potential performance penalty. First, the added capacitance of the replicated circuitry would slow down the processor. Second, on-chip caches and similar circuitry providing speedups, would have to be decreased in size or even removed in order to accommodate the additional circuitry.

Techniques for implementing micro rollback with low area and performance overhead were developed throughout this dissertation. For a processor, these involve introducing a delayed-write buffer (DWB) which delays commitment of changes to storage, such as a large register file. Efficient methods for checkpointing and restoring state registers, a stack of registers (e.g. *last_pc, pc, next_pc*), or register files modified less frequently than once per cycle were also developed. For memory systems (e.g. caches), we have shown that micro rollback can be implemented using a DWB similar to the one used for the register file.

Detailed layouts and extensive simulations of key building blocks have confirmed that micro rollback can be implemented with little area and performance overheads. The area overhead for adding micro rollback comes mainly from the extra storage required for saving the uncommitted modifications and, to a lesser extent, from the more complex control circuitry. The extra processing required for micro rollback occurs mainly in parallel with normal operation. The small performance degradation is due mostly to the longer buses, a small increase of the fan out of some circuits, an extra level of multiplexors, and larger decoders.

In a multi-module system, a rollback signal generated by one module must be sent to the rest of the system so that a consistent state can be reached after all the modules have rolled back. We have developed a mechanism for logging the occurrence of recent intermodule transactions. These logs are used to coordinate rollbacks, determining the distance that each module should roll back in order to maintain a consistent system state.

We have shown that it is possible to implement hybrid systems, consisting of modules capable of rollback and modules not capable of rollback. Rollback domain interface units (RDIUs), serving as the interface between the different module types, have the capability to buffer data transfers, delaying their delivery to their destination or replaying them when necessary. Several RDIUs were described for a micro rollback capable processor (MRCP) interacting with standard memory subsystems (e.g. caches supporting burst accesses and pipelined accesses). It was shown that simple RDIUs can also be used to interface an MCRP with loosely-coupled and tightly-coupled coprocessors which are not capable of rollback.

We proposed the integration of micro rollback with techniques designed to increase the performance of uniprocessors, such as out-of-order execution, branch repair, and register renaming. It was shown that these techniques are compatible with micro rollback. We believe that future generations of VLSI processors will incorporate some of these speedup techniques. Our work should encourage computer architects to consider adding micro rollback and speedup techniques simultaneously so that both high performance and a high level of fault tolerance can be achieved.

Bibliography

[AMD87] Advanced Micro Devices, *Am29000 Streamlined Instruction Processor User's Manual.* 1987.

[Arya85] S. Arya, "An optimal instruction scheduling model for a class of vector processors," *IEEE Transactions on Computers* **34**(11) pp. 111-122 (November 1985).

[Aviz71a] A. Avizienis, G. C. Gilley, F. P. Mathur, D. A. Rennels, J. A. Rohr, and D. K. Rubin, "The STAR (Self-Testing-And-Repairing) Computer: An investigation of the theory and practice of fault-tolerant computer design," *IEEE Transactions on Computers* **C-20**(11) pp. 1312-1321 (November 1971).

[Aviz71b] A. Avizienis, "Arithmetic Error Codes: Cost and Effectiveness Studies for Application in Digital System Design," *IEEE Transactions on Computers* **C-20**(11) pp. 1322-1330 (November 1971).

[Brow90] E. W. Brown, A. Agrawal, T. Creary, M. F. Klein, D. Murata, and J. Petolino, "Implementing Sparc in ECL," *IEEE Micro* **10**(1) pp. 10-22 (February 1990).

[Cart76] W. C. Carter and C. E. McCarthy, "Implementation of an Experimental Fault-Tolerant Memory System," *IEEE Transactions on Computers* **C-25**(6) pp. 557-568 (June 1976).

[Cast82] X. Castillo, S. R. McConnel, and D. P. Siewiorek, "Derivation and

Calibration of a Transient Error Reliability Model," *IEEE Transactions on Computers* **C-31**(7) pp. 658-671 (July 1982).

[Cens78] L. M. Censier and P. Feautrier, "A New Solution to Coherence Problems in Multicache Systems," *IEEE Transactions on Computers* **C-27**(12) pp. 1112-1118 (December 1978).

[Chau88] S. Chau and D. Rennels, "Design Techniques for a Self-Checking Self-Exercising Processor," *Proceedings of the International Workshop on Defect and Fault Tolerance in VLSI Systems*, pp. 191-202 (October 1988).

[Chow89] P. Chow, *The MIPS-X RISC Microprocessor,* Kluwer Academic Publishers (1989).

[Ciac81] M. L. Ciacelli, "Fault Handling on the IBM 4341 Processor," *11th Fault-Tolerant Computing Symposium*, pp. 9-12 (June 1981).

[Cmel91] R. F. Cmelik, S. I. Kong, D. R. Ditzel, and E. J. Kelly, "An analysis of MIPS and SPARC Instruction Set Utilization on the SPEC Benchmarks," *Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pp. 290-302 (April 1991).

[Cray77] Cray Research Inc, *CRAY-1 Computer System. Hardware Reference Manual.* 1977.

[Cray84] Cray Research, *Cray X-MP Computer Systems Four-Processor Mainframe Reference Manual.* October 1984.

[Davi81] R. P. Davidson, M. L. Harrison, and R. L. Wadsack, "BELLMAC-32: A Testable 32 Bit Microprocessor," *1981 International Test*

Conference Proceedings, pp. 15-20 (October 1981).

[Davi85] H. L. Davis, "A 70-ns Word-Wide 1-Mbit ROM With On-Chip Error-Correction Circuits," *IEEE Journal of Solid-State Circuits* **SC-20**(5) pp. 958-963 (October 1985).

[Down64] R. W. Downing, J. S. Nowak, and L. S. Tuomenoksa, "No. 1 ESS Maintenance Plan," *Bell System Technical Journal* **43**(5) pp. 1961-2019 (September 1964).

[Doyl81] E. A. Doyle, "How Parts Fail," *IEEE Spectrum* **18**(10) pp. 36-43 (October 1981).

[Froh77] R. A. Frohwerk, "Signature Analysis: A New Digital Field Service Method," *Hewlett-Packard Journal*, pp. 2-8 (May 1977).

[Glas85] L. A. Glasser and D. W. Dobberpuhl, *The Design and Analysis of VLSI Circuits*, Addison-Wesley (1985).

[Groh90] G. F. Grohoski, "Machine Organization of the IBM RISC System/6000 processor," *IBM Journal of Research and Development* **34**(1) pp. 37-58 (January 1990).

[Gros83] T. R. Gross, "Code Optimization of Pipeline Constraints," *Doctoral Dissertation*, Computer Systems Lab., Stanford University, (1983).

[Grov90] R. D. Groves and R. Oehler, "RISC System/6000 Procesor Architecture," *IBM RISC System/6000 Technology*, pp. 16-23 (1990).

[Hamm50] R. W. Hamming, "Error Detecting and Error Correcting Codes," *The Bell System Technical Journal* **29**(2) pp. 147-160 (April 1950).

[Henn90] J. L. Hennessy and D. A. Patterson, *Computer Architecture A Quantitative Approach,* Morgan Kaufmann, San Mateo, CA (1990).

[Hopk78] A. L. Hopkins, T. B. Smith, and J. H. Lala, "FTMP — A highly reliable fault-tolerant multiprocessor for aircraft," *Proceedings of the IEEE* **66**(10) pp. 1221-1239 (October 1978).

[Horo87] M. Horowitz, P. Chow, D. Stark, R. T. Simoni, A. Salz, S. Przybylski, J. Hennessy, G. Gulak, A. Agarwal, and J. M. Acken, "MIPS-X: A 20-MIPS Peak, 32-bit Microprocessor with On-Chip Cache," *IEEE Journal of Solid-State Circuits* **22**(5) pp. 790-799 (October 1987).

[Hwu87] W. W. Hwu and Y. N. Patt, "Checkpoint Repair for Out-of-order Execution Machines," *The 14th Annual International Symposium on Computer Architecture*, pp. 18-26 (June 1987).

[Inte87] Intel, *80387 Programmer's Reference Manual.* 1987.

[Inte89] Intel, *80960CA 32-bit High Performance Embedded Processor — datasheet.* September 1989.

[John87] M. Johnson, "System Considerations in the Design of the Am29000," *IEEE Micro* **7**(4) pp. 28-41 (August 1987).

[John89] W. M. Johnson, "Super-Scalar Processor Design," *Doctoral Dissertation*, Computer Systems Lab., Stanford University, (June 1989).

[Kane87] G. Kane, *MIPS R2000 Risc Architecture,* Prentice Hall (1987).

[Kate83] M. G. Katevenis, ''Reduced Instruction Set Computer Architectures for VLSI,'' Report No. UCB/CSD 83/141, Computer Science Division (EECS), University of California at Berkeley, CA 94720 (October 1983).

[Kell75] R. M. Keller, ''Look-Ahead Processors,'' *Computing Surveys* **7**(4) pp. 177-195 (December 1975).

[Kohn89] L. Kohn and S.-W. Fu, ''A 1 000 000 Transistor Microprocessor,'' *1989 International Solid-State Circuits Conference Digest of Technical Papers*, pp. 54-55 (February 1989).

[Koo87] R. Koo and S. Toueg, ''Checkpointing and Rollback-Recovery for Distributed Systems,'' *IEEE Transactions on Software Engineering* **SE-13**(1) pp. 23-31 (January 1987).

[Lamp78] L. Lamport, ''Time, Clocks, and the Ordering of Events in a Distributed System,'' *Communications of the ACM* **21**(7) pp. 558-565 (July 1978).

[Lamp79] L. Lamport, " ''How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs",'' *IEEE Transactions on Computers* **C-28**(9) pp. 690-698 (September 1979).

[Lee86] D. D. Lee, ''Data Path Design Considerations for a High Performance VLSI Multiprocessor,'' University of California at Berkeley Technical Report UCB/CS Division (November 1986).

[Lee89] D. D. Lee, S. I. Kong, M. D. Hill, G. S. Taylor, D. A. Hodges, R. H. Katz, and D. A. Patterson, ''A VLSI Chip Set for a Multiprocessor

Workstation — Part I: An RISC Microprocessor with coprocessor interface and support for symbolic processing,'' *IEEE JSSC* **24**(6) pp. 1688-1698 (December 1989).

[Lee84] J. K. F. Lee and A. J. Smith, ''Branch Prediction Strategies and Branch Target Buffer Design,'' *Computer* **17**(1) pp. 6-22 (January 1984).

[Lian90] M. T. Liang, ''Micro Rollback on a VLSI RISC: Design and Implementation of the UCLA Mirror Processor,'' Computer Science Department Technical Report CSD-900042, University of California, Los Angeles, CA (December 1990).

[Logr72] L. Logrippo, ''Renamings in program schemas,'' *Proceedings of the IEEE 13th Annual Symposium on Switching and Automata Theory*, pp. 67-70 (October 1972).

[Mack78] D. Mackie, ''The Tandem 16 NonStop System,'' *Infotech*, pp. 145-161 (1978).

[Mak82] G.-P. Mak, J. A. Abraham, and E. S. Davidson, ''The Design of PLAs with Concurrent Error Detection,'' *Proc. 12th Int. Symp. Fault-Tolerant Computing*, pp. 303-310 (June 1982).

[McCl85] E. J. McCluskey, ''Built-In Self-Test Techniques,'' *IEEE Design and Test* **2**(2) pp. 21-28 (April 1985).

[Mead80] C. Mead and L. Conway, '','' in *Introduction to VLSI Systems*, ed. Addison-Wesley , (1980).

[Mele89] C. Melear, ''The Design of the 88000 RISC Family,'' *IEEE Micro*

9(2) pp. 26-38 (April 1989).

[Moto85a] Motorola, *MC68881 Floating-Point Coprocessor User's Manual.* 1985.

[Moto85b] Motorola, *MC68020 32-Bit Microprocessor User's Manual,* Prentice-Hall, Englewood Cliffs, NJ (1985).

[Muld91] J. M. Mulder, N. T. Quach, and M. J. Flynn, ''An Area Model for On-Chip Memories and its Application,'' *IEEE Journal of Solid-State Circuits* **26**(2) pp. 98-106 (February 1991).

[Mura89] K. Murakami, N. Irie, M. Kuga, and S. Tomita, ''SIMP (Single Instruction stream/Multiple instruction Pipelining): A Novel High-Speed Single-Processor Architecture,'' *16th Annual International Symposium on Computer Architecture,* pp. 78-85 (May 1989).

[Pate82] J. H. Patel and L. Y. Fung, ''Concurrent Error Detection in ALUs by Recomputing with Shifted Operands,'' *IEEE Transactions on Computers* **C-31**(7) pp. 589-595 (July 1982).

[Patt82] D. A. Patterson and C. H. Sequin, ''A VLSI RISC,'' *Computer* **15**(9) pp. 8-21 (September 1982).

[Perr89] T. S. Perry, ''Intel's Secret Is Out,'' *IEEE Spectrum,* pp. 22-28 (April 1989).

[Rand78] B. Randell, P. A. Lee, and P. C. Treleaven, ''Reliability Issues in Computing System Design,'' *Computing Surveys* **10**(2) pp. 123-165 (June 1978).

[Rao89] T. R. N. Rao and E. Fujiwara, *Error-Control Coding for Computer Systems,* Prentice Hall (1989).

[Renn86] D. Rennels and S. Chau, "A Self-Exercising Self-Checking Memory Design," *16th Fault-Tolerant Computing Symposium,* pp. 378-363 (1986).

[Reyn78] D. Reynolds and G. Metze, "Fault Detection Capabilities of Alternating Logic," *IEEE Transactions on Computers* **C-27**(12) pp. 1093-1098 (December 1978).

[Rowe88] C. Rowen, M. Johnson, and P. Ries, "The MIPS R3010 Floating-Point Coprocessor," *IEEE Micro* **8**(3) pp. 53-62 (June 1988).

[Sedm80] R. M. Sedmak and H. L. Liebergot, "Fault Tolerance of a General Purpose Computer Implemented by Very Large Scale Integration," *IEEE Transactions on Computers* **C-29**(6) pp. 492-500 (June 1980).

[Sher84] R. W. Sherburne, M. G. H. Katevenis, D. A. Patterson, and C. H. Séquin, "A 32-Bit NMOS Microprocessor with a Large Register File," *IEEE Journal of Solid-State Circuits* **SC-19**(5) pp. 682-689 (October 1984).

[Siev82] M. Sievers and D. A. Rennels, "An LSI Totally Self-Checking Hamming Coded Memory Interface," *International Symposium on Circuits and Systems,* pp. 1176-1179 (May 1982).

[Sit89] H. P. Sit, M. R. Nofal, and S. Kimn, "An 80 MFLOPS Floating-Point Engine in the Intel i860 Processor," *International Conference on Computer Design,* pp. 374-379 (October 1989).

[Smit88] J. E. Smith and A. R. Pleszkun, "Implementing Precise Interrupts in Pipelined Processors," *IEEE Transactions on Computers* **C-37**(5) pp. 562-573 (May 1988).

[Smit90] M. D. Smith, M. S. Lam, and M. A. Horowitz, "Boosting Beyond Static Scheduling in a Superscalar Processor," *Proc. 17th Annual Symposium on Computer Architecture, Computer Architecture News* **18**(2) pp. 344-354 ACM, (June 1990).

[Sohi87] G. S. Sohi and S. Vajapeyam, "Instruction issue logic for high-performance, interruptible pipelined processors," *Proceedings of the 13th Annual Symposium on Computer Architecture* **15**(2) pp. 27-34 (June 1987).

[Sohi90] G. S. Sohi, "Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers," *IEEE Transactions on Computers* **39**(3) pp. 349-359 (March 1990).

[Sun91] Sun Microsystems Inc, *The SPARC Architecture Manual, Version 8*. January 1991.

[Take80] K. Takeda and Y. Tohma, "Logic Design of Fault-Tolerant Arithmetic Units Based on the Data Complementation Strategy," *10th Fault-Tolerant Computing Symposium*, pp. 348-350 (October 1980).

[Tami88a] Y. Tamir, M. Tremblay, and D. A. Rennels, "The Implementation and Application of Micro Rollbacks in Fault-Tolerant VLSI Systems," Computer Science Department Technical Report CSD-880004, University of California, Los Angeles, CA (January

1988).

[Tami88b] Y. Tamir, M. Tremblay, and D. A. Rennels, "The Implementation and Application of Micro Rollback in Fault-Tolerant VLSI Systems," *18th Fault-Tolerant Computing Symposium*, pp. 234-239 (June 1988).

[Tami90] Y. Tamir and M. Tremblay, "High-Performance Fault-Tolerant VLSI Systems Using Micro Rollback," *IEEE Transactions on Computers* **39**(4) pp. 548-554 (April 1990).

[Tami91] Y. Tamir, M. Liang, T. Lai, and M. Tremblay, "The UCLA Mirror Processor: A Building Block for Self-Checking Self-Repairing Computing Nodes," *21st Fault-Tolerant Computing Symposium*, (June 1991).

[Thor70] J. E. Thornton, *Design of a Computer, the control Data 6600*, Scott, Roresman and Company, Glenview, Ill. (1970).

[Toma67] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Research and Development* **11**(1) pp. 25-33 (January 1967).

[Trem88] M. Tremblay, P. Trajmar, D. Dobrikin, and F. Daneshgaran, "A 32-Bit RISC Processor with Micro Rollback," *CS258 Class Project Report*, Computer Science, University of California at Los Angeles, (June 1988).

[Trem89a] M. Tremblay and Y. Tamir, "Support for Fault Tolerance in VLSI Processors," *International Symposium on Circuits and Systems*, (May 1989).

[Trem89b] M. Tremblay and Y. Tamir, "Fault-Tolerance for High-Performance Multi-Module VLSI Systems Using Micro Rollback," *Decennial Caltech Conference on VLSI*, (March 1989).

[Tsao82] M. M. Tsao, A. W. Wilson, R. C. McGarity, C.-J. Tseng, and D. P. Siewiorek, "The Design of C.fast: A Single Chip Fault Tolerant Microprocessor," *Proc. 12th Int. Symp. Fault-Tolerant Computing*, pp. 63-69 (June 1982).

[Wald85] D. E. Waldecker, C. G. Wrigh, M. S. Schmookler, T. G. Whiteside, R. D. Groves, C. P. Freeman, and A. Torres, "ROMP/MMU Implementation," *IBM RT Personal Computer Technology*, pp. 57-65 (1985).

[Wens78] J. H. Wensley, L. Lamport, J. Goldberg, M. W. Green, K. N. Levitt, P. M. Melliar-Smith, R. E. Shostak, and C. B. Weinstock, "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control," *Proceedings of the IEEE* **66**(10) pp. 1240-1255 (October 1978).

[West85] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design A Systems Perspective*, Addison-Wesley (1985).

[Will91] T. E. Williams and M. A. Horowitz, "A Zero-Overhead Self-Timed 160ns 54b CMOS Divider," *Proceedings of ISSCC*, pp. 98-99 (February 1991).

[Wils85] D. Wilson, "The STRATUS Computer System," *in Resilient*

Computer Systems, pp. 208-231 Collins, (1985).

[Yen87] M. M. Yen, W. K. Fuchs, and J. A. Abraham, ''Designing for Concurrent Error Detection in VLSI: Application to a Microprogram Control Unit,'' *IEEE Journal of Solid-State Circuits* **SC-22**(4) pp. 595-605 (August 1987).